Carnegie Mellon

School of Computer Science

Venari Note 1 May 29, 1990

The Venari Project: Goals and Plans

Jeannette M. Wing

This note presents the main goals of the Venari Project. Venari means “to hunt” in Latin. The thrust of this project is addressing the problem of *search*, hence the name.

The project is meant to be open-ended and ambitious. We intend to provide a loose framework to support interesting language and systems research at CMU for interested faculty, research staff, and graduate students. We touch upon areas in programming and specification language design, semantics, and implementation; concurrent and distributed systems; databases and persistent objects; and software development libraries and environments. We base our new ideas from our experience gained from the Avalon, Camelot, and Ergo Projects; we continue to interact with Ergo faculty, and to a lesser extent, people on the Coda and Mach Projects. This note refers to a paper on Avalon/Common Lisp and three other Venari notes, so the interested reader may wish to read these other documents for further background, motivation, and information.

Though the project is open-ended, we do have some specific goals and plans. This note describes some of these, and in particular, some plans for the first year of the project (Section 6).

1. Overall Goal

> *The primary goal of the Venari Project is to provide software support for storing, accessing, and retrieving objects based on their semantics.*

Objects range in semantic complexity. A text file has little semantics. It is just an uninterpreted string of characters. Accessing a file can be done by its name or by its contents. If we are lucky enough to know the file’s name, we can simply perform the query “get me the file named F.” Suppose instead that we do not know the name of the file, but that we have a facility that implements string matching. Then, we can simply perform the query “give me all the files that contain the substring S.” Of course, current file systems already provide us with facilities beyond these limited access capabilities. The Unix File System and its distributed look-a-likes (e.g., Andrew) provide *grep* facilities that implement efficient and sophisticated string and pattern matching algorithms over names and contents of both directories and files, possibly located at sites remote from the query site. As numbers of files and users grow, we rely more and more on these facilities to find the files that we are interested in.

Taking a step beyond files, we find another class of objects in relational databases. A database record can be viewed as a set of key-value pairs. We typically access records through various predicates on their keys and values, e.g., “give me all records of students who have passed all the Quals.” Here, we almost never access records through their names, i.e., unique identifiers, but instead through their contents, i.e., semantics.

Before we hone in on our particular application domain, let’s consider other kinds of objects and how we might formulate queries based on their “semantics.”

Bibliography entries in a card catalog: Give me all those papers written by Hoare published in CACM.

Module in a program library: Give me a procedure that sorts arrays of integers.

Theorem in a proof library: Give me a proof of the theorem about the commutativity of insert for sets.

Or even graphics and sound:

Map in an atlas: Display a street map of Pittsburgh that shows the Oakland neighborhood.

Face in mug file: Display a picture of the man convicted of robbing the Centre Avenue 7-11 store last July.

Musical pieces in a CD library: Play the recording containing <musical staff and notes go here>.

We are interested in the problems of managing these kinds of objects. We assume they are stored in some kind of *repository*, e.g., card catalogs, program libraries, proof libraries, atlases, police station mug files, CD libraries. We want to be able to search through a repository and retrieve objects of interest. Retrieving objects involves performing a *match* that ranges in algorithmic complexity, depending on the semantic complexity of the object (e.g., checking equality between two strings, checking correspondence between two two-dimensional bit patterns, proving an implication in predicate logic).

2. Our Context: Software Development

Software development can use facilities that provide general search and retrieval of different object repositories. We envision in the development of large software systems the need to access repositories whose objects have rich semantic content, e.g., macro libraries, program module libraries, specification libraries, and proof libraries, as well as more mundane repositories like the file system.

The Venari Project will initially focus on program module libraries. We make the following two assumptions:

1. For a given programming language there exists a library of program modules.
2. Each program module has a *descriptor* that specifies the module’s semantics. This descriptor might range in semantic complexity; it could simply be the module’s name, it could be its signature, or, for a procedure module, it could be a pair of predicates (e.g., its pre- and post-conditions).

Extant program module libraries include the Ada repository, the Ada Generic Library (for list processing), [1] the Smalltalk class hierarchy, MacApp, Mathematica, Scratchpad, and numerous FORTRAN libraries. Programming languages such as C, CLU, ML, Common Lisp, and Modula-3 also provide their users with directories of standard routines, functions, and/or data types.

In practice, our second assumption would be harder to discharge than our first since none of the libraries with which we are familiar contain formal specifications (e.g., pre- and post-conditions) as descriptors of modules. However, most libraries, especially for strongly-typed languages, associate with each module at least name and signature information and some informal commentary; this kind of information will serve initially as our module descriptors. Our hope and expectation is that as formal methods gain in usage, familiarity and popularity, programmers will be willing to attach more formal specifications to their modules as a way not only to enhance software reliability and reusability but also to document their code more precisely.

Our first assumption implies, by our use of "library," that our objects of interest are *persistent*, i.e., their lifetime exceeds the lifetime of any computation, e.g., search, accessing them. There are many degrees of persistence and many degrees of *atomicity* of a computation that we need to support. These are discussed in Section 4.1.

