Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2006-015 December 10, 2002

Amorphous Infrastructure for Language Implementatio

Ryan Newton and Jacob Beal

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Amorphous Infrastructure for Language Implementation

Ryan Newton Jake Beal

December 10, 2002

Abstract

We propose a method for the robust implementation of simple graphical automata on an amorphous computer. This infrastructure is applied to the implementation of purely functional programming languages. Specifically, it is used in conjunction with data-flow techniques to implement a toy language homologous to recurrence equations, exploiting control-flow parallelism through parallel operand evaluation. Also, data parallelism is explored in a separate implementation, in which a simple mark-up syntax enables Scheme programs to perform spatially-distributed tree-walking without modifying their semantics. This addition enables an idiomatically expressed interpreter to be trivially instrumented, producing a spatially distributed universal machine, and once again achieving control flow parallelism in the interpreted language.

Motivation

While amorphous computing ([1] [5] [6] [8]) has experienced success in fields for which there are strong biological precedents, such as pattern formation, regeneration, and morphogenesis, little work has been done to date on how to accomplish more traditional tasks on an amorphous computer. Specifically, how can we execute programs written in traditional programming languages?

The most obvious explanation for the lack of amorphous back-ends for compilers is that serial programming languages are incompatible with the massively parallel amorphous substrate. This statement is more or less true. However, we argue that imperative features are primarily at fault. Hence, while it would be preferable to address the whole class of "traditional programming languages", we tackle the smaller problem of addressing purely functional ones. Our approach proceeds by building virtual hardware in amorphous space; by targeting purely functional languages, we remove awkward synchronization constraints on the virtual hardware. In contrast, imperative languages make ordering of events critical, thereby demanding a synchronized flavor of virtual hardware. Further, effects eliminate the possibility of systematically re-instantiating subcomputations—a large potential source of robustness to hardware failure.

While functional languages fit the substrate better than imperative, the fit is far from "natural". A much more efficient treatment is possible for problems with a morphology more closely matched to an amorphous computer. Nonetheless, if amorphous computers become a physical reality, there exist situations in which one would want to run

'normal' programs on an amorphous computer. We theorize that it may be desirable to embed pieces of well-understood traditional logic to monitor and orchestrate the operation of a large amorphous system. Further, there may be contexts in which amorphous computers are the only alternative, because a properly utilized amorphous computer can provide certain physical benefits over its traditional counterparts: surviving tears and punctures and squeezing into any conceivable form factor.

In addition to these niche applications, language implementation provides a demanding context to explore various amorphous machines and tools—native tools which can equally well serve in the implementation of distributed-programming languages, or, more poignantly, may serve as primitives in the creation of uniquely amorphous languages.

1 Introduction

We will show how a data-flow graph formed from completely asynchronous collection of nodes, communicating with unreliable messages, is capable of implementing pure languages. The data-flow approach captures the limited parallelism implicit in evaluation of operands, which can be exploited by some algorithms but by no means constitutes a platform for distributed computing. However, our techniques for robust data-flow graphs apply equal well to the the implementation of other graphical automata. Hence they are likely capable of answering the more important question: How can we use established distributed algorithms straight out of the box on an amorphous computer? Whether or not this method is effective is not treated here. (But Appendix B relates our model to existing automata.)

Our data-flow abstraction is restricted to a static graph; nodes are neither created nor destroyed at runtime. Therefore, the implemented language has no heap and is not capable of allocation. To explore dynamism we propose a method for instrumenting ordinary Scheme programs, such that, when run on the HLSIM [10] platform, they construct and traverse spatially distributed data structures—in parallel.

This tool for spatial distribution can be combined with the robust-graph mechanisms presented in next section. These tools are the primary contribution of this paper, and provide reusable components at multiple granularities:

- The technique for implementing graphical automata used in this paper may be adapted to provide infrastructure for a variety of computations.
- The independent amorphous widgets discussed herein are general-purpose, and the spatial distribution system is similarly portable.

Since we are automatically compiling hardware (of a sort) our approach is vaguely related to the application-specific hardware efforts of the Phoenix group at CMU [3]. But, because they follow the imperative paradigm, their intermediate language as well as their compilation target differs greatly from ours [4]. On the other hand, since we use a mark-up language to enable implicit parallel programming, our approach has weak connections to projects such as the Jade language [9], but again, only weak connections.

2 Tools for Robustness

Here we consider the tools necessary to effectively build and maintain collections of spatial entities capable of communicating with each-other through dedicated channels (wires) constructed on an amorphous computer. The first result of this inquiry will be a number of modules implementing various tools. The second, complementary piece is a compiler to transform graph descriptions into amorphous programs that utilize said modules—programs suitable to be run in parallel on a multitude of identical computing elements.

The ActiveGradients Module

The ActiveGradient module has some new facilities which merit mention, above and beyond the active gradients used in previous work.[1] First is the introduction of versioned gradients, where a lower hop-count may be superseded by a higher hop-count if the higher count has a higher version number—important when you've got moving sources for a gradient. Second is that gradients can carry an arbitrary boundary test function with them (examples include bounding by hop-count, presence of another gradient, or hop-count past the boundary defined by another bounding function) and this function can be changed by a new version of the gradient.

The PersistentNode Module

The purpose of the PersistentNode module is to make self-repairing blobs that scootch away from congestion, damage, and network edges. To do this, the node calculates a "pull" of good processing resources in each direction and moves towards the best. Any given node is created with a name and a radius, and forms a region of that size bearing that label.

