ML Grid Programming with ConCert *

Tom Murphy VII

Carnegie Mellon

tom7@cs.cmu.edu

Abstract

Grid computing has become increasingly popular with the growth of the Internet, especially in large-scale scientific computation. Computational Grids are characterized by their scale, their heterogeneity, and their unreliability, making the creation of Grid software quite a challenge. Security concerns make the deployment of Grid infrastructure similarly daunting.

We argue that functional programming techniques, both well-known and new, make an excellent practical foundation for Grid computing. We present a prototype Grid framework called *ConCert* built entirely in Standard ML which allows for the trustless dissemination of Grid programs through the use of certified code. The framework is fault-tolerant and relatively easy to implement, owing to a simplified network abstraction. This network abstraction is tedious to program for directly, so we present a high level ML-like language *Grid/ML* and a compiler *Hemlock* for the language.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages; D.1.1 [*Software*]: Programming Techniques—Applicative (Functional) Programming

General Terms Languages, Reliability, Security

Keywords Grid Computing, Certified Code, Distributed Computing, Functional Programming, Compilers

Introduction

A Computational Grid is a large scale aggregation of computers, often designed for scientific computing. There are many active and as-yet-unrealized visions of the Grid. We take the view that analogizes with the electrical power grid: a vast connection of computational resources, accessible to all participants. We hope that such a network can be built in a peer-to-peer fashion from computers owned by volunteer internet users.

In spite of its tantalizing potential, the Grid remains difficult to deploy, and difficult to program for. Some special challenges encumber it:

Security. Grid programs are by nature network applications, which makes them especially susceptible to remote attacks. In order to convince users to donate their unused cycles, they must do so at negligible risk. Donors should be protected from both malicious Grid programmers and imperfect ones.

Failure. Because the Grid consists of home computers with intermittent usage patterns and network connections, we must expect that nodes may fail at any time. Failure is a problem for programmers, who must write their programs to expect, and tolerate, failure.

Distribution. The distributedness of Grid applications is often the very point, but it comes with its costs, too. For instance, distributed concurrent programs are difficult to schedule efficiently.

There are two main contributions of this paper. We present a model for Grid computing called *ConCert* that draws on ideas and techniques from functional programming to provide partial solutions to the above difficulties. We also describe the implementation of this model. It consists of a peer-to-peer application that implements the ConCert network, called the *Conductor*; a high-level language for Grid programming called *Grid/ML*; and its compiler *Hemlock*.

For security we use type safe languages and certified (or proof-carrying) code. Though these ideas are not new, the ConCert project is one of few extant applications that consume certified code technology. Therefore we provide valuable lessons about usage scenarios and requirements for such technology. In order to deal with the particular contours of fault-tolerant distributed programming, we design our network substrate with failure recovery and a simple, local scheduling policy in mind. We do so by embracing the pure functional paradigm: Grid applications are split into series of deterministic functions whose results are memoized by the network.

While this design greatly simplifies the design and implementation of the framework software, it also introduces new problems. Programming directly against this network abstraction is very tedious; programmers usually need to apply program transformations by hand in order to achieve standard concurrent programming idioms such as thread synchronization. Through the traditional compilation techniques of closure- and CPS-conversion we are able to automate this and present a high level language that makes Grid programming quite pleasant. Grid/ML is based on core SML. In Grid/ML we are able to express a few standard techniques for fault-tolerance, including message logging and checkpointing.

The remainder of this paper is organized as follows. First, we present our network abstraction, motivating it with the implementation of our Conductor software (Section 1). Next, we present the Grid/ML language with a few small examples (Section 2). We then describe the Hemlock compiler (Section 3) and the special challenges we face compiling Grid/ML for the ConCert network, followed by further examples of Grid/ML programming (Section 4). Each is the subject of ongoing research, so we conclude with an in-depth discussion of our next steps (Section 6).

* The ConCert Project is supported by the National Science Foundation under grant ITR/SY+SI 0121633: “Language Technology for Trustless Software Dissemination”.

ML’06 September 16, 2006, Portland, Oregon, USA.

1. The ConCert Network

A Grid generally consists of a number of distributed, concurrently running processes. In typical Grids the processes communicate with one another via some API—maybe nothing more than internet sockets. Though successful in scientific computing, Grid APIs such as Globus [14] usually take the dissatisfying perspective that Grid applications are literally programs running on separate computers that communicate over the internet. That is, there is little abstraction. One ultimate goal of the ConCert project is to take a more high-level language-based view in order to answer the question: What is it like to program the Grid as a new kind of computer?

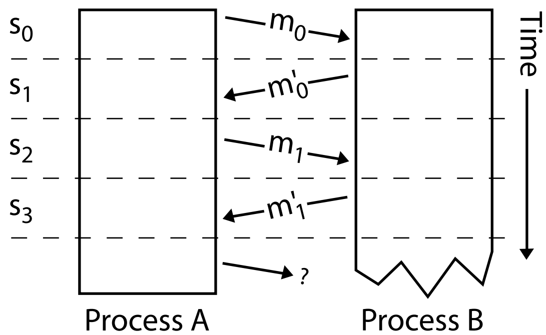


Figure 1. Grid processes engaged in a protocol

Although this perspective can be seen as essentially aesthetic, there is a more serious issue involved with the low-level view due to the fact that Grid processes can fail. Suppose processes A and B are exchanging data via some protocol, as illustrated in Figure 1. Now suppose process B fails after the step marked s_3. This leaves A in a problematic state: it is part-way through a protocol, with nobody to talk to. In a sense the imperative nature of failure has exposed the imperative implementation of communication. A typical solution to this problem makes the communication functional via memoization. If we store all the messages sent to and from B, then we can restart B from the beginning, and replay messages m_0 and m_1 to it. (For this to work, we assume or check that B sends back the same messages m'_0 and m'_1 in response, implying that B is deterministic given particular inputs.) If B is communicating with several processes, we need to replay all the messages sent by all of those processes, in the correct order. This technique is called message logging [18].

