Chapter 10

Conclusion

10.1 Summary

The solutions to many programming problems can naturally be expressed as the composition of procedures that manipulate lists and trees. By capturing common idioms in reusable pieces, this modular program organization facilitates reading, writing, modifying, and reasoning about programs. Unfortunately, issues of space, time, and operation scheduling often compel programmers to manually interweave several idioms into a single complex loop or recursion. The result is a monolithic program that is significantly harder to construct, understand, and extend than the modular version.

I have developed a technique that enables programmers to achieve the desirable space characteristics and operation scheduling of intricate loops and recursions in modular list and tree programs. The technique is based on a lock step processing model that allows networks of aggregate data operators called *slivers* to capture the fine-grained operational behavior of monolithic procedures. Slivers communicate via *synchronized lazy aggregates*, standard interfaces that transmit not only elements but control information that specifies when the elements are computed. Control information is in the form of *synchrons*, a novel barrier synchronization mechanism that is fundamentally tied to automatic storage management. Synchrons enable the barriers represented by strict monolithic procedures call to be simulated by a network of concurrently executing slivers.

The sliver technique combines the simplicity and expressiveness of lazy data with the synchronization and storage efficiency of interprocess communication channels. The sliver approach is a dynamic version of Waters's series technique that is able to express general linear and tree-shaped recursive computations in addition to iterations. By allowing the complexities of synchronization to be hidden from the programmer, slivers improve upon the concurrency and synchronization techniques suggested by Hughes for controlling space behavior in modular programs.

I have implemented a tower of languages (EDGAR, OPERA, and SYNAPSE) that support the lock step processing model. EDGAR is an explicit demand graph reduction model in which the operational details of concurrency, synchronization, and non-strictness are formalized. OPERA is a dialect of Scheme in which these operational features are embedded. SYNAPSE is a collection of higher order OPERA procedures that manipulate synchronized lazy lists and trees. I have shown how a number of intricate computations can naturally be expressed as modular combinations of SYNAPSE procedures, and that the resulting programs mimic the detailed space behavior and operation scheduling of hand-crafted loops and recursions. Synchronized lazy aggregates can also be used in conjunction with conventional strict and lazy aggregates to partition a computation into loosely coupled networks of tightly coupled components.

The slivers technique addresses the space overheads, but not the time overheads, associated with aggregate data programs. While sliver programs have the same time complexity as corresponding monolithic programs, prototype implementations indicate that they suffer significant time overheads due to the manipulation of intermediate data and the management of fine-grained concurrency and synchronization. Nevertheless, I expect that it will be possible to eliminate these overheads in many cases by employing techniques similar to those used by Waters to compile series programs into monolithic loops and recursions.

Despite the pragmatic drawbacks, the work described here represents an important step towards reducing the tension between modularity and control in programming. At the very least, slivers and synchronized lazy aggregates constitute an executable specification language for expressing operational characteristics of programs. More importantly, the notions of computational shape and lock step processing developed here provide new ways for programmers to think and talk about computation.

10.2 Contributions

Here is a summary of the major contributions of this work:

- I have presented a detailed analysis of why existing modularity techniques fail to preserve important operational characteristics of monolithic loops and recursions.

- I have laid the groundwork for a theory of *computational shape* that describes how the operational processing patterns of a computation can be composed from the patterns of its components. The preliminary notion shape presented here successfully captures intuitions about linear and tree-structured computations.

- I have developed a lock step model of computation that allows programs written in the aggregate data style to control space behavior and operation scheduling in a modular fashion. The model introduces two new abstractions that successfully encapsulate the details of lock step processing: a *sliver* is a processing module and a *synchronized lazy aggregate* is a data structure that guarantees the lock step processing of the slivers that produce and consume it. I have demonstrated the utility of the lock step processing model in the context of SYNAPSE, a list and tree processing language.

- I have invented a novel synchronization mechanism, the *synchron*, that interacts with automatic storage management to provide a general form of barrier synchronization. Synchrons are the key technology for implementing synchronized lazy aggregates and achieving the lock step processing of slivers.

- I have designed a semantic framework, *explicit demand graph reduction* (EDGAR), that models concurrency, synchronization, non-strictness, and side effects in an intuitive way. I have implemented a graph-based interpreter based on EDGAR and used it as a foundation for OPERA, a concurrent dialect of Scheme that supports synchrons.

- I have implemented a graphical program animator, the DYNAMATOR, that visually illustrates the step-by-step execution of the EDGAR model. The DYNAMATOR shows great promise as a tool for debugging and pedagogy.

10.3 Future Work

10.3.1 Expressiveness

The SYNAPSE implementation validates the lock step model upon which slivers are based, but it is far from an ideal system for expressing computations in a modular way. While it is fairly easy to express a wide range of linear computations in terms of existing synquence slivers, there is still much functionality missing. More linear primitives, along the lines of those developed for series [Wat90], need to be added.