A node operates on a 4-stroke cycle of inward and outward data-flow. First, a gradient of size $3r$ is sent out by the first member of the node to timeout (this may be thought of as a "leader" node). Second, every node within $2r$ of the center calculates its heuristic value by summing the heuristic values of its successors, adding one, and dividing by its system load (estimated any way you want; the default estimator counts 1 load point for being in a node and 0.5 points for being within hearing range of a node's step 1 gradient). This then gives a higher heuristic value in the direction of regions with greater local density, less voids, and lower congestion. Third, the nodes compete as to which has the best heuristic value and wait for the numbers to converge; in most cases, the initiator of the gradient will be the high value. Finally, the winner designates its highest-valued neighbor to send the new gradient, and that node's timeout is advanced to ensure that it is most likely to send the new gradient.

Under this system, nodes drift at a rate of one hop approximately every $4 * r + 2 * timeout$ cycles, and tend to space themselves with $4r$ hops of separation.

[1] side note: these basicalaly send and maintain a gradient. When a processor stops hearing from lower hop-counts, it marks itself as "dead". When a dead processor stops hearing from higher dead hop-counts, it deletes the gradient. Thus life and death flow outward and deletion flows inward.

The PersistentWire Module

The biggest challenge in building the persistent wire module comes from the fact that its endpoints can move. The default endpoint which this module is supplied with are persistent nodes, but any predicate can be chosen to determine endpoints. The difficulty arises when the communication lag between two endpoints rises to the order of the frequency at which those endpoints are expected to move. So if a round-trip between two points takes 20 hops, and the endpoints move every 20 hops, then the information used to build the connection is out of date and may be useless or misleading.

To handle this difficulty, the wire-building process is directional and proceeds from both endpoints (if the wire is initiated at a single endpoint, then the first process initiates its mirror upon contact with the other endpoint). Going forward, an endpoint sends out a distance-limited versioned gradient which updates to new versions on a regular bases. Initially, we choose an arbitrary starting expected distance within which to search for its neighbor. If no contact is reported within the expected lag time, it doubles the distance and resets its wait. This is a continuous process, so wire-building adapts transparently to disruptions and loss of contact. Once the search process reaches the other endpoint, it sends the hop-count back down along a chain of predecessors to the source—although the source may have moved, the new versions of the gradient provide updated predecessor information. This chain of predecessors, and everything distance r becomes part of the wire.

Since new versions of the gradient are being sent out on a regular basis, the wire may change routes often (though the chain of predecessors will avoid changing when it can). In fact, the whole wire will sometimes appear to have regular ripples moving in both directions, as both endpoints update. Even though the wire may be globally disconnected, however, information traveling along the wire will find that it is locally connected, since the breaks are traveling as well.

Code-Gas

The Code-Gas module is designed to store data distributed widely across the network, so that any catastrophe has only a low probability of killing a given piece of data. Each unique piece of data added to the code-gas module becomes a species of particle in a gas-like medium. Particles randomly walk from node to node. The probability of going to a given node is inversely proportional to the number of neighbors that node has, which equalizes the expected concentration of particles in any given node.

At each step, any given particle has a chance of cloning itself or of dying. The probability of reproducing is allocated so that locally rare particles are more likely to reproduce than common ones: $P_r(s) = k_r * (1/n_s^2)/(\sum_i (1/n_i))$, where k_r is a constant and n_s is the number of particles of a given species at the node. The probability of dying $P_d(s) = k_d * n_s$, where k_d is a constant and n_s is the number of other particles at the node. Thus, the expected density of particles per node is k_r/k_d, and the number of particles in a species will tend to equalize and spread evenly across the network.

A particle of data in a processor may be fixed by that processor. It is expected that a processor will do this when it wants to use that data. A piece of data which is fixed may not die or leave the node (though it may reproduce, and additional copies of its species are not affected)

In many ways, this system is similar to the holistic storage techniques used by Butera [5] with the exceptions that the code-gas calculations use only local information, and regulate the number of copies of a piece of data, so that changing the number of species of data stored by the system does not change the load per node, but rather changes the degree of redundancy afforded each piece of data.

Compilation to amorphous program

The compiler for graph structures is thus quite straightforward. Code-gas is used to distribute seeds for each node in the graph. Thus, it is likely that enough information to reconstruct the entire graph will exist in any proportionally large region of the network. A starting seed is elected to develop into its respective node. (Minimum Id is one method for selection.) Once established, this node summons its neighbors from their slumber (by triggering the development of a single seed for each); they, in turn, awaken their neighbors, and so on.

Robustness

Each tool presented here has its own method of robustness. Wires heal breaks and reroute around large disruptions. Nodes can regenerate from even a single surviving processor. In section 4 we will see that, in the construction of data-flow machines, these components can be synergistically combined to further augment each other's resiliency. Namely, wires, the more spatially distributed entity, will take on a large share of the job of storing state.

3 A Tool for Spatial Distribution

Consider an arbitrary Scheme program running on an individual processor in an amorphous matrix. This program may easily overflow the small memory of its host. It would hence be desirable to utilize the memories of nearby processors as remote storage. We look at a method of automatically transforming Scheme programs to display this behavior, with the assistance of a small manual mark-up.

The graphical machines implemented in the last section are essentially static. Their *implementation* is dynamic in a number of ways: nodes may die off, replacements grow from seeds, and the parts of the system are physically active: wires maintain and reroute themselves, nodes drift to alleviate congestion. However, the abstraction is not capable of dynamically adding or removing nodes. Since we will apply this machine to language implementation, this will mean that our language will be incapable of allocation (with the exception of its call-stack, which will take the form of data accumulating in the wires). In contrast, this section aims solely at developing a general-purpose protocol for dynamically sequestering memory from the surrounding amorphous media.

The simplest general purpose method to utilize foreign memory is to form a chain of processors trailing out from the current one, and allow the original processor to operate on a virtual address space formed from the concatenation of the physical memories of the processor chain (minus the memory used for establishing this abstraction). This

is an undesirable method for a number of reasons [5, pg.166]. The first and most obvious improvement is to use a tree to index the virtual memory, making access time logarithmic in the size of the memory, rather than linear. Second, some sort of robustness to failure of individual processors is a mandatory requirement for practical amorphous algorithms.