Our basic strategy in designing the ConCert network is to build message logging into the Grid by representing these processes as deterministic functions whose results are memoized. As we do, our network will not obviously be as expressive as a network of programs communicating via internet sockets. However, through some idioms and compilation techniques we will be able to regain some of this expressiveness. Section 6 briefly discusses our current efforts to regain other expressiveness, such as awareness of location.

1.1 Cords

ConCert is a peer-to-peer network of interconnected *nodes*, each running the same *Conductor* software. This software has several duties discussed in a previous report [7]; two are relevant here. The first duty is to maintain a queue of pending work. The second is to "steal" work from other nodes (or perhaps the node itself if its queue is not empty), verify and run the work, and memoize the result. By analogy with *thread* and musical pun on *chord*, we call these units of work *cords*.

Cords are bits of deterministic, certified code. Because we are agnostic about the kind of data that may be manipulated by these cords, each can be thought of as a function of type `byte vector` $\rightarrow$ `byte vector`. In other words, cords are responsible for marshalling their own data structures into bytes suitable for transmission over the network.

Cords cannot communicate directly with other cords. Therefore, not only are cords deterministic, but they do not block waiting for any network events. This simplifies scheduling greatly. However, a cord can have *dependencies* on the results of other cords. Such a cord cannot be run until all of its dependencies are filled, at which time it will be given the results of all the cords it depends on as a second argument, its *witness*. Thus we can refine the type of cords to `cord vector` $\times$ `((witness` $\times$ `byte vector)` $\rightarrow$ `byte vector)`. Here the `cord vector` is the set of dependencies. The type `witness` is a vector of byte vectors; the results of the cords it depends on. An apt analogy is that cords are like a compiler's *basic blocks*, except that they are split by communication structure, rather than by control flow structure.

The essential feature that allows us to do anything interesting with cords is that they can spawn other cords. The cord code also receives a spawning function as one of its arguments, along with abstract types and constructors for forming dependency vectors (etc.). In the actual implementation these are accompanied by some other uninteresting arguments.[1] For most of the discussion here we ignore these extra arguments.

In practice, a cord can only re-spawn its own code with a new argument and dependencies, since it would otherwise have to conjure up certified code from some place (we do not support "run time code generation" for lack of compelling applications). This is no problem, however, as we can use the cord's argument to give a single cord unlimited potential "entry points." ConCert caches the components of cords, so once a host has received and certified a cord, it does not need to download or certify future cords spawned by that one.

Conceptually we can think of each cord as literally, recursively, containing all of the cords that it depends on, according to the type given above. However, this is highly inefficient in practice. We instead uniquely identify each cord using cryptographic hashes; in ConCert a cord identifier is a triplet `(hash(`*deps*`), hash(`*arg*`), hash(`*code*`))`. We use the SHA-1 algorithm [4] to hash cords, which produces a 160 bit hash. The chances of hash collisions by chance are therefore negligible. The best publicly known attack against SHA-1 takes 2^{63} operations [30], which is perhaps feasible for a malicious party with significant resources. However, we do not yet protect against other, cheaper forms of malice (Section 6). Hash collisions only disrupt the network algorithms, and do not affect the safety of the certified code that is run.

When spawning a new cord, the spawning cord receives the spawned cord's id. Because cords are deterministic given their arguments, this identifier also uniquely determines the *result* of the cord, presuming that the cord terminates.

Scheduling higher-order cords with dependencies requires network-wide lookup of dependencies and retrieval of results. We achieve this by the implementation of a network-wide distributed hash table [27]. We also use this hash table for redundancy in order to implement failure recovery.

Failure recovery happens as follows. When a cord c_1 in some node's queue depends on another cord c_2, which the node decides has failed to complete, the node will attempt to restart c_2. Often, its code, argument, and dependencies can be recovered (by lookup in the hash table). If not, c_1 fails as well and is removed from the queue. In a catastrophic situation, failure may propagate to the parent of all cords, which is the *client* (described below). If this client is still running, then it will have retained the materials to

[1] Access to these resources could also be achieved by dynamic linking; we choose entirely closed code for its relative simplicity.

restart any cords it is waiting for. If it is not, then there will be no way for anyone to observe the result (the application has essentially terminated), so there is no reason to try to restart any involved cords.

Cords can't produce effects, so the final piece of our network is the concept of a *client*, which is a program that connects to the Grid to run work on it. A client interfaces with a local conductor by seeding it with cords (probably as the result of some user input), and retrieving the results of cords that it submitted. The client is not restricted as cords are; it may be nondeterministic, effectful, and may block waiting for a cord's result. On the other hand, it is not mobile or certified (necessarily), and it is clearly not tolerant to (its own) failure. In the Grid/ML language, the client and cord code are written as part of the same source file, but the programmer must be aware of the distinction.

1.2 Implementation

The ConCert conductor is written in Standard ML and runs only on x86 Linux. For our certification framework we use TALx86 [21], a particular implementation of typed assembly language for x86 processors. In principle we support multiple certification frameworks and tunable safety policies, though only a TALx86 checker and loader exists currently. A brief description of this part of the Conductor may be interesting to those developing code certification frameworks; in order to write an LVR ("loader, verifier, and runner") we need to be able to:

- Test if code (read from a file) passes certification
- Dynamically load (from a file) and execute closed code with arguments. One of these arguments will be a function for spawning new cords, which needs to communicate with the Conductor over a UNIX socket
- Retrieve the code's result and send it over a UNIX socket

Certification in the first item can be parameterized by a safety policy. For our current implementation, the only possible safety policy is type safety. In certification systems under development, this can include properties like resource bounds [29].

It is good if the second item does not depend on the first. The reason is that we usually run the same piece of cord code several times on different arguments in the course of execution. We wish to be able to cache the result of certification to avoid paying the (often substantial) cost multiple times. Unfortunately the TALx86 dynamic loader [12], which itself is written safely in TALx86, has no choice but to type check on each dynamic load.