In general, we make no assumption about the relative location of the query site and the library site(s); they can be physically distinct. Moreover, a library may itself be logically and/or physically partitioned into separate pieces. For example, it might be logically separated into a set of object descriptors and the set of objects themselves with a correspondence between the two sets; hence, in practice we may actually deal with more than one library, e.g., a specification library and a code library, with a mapping providing links between specifications and code. Looking further into the medium-term future, we envision also dealing with proof libraries with links between theorems and proofs and between theorems and (program module) specifications. Finally, because of physical memory limitations, and/or a desire to enhance availability and reliability, a library of objects may physically be split and/or replicated across multiple sites.

We also make no assumption about the homogeneity of sites. Typically, we would perform a query on a local workstation (e.g., an RT) that requires a search over objects residing remotely, accessible through an object server (for now, read "file server") running on a machine of different type (e.g., a Micro-Vax).

Management of all of these kinds of objects on a large scale strain current file system and database technology in two ways related to scale: complexity of the objects themselves and quantity of objects to be managed.

How then do we plan to achieve our vision?

- The fundamental function we need to provide is *match*: When searching in an object repository, all objects whose descriptors match that of a given query are returned.
- Since we are dealing with objects of varying degrees of persistence and computations of varying degrees of atomicity, we need to provide some primitive notion of *persistent* storage.
- Since we may deal with repositories remote from the query site, we need to support *distributed* queries.
- To further optimize search we may wish to perform *concurrent* searches over a single library and certainly concurrent searches across different libraries.

[1] D.R. Musser and A.A. Stepanov, *The Ada Generic Library*, Springer-Verlag, 1989.

Below we discuss each area in turn. We view the first subproblem as our motivating *application* (Section 3) for the underlying *infrastructure* (Section 4) provided by solutions to the other three. In Sections 4.1, 4.2, and 4.3, we briefly provide some motivation and sketch out some open issues related to the issues of persistence, distribution, and concurrency; Venari Note 2 addresses these issues in somewhat more detail. We have made the decision to prototype our infrastructure in ML and justify our language implementation choice in Section 5.1.

3. Application: Searching and Matching

3.1. Context

As mentioned, we assume each object has a *descriptor*. Example object descriptors for a program module are a C function name, an ML signature, an Ada package specification, an Anna specification, and a Larch interface. We use object descriptors, not just to associate information with objects, but as the basis for comparison in our queries. Given a query Q, we retrieve one or more objects each of whose own descriptor, D_i, *matches* Q. Technically, Q and D_i must be in some formal logical language, which serves as a query language, and we must check the following implication for each i: $D_i \Rightarrow Q$. In practice, object match must be computable by some *match engine* within reasonable time bounds.

The notions of object descriptors, object match, and match engines have been implemented in various guises already. Consider the following three different communities within computer science:

- The operating systems community would call object descriptors "file names" (with perhaps some protection bits) and object match "pattern matching" where a query might be a regular expression over strings. The UNIX *grep* facility is an indispensable match engine.
- The software engineering community would call object descriptors "formal specifications" and object match "satisfaction." The match engine would be some kind of theorem prover or proof checker. Verification technology has not reached the point where object match can be done automatically, but it can be done for restricted domains and/or restricted subsets of certain specification languages. Prolog is a notable example where unification is used for object match, i.e., finding instantiations for variables in a given query.
- The programming language semantics community would call object descriptors "types" (partial-semantics predicates) and object match, "type checking and inference." The match engine would be a type checking or inference algorithm. Matching amounts to determining if a query is a subtype of an object descriptor. Although for some languages type checking and inference is inefficient or undecidable, again there are restricted subsets of higher-order typed lambda calculi in which type inference can be efficiently performed. ML is a notable example.

The goal of our work is to move beyond simple string pattern-matching as currently done for file retrieval, but not necessarily require full theorem-proving capability as needed for machine-assisted program verification.

3.2. Match Predicate

Since we have distilled the research problem down to computing the predicate $D_i \Rightarrow Q$, we now face the matter of how to check whether the predicate holds for each object i. One approach is to define a language with which we can write the D_i's and Q, define the formal meaning of $\Rightarrow$, and build a semantic checker (e.g., a theorem prover) for determining match. Though this approach is feasible, it restricts the query and description language to be a single, presumably "standard" language (e.g., first-order logic with a built-in set of function and predicate symbols). Hence, it restricts the notion of match to that of the interpretation of that language (e.g., the usual notion of implication in first-order logic).

An alternative approach is to permit multiple query languages and provide a means to use them together. We want to build an environment within which we can experiment with existing and future match engines, each of which comes equipped with its own specialized query language. Each query language is specific to the types of objects being sought and the match engine performing the search. We immediately gain the ability to use different existing languages that express entities such as types, function signatures, and algebraic specifications as different query languages. Although the query languages may not be computationally complete themselves, our implementation language, ML (see Section 5.1), can "tie" them together and provide a means to do transitive closure or recursive queries. Most importantly, with our approach the query languages will be directly matched to the objects they describe (e.g., ML type signatures describe ML functions; Ada package specifications describe Ada package bodies).

One justification for our approach is the recognition that various communities already have their own notions of "descriptor" and "match;" we need only define a generic *match* predicate. We would make it possible to invoke this generic predicate with each different flavor of query given a specific repository of persistent objects to be searched.