A high-level approach

A high level solution to the problem is to redefine `cons`, `car` and `cdr`, such that `cons` has the potential to create cells on neighboring processors, returning some sort of pointers which `car` and `cdr` can follow. We operate on a subset of Scheme wherein pairs are the only non-immediate datatype, and {`cons`, `car`, `cdr`, `set-car!`, `set-cdr!`} are reserved names. These scheme programs are allowed to contain side-effects, but we will end up giving `cons` a lazy (but speculative) semantics, so the user is warned not to depend on any side effects that occur inside an operand expression of a `cons`. If one feels this results in too sloppy of a dialect, they are free to either abstain from the use of side-effects altogether, or to modify `cons` to wait for its operands to complete their evaluation. We prefer to harness some of the power of lazy evaluation, while not giving up entirely the power of mutation. Finally, before we handle the programs, we assume they have already been processed with the following standard transformations:

- Assignment conversion: variables are no longer mutable, having been converted to mutable pairs.
- Closure Conversion and Lambda-lifting: procedures have no free variables, all procedures are top-level.
- As well as the nonstandard conversion:
  Cons-lifting: Every call to cons in the source code is wrapped in a procedure and lifted to top-level. Free variables becoming formal parameters. Hence all cons calls become viable entry points. The procedures resulting from this lifting are called with a special syntax: `(cons-call` $\langle procedure \rangle$ $\langle fv \rangle$ `...)`

A fixed heap size is assigned, which limits how many cons cells each processor can hold. Since every processor has a randomly assigned Id (unique with arbitrarily high probability), such a cons cell may be referred to by a pointer of the form $\langle hostid, heapaddress \rangle$.

As the running Scheme process, we wish to foist cons cells off onto our neighbors, referring to them by their pointer. However, the job of referencing an arbitrary pointer is an awful one; an arbitrary processor in an amorphous network can only find another by releasing a gradient with radius greater than or equal to the minimum hop-count between the two processors. Indeed, this is what is done in the case of "lost pointers". But establishing a system of locally organized pointers can minimize or eliminate this occurrence.

For the purpose of dereferencing pointers, we build a network as follows. Nodes in the network are isomorphic to processors in the computer. Links in the network

represent the directional knowledge of one node about the location of another. All processors know (or can acquire) the Id's of their immediate neighbors, which amounts to sufficient knowledge of their location: their names and the fact that they are "next to me". Hence in the starting network, all processors are bidirectionally linked to their neighbors. This local knowledge forms the bedrock. If a pointer residing in a cons cell within the heap of a given processor is to be dereferencable using this baseline network, its $hostid$ must refer either its own processor or to an immediate neighbor. For now we require that all pointers be dereferenceable in such a fashion; later we will allow the building of non-local links in the network enabling some access of pointers with distant hosts.

Next we discuss a communication infrastructure which supports the actions alluded to here, allowing accesses, writes, and the allocation of adjacent memory. After that, with our tools for allocation, access and mutation in hand, we will come to this question: how to build a useful structure of pointers across the network of links? Local allocation by the root processor is clearly insufficient; it only provides access to memory available in the immediate vicinity of the root. The obvious solution would be allocating to further remote sites. We find this inefficient, and will instead pass on the allocating process itself.

The communication system

- *allocate(cons-name, values-for-free-vars)*:
- *access(path, heap-address)*
- *write(path, heap-address, value)*
- *call(path, heap-address, name, actual-parameters)*

We now introduce a communication system for distributing and retrieving cons cells, consisting of the four messages shown above, and a handful of implicit helper messages. The messages above assume underlying infrastructure which allows processors to send messages to specific neighbor, that is, other neighbors ignore the message. Further, the messages are introduced in conjunction with an implicit receipt system, which allows the receivers response to be channeled back to the original message-sending thread on its host processor (which blocks until it receives this response). We assume the presence of a standard thread system in the programming platform of the amorphous computer (or simulator). Operators and operands will be evaluated on separate threads in the host processor.

Generally speaking, a storage network is grown with the *allocate* message. All processors in the amorphous matrix are either part of the network or outside of it. All processors with cons cells in their heaps are inside it. The *allocate* message is directed at processors outside of the network, bringing them into it. *access* and *write* travel through the interior of the network, retrieving and modifying data respectively. The *call-child* message will be addressed below.

All the messages aside from *alloc* take *path* and *heap-address* arguments. The heap-address argument refers to a location in the heap of the processor receiving the message, indicating a cons cell. This cons cell acts as a root node, and the path argument denotes a tree index (through cars and cdrs) relative to it.

Handling *access* and *write* messages is thus straightforward. If the path is null, we operate on the cell in our heap. Otherwise we refer to the pointer in the car or cdr field of the cell (depending on the first symbol in the path). We take the processor-id and heap-address from this pointer, and broadcast a new message to this processor id with the heap-address from this pointer and the original path argument minus its first character. We wait on the receipt to this message before formulating our own.

Our notion of pointer is here revised to include a third field, the path—implemented as a string of 'a' and 'd' characters signifying car's and cdrs. The first and second fields of the pointer indicate a cons cell with a known location, the path allows other cons cells in the subtree of the known cell to be reached. The Scheme process running at the root of this tree can access and modify any cells stored in its own heap, and hence any cells in their subtrees. When *access* returns its response if the response is itself a pointer, its path field is updated as it moves up the tree, and when it is returned through `car` or `cdr`, it is annotated with a correct path field.