The LVR is simple and can be written at a fairly low level. Our TALx86 loader is written in Popcorn (with a tiny amount of C for Unix sockets), which is a safe C-like language that compiles to TAL. The experimental LVR for TALT, the project's next generation foundational typed assembly language [11], is simply written in C and assembly.

1.3 ConCert Applications

Before developing our high-level language Grid/ML, we wrote several applications directly against the ConCert abstraction. For these we wrote our cord code in Popcorn, with clients written in Standard ML or Popcorn.

Ray tracer. We developed a Grid ray tracer based on the specification from the 3rd ICFP programming contest [3]. Ray tracing is a naturally parallelizable task; arbitrarily small chunks of the image can be rendered on different machines to get an almost linear speedup. We can spawn all of our cords from the client (never recursively), and don't need dependencies. Despite this application's simplicity, it is actually most similar to current large-scale Grid applications such as SETI@Home [2], which often have massive amounts of data that can be processed independently.

Chess Engine. Next, we developed a Grid chess player based on the Jamboree algorithm [19]. Naïve game tree search is also easily parallelizable, however, in order to avoid an unmanageable blowup, serial pruning strategies such as Alpha-Beta are necessary. Jamboree attempts to balance between these two extremes. Unlike the raytracer, this algorithm requires parallelism at depth greater than 1, so we manually apply CPS and closure conversion to implement fork-join parallelism with cords.

2. The Grid/ML Language

Despite our demo applications, Popcorn and TAL Grid code is much too difficult to develop and maintain. Our answer is a high level functional language called Grid/ML based on core SML.

The extensions for Grid programming are actually quite simple. We add the types and primitives from Figure 2.

```
type α task
val spawn   : (unit → α) → α task
val syncall : α task vector → α vector

fun sync t = sub(syncall [| t |], 0)
```

Figure 2. Grid/ML Grid Primitives

In other words, Grid/ML has simple fork-join parallelism, where the abstract type α `task` represents a forked task that will return a result of type α. The `spawn` primitive takes a suspension and begins running it on the Grid. The `syncall` primitive waits for all of the supplied tasks to finish and then returns each of their answers. We define `sync`, which will be used in some examples, as a `syncall` on a singleton vector. Tasks will be compiled into cords. Because we allow tasks at any type, the language implementation will have to marshal values of arbitrary type into `byte vectors`. In addition we will have to deal with the blocking aspect of `syncall` especially, since cords are not allowed to block.

The story is slightly more complicated. The `task` primitives are for writing Grid code (which will become cords), but recall that we also write the client code in the same source file. (We do this because it is almost always necessary to share data structures between the client and Grid code, and Grid/ML does not support any sort of separate compilation or libraries.) Therefore we also have a type α `job` and associated operations `submit` and `waitall`. For client code it is easy to provide other primitives on jobs, such as a non-blocking query as to the status of a submitted cord. Aside from marshalling, these correspond to direct calls to the same ConCert client library used to write our Popcorn demos.

```
let
   fun job () =
       let val t = spawn (fn () => "hello")
       in  sync t
       end
   val j = submit job
in
   print (wait j)
end
```

Figure 3. Simple Grid/ML Example

A simple example in Figure 3 illustrates the use of these primitives. In this example, the body of the function `job` runs on the Grid, and the rest is client code. The Grid code spawns its own task and immediately syncs on the result. Note that client code and Grid code have separate and unequal capabilities: Only the client code can perform I/O, and only the Grid code can `syncall`. Violations of this are currently checked dynamically. (Our work on modal type systems, described in Section 6, is intended to make this distinction static, among other things.)

Because they will be compiled into cords, any `spawn` in a Grid/ML program can be thought of as a checkpoint—a common fault-tolerance technique. If a task fails while running on the Grid, then it can be restarted from a checkpoint (usually the most recent one, depending on the extent of the failure). It will only be restarted if some other task is currently `sync`ing on it. The Grid/ML programmer can easily induce the creation of extra tasks in order to checkpoint long computations.

```
datatype α cxpoint =
   Done of α
 | More of cxpoint² task

fun iter(n, r) =
  if n = 1000000
  then Done r
  else let fun rest () = iter(n + 1, f r)
       in
           if n mod 50000 = 0
           then More(spawn rest)
           else rest ()
       end

fun getanswer cp =
  case cp of
    Done a => a
  | More t => getanswer (sync t)
```

Figure 4. Encoding Checkpoints in Grid/ML

Figure 4 contains an example that uses checkpoints. The function `loop` is intended to compute the iteration of `f` on `r` a million times. In order to avoid losing intermediate results should the computation fail, the result datatype α `cxpoint` allows the return of intermediate progress (`More`) in addition to the final answer (`Done`). Now the caller, `getanswer`, `sync`s on any checkpoints that it receives, and succeeds when the final answer is reached. Each `sync` may cause the associated checkpoint to restart if it detects failure or a timeout.

Though explicative, this example is actually overkill. Due to our compilation strategy, `sync`s themselves also act as checkpoints. We can write a function that checkpoints the currently executing task and then continues; its code is simply:

```
fun checkpoint () = syncall [| |]
```

That is, `syncall` of the empty vector of tasks induces a checkpoint. The reason for this will be clear when we explain the implementation of `syncall` in the next section. We will then be able to give other interesting examples of functional programming with cords.

3. The Hemlock Compiler

Hemlock is our compiler for Grid/ML. It transforms a Grid/ML program into a TALx86 client and cords. Hemlock is also written in Standard ML. It is named after the nonpoisonous state tree of Pennsylvania—not Socrates' fatal cocktail!

[2] We deviate from SML syntax here: `datatype`s are forced to be uniform, so we do not even mention α when we make recursive reference to `cxpoint`. Datatypes are compiled as μ-sums.

Hemlock is much like a standard (whole-program) compiler for a typed functional language. It has three special requirements. First, it must generate well-typed TALx86 code, so the compiler must be certifying. Second, it must be continuation-based in order to enable the compilation of `syncall`. Finally, we must be able to marshal any value at runtime into byte vectors. To expedite the implementation of marshalling, we use a unityped representation (this also makes certification somewhat simpler).