Syndrites are in a considerably more embryonic state than synquences. The current set of syndrite slivers is very weak. The mechanisms for specifying different orders and directions of sequential tree traversals are rather ad hoc; better abstractions for sequential processing of trees need to be found. It would also be helpful to develop a sliver that sequentializes parallel tree manipulations in a reusable way. Designing slivers whose processing shapes are dynamically determined rather than being statically defined is another important avenue of exploration. For example, imagine a syndrite that performs a left-to-right pre-order walk if its argument is even, but a right-to-left post-order walk if its argument is odd.

To drive the design of an expressive suite of syndrite slivers, the sliver technique needs to be tested out on a wide range of complex tree-shaped computations. I have only experimented with a few relatively simple tree programs: free and bound variable analysis, alpha renaming, lambda lifting, and pattern matching. At the very least, slivers should be used to duplicate results from attribute grammar research within the aggregate data style. A more ambitious undertaking would be a detailed study of existing tree programs along the lines of Waters's study of iterative programs [Wat78, Wat79].

Compilers and interpreters are particularly relevant candidates for study. One of the main pragmatic motivations for slivers was to be able to describe compilers and interpreters in a modular, but still efficient, fashion. Currently, interpreters are not easily accomodated by SYNAPSE because the dynamic evaluation trees generated by an interpreter tend to exhibit complex dependencies between subtrees. For example, when evaluating an application node in a Scheme interpreter, the environment associated with the result of evaluating the operator subtree is needed in the subtree for evaluating body; this dependency does not correspond to an existing syndrite sliver.

Filtering is an area that requires more work. While reusable filtering works surprisingly well for synquences, the details of syndrite filtering have yet to be ironed out. Compacting the elements of a filtered list can often be accomplished using the loose coupling afforded by stream buffers; but this technique can lead to deadlock in situations with fan-out. The default gap handling properties of mappers and accumulators is often inappropriate when multiple inputs are involved. The existence of a large corpus of tree examples would help in the design of better filtering constructs.

Cyclic dependencies are another area requiring further investigation. Some problems naturally decompose into components that exhibit cyclic dependencies. For example, in the FX language [GJSO92], name resolution and type reconstruction are conceptually distinct stages of the implementation that happen to depend on each other. In practice, the implementations of these stages must be manually interwoven. Slivers offer the possibility that this process could be expressed as a cyclic combination of a name resolution sliver and a type reconstruction sliver. While I have successfully implemented a cyclic Fibonacci network based on slivers,[1] I have not yet developed robust enough abstractions for experimenting with more complex examples.

Tail calls should be expressed in a more elegant manner. The tail call of a monolithic procedure is currently simulated in SYNAPSE slivers by using a special calling mechanism that does not rendezvous at an up synchron. But this requires two versions of every calling abstraction. Is it possible to design a cleverer calling convention that automatically waits on an up synchron only if there is pending work to be done?

10.3.2 Pragmatics

In order for synchronized lazy aggregates to be a practical technique, it is necessary to significantly reduce the time and space overheads exhibited by the current SYNAPSE and OPERA implementations. This area is ripe for exploration.

The time overhead is by far the biggest problem. While SYNAPSE programs have the

[1] It turns out that there can never be a cycle consisting purely of slags within a sliver network. At least one connection in every cyclic dependency must be a non-slag.

same order of growth in time as the monolithic programs that they model, preliminary results indicate that they take more than an order of magnitude more execution steps than monolithic programs executed under the same graph reduction model. The extra steps are due to the following sources:

- Packaging and unpackaging the elements of slag.
- Manipulating and rendezvousing on synchrons.
- Manipulations for handling filtering appropriately.
- Handling of `lazon`s, `eagon`s, and touches that control fine-grained behavior.

Because the comparisons were performed in a graph reduction model, these overheads do not include any overheads of the graph reduction model itself. In practice, straightforward implementations of the fine-grained concurrency and reference-counting garbage collection implied by EDGAR and OPERA are likely to incur additional significant time penalties compared to traditional sequential implementations of monolithic programs.

While SYNAPSE programs can exhibit the same order of growth in space as a monolithic program, they still have higher constants due to the memory requirements of slags. For example, a constant-space iteration composed out of two SYNAPSE slivers requires extra storage for the intermediate slag structure necessary to communicate a single element from one sliver to another.

There are two basic approaches for reducing the overheads associated with SYNAPSE

1. The *dynamic approach* is to develop efficient mechanisms for handling the fine-grained concurrency, synchron synchronization, and non-strictness required by OPERA.
2. The *static approach* is to develop static analysis and compilation techniques that remove the sources of overhead.