When a processor wishes to create a new cons cell, it searches for a volunteer, first in itself and then immediate neighbors. A processor "volunteers" in response to a request when the predicate function `available?` returns true. For now, the `available?` function simply returns true when heap is non-full, but improvements are certainly possible (Appendix A). The search for volunteers among neighbors is done by broadcasting the *request-alloc* messages—a helper to *alloc*. When a volunteer is found, allocation is initiated with the *alloc* message.

Cons: lazy vs. strict evaluation

Scheme is a strict language. Consider the program: `(cons 1 (cons 2 (cons (cons 3 ()) (cons 4 ()))))` which produces the list `(1 2 (3) 4)`. With a call-by-value semantics, the first cons to actually be evaluated will be `(cons 3 ())` or `(cons 4 ())`. This calling-convention has disastrous effects for our spatial allocation. The *leaves* of the tree are allocated first, and are allocated next to the root processor! Consider figure 1. For ease of visualization, we consider a heap-size of 1 such that cons cells can be equated with their processor hosts. Dotted lines represent the pointers, and solid arrows represent the action of allocating from a processor to a neighbor. The diagram on the right shows the sloppy sort of layout we achieve with this approach. The diagram to the left shows the layout that results from the following strategy.

Spatially, we want our trees to grow away from the root. The solution has two components: we must give cons a lazy semantics, and we must pass its operands (delayed computations) to child nodes. Hence a call to `cons` returns immediately, while a call to `car` may have to wait. Within the lazy semantics, we are use speculative evaluation. That is, the recepient of an *alloc* message doesn't need to wait on a `car` or `cdr` to begin evaluating the computation its received; it begins immediately.

This method will create the desired layout shown on the left of figure 1. However, the task of passing a subcomputation to a child amounts to the task of marshaling an arbitrary function across a boundary. Here we take the approach of transmitting the code for the function, along with the values of its free variables at the time when it is transmitted. This approach critically depends on the absence of mutation in variable bindings; the only mutation allowed is within cons cells, which are shared globally

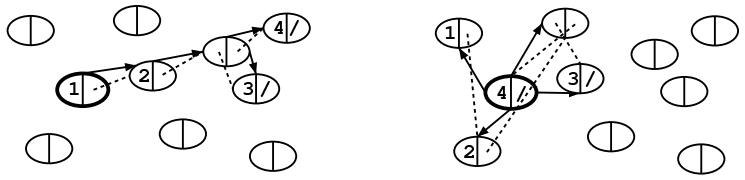

Figure 1: Lazy evaluation allows proper layout (left). Strict evaluation results in leaves being emitted before their parents (right). Dotted lines represent the pointers, and solid arrows represent the action of allocating to one processor from another.

on the network by all running computations. One might worry about ordering the accesses and writes coming from different computations in different locations; the rather convenient answer is that we don't have to; Scheme semantics does not guarantee a specific order of evaluation for operands, and parallel computations will only result from evaluation of separate operands.

As mentioned prior, all `lambdas` and `cons` have been lifted, and had their (formerly) free variables converted to explicit parameters. When `call-cons` is used, it either begins evaluating the cons locally, or it passes the `cons` off to a neighbor by means of the *alloc* message. Now we see the meaning of the *values-for-free-vars* component of the *alloc* message; these are the parameters for the top-level procedure named by *cons-name*. The receiver of an *alloc* message makes a dispatch to the named procedure with the given arguments.

One problem which arises once we have code executing on foreign processors, is that lost-pointers now become a possibility. A program executing on one branch of a tree may wish to access data laterally across from it on another branch, or in another, disconnected tree entirely. One solution to the former problem would be to track references to a cell (bidirectional pointers) so that an access can move up the tree as well as down. Our solution is to handle both cross-branch and inter-tree references by forming a gradient, and connecting the two processors with a wire (as seen in the PersistentWire module). `car` and `cdr` are responsible for updating the path fields of the pointers they return. Now when *a*lloc (and later *call-child*) carry a pointer *down* the tree, they must invalidate its path field, setting it to a distinct *no-path* value.

Call-child

We have painted a picture of a system that allows allocating procedures to travel over the matrix to doing their allocation. Further, we stipulated that, in the transformed program, operands and operators are be evaluated by separate threads. Thus allocating procedures can branch out over the matrix and operate in parallel.

The problem that remains concerns recursive procedures which access the these data-structures. A procedure doing only accesses would be left down at the root, reaching out to its data with a longer and longer arm. Or worse, consider the common pattern of a function walking over a tree to produce a new tree. Such a process would start off at the root, then begin off on a branch of its own (since `cons` migrates), possibly getting further away from the tree its accessing, and resulting in a stream of lost-pointer

references.

In this section we talk about the fourth message, *call-child*, which can ameliorate this situation. This message is similar to the *alloc* message, in that it contains the name of a top-level procedure (not a cons-lifted procedure in this case), along with actual parameters to instantiate it. We provide a single mark-up syntax, `possible-branch`. One places this in their source within definitions section of a `lambda`, as follows:

```
(lambda (x y z)
  (possible-branch y)
  ...)
```

By this, the user means that this procedure is doing a recursion over the data-structure referred to by argument `y`. This mark-up generates a chunk of code that checks to see if `y` is a pointer, if it is, *call-child* is used to pass off the procedure call to the host of the pair pointed to by `y`. This enables recursive procedures—such as one summing the numbers in a tree—to travel over the tree structures on which they operate, both getting closer to the data being accessed, and branching in a divide-and-conquer fashion.

The responsibility for this mark-up presently falls on the user. Automating the placement of this mark-up is a tricky problem not treated here. Multiple arguments may have pair values some or all of the time. Only one argument should be marked with `possible-branch` (otherwise an insane bouncing back and forth occurs). Deciding which one would likely require a combination of run-time and static analysis. Consider the procedure subst for replacing subexpressions: `(subst ls new old)`. It is clear to us that while both `ls` and `old` might be lists, we want the procedures control flow to follow `ls`. Or in the case of the `copy` procedure in Appendix C, no mark-up is desired at all.