We will generate a single piece of cord code for our Grid/ML application. Recall that cords take arguments; the argument to this code will be a (marshalled) closure to run. In this way the single piece of code can actually represent any number of operations. The standard portions of the compiler are described briefly, and at the appropriate stages we explain any special considerations relating to these three aspects.

Rather than use a parser generator tool, Hemlock's parser is written using combinators [17]. This allows us to handle `infix` declarations by making parsing context-sensitive. We support the entire grammar of core SML (except where we intentionally diverge) in about 400 lines of code.

Parsing is followed by fairly standard elaboration into a typed intermediate language. Type-checking and translating the Grid/ML primitives at this stage is trivial; they are primitive in the intermediate language and have analogous types there. Datatypes are translated as polymorphic mutually-recursive sums. At this point we translate into a unityped CPS language, which introduces our first point of interest.

3.1 Continuation Passing Transformation

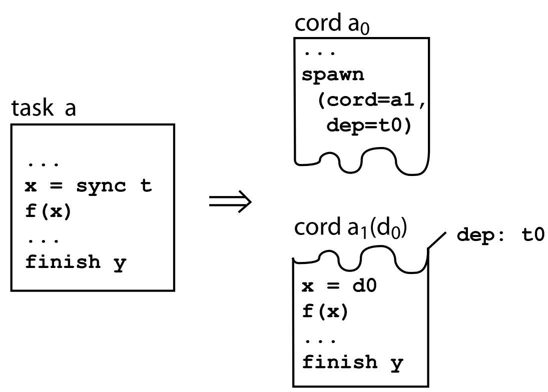


Figure 5. Transform of `sync`, first attempt

Our CPS language is closely based on Appel's [5]. As we translate, we have the ability to grab the current expression's continuation and reify it as a function, which we will use to translate our Grid primitives. Otherwise this translation does not differ substantially from the standard.

We explain the translation of the Grid primitives at a high level, because the details are quite confounding. As remarked earlier, `syncall` is the source of complication. We look at the unary case of `sync` for simplicity (of course the n-ary `syncall` case can easily be coded up by iterating `sync`, although this is less efficient.) Each task will be split into two cords every time that it does a `sync`; a first cut appears in Figure 5. When translating a task into a cord and encountering `sync` on a task `t`, we terminate the current cord after spawning its continuation. The continuation has a *dependency* on the the argument to `sync`. In this way we avoid actually blocking within a cord, but nonetheless implement the blocking semantics of `sync`. Recall that each cord is invoked with the results of each of its dependencies; for a_1 that is d_0, the result of t_0.

Alas, this naïve translation doesn't work. The reason is that the task `t` may itself do a `sync`, and presumably some other task or client is expecting to receive the result of task `a`. However, we can no longer look to the (first) translated cord to find the answer to a task, because the task may be split across many cords due to `syncs`. Note that we even ignored the return value of a_0 in Figure 5!

```
cord a0 =
 ...
 ca1 = spawn (cord=a1, deps=t0)
 finish (FWD ca1)

cord a1(d0) =
 (case d0 of
    ANS x =>
       f x
       ...
       finish (ANS y)
  | FWD t1 =>
       ct1 = spawn(cord=a1, deps=t1)
       finish (FWD ct1))
```

Figure 6. Revised code for a_0, a_1

A revised version appears in Figure 6. Now, all cords return a sum: Either `ANS` with the final answer for the associated task, or `FWD` with the next cord in the sequence. (This sum type is still marshalled into a `byte vector` as before.) As a_0 terminates, it forwards to its continuation, so that any cord depending on a_0 will be able to direct its attention to a_0's continuation instead. We see exactly this behavior in a_1, which has a dependency on the result of t_0. The cord a_1 must must check if it has received the final answer (in which case it continues as before) or a forwarding message. If it is a forwarding message, it respawns itself with a dependency on this new cord, and returns its own corresponding forward message. At the normal end of a task, we wrap the result with `ANS`.[3] To translate `spawn`, we simply spawn the supplied function after wrapping it so that its result `x` becomes `finish (ANS x)`. Cords spawned through the Grid/ML mechanism have no dependencies.

This aspect of the translation is very similar to the α `cxpoint` device that we used in Section 2. In fact, now we can see why `syncall [| |]` implements a checkpoint. An empty `syncall` causes a new cord to be spawned with no dependencies, so it executes immediately. The previous cord terminates with a `FWD` to the new cord. To resolve the `FWD`, any task waiting on this one does essentially the same case analysis and loop that was done in the `getanswer` function from Figure 4.

At this point in the compiler we introduce the concept of marshalling, so we have primitives that convert from any type to byte vectors, and vice versa. During CPS conversion, calls to `marshal` are inserted when a value is returned from a cord, and calls to `unmarshal` are made on every value received as the result of a dependency. Also recall that we really generate a cord with one entry point whose argument is a function to call; at this entry point we unmarshal the argument function and invoke it. When we want to spawn a function, we really spawn our own code with the marshalled function as an argument. The implementation of marshalling will be discussed in Section 3.3; hopefully its uses are clear.

[3] For performance reasons, the Conductor supports forwarding cords directly, which is what the compiler actually generates. These special cords have little overhead and can be run immediately to update the continuation with the new dependency.

The CPS conversion is followed by an optimization pass. Function inlining, β- and η-reduction are particularly important because of the somewhat naïve generation of continuation join points in the translation. Otherwise, this optimization pass is fairly standard.

3.2 Backend

In order to reify higher order functions as data, we run a standard closure conversion algorithm to make function values into a pair of closed code and an environment. Environments are no more difficult to marshal than records, but code pointers will need some special attention. Although closure conversion is unsurprising to the implementer of functional languages, it is sorely missing from some related languages (Section 5), which force the programmer to perform it manually.