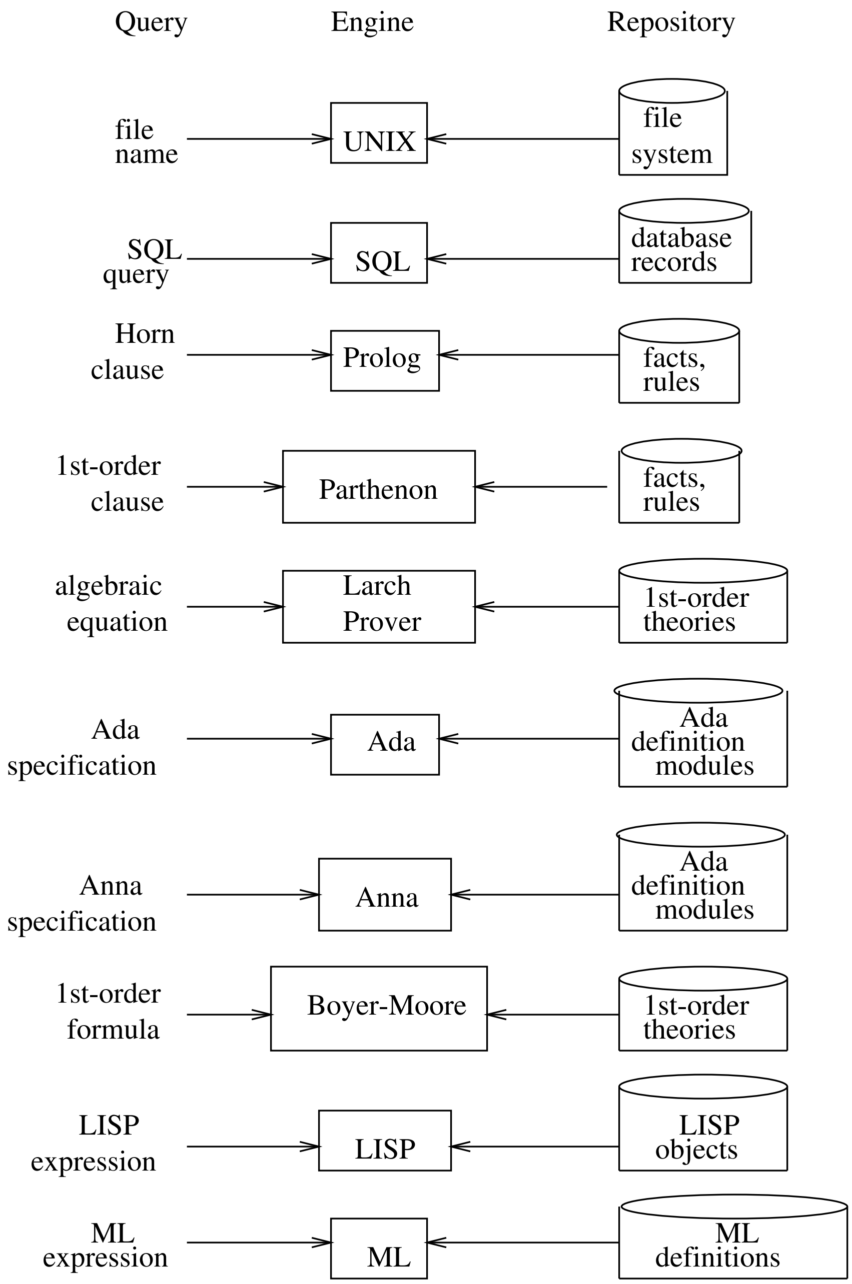
As shown at the bottom of Figure 1, the arguments to the *match* predicate are a query, a match engine, and an object repository:

```
match: query X engine X repository -> objects.
```

The picture shows various match engines in rectangles and corresponding object repositories to the right of each engine. As of today, except for UNIX and Lisp, these engines work in isolation. Thus, for example, without some amount of hand-crafting, it is impossible to search through a Prolog database and an Ada package library with a single query.

Figure 1 illustrates our approach's flexibility and extensibility. First and most important, our framework allows us to add more engines as they become available. We can also build immediately from engines available today (as is true of all shown in the figure, plus others not shown) and incorporate future engines as they are developed *independently*. Second, we can replace any of the individual engines that are extremely computation-intensive (e.g., the Boyer-Moore theorem prover) by newer, perhaps more efficient, versions without changing its abstract interface. The Parthenon engine, for example, has functionality similar to the Prolog engine but since it uses a parallel search and backtracking strategy on a multiprocessor, it outperforms standard Prolog. [2] Third, as theoretical research advances with respect to handling (automatically) richer subsets of logic (possibly higher-order), we can exploit more and more type information in our object

[2] S. Bose, E.M. Clarke, D.E. Long, and S. Michaylov, "Parthenon: A Parallel Theorem Prover for Non-Horn Clauses," in *Proceedings of the Fourth IEEE Symposium on Logic in Computer Science*, 1989, also CMU-CS-88-137.



match (query, engine, repository)

management approach. For example, an Ada environment today can easily support search through an Ada package library based purely on the signature information of an Ada package. However, were a rich enough subset of first-order logic supported by some theorem-proving engine, then an Anna specification or Larch equivalent (i.e., signature information plus pre- and post-conditions) could be used. Fourth, instead of exact match, we can substitute partial match predicates where it makes sense or would speed up performance. Each of the engine-specific instances invokable through the *match* predicate can be defined to return an exact, subset, or superset of objects that "match" the given query, depending on the application and user's intent. One obvious generalization would be then to allow for all such types of match.