With the system described here, tree walks that produce new trees will produce them approximately in place, alongside the tree being walked. Since the cons cells we have dealt with do not migrate between processors to relieve crowding (which would reap havoc on the pointer system) this case remains inefficient. But the punch-line is: since this mechanism is framed in terms of processors and communication links between them, it is feasible substitute the robust nodes and wires of the prior section for the fragile processors and links of this one. (This would require a dynamic graph, unlike the data-flow graphs to be used in section 4.) Were this accomplished, new trees allocated alongside old ones could potentially drift apart so as not to crowd each other.

Garbage Collection

A final note should be made about the task of garbage collection within this framework. As is standard within amorphous computing, we cause parts we don't want to die by cutting them off from a life-giving feed. In this case, we have the root node send pulses of liveness-preserving through its pointer tree. Cons cells not 'pinged' in this way for a certain length of time, die off.

⟨program⟩ ⟶
    `(letrec ((`⟨identifier⟩⟨proc⟩`)*)` ⟨exp⟩`)`
⟨proc⟩ ⟶ `(lambda (`⟨identifier⟩$^+$`)` ⟨exp⟩`)`
⟨exp⟩ ⟶ ⟨number⟩
    | ⟨identifier⟩
    | `(if` ⟨exp⟩ ⟨exp⟩ ⟨exp⟩`)`
    | `(`⟨primitive⟩ ⟨exp⟩ ⟨exp⟩`)`
    | `(`⟨identifier⟩ ⟨exp⟩`*)`
⟨primitive⟩ ⟶ `+` | `*` | `-` | `=`

Figure 2: Grammar for targeted mini-language

```
(Letrec ((fact
         (lambda (n)
           (if (= 0 n) 1
              (* n (fact (- n 1)))))))
  (fact 6))
```

Figure 3: Source code for figure 4

4 Applications: Data-Flow Machine

Our first approach to the problem of language implementation takes a data-flow perspective: given a program to execute, we generate a visible graph of variables and operations in the amorphous matrix that, when run, simulates the execution of the program. In this approach we implement the language of figure 2. This grammar defines a highly restricted, syntactically Scheme-like language for computing recursive numeric functions, and has the anticipated semantics.

Compiling a Data-flow Graph

The first step in our compilation, prior to generating code for the amorphous processors, is to convert input programs into a data-flow intermediate language. As an example, consider the code of figure 3, and the resultant graph in figure 4.

There are six kinds of nodes in this language: `variable`, `constant`, `primitive`, `lambda`, `call`, and `if` as well as a dummy node named `output`. In the diagram, all data-pipes drawn with arrowheads are restricted to one-way transmission; those drawn with dots as endpoints can support two-way communication (merely a shorthand for separate to and fro pipes). The message passing semantics is quite lenient: messages may be duplicated or lost, in fact, they can even be defferred (the wire serves as a buffer until the node wishes to consume the message), but they may not be garbled or delivered to the wrong destination.

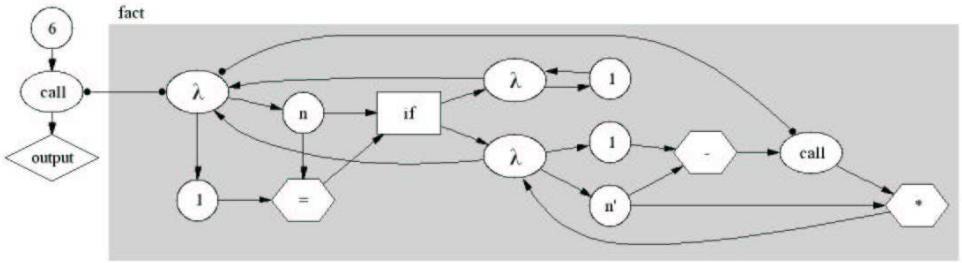

Figure 4: Data-flow graph generated from code of figure 3

⟨constant⟩ ⟶ ⟨input⟩ ⟨output⟩
⟨variable⟩ ⟶ ⟨input⟩ ⟨output⟩*
⟨primitive⟩ ⟶ ⟨input⟩ ⟨input⟩ ⟨output⟩
⟨lambda⟩ ⟶ ⟨bidirectional⟩* ⟨output⟩* ⟨input⟩*
⟨call⟩ ⟶ ⟨input⟩* ⟨bidirectional⟩ ⟨output⟩
⟨if⟩ ⟶ ⟨input⟩ ⟨input⟩* ⟨output⟩ ⟨output⟩

Figure 5: Grammar describing the pipe structure of nodes

The Analogue to the Stack Frame

Typically the instantiation of a new procedure call elicits the allocation of a new stack frame in which to store values of local variables along with a return pointer. In our data-flow languag, new procedure calls result in the creation of a new, unique *call-tag*. All values in the system are transmitted in conjunction with their call-tag (i.e. $\langle \alpha, 3 \rangle$ and $\langle \beta, 9 \rangle$ are potential messages). The `call` node is responsible for allocating new call-tags.

Messages bounce around in the pipes until the receiving node chooses to consume them. Hence, pipes serve as registers as well as a transmission mechanism.[2] In fact, accumulation of data in pipes is the only form of run-time *allocation* that this system supports. For example, a non-tail recursive function might go through an α, β and γ call before terminating, in which case messages with all three of these tags will coexist in the function's graph. This mechanism has an added benifit, not only can these tags coexist, multiple calls of the same function can happen simultaneously from separate call sites without interfering with eachother. This will become more clear with the elucidation of the node semantics.

[2]In standard graphical automata, the state held in our wire would probably be modeled by "port buffer" state within the nodes.