Unfortunately, the dynamic approach does not address many of the sources of SYNAPSE inefficiency. In particular, it does nothing to reduce the allocations, stores, and reads from intermediate data structures, nor does it remove the extra manipulations associated with filtering. Nevertheless, if other overheads are brought down to low enough levels, these remaining overheads may be deemed acceptable for certain classes of programs. Since programmer time is often more valuable than machine time, the ease with which a program can be written can be more important than how fast it is.

The static approach holds more promise for increased efficiency. Ideally, it should be possible to eliminate the need for run-time overhead completely by compiling many sliver networks into efficient monolithic recursive procedures. Techniques that are particularly worth exploring in this context include series compilation [Wat91], deforestation [Wad88], partial evaluation [WCRS91], and attribute grammar evaluation [DJL88]. Series compilation is based on representing a series operator as collection of code fragments that characterize different aspects of the operator; compilation consists of gluing together corresponding fragments for a network of series operators. This approach appears promising for slivers, but the details need to be worked out. Both deforestation and partial evaluation techniques can have trouble dealing with general recursions, but the limited forms of recursions in SYNAPSE and the explicit synchronization information available may provide enough constraints to make the techniques applicable.

It is worth emphasizing that requiring slivers networks to be compiled into monolithic recursive procedures will result in reduced expressive power. Since any such compilation technique would need to determine the structure of a sliver network at compile time, many of the more expressive aspects of slivers (e.g., higher-order slivers, repeated application of a sliver, conditionally choosing a sliver at runtime) would have to be curtailed. Of course, an option would be to compile only those networks that are statically determinable, and leave the more expressive cases to be handled by existing dynamic methods. But if the goal is to eliminate the run-time overheads of features like fine-grained concurrency, this option is not viable.

If dynamic overheads are deemed allowable, then other forms of static analysis (in addition to the compilation techniques mentioned above) can be used to reduce these overheads. For example, using the technique of *effect analysis* [LG88], it may be possible to distinguish expressions that need to be evaluated concurrently from those for which sequential evaluation is sufficient.

10.3.3 Computational Shape

The notions of computational shape introduced in Chapters 4 and 5 are alluring but very preliminary and informal. An important area of future research is to formalize these notions into a *shape calculus* that can be used as a basis to help both humans and machines reason about programs. A shape calculus would capture the essential interfaces, call structures, and internal dependencies of process fragments like slivers and use this information to determine both pairwise sliver compatibility and the processing shapes of lock step components.

A shape calculus could be the basis for many practical tools. In analogy with type systems, it is possible to imagine a *shape system* in which the shape of sliver network can be determined from the shape of its components by *shape reconstruction*. Shape analysis would be helpful for conservatively predicting whether a sliver network could deadlock. Any sliver compilation technique would presumably have to employ some sort of shape analysis.

Shape-based reasoning shows promise as a new way to approach program transformations. Existing algebraic techniques may benefit by incorporating a notion of shape. Based on a library of slivers partitioned according to function, it may be possible to improve the operational characteristics of a given program by replacing some slivers with differently-shaped, but functionally equivalent, slivers.

Another direction to explore is extending the simple linear and tree shapes suggested by this research. Are there processes shaped like arrays, DAGs, and graphs? What process shapes are suitable for describing data parallel computations? How can the process shapes generated by fancy control constructs (e.g., non-local exits and continuations) be expressed? Is it possible to convert a process of one shape to one of another shape? (For example, perhaps sequential tree walkers can be described as linear processes that are appropriately "bent" to form a tree.)

It would also be worthwhile to explore synergies between the notion of shape developed here and other notions of data, communication, and computation shapes. The dependency analysis underlying many attribute grammar systems [DJL88] is relevant to the design of the shape calculus. It may well turn out that the shape calculus amounts to a modular version of this analysis. Jay and others are currently developing a theory of "shapely types" that factors data structures into their shapes and their elements. For example, the same sequence of elements can be stored in a list, an array, or as the leaves of different kinds of trees; a shapely type describes the structure holding onto the sequence [Jay]. Size and access inference for data parallel programs [CBF91] is another line of work that strongly suggests a notion of process shapes. In his work on paralations [Sab88], Sabot described an intriguing notion of communication shape.

10.3.4 Synchronization

Synchrons are a powerful synchronization mechanism that may have many other uses beyond their role in synchronized lazy aggregates. For example, it might be possible to use them as the basis for a general temporal constraint solver. The time-based nature of synchrons suggests a *temporal calculus* in which actions are specified to happen before, after, or between synchrons. Early versions of the OPERA supported `before`, `after`, and `between` constructs on synchrons, but they were ultimately abandoned because their semantics was never satisfactory. Working out the details of a temporal calculus would be an interesting project.

Synchrons are currently implemented using reference counting garbage collection. Reference counting, which requires explicit deallocation of every object, has a high overhead compared to techniques (like stop and copy) that only examine live data. Is it possible to implement synchrons