We plan to build a *framework* into which we can incorporate multiple match engines. The framework will support multiple query languages. Each match engine serves as a browsing facility through its corresponding object repository. The first browsing tool we will build is one for ML.

3.3. Scenario

Example 1: We can use the Larch Shared Language (LSL) [3] as a basis for one of our query languages and the Larch Prover (LP) [4] as one of our match engines. Consider the following sample use of the generic *match* predicate. It retrieves all first-order theories for which the two stack axioms given in the query hold. (The concrete syntax used below is meant to be suggestive, not definitive.)

```
match ( {F(H(s,e)) == s, G(H(s,e)) == e },
        "Larch Prover",
        "Larch Handbook" )
```

The first argument to *match* is a query in the form of a set of two equations written in LSL. The second argument is the name of a match engine, the Larch Prover, which takes as input a set of equations. The third argument is the name of an object repository, the Larch Handbook, that stores first-order theories. Note that names in the query are meta-variables bound during satisfaction to actual names (like "push,", "pop," and "top") given in axioms of retrieved objects.

Example 2: Now consider a second use of the generic *match* predicate. We use Larch interface languages, e.g., Larch/Ada, to describe the semantics of program modules, e.g., Ada packages. The instantiation of *match* below retrieves all Ada procedures that return the top element of a stack, modifying the stack object as a side effect.

```
match ( [ procedure P (in out s: stack, out e: elem)
                pre: ~isEmpty(s)
                post: (e = top(s) & s' = pop(s)) ],
            "Larch/Ada Interface Checker",
            "Ada package library" )
```

---

[3] J.V. Guttag, J.J. Horning and J.M. Wing, "The Larch Family of Specification Languages," *IEEE Software*, September 1985, pp. 24–36.

[4] S.J. Garland and J.V. Guttag. "An overview of LP, the Larch Prover," in *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, Chapel-Hill, N.C., 1989, pp. 137–151.

The first argument to *match* is a query written as a Larch/Ada interface specification. It contains signature information plus pre- and post-conditions. The pre-condition requires that the procedure not be called with an empty stack; the post-condition asserts that upon return, the procedure will remove and return the top element of the input stack, modifying the original stack value. The second argument to *match* is the name of a different match engine, the Larch/Ada Interface Checker (which does not exist). The third argument is the name of the object repository, the Ada package library. Note again that procedure names and arguments within queries are meta-variables. Thus, a procedure's exact name is irrelevant; as long as its signature matches and its behavior satisfies the pre- and post-conditions, then it is returned.

Example 3: We can use match engines together in a single *mixed* query, perhaps requiring searches through more than one object repository. Consider the following query:

> Give me all Ada packages that contain a procedure that given a stack argument, removes and returns the top element.

In principle we might invoke three separate match engines: a type checker for the procedure signature, a Larch/Ada checker for the procedure's pre- and post-conditions, and LP for verifying other properties of the types of the procedure's arguments. The search would be over two separate libraries: the Ada package library and the Larch Handbook. The result of this mixed query would be the set of procedures that not only have the correct signature and satisfy the pre- and post-conditions of the given query, but also ensure the semantics of the types (stacks and elems) of its arguments.

In general, we envision that our framework would provide a way for users to use multiple match engines. If initially only an approximate match is necessary, a user can use a simple, but fast, match engine. The user can then invoke more sophisticated match engines to obtain more precise matches over the objects returned from the first match. For example, suppose a user wants a procedure that sorts arrays of elements. The first query might simply use a grepper to retrieve all procedures and packages that contain the string "sort." Of the returned modules, the second query might retain only those whose signatures matches "array[T] -> array[T]." A third query might retain only those whose pre- and post-conditions match the query informally expressed as "**pre**: input array is not empty; **post**: returns a new array containing the same elements as input array arranged in sorted order." Finally, given that only a few modules remain, the user might be willing to pay the cost of using a theorem prover to show that some desired property P holds, where P is not explicitly stated in the specification, but provable from it. In our example, P might be the property that elements of the input array are not modified by the sort procedure. Performing this last step in a *pipelined query* would be particularly useful when the user is faced with choosing components for a critical part of a large system; here, the user could gain some assurance that additional desired properties hold. Notice that even for this simple example, the user might miss objects that could match (e.g., if a sort procedure does not contain the string "sort"; if a generic sort procedure takes the ordering relation as another argument; if the pre-condition is true; if the "no change" property holds of the procedure but the prover is too weak or the specification lacks the information needed to show it).

3.4. Open Issues

Extending Existing Specification Languages and Tools

In the LSL stack example, we naively "used" LP as a match engine. Currently, LP is used as an interactive proof checker; it cannot and is not intended to be used as an automatic theorem prover. To use LP as a match engine, we would need (1) to define a restricted, but sufficiently rich subset of LSL to be a query language and (2) to define a computationally tractable notion of object match. (Defining it in terms of theory equality for exact match, or theory containment for partial match, would make checking for it undecidable.)