Node Semantics

All nodes operate asynchronously according to the following rule:

When the desired inputs are available, with a common call-tag S, fire, discharging one or more S-tagged messages into the appropriate outgoing pipe(s).

The semantics of each node are as follows:

- `constant`

  Accepts a single input, produces on its output some number of messages (possibly over a period of time) tagged with the call-tag of the input, but with the value of the constant.

- `variable`

  Accepts a single input containing appropriately tagged values for the formal parameter. It produces some number of output messages on all of its outgoing pipes, each output message identical to the input message.

- `primitive`

  Accepts two inputs and produces a single output. When messages $\langle S, x\rangle$ and $\langle S, y\rangle$ are waiting on the input pipes (respectively), the `primitive` produces some number of output messages of the form $\langle S, z\rangle$, where z is the result of computing that particular primitive for x and y.

- `lambda`

  `lambda`s have an arbitrary number of bidirectional pipes correlating with the number of call-sites for that `lambda`. The input portion of this pipe provides incoming actual parameters, which must have formal parameter names (or indices) for disambiguation. Together with the call-tag, an actual parameter is of the form $\langle S, name, value\rangle$. The output portion of this pipe serves the role of a return address.

  In addition to these bidirectional pipes, `lambda`s have an arbitrary number of additional outputs which instantiate the body. There will be one of these pipes to each of the leaves (`constant` and `variable`s) in the body. On each call, `variable` nodes are sent the appropriate actual parameters, and `constant`s are sent a dummy value with the appropriate call-tag. This enables the constant to begin broadcasting some appropriately tagged values.

  Finally, an arbitrary number of additional input pipes bring values returned from the body to the `lambda`. A message on any of these pipes results in a dispatch to the return addresses.

- `call`

  A `call` node possesses an arbitrary number of inputs (actual parameters), a single bidirectional pipe, and a single output pipe. When all inputs $\langle S, a_1\rangle \ldots \langle S, a_n\rangle$ are present, they are tagged with a new call-tag and with the appropriate formal parameter names, becoming $\langle T, f_1, a_1\rangle \ldots \langle T, f_n, a_n\rangle$, and are all sent down

the bidirectional pipe (presumably to a `lambda` node). Again, each may be sent multiple times with some temporal pattern of the nodes choosing. When a response, $\langle T, value \rangle$ is heard on the bidirectional pipe, messages of the form $\langle S, value \rangle$ are issued to the single output port.

- `if`

  An `if` node receives an input representing the *test*, it has two outputs representing the *consequent* and *alternative*. The consequent and alternative, will both have been converted to `lambda` nodes (see below for explanation). It should be noted that "bidirectional pipe" is just a notational shorthand for "one input + one output pipe". In the case of the consequent and alternative `lambda`s The actual-parameter pipe will come from the `if`, but the return-address pipe will go to the `if`'s return address.

  The `if` has an arbitrary number of input pipes coming from variable nodes, in this manner it collects values for the free variables of its consequent and alternative expressions. `if` waits until it receives a boolean on its primary input, then it transmits the free-var values to the appropriate branch, triggering a call of its `lambda`.

Now to explain the introduction of extra `lambda` nodes that is seen in figure 4. Parts in our machine operate asynchronously. Yet it is unacceptable to execute procedure calls in un-taken branches of `if`s. Thus, some constraint upon control flow for branches must exist. This is accomplished by wrapping the consequent and alternative branches of if-expressions with lambda-expressions. These anonymous `lambda`s need no `call` nodes; the `if` instantiates them with the current call-tag. The purpose is to deprive un-taken branches of the value messages they would require to compute. Because the leaf nodes in the `if` branches (`constant`s and `variable`s) now depend on the the inner `lambda`s for instantiation, this goal is accomplished. The motivation for this technique was to maintain asynchronous firing, without the need for explicit activation wires (as are used in other graphical intermediate languages). We realize that this decision could reasonably be reversed, and activation wires could be added only to `call` nodes, which would save the extra wires needed to pipe variables values through `if` nodes.

Construction and Execution of the Amorphous Program

The tools of section 2 make implementation of the data-flow intermediate language straightforward. Seeds for each node in the graph are distributed through a Code-gas. At least one seed of each type (for each node in the graph) is coaxed into budding. Because of our message passing semantics, it woudl even be acceptable if duplicate physical nodes exist for a given abstract node.

Robustness

When this data-flow language is implemented with the tools of section 2, the system's acquires a baseline robustness (to small-scale cell deaths) that comes from robustness in the primitives that make it up. In the case of large-scale, regional cell-deaths, our system still has means for perseverance.

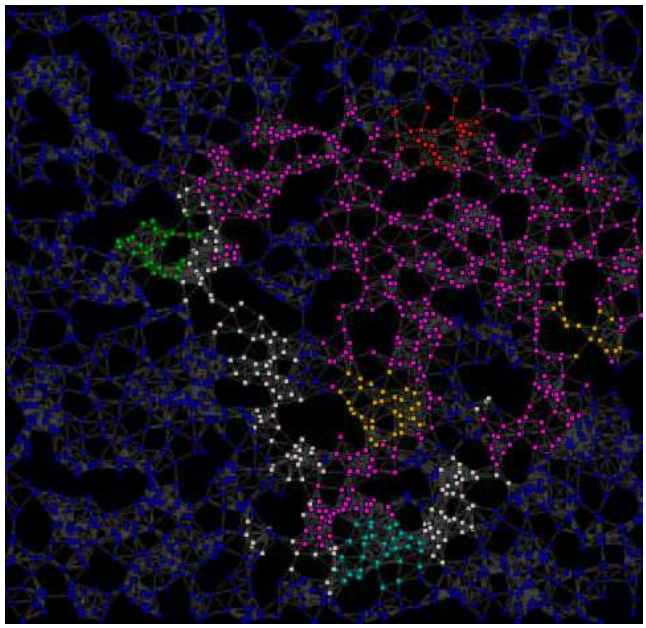

Figure 6: Execution of (lambda (x) (+ x 3)). The green node is a `lambda`, the yellow nodes are `primitives` and `constants`, the turqoise node is an `operator`, and the red node serves as a helper extension to broadcast `lambda`'s message to the variables and constants.