Following closure conversion, we essentially have assembly language, and are ready for translation into TALx86. We choose the following TAL type `ttt` to uniformly represent all values:

```
ttt = ^+[*[S( INT_T ), B4],
         *[S( STR_T ), string],
         *[S( REF_T ), `ttt],
         *[S( SUM_T ), B4, `ttt],
         *[S( IND_T ), B4],
         *[S( TUP_T ), (array `ttt)],
         *[S( COD_T ), codeptr]]
```

Here, `ttt` is a pointer (`^`) to a disjoint union (`+`) of several possibilities. Each is a product (`*`) with some singleton type (`S()`) as its first field; the tags `*_T` are each distinct integer constants (this is just the tag field normally used in uniform representations). `INT` and `STR` are straightforward integers and strings (B4 is the type of four-byte machine words). `REF` is a reference cell; note that types are automatically recursive in TAL. `SUM` consists of an object language tag (indicating the arm of the sum) and then a value. Since all types are represented the same way, there is no need for dependency here. `TUP` is followed by a variable-length array, which is used for vectors and tuples. `COD` contains a code pointer; we arrange that all code pointers inside values have the same type by using a uniform calling convention. The `IND` tag is only used during unmarshalling, and will be explained shortly.

Given this representation, generation of type-safe TAL code is not difficult (though the resulting code is allocation heavy and has many tag checks). Performance is acceptable for a prototype, and could be improved significantly with simple local unboxing optimizations. The payoff of this representation comes in the implementation of marshalling and unmarshalling.

3.3 Marshalling

We wish to write functions `marshal` : `ttt` $\rightarrow$ `byte vector` and `unmarshal` : `byte vector` $\rightarrow$ `ttt option`. These functions must be well-typed TAL and must deal with all data, including code pointers and cyclic references. We also wish to preserve sharing. We give our marshalling algorithm along with a checklist of necessary features for designers of certified code frameworks.

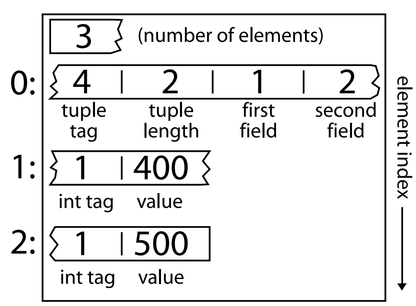
The format of a marshalled term is an array (encoded as a string) of subterms represented as strings. Each subterm is a constructor applied to some integers, which are the indices of other subterms in the array. For example, the tuple `(400, 500)` has subterms `400` and `500` and can be represented as:

```
loop () =
    if empty(waitq)
    then done
    else p = pop_head(waitq)
         s = concat(tagof(p),
                    map getindex
                     (subterms of p))
         push_tail(outq, s)
         loop()

getindex p =
    case (lookup p in tmap)
     of SOME i => return i
      | NONE =>
         i = nextindex
         increment nextindex
         insert(tmap, p → i)
         push_tail(waitq, p)
         return i

marshal p =
    clear queues
    insert(tmap, p → 0)
    push_tail(waitq, p)
    loop ()
    return string(rev(outq))
```

Figure 7. Pseudocode for Marshal



3.3.1 Marshal

The `marshal` function crawls over a term of type `ttt` and produces an array as described, which it then encodes as a string.

Pseudocode for our marshalling function is given in Figure 7. It makes use of several data structures. The queue `outq` is a list of processed subterms. It will be linearized to create the output array.

The map `tmap` is a map from `ttt` pointers p to integers i. Each i is the position that the subterm at p will have in the final array, if it is known. If p is in the domain of `tmap`, then it has already been marshalled (and is in `outq`) or is on the `waitq`. We represent the map as a binary tree; more efficient representations such as hash tables are not possible because TALx86 only has inequality operations (less than, greater than, and equal) on pointers.

The queue `waitq` is a queue of pending pointers p. We insert at the tail and dequeue from the head. They appear in the queue in the exact order that they will appear in the output array. These are terms that we've forward-referenced as members of some other terms, but have not yet marshalled.

The integer `nextindex` simply gives us the next available index in the array. It is also the sum of the lengths of `outq` and `waitq`.

To marshal a term, we clear our data structures and then initialize the wait queue and map by assigning the term position 0. We repeatedly process items from the `waitq` until it is empty. To process one, we look at its tag and look up each of its subterms in the map (if any). If they have already been assigned indices, then we use those indices; otherwise we assign them the next available indices and put them in the wait queue.

Because we cannot transmit code pointers, we also generate a static array `ctab` at compile time, which contains all of the code pointers in our program. (Actually, we can optimize this table somewhat by omitting code that can never be part of an escaping closure.) To marshal a code pointer, we search through the table for its index, and ship that integer. (This process does not appear in the pseudocode above, but is completely straightforward.) Again, we cannot use hashing here because we have only inequality on pointers.

The marshalling code is about 1,000 lines of TALx86 code, a mix of hand-written TAL and the output of the Popcorn compiler.

3.3.2 Unmarshal

```
firstpass (arr, s) =
  foreach i in 0..(arr.len - 1)
    tag = get_next(s)
    case tag of
       ...
     | COD =>
         codidx = get_next(s)
         arr[i] = new_ttt(COD, ctab[codidx])
     | REF =>
         subidx = get_next(s)
         arr[i] = new_ttt(REF, arr[subidx])
       ...

flatten (arr) =
  foreach i in 0..(arr.len - 1)
    foreach subterm field f in arr[i]
       if f.tag = IND
       then f := arr[f.val]

unmarshal s =
  n   = num_elts(s)
  arr = new_array(n)
  foreach i in 0..(n-1)
    arr[i] = new_ttt(IND_T, i)
  firstpass(arr, s, idx)
  flatten(arr)
  return arr[0]
```

Figure 8. Pseudocode for Unmarshal

When we receive a string s, we need to unmarshal it into a term of type `ttt`. Recall that our marshalled string represents an array of n subterms; the first thing we do is create an array of `ttt` pointers with size n, called `arr`.

To handle cycles, we have a two-pass unmarshalling algorithm, which is given in Figure 8. Here is where we use the indirect tag `IND_T` that's part of our `ttt` type. A pointer to a `ttt` with an indirect to i means that the pointer should be to the contents of `arr[`i`]` instead. We then "tie the knot" in a second pass, removing all indirections. We begin by initializing `arr[`i`]` to an indirection to i for each i.