As in the Larch/Ada example, to use Larch interface languages as query languages, we would need (1) to build new match engines that handle Larch interface languages (LP handles only LSL) and (2) to use these languages and engines in conjunction with LP.

Partial Match

The Larch/Ada example shows two ways in which partial, instead of exact, match would be useful. (1) Exact match would preclude finding a procedure with pre-condition **true**, though it may have the same post-condition. One kind of partial match could yield a procedure with a weaker pre-condition (and perhaps stronger post-condition) that still may be useful to the Ada programmer. (2) Exact match returns only procedures that satisfy the query. A second kind of partial match would return all Ada packages that contain a procedure that satisfies the query. Of course, a combination of both kinds of partial matches would be meaningful as well.

Exploring the Match Design Space

At one extreme, we require only simple string and pattern matching for searching through files and directories; Grep-like facilities suffice for this need. At the other extreme, we require automatic theorem-proving capability, which is clearly unrealistic as well as impractical. However, in between the two extremes lies not a linear spectrum, but a large design space. One axis represents expressibility of the semantics of the objects; the other axis represents the complexity of search.

We have already explored one design point in this space: We have experimented with using Lambda Prolog to gain expressibility through its basis in higher-order logic instead of first-order logic. We gain the immediate advantage of using Lambda Prolog's higher-order unification to perform matching "for free." In our experiment we used Lambda Prolog (instead of the Larch Shared Language) as the assertion language for a Larch interface specification. We used unification as our match engine. We gain some expressibility in using Lambda Prolog as our assertion language since LSL is a subset of first-order; however, we also lose some, since LSL is an equational subset of first-order logic and Lambda Prolog (though higher-order) does not handle equations. Hence, we gain the ability to do match "automatically" which would be impossible using the Larch Prover, but lose somewhat in expressibility. See Venari Note 3 for a description of our Lambda Prolog matcher.

Finally, acknowledging that (1) string pattern-matching is extremely useful in pruning the search space and (2) program modules do not yet have formal specifications, we have implemented a *grepper* in ML (see Venari Note 4) based on the gnu-emacs grep facility.

4. Infrastructure

4.1. Persistence

Degrees of Persistence

What is common in all the examples listed in Section 1 is the *persistent* nature of the objects (bibliography entries, program modules, theorems, maps, faces, music). In principle, an object is *persistent* if its lifetime is forever, i.e., once created it is never destroyed. In practice, an object may vary in degrees of *persistence*:

a *recoverable* object stored in stable storage survives all hardware failures; a *failure-atomic* object stored in non-volatile memory survives soft crashes; a heap-allocated object's lifetime is typically greater than a stack-allocated one's in the course of a program's execution. Maintaining higher degrees of persistence incurs a performance cost. This cost has to be weighed against the cost of recreating the object from scratch (if that even can be done at all). For example, saving the symbol table from a large compilation might make the subsequent recompiles execute faster. The symbol table could also be reused for purposes other than compilation.

Degrees of Atomicity

Analogous to varying degrees of persistence are degrees of atomicity ("all-or-nothing") of a given computation. We identify (at least) these four levels of atomicity:

1. At the lowest (system) level, we assume reading and writing to physical (e.g., disk) memory are atomic operations.

2. At the next higher (system/language) level, we assume operations on objects are atomic. For example, reading and writing to a file, inserting and deleting elements from a set, and adding a new key-value binding to a symbol table and updating an existing one are all examples of atomic operations. Note that we rely on the type semantics of an object (e.g., files, sets, symbol tables) to define the units of atomicity. Supporting this kind of atomicity is the heart of the design and implementation of persistent store for the Venari Project. In reality, the bits representing an object are stored on some physical device; we want to provide the illusion that we can access these objects through their operations atomically.

3. At the next (language) level, we may wish to provide support for *transactions*. Here, we treat a sequence of operations on objects as an atomic unit of computation. Through standard failure-atomic semantics, a transaction completes successfully, in which case we say it *commits*; or it fails, in which case we say it *aborts*. If it commits, all of its effects are visible to the outsider observer; if it aborts, none of its (partial) effects are visible. Numerous synchronization mechanisms like locks and/or timestamps are used to coordinate access to objects by concurrent transactions.

4. At the next (language/application) level, we may wish to provide support for *sessions*. We distinguish between transactions and sessions based on duration in real time. Transactions typically last on the order of seconds or minutes (e.g., read bank account, withdraw money, commit final balance). Sessions typically last on the order of hours or days (e.g., modify five C++ classes, compile revised program, save updated symbol table). Currently, synchronization of sessions is done at best by facilities like RCS and more typically by word-of-mouth ("Send me a message when you're done with that file.")

For each of the degrees of persistence and atomicity, there is an analogous notion of "recoverability." For example, at the transaction level, we would recover from a hard crash by "replaying" a log. The system maintains a log containing sufficient information about objects to resurrect objects to some consistent state following a crash. At the application level, e.g., for sessions, the user might have to handcode recovery code per application and/or painstakingly reexecute by hand all changes made to objects since the last saved session.