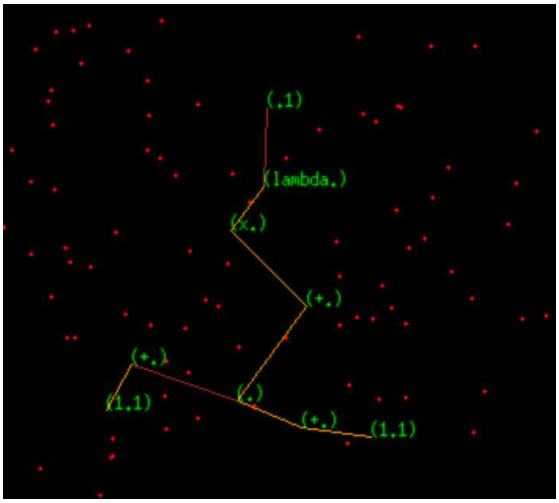

Figure 7: Execution of ((lambda (x) (+ (+ x x) (+ x x))) 1). The cons cells are drawn $(\langle car \rangle.\langle cdr \rangle)$. The cell `(.1)` is the root of the computation. This snapshot was captured right after 1 was substituted for the occurrences of 'x' within the lambda's body.

Consider losing an entire node to hardware failure. The code-gas contains distributed seeds for each node in the graph: any sufficiently large area of the matrix should contain a full description of the graph. Since the nodes themselves contain no internal state it is acceptable to replace a lost node with a new copy grown from a seed. If part of the wires attached to the lost node survive, and manage to reconnect to the new node, we may salvage the messages that were in transit. If not, hopefully the source of the messages is still broadcasting them. If not, then we may yet be able to restart the computation at from the `call` node for the procedure which sustained damage (once the function is regenerated). This is a perk of a purely functional system; if we know the function f was called with arguments $\langle \alpha_0, v \rangle \ldots \langle \alpha_n, v \rangle$ before it sustained damage, we can simply repair the function and call it again with identical arguments.

5 Applications: Distributed Interpreter

Here we build an interpreter that dynamically spreads out its storage and control flow across the surface of an amorphous computer. Using the tool for implicit spatial distribution from section 3 makes this is exceedingly straightforward. In fact, the code from Appendix C does the job: an idiomatic interpreter with two `possible-branch`'s included. A visualization of the result can be seen in figure 7. (Note, these computing elements are not sparse, the view is merely zoomed.) In this picture, a heap-limit of one is used, such that each processor hosts a maximum of one cons cell. The cons cells are drawn $(\langle car \rangle.\langle cdr \rangle)$, where $\langle car \rangle$ and $\langle car \rangle$ are symbols, numbers, or the null string, which denotes a pointer. This snapshot was captured right after 1 was substituted for the occurrences of 'x' within the `lambda`'s body.

Behavior of the interpreter

The two significant functions in this interpreter are `eval-exp` and `subst`. This evaluator works by beta-reduction; `eval-exp` uses `subst` to perform applications by substituting operand expressions for references to formal parameters within a lambda's body. Both of these functions do tree walks over a list structure. `subst` is destructive, walking over a parse tree and mutating portions of it. `eval-exp` is mainly a gatherer, crunching numbers together and bringing them back.

This interpreter processes a call-by-value version of the λ-calculus, augmented with numbers, primitives and if. As an effect of executing this interpreter with the spatial-distribution system, evaluation occurs in parallel on operands and operators.

6 Conclusions

Purely functional languages can be implemented on an amorphous substrate reasonably effectively. The same is not necessarily true of imperative languages, but this question has yet to be answered. A related and potentially promising area is the implementation of graphical automata used in distributed algorithms. Such implementations have the potential to be robust and possibly efficient. We have provided general tools to enable such an effort. However, even if amorphous computers effectively subsume

traditional distributed algorithms, the reverse is not necessarily true. Purely computational tasks should be specified in a way that abstracts over the spatial details of an amorphous computer, but there are a number of spatially dependent amorphous tasks, such as the the formation of specific two-dimensional shapes. Hence, uniquely amorphous languages need to incorporate an element of spatial control. We remain blind to what form these future amorphous languages will take, but we hope that some of the primitives discussed here are helpful to their formation.

References

[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R. Weiss. Amorphous computing. *MIT AI Memo*, 1999.

[2] Paul C. Attie and Nancy A. Lynch. Dynamic input/output automata, a formal model for dynamic systems. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 314–316. ACM Press, 2001.

[3] Mihai Budiu. Application-specific hardware: Computing without cpus. *SOCS-4*, October 2001.

[4] Mihai Budiu and Seth C. Goldstein. Pegasus: An efficient intermediate representation. *CMU CS Technical Report*, May 2002.

[5] W. Butera. *Programming a Paintable Computer*. PhD thesis, MIT, 2002.

[6] D. Coore. *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*. PhD thesis, MIT, 1999.

[7] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. *IOA: A Language for Specifying, Programming, and Validating Distributed Systems*, 1997.

[8] N. Radhika. *Programmable Self-Assembly using Biologically-Inspired Multiagent Control*. PhD thesis, MIT, 2002.

[9] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of jade. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):483–545, 1998.

[10] High Level Simulator. http://www.swiss.ai.mit.edu/projects/amorphous/.