In our first pass, we simply read elements from the input string s, and create terms to populate `arr`. Code pointers are small integers, which we look up in our code table `ctab`. If the element indicates a term with subterms (such as a reference cell), then we retrieve those subterms from the array. If we have not yet unmarshalled that element, we will use the indirection in the cell. Because of cycles, there may be no way to order the elements such that we avoid indirections.

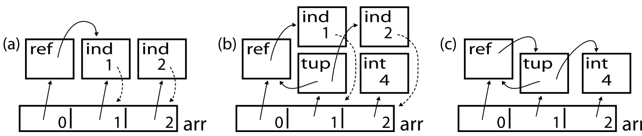


Figure 9. Unmarshalling Example

Figure 9 shows the process of unmarshalling a self-referential tuple. Solid arrows indicate literal pointers, and dotted arrows indicate indirections through the array. Part (a) shows the state of affairs after writing the first term into `arr[0]`. The `ref` term created will contain whatever ends up in `arr[1]`.

After filling in the remainder of the array slots, we have created all of the terms, but all forward references are through indirect pointers (Figure 9(b)). We then make a flattening pass to remove these indirections. For each term in the array, we look at its subterms. If any is an `ind`(i), we rewrite it in place to point directly to the contents of `arr`(i). For instance, in Figure 9(c), we encounter `ref(ind(1))`. We must rewrite this to the recursive structure `ref(tuple(ref ..., int(4)))` without modifying the location of the `ref`, which is pointed to by other nodes.

After flattening, the first element of `arr` holds the root of our term, so we return that to complete unmarshalling.

The implementation of unmarshal is also approximately 1,000 lines of TAL code.

The marshalling algorithm preserves sharing, handles cycles, and is simple enough to be coded directly in assembly language. Marshalling could also be extended to perform hash consing, which would reduce the size of marshalled data.

In order to implement marshalling this way, a certification framework needs to minimally support inequality on code and data pointers (coercions to integers for hashing enables better data structures), and the imperative overwriting of fields inside objects of sum type.

3.4 Other Features

Hemlock has a few other special considerations. As remarked earlier, Grid/ML supports reference cells. However, these reference cells cannot be used to communicate across cords; we do not wish to violate our no-communication invariant. Each time we marshal, we copy the heap, creating a new copy of each reference cell. Because of the compilation strategy we use, the identity of references cannot be maintained across `spawns` and `syncs`, either. However, such "local" references can still be useful; for instance, they can be used to safely implement benign effects such as memoization or the balancing of a splay tree, and they can be used to generate cyclic data structures that are then used functionally.

We also need to do two special things to implement exceptions. First, exceptions that reach the end of a cord are propagated to any cords that depend on it. This is accomplished by assigning a distinguished bit to marshalled data that indicates "exception"; when unmarshalled, the value will be passed to the current exception handler rather than become the argument or fill a dependency slot. This is essentially just the monadic interpretation of exceptions where a cord returning type τ becomes a cord returning type τ `+ exn`.

Second, we implement the SML `exn` type, which creates new tags at runtime. In fact, we support a full extensible datatype mechanism, of which `exn` is a special case. Typical implementations use an incrementing counter to generate distinct integers for tags. We can't use a counter because we can't coordinate the increment of that counter across independent cords. Therefore, we use 64-bit random numbers as tags. Think of these tags as *nonces*, in other words, we are doing a form of cryptographic typechecking. Of

```
(* parallel tuple construction *)
fun &&(f1, f2) =
    let val t1 = spawn f1
        val t2 = spawn f2
        val res = syncall [| t1, t2 |]
    in
        (sub (res, 0), sub (res, 1))
    end
infix &&

(* ... its use *)
case P of
    A /\ B =>
        AndIntro
        ((fn () => prove (G ==> A)) &&
         (fn () => prove (G ==> B)))
  | ...
```

Figure 10. Parallelism in our theorem prover

```
datatype α stream =
   Empty of unit
 | Cons of α × stream task

fun consumer (ps : stream task) =
   (* fetch the next proof from server *)
   (case sync ps of
      Cons(pf, next) =>
             (* use the proof, then loop *)
             ... pf ...
             consumer next
    | Empty => ...)

fun server state =
   let
       (* generate a proof and next state *)
       val (pf, nextstate) = ...
   in
       (* send the proof to any listener,
          and begin computing the next *)
       Cons(pf,
            spawn (fn () => server nextstate))
   end
```

Figure 11. Modeling one-way communication

course, we still want our cords to be deterministic, so the "random" numbers are actually generated pseudo-randomly using the cord's own id as a seed. Collisions by chance are very unlikely–about 4 billion exception tags must be generated before we expect two to be equal. Intentional collisions are easy to generate by writing TAL code or the binary marshaled format directly. In either case, collisions simply cause a run-time error, not a violation of safety.

4. More Examples

We have implemented a simple theorem prover for propositional intuitionistic logic in Grid/ML as a test application. Theorem proving has many opportunities for parallelism, for instance, to prove the proposition $A \wedge B$ we independently prove A and B. The code is quite simple; the only place that we use our Grid primitives is in the implementation of parallel tuple construction `&&` (Figure 10).

In bottom-up theorem provers for Linear Logic [8] it is sometimes necessary to return not one proof but a *stream* of different proofs (there may be multiple incomparable ways to use resources to achieve a goal). We might like to model this as a *proof server* that sends proofs to a waiting *consumer*. This kind of communication is also modelled easily in Grid/ML. In Figure 11 the server returns a `proof stream`, which is a `proof` paired with a new server to `sync` on for the next proof. The new server is started as the first server concludes, so the client and server do execute in parallel. More complex communication, such as multi-direction and multi-participant communication can be modelled as well, with varying degrees of faithfulness. No matter how the system is used, we retain our automatic failure tolerance through what amounts to message logging.