Open Issues

We plan initially to focus our attention on building support for the system/language level of atomicity (number 2 above). This requires:

- Building an object interface to a recoverable virtual memory (RVM) interface provided by the operating system (currently being built by Hank Mashburn).
- Adding necessary linguistic extensions to ML to permit programmers to distinguish between objects allocated from volatile memory versus recoverable memory. The two possibilities are declaring a type to be persistent, in which case all objects of that type are persistent, or declaring an object to be persistent.
- Extending/modifying the semantics of ML to characterize formally what persistent objects are.

We will also explore linguistic extensions to ML for providing transaction semantics. There are at least two possibilities: (1) Since ML is mostly functional, we could implicitly associate *begin transaction* and *end transaction* delimiters around code that accesses mutable objects (ref types); or, (2) we could provide new control structures that let programmers explicitly delimit transaction boundaries. The nature of these control structures might in fact look like a function call and might in fact be implemented in terms of call/cc.

Once we have an object interface to a persistent store, we also plan to investigate and build indexing schemes to the persistent store that can expedite search, similar to indices provided by conventional databases.

4.2. Distribution

In the traditional model, an RPC interface serves two purposes. It defines both the calling interface between a client and server and the boundary along which a computation is distributed. The caller (client) is also the initiator process of some computation; the callee (server) is also some initiated process executing on behalf of the caller.

Here, we separate the two purposes. We use the terms *client* and *server* to distinguish between the caller and callee. This client/server distinction defines the interface between the facilities provided by the server programmer, and those provided by the client programmer, just as is true for interfaces in non-distributed programs. *Extending a server's interface* is the mechanism by which clients can define and add application-specific functions to a server's fixed set of functions.

We use the terms *local* and *remote* to distinguish between the initiator and initiated processes of a distributed computation. The local/remote distinction serves to define the boundary along which a computation is distributed. A computation, e.g., a query, is local if it is performed at the process initiating the computation, while a computation, e.g., a search and match, is remote if it occurs at some process different from the initiator. *Remote evaluation* is the mechanism by which we can express this change in computational locus.

We first discuss remote evaluation and then extending server interfaces.

4.2.1. Remote Procedure Call, Remote Evaluation

Since we expect to perform queries on objects that reside remotely from the query site, we need minimally to provide for a means to do remote procedure call (RPC). RPC is a well-understood language mechanism and supported by many different distributed operating systems. Most assume a call-by-value semantics for RPC. We will minimally provide RPC in ML for base types, initially ignoring ref types, but including structured types (e.g., lists, tuples, and records).

Of greater interest, however, especially in the context of a higher-order functional language, is what passing more complex objects (functions, closures, and continuations) means, how it can be achieved, and what greater computational and performance capabilities it could provide the programmer.

For instance, we would like to support the notion of *remote evaluation*, where uninterpreted code is passed as an argument to a remote site and remotely evaluated (on data residing remotely and/or on data passed along with the code). Cast in these terms, we view remote evaluation as nothing different from remote procedure call where one of the arguments is a function, i.e., code. Though the semantics may be similar to RPC, the possible performance benefits gained distinguish it in motivation. Typically remote evaluation is used to move a computation to where the data resides and/or to exploit the computational power of some other processor. Remote evaluation allows users to ship a function once and have it execute remotely many times, thereby saving the cost of doing multiple RPC's locally. For example, instead of doing an RPC of a single function, F, 100 times in a while loop locally, we could create a new local function G that has the effect of shipping F to the remote site and executing F 100 times remotely, thereby paying the cost of one RPC (of G) instead of 100 RPC's (of F).

Open Issues

Providing general support for distributed queries requires:

- Building RPC for base types. We will use Mach primitives to "hide" much of the complexity of this from the programmer. We will pass booleans, integers, strings, records, lists, and tuples. For user-defined data types, we will initially pass their representation types as expressed in terms of built-in types. Later, we will investigate support for user-defined encode and decode functions (e.g., as done in Avalon/C++ and Argus) for enhancing efficiency.
- Handling ref types (mutable objects). This "feature" is a thorny issue that raises questions orthogonal to our work (especially for ML). Some problems include: (1) If pointer structures are not copied entirely, a passed mutable object may contain *remote references* (pointers to an object at the local site). Locating the remote objects entails *callbacks*; (2) If entire pointer structures are copied by value, maintaining consistency among *copies* across sites requires implementing efficient replication algorithms; (3) Is it worth preserving sharing structure, or partial sharing (i.e., to a specified depth), for mutable objects?
- Building support for RPC for more sophisticated types, in particular functions, closures, and continuations. Note that the notion of two or more "ML's" communicating is foreign to a standard ML "read-compile-print" top-level "interpreter." What assumptions about pervasives can be made at the different ML's? What parts of the environment need to be passed? [5]
- Extending ML to let programmers implicitly or explicitly denote remote evaluation. How much location transparency should we support? How much control do we want to provide the programmer in denoting what resides locally, what should be evaluated remotely and where, etc.?
- Extending the semantics of ML to deal with RPC and hence, remote evaluation. Does RPC give us exactly the desired semantics for our intuitive notion of remote evaluation? I.e., is there a semantic distinction between RPC and remote evaluation? If so, does ML reveal it?