Appendix

A Notes: limitations of current implementations

Data-flow compiler

The current implementation uses Beal's Java version of the HLSIM simulator. It supports all of the widgets described in section 2, with the exception that PersistentNode's

centers are assumed never to die, thus eliminating the third stage of their four stage cycle. The `call` and `if` nodes remain incomplete; the demos operate by manually instantiating a `lambda` node. Further, the ability to restart function applications is not yet operational in the current implementation.

Distributed interpreter

The current implementation uses the HLSIM platform [10]. Garbage collection is incompletely implemented, but messages to support dependency by the tree on the root *are* in place, and are used for the deletion of misallocated cells (as well as the inhibiting further allocation by misallocated cells). Misallocated cells occur when a child manages the volunteer handshake to accept an allocation, but receipt to the allocation fails to get back to the parent, and the parent takes its allocation elsewhere.

Thin, simple wires are in operation, as well as transmission of messages using them. When the interpreter performs a beta substitution involving a procedure as argument, it needs to copy the code of the input procedure into one or more locations in the the code of the body. This is a lateral subtree copy across the branches of the tree, and should be supported transparently through "lost pointers", but is not working presently. Thus, the interpreter cannot handle first class procedures.

Possible Improvements

- The `available?` predicate presently returns true if there's any space at all in the heap, and false otherwise. However, completely saturating up the heap of a given processor is highly undesirable. A simple improvement would be to make `available?` probabalistic, such that, it returns true with higher probability when the heap is mostly empty, and lower probability when it is mostly full.
- Messages which bounce around in wires should switch into a slow-mode if not consumed for some period of time. Then a node could issue 'flush' commands when it wants to retrieve This makes wires more closely resemble registers.
- The spatial distribution mechanism should spill over into neighboring processors for an overflowed stack as well as an overflowed heap.

B Input/Output Automata

There exist many formalisms for distributed computing. However, it has been claimed that such models fail to provide "a comprehensive descriptive framework with the formal power of Turing's work" [5, pg.167].

This paper deals with building node-and-wire machines on an amorphous substrate. As such, it is inevitable to ask how established models of distributed computation might inform our efforts.

State machine models such as I/O Automata [7] typically incorporate a deterministic message passing semantics. The model that we use in our data-flow implementations instead use a nondeterministic semantics wherein messages can be duplicated,

lost, and reordered, but not garbled. Thus our data-flow model could be framed as a nondeterministic I/O Automata.

We chose not to incorporating message-preserving abstraction because, as will be shown, it is unnecessary for our data-flow application and merely adds overhead. On the other hand, the interpreter of section 5 relies on guaranteed message delivery as well as dynamic extensions: the ability to spawn and delete nodes at run-time. Thus this could be cast as a Dynamic I/O Automata as defined by Lynch and Attie [2]. However, it should be noted that in our actual spatial-distribution system implementation we liberally break the abstraction to use cheap, fast non-guaranteed messages, essentially, we make use of both the UDP and TCP protocols.

C Code for a simple interpreter

Note, the following grammar and interpreter use a compressed form for representing programs, which saves cons-cells by using improper lists.

⟨expr⟩ ⟶ ⟨number⟩
| ⟨identifier⟩
| (`lambda` . (⟨var⟩ . ⟨exp⟩))
| (`if` . (⟨exp⟩ . (⟨exp⟩ . ⟨exp⟩)))
| (⟨primitive⟩ . (⟨exp⟩ . ⟨exp⟩))
| (⟨exp⟩ . ⟨exp⟩)

```
;; This is a destructive substitute:
(define subst
  (lambda (body new old)
    (possible-call-child body)
    (cond
     ((integer? body) body)
     ((symbol? body) (if (eq? body old)
                         (copy new)
                         body))
     (else
      (let ((ar (car body))  ;; Be careful with c*rs,
            (dr (cdr body))) ;; they're expensive now.
        (cond
         ((eq? ar 'lambda)
          (let ((v (car dr)))
            (if (not (eq? v old))
                (set-cdr! dr (subst (gcdr dr) new old)))
            body))
         ((primitive? ar)
          (set-car! dr (subst (gcar dr) new old))
          (set-cdr! dr (subst (gcdr dr) new old))
          body)
         (else
          (set-car! body (subst ar new old))
          (set-gdr! body (subst dr new old))
          body)))))))

;; This is a beta-reducing valuation function:
```

```
(define eval-exp
  (lambda (expr)
    (possible-branch expr)
    (if (integer? expr) expr
        (let ((ar (car expr)) (dr (cdr expr)))
          (cond
           ((primitive? ar)
             (let ((op1 (eval-exp (car dr)))
                   (op2 (eval-exp (cdr dr))))
               (apply-primitive ar op1 op2)))
            ((eq? 'if (car expr))
               (if (eval-exp (car dr))
                   (eval-exp (car (cdr dr)))
                   (eval-exp (cdr (cdr dr)))))
            ((eq? 'lambda (car expr)) expr)
            (else
              (let ((rator (eval-exp (car expr)))
                    (rand  (eval-exp (cdr expr))))
                (let ((formal (car (cdr rator))))
                  (eval-exp (subst rator rand formal)))))
            )))))

;; Copy does not use possible-branch because doing
;; so would create a large number of lost pointers.
;; further, it forces evaluation of cons'operands.
;; Thus a maximum of one lost-pointer (and subsequent
;; wire build, should occur as a result of copy).
(define copy
  (lambda (t)
    (if (pair? t)
        (let ((ar (copy (car t)))
              (dr (copy (cdr t))))
          (cons ar dr))
        t)))

(define (primitive? p) (memq p '(+ - * =)))
(define (apply-primitive prim op1 op2)
  (case prim
        ((+) (+ op1 op2))
        ((-) (- op1 op2))
        ((*) (* op1 op2))
        ((=) (= op1 op2))))
```