Our intention is that these examples are fairly unsurprising, because we argue that Grid/ML provides a natural and high-level abstraction that makes fault-tolerant functional programming simple and transparent.

5. Related Work

The ConCert network can be seen as a generalization of Cilk-NOW [6]. Cilk-NOW is an extension of the C language for writing parallel programs, and a runtime system that permits their execution on networks of workstations. In the Cilk-NOW language, programmers write their code in manually CPS- and closure-converted style. Like ConCert, Cilk-NOW only allows functional programs with no extra-lingual communication. But because we automatically perform the requisite program transformations and have higher-order tasks and other functional language features, programming in Grid/ML is much less constrained.

Chothia and Duggan [9] give a language based on the Pi Calculus for fault tolerant distributed computing. The work is primarily focused on the distributed maintenance of log tables, which are used to achieve atomic transactions (another common technique for fault tolerance). In particular, they use cryptography to protect against forged messages from attackers trying to disrupt these logs. This goal is very ambitious; we do not even address malicious claims about the result of a cord (the analog in ConCert). Alas, there is no high level language and no implementation yet.

Jocaml [10] is another ML-based language and implementation for distributed computing, which is based on the Join Calculus [15]. It also does automatic marshalling of data structures—including closures—and has a much richer set of communication and mobility primitives. This includes so-called *join patterns*, which completely subsume our conjunction-only dependencies. However, Jocaml is not certifying, and does not automatically tolerate failure. Therefore, we argue that Grid/ML is more appropriate for trustless, fault-prone networks. Nonetheless, enriching our language for dependencies is likely to give Grid/ML more expressive power without sacrificing fault-tolerance, and is worth investigation.

Acute [26] is a new ML-like language for distributed computing. The language is tailored to a dynamic linking setting where interacting programs across hosts may be upgraded to different versions while the program is running. The general mechanism in Acute for controlling locality is its dynamic rebinding system. This allows the programmer to mark libraries so that references to them are rebound to local versions when code arrives at a site—rather than sending the library along with the code. In contrast, code in Grid/ML has a uniform view of the network; cords must contain any application library code that they use, and always dynamically bind to the same fixed set of host resources no matter where they are run. Another advantage of the Acute language over Grid/ML is that Acute supports modules and preserves abstraction through marshaled data. We plan to introduce abstract types in the next iteration of Grid/ML, which is described briefly below. Like Jocaml, Acute does not certify the safety of mobile code, nor provide transparent failure recovery.

6. Evaluation

Because the systems described are still the subject of ongoing research, we dedicate a significant portion of the discussion to evaluating our current status and discussing future and current research.

Backend and runtime. Currently, our backend is unityped. For performance reasons, and in order to better ensure the correctness of the compiler, we would like to preserve types through compilation and move away from unityped representations. The main challenge in a heterogeneous representation with polymorphism is marshalling: We cannot marshal a term unless its representation is known. At least two solutions are available. Since Hemlock is a whole-program compiler, we could monomorphize the entire program so that the concrete representation of every value is known at compile time. Alternatively, we could use an intensional type analysis [28, 13, 31, 16] type system and pass representation information along with each run time value. Only the latter approach allows for separate compilation.

Garbage collection on the Grid is another concern. Because we want the Grid to run indefinitely, we need to collect stray cords and remove items from our memo tables. There are several algorithms for distributed garbage collection [25], though the problem is much more difficult than garbage collection within a single heap. Fortunately, we can tolerate even non-conservative garbage collection by interpreting premature collection simply as failure. This gives us simpler-than-usual requirements.

Locality. Grid/ML lacks some expressive power that is useful for Grid programming. In particular, tasks have no locality—they have no way to tell where they are running, nor the ability to ask to run in a certain place. For scientific computing, this is often the very point of Grid computing. Resources such as supercomputers, sensing equipment, or data farms are at fixed locations, and we need to migrate our code to those locations in order to make use of them. My thesis work is a successor to Grid/ML for location-aware distributed computing [22]. The language, called ML5, has a modal type system [23] based on the modal logic S5 [20]. In this language, expressions and bound variables are typed relative to locations, which gives us a static guarantee that the distributed program will make use of local resources safely. (For instance, the runtime errors that can arise in Grid/ML from using client resources in a cord are ruled out statically in ML5.) Nonetheless, we have evidence that many real Grid applications can be expressed with this limited set of primitives: applications such as SETI@Home [2], GIMPS [1], and countless other massively-parallel tasks can be implemented with unit-depth fork-join parallelism.

Result certification. Certified code does a good job of protecting node owners from malicious or imperfect code providers. Unfortunately, it does little to protect the users of the Grid against malicious nodes. For instance, it would be simple to create a counterfeit Conductor that published bogus results for any cord it saw. Sometimes this can be detected by the application—a factoring application can easily test if the factors returned have the correct product. Yet often we cannot easily distinguish good results from bad. We intend to leave such answer certification to the application developer, but need to provide support for certification strategies, and methods for exiling hosts that are caught cheating. A useful property of our approach is that cords are deterministic, and a cord's effect on the network can be summarized by its result. One way to protect against bogus results is to run the same cord on multiple, randomly chosen hosts, and compare the results. As long as cheating hosts are in the minority, we can vote to choose the correct result, and ignore the bogus ones. When we identify unreliable hosts, we can ignore future results from that same host. We believe it is also possible to share evidence of fraud with other hosts, so that unreliable hosts are soon ignored by the whole network [24]. The method is based on digital signatures: Each host signs the results that it returns, avowing that they are correct. The "evidence" that host H is fradulent is a message signed by X stating that the result of cord C is R, where R is incorrect. Since any host can run the deterministic cord C and see that the result R is wrong, it can independently verify that host H is unreliable. We also need to prevent a malicious person from creating many aliases for the same host, because each would need to be banned separately. To do this, we propose that the cryptographic keys that identify a host have a particular property that makes them computationally difficult to generate. Thus, each host joining the network pays an up-front, one time cost (on the order of a day's worth of computation). A malicious host must pay this cost every time it is banned, in order to re-join the network. These ideas are quite preliminary and we have not implemented them in any system.