[5] Unlike our Avalon/Common Lisp design, I see no reason to support pass evaluators themselves since eval is not even a first-class citizen in ML. Note that we do not intend initially to support passing structures, signatures, or functors for the same reason.

4.2.2. Extensible Servers

Similar to the idea of remote evaluation is that of *remote definition*. In the client/server model, remote definition is the ability to let clients define and add functions beyond a server's exported interface. These extensions are functions defined in terms of the server's fixed interface. Depending on what degree of protection is desired, these extensions are available to the client only, to whom the client grants access, or if no protection is provided, to anyone who has access to the server.

There are several advantages to providing extensible servers. The client programmer gains flexibility by tailoring the server interface to the needs of his or her application. Concrete examples of software with extensible interfaces are Emacs and Postscript. The programmer also can greatly enhance the application's performance by avoiding the need to pass code repeatedly. For example, NeWS, an extensible windowing system, can support the smooth rubber-banding of spline curves, while X, which essentially uses the standard RPC paradigm, has difficulty smoothly rubber-banding even straight lines.

In the context of ML, we viewed remote evaluation abstractly as RPC; similarly, we view remote definition as the application of a functor on a structure. [6] The result of the application is a new (extended) structure.

Open Issues

As for the other issues discussed so far, support for extensible servers entails extensions to ML at the syntactic, semantic, and implementation levels. Specifically, it requires:

- Building support for remote functor applications.
- Extending ML to let programmers denote remote definitions. What do clients and servers look like? How does a client perform remote functor applications? What restrictions do we need to place on the safety of such applications?
- Extending the semantics of ML to deal with remote definitions. Does remote functor application give us exactly the semantics desired for extensible servers?
- Investigating support for degrees of server access. This entire area of protection and security is orthogonal to the Venari Project, but clearly of importance when we intend to support not only extensible servers but also data and function caching at the local site. [7]

4.3. Concurrency

We motivated the need for persistence by our assumption that our objects of interest are stored in libraries. The correctness of the state of the system relies on being able to access objects whose lifetimes are potentially forever. In this sense, persistence is more fundamental than our need to support distribution or concurrency. As for distribution, we motivate our need for concurrency based on performance reasons alone. Just as we

[6] See sml files in /afs/cs/project/avalon-1/member/ldl/sml/examples/bib for a concrete example–the Avalon/Common Lisp bib example redone in ML.

[7] S.M. Clamen, L.D. Leibengood, S.M. Nettles and J.M. Wing, "Reliable Distributed Computing with Avalon/Common Lisp," in *Proceedings of IEEE Computer Society 1990 International Conference on Computer Languages*, March 1990, New Orleans.

imagine a complex query occurring across multiple sites, we imagine that a search-and-match computation can exploit concurrently executing threads of control.

There are two levels of concurrency we plan to support: "light-weight" threads of control and "heavy-weight" threads of control.

Of immediate interest is support for light-weight processes. In particular, we would like to perform concurrent search through a library on behalf of the same or even different queries. Such concurrent search requires multi-threaded servers. Support for light-weight threads will also let us implement concurrent garbage collection algorithms where one or more *collector* threads execute in parallel with the one or more *mutator* threads of the main program. Moreover, anticipating support for a persistent heap as well as a regular (volatile) heap, we can implement concurrent garbage collection of the persistent heap in parallel with normal garbage collection.

Initially we will identify heavy-weight processes with transactions and rely on standard implementation mechanisms, e.g., two-phase read/write locks, to provide transaction semantics. Earlier remarks related to transactions, e.g., recovering from a log, explicit linguistic extensions to ML, hold here. Support for concurrent (top-level) transactions lets an application programmer perform multiple searches, possibly at different sites, in parallel, and lets multiple clients access the same server. Recovery code can be run as a separate background activity as well. Finally, we envision needing to support not just concurrent (top-level) transactions, but also concurrent nested transactions in order to give programmers even finer control over concurrent activities. For example, we might want to let different transactions concurrently make updates to different disjoint branches of a tree structure, but perhaps have all changes contingent upon a top-level commit.

Open Issues

- Implementing and providing a threads package for ML. We are doing this initially for a C-threads-like interface in Mach.
- Defining an appropriate model of computation for concurrent ML.
- Adding linguistic support to "hide" the shared-memory model and provide the illusion of a message-passing model, as in Reppy's events.[8]
- Synchronizing multiple clients at a single multi-threaded server, hence requiring some kind of scheduling, pre-emption (possibly to break deadlocks), locking, etc.
- Designing, implementing, and providing semantics for concurrent (nested) transactions for ML.

Finally, as we build up our infrastructure, we need to write sample applications to exploit each (ML) language extension, in particular to test out their interactions.

[8] J.H. Reppy, "First-class Synchronous Operations in Standard ML," TR 89-1068, Dept. of Computer Science, Cornell University, Deember 1989.

5. Implementation Issues

5.1. Implementation Language

We have chosen to implement our software support in Standard ML of New Jersey. [9] ML provides compile-time type-checking (actually, type inference), polymorphism, higher-order types, modules, and signature mappings as well as useful control features like call/cc and exception handling. It is “mostly” functional, meaning that side-effecting code can usually be easily identified and isolated from the rest of the code. Moreover, it has a completely formal semantics, meaning that any extension we make can be studied in the context of a well-defined framework. Finally, ML has wide support at CMU; many faculty and graduate students are experts on its semantics and implementation. ML runs on many machines and is even taught in the undergraduate programming languages course.

We chose a functional language because of our interest in passing functions as first-class objects (see previous discussion on remote evaluation). We chose ML over Scheme because of ML’s strong typing and modules. We chose ML over Common Lisp because of the simplicity, minimality, and elegance of ML’s features and of the existence of a formal semantics.

However, by choosing ML, we created some work for ourselves: ML has no notion of persistence or distribution. Only recently have people been adding concurrency, but for a message-passing paradigm (e.g., Sufrin’s CSP and Reppy’s events), not a shared-memory paradigm as we would want initially in order to support a threads package for ML.

5.2. Operating System Support, Machines

Our initial implementation will be on the Mach operating system. We will assume homogeneity among machines initially, eventually addressing heterogeneity through either an ML-C or abstract interpreter approach, depending on which will give us better performance. As our RT’s are replaced, we expect to run ML on PMax’s and/or other machines with similar computational power and memory capacity. See Venari Note 2 for further ML-related implementation details.

6. Plans

We will spend approximately three years building software support for the Venari Project. This support includes the application, i.e., the framework plus various match engines, and the infrastructure, i.e., extensions to ML.

For the first year, we intend to work on the two major parts of the Venari Project, the application and the infrastructure in parallel. Hence, to make progress on the application, we assume a single site (no distribution) and a single thread of control (no concurrency), and we will initially use the file system as our means of guaranteeing persistence. At the same time, we will develop the necessary ML run-time support for persistence, distribution, and concurrency; and design and define appropriate linguistic extensions for the

[9] The reader should not confuse the language in which we implement our software support with any one of the program library languages.

ML programmer. When sufficient infrastructure is in place, we can then start exploiting the ML extensions in our application.

Here is a more detailed plan for the first year.

I. Building the framework plus match engines:

Spring, Summer 1990:

- *Framework:* Build ML browsing environment. This will provide the "glue" for the framework as well as give us a reasonable implementation environment. The environment will include interfaces ("hooks") to various match engines, e.g., the ML-grepper.
- *Match Engines:* Write up work on Lambda-Prolog match engine.
- *Libraries:* Continue collecting software libraries.

Fall 1990:

- *Framework*: Document browsing environment.
- *Match Engines:* Incorporate existing match engines. Investigate feasibility of specification languages and tools (e.g., Larch) with appropriate extensions as use for query language and match engines.

Spring 1991:

- *Integration:* Develop prototype application using environment and its interface to match engines and software libraries.

The above effort will occur in parallel with:

II. Adding extensions to ML:

Spring 1990:

- *Distribution:* Investigate feasibility of passing intermediate representations (lambda representations and continuation-passing-style (CPS) lambda representations) between two ML's.
- *Concurrency:* Design threads-like package for ML.
- *Persistence:* Design heap management (allocation and garbage collection (GC)) for persistent objects for ML. Since this will be built on top of Mashburn's recoverable virtual memory (RVM) system, for short, we sometimes call this entire effort RVM+GC. Investigate how persistence affects semantics of ML.

Summer 1990:

- *Distribution:* Design and implement RPC for ML types, excluding refs, but including functions and closures, and possibly continuations. Design and implement prototype for an extensible server defined remotely.

- *Concurrency:* Implement ML threads interface in Mach.
- *Persistence:* Design and specify runtime system support for persistence in ML.

Fall 1990:

- *Distribution:* Give small demo for IC. Continue implementation effort. Study semantics of remote evaluation and remote definition in ML context.
- *Concurrency:* Document extensions to multi-threaded ML. Design ML linguistic support for light-weight processes (or just use Reppy's interface). Investigate design and support for heavy-weight processes.
- *Persistence:* Implement persistent garbage collection.

Spring 1991:

- *Documentation:* Document (barebones) ML extensions for general ML community.
- *Implementation:* Continue polishing implementation. Investigate support for further extensions, e.g., remote refs, transactions.
- *Semantics:* Finish semantic extensions to (barebones) concurrent, distributed, persistent ML.
- *Application:* Begin developing a prototype application using ML infrastructure.

In the second year of the project, we will continue the design, definition, and implementation of ML extensions as well as the building of our framework and match engines. The primary focus will be to build a prototype application that demonstrates the use of multiple match engines for a distributed query and concurrent search over some existing (or hand-tailored) software library.

In the third year of the project, we plan to integrate all these efforts and make some parts of it more generic, e.g., the browsing environment need not be for ML, the lowest-level operating system support need not be for Mach. We hope to focus on the problem of search by designing and incorporating more sophisticated match engines. These engines may be more sophisticated either by the inherent complexity of the match (and/or search) algorithm itself (and hence, in principle rely on more semantic information about an object) or by the implementation of a given algorithm, e.g., using a parallel theorem-prover or exploiting multi-processor capabilities to parallelize searching and/or matching.