Portability. One of the characteristic qualities of Grids is their heterogeneity. The Conductor currently runs on Linux and Windows (using Cygwin). Because the object file format for TAL on the two platforms is different (and incompatible), we also support two architectures: x86/ELF/Linux and x86/COFF/Windows. We support both architectures on the same grid by using the security policy to ensure that only Windows code is run on Windows hosts and Linux code on Linux hosts. An application can currently be compiled for Windows or for Linux and then run on the subset of the clients in the network that use that same OS.

Supporting multiple architectures for the same application is more difficult. Because we identify cords by hashes of their code (among other things), we'd need to extend this to multiple different code blocks for different architectures. We would also need to specify safety policies and provide certificates separately for each code block.[4] The biggest difficulty comes when attempting the automatic marshalling done with Hemlock. Because one cord's answer (running on architecture A) may be used by another cord (running on architecture B), the code for each architecture must agree on the format of the marshalled data. In particular, closures must be represented in the same way, which seems to imply that a single compiler must produce the code for each architecture simultaneously. This seems troublesome.

Bytecode-based solutions circumvent these issues by making the code the same for every platform. On the other hand, x86 machine code has through its ubiquity become—in some sense—a sort of portable bytecode itself. On all modern compatible processors, x86 is not executed directly, but dynamically compiled down to a RISC or VLIW instruction set by the CPU. Working the more cynical angle, it may be most economical to achieve the portability of Java bytecode by simply calling x86 a "bytecode," and then providing virtual machines for any architecture without hardware support. This is likely to be more tractable than typical emulation tasks because we typecheck a particular subset of the machine code for which there is well-understood semantics.

6.1 Conclusion

We have presented a system for fault-tolerant Grid programming in ML. In addition to providing a source language based on ML, we embrace ideas from functional programming in many components of the system, and argue that functional ideals are an excellent match for Grid computing in practice.

[4] In fact, there is no reason to believe *prima facie* that two architectures can necessarily even admit the same safety policies!

7. Acknowledgements

The author wishes to acknowledge the helpful guidance of his advisors, Robert Harper and Karl Crary, as well as the work of the ConCert "tiger teams," who designed the ConCert prototype and wrote many of the original applications.

References

[1] GIMPS, the great internet mersenne prime search, http://mersenne.org/.

[2] SETI@Home, http://setiathome.ssl.berkeley.edu/.

[3] The third annual ICFP programming contest, http://www.cs.cornell.edu/icfp/.

[4] Secure hash standard, August 2002. FIPS 180-2, National Institute of Standards and Technology.

[5] Andrew W. Appel. *Compiling With Continuations*. Cambridge University Press, 1992.

[6] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, 1997.

[7] B. Chang, K. Crary, M. DeLap, R. Harper, J. Liszka, T. Murphy, VII, and F. Pfenning. Trustless grid computing in ConCert. In M. Parashar, editor, *Grid Computing – Grid 2002 Third International Workshop*, pages 112–125, Berlin, November 2002. Springer-Verlag.

[8] Bor-Yuh Evan Chang. Iktara in ConCert: Realizing a certified grid computing framework from a programmer's perspective. Technical Report CMU-CS-02-150, Carnegie Mellon, 2002.

[9] T. Chothia and D. Duggan. An architecture for secure fault-tolerant global applications. In *Workshop on Principles of Dependable Systems (PODSY)*. IEEE Press, June 2003.

[10] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[11] Karl Crary. Toward a foundational typed assembly language. Technical Report CMU-CS-02-196, Department of Computer Science, Carnegie Mellon University, December 2002.

[12] Karl Crary, Michael Hicks, and Stephanie Weirich. Safe and flexible dynamic linking of native code. In *2000 ACM SIGPLAN Workshop on Types in Compilation*, September 2000.

[13] Karl Crary and Stephanie Weirich. Flexible type analysis. In *International Conference on Functional Programming*, pages 233–248, 1999.

[14] Ian Foster and Carl Kesselman. The Globus project: a status report. *Future Generation Computer Systems*, 15(5–6):607–621, 1999.

[15] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the Join-calculus. In *23rd ACM Symposium on Principles of Programming Languages*, January 1996.

[16] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *POPL 1995: The 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

[17] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[18] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proc. 7th Annual ACM Symp. on Principles of Distributed Computing*, pages 171–181, Toronto (Canada), 1988.

[19] B. Kuszmaul. The StarTech massively parallel chess program. *Journal of the International Computer Chess Association*, 18(1):3–19, 1995.

[20] Clarence Irving Lewis. *A Survey of Symbolic Logic*. University of California Press, 1918.

[21] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and

Steve Zdancewic. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, Georgia, May 1999.

[22] Tom Murphy, VII. Modal types for mobile code (thesis proposal). Technical Report CMU-CS-06-112, Carnegie Mellon, Pittsburgh, Pennsylvania, USA, Feb 2006.

[23] Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. Technical Report CMU-CS-04-105, Carnegie Mellon University, Mar 2004.

[24] Tom Murphy, VII and Amit K. Manjhi. Anonymous identity and trust in peer-to-peer networks. http://www.cs.cmu.edu/˜tom7/papers/.

[25] David Plainfossé and Marc Shapiro. A survey of distributed collection techniques. Technical report, BROADCAST, 1994.

[26] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: high-level programming language design for distributed computation. In *International Conference on Functional Programming (ICFP) 2005*, Tallinn, Estonia, September 2005.

[27] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[28] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. *ACM SIGPLAN Notices*, 35(9):82–93, 2000.

[29] Joseph Vanderwaart and Karl Crary. Foundational typed assembly language for grid computing. Technical Report CMU-CS-04-104, Department of Computer Science, Carnegie Mellon University, February 2004.

[30] Xiaoyun Wang, Yiqun Yin, and Hongbo Yu. Finding collisions in the full sha-1. In *Crypto ’05*, August 2005.

[31] Stephanie Weirich. Type-safe cast: Functional pearl. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 58–67, Montreal, Canada, September 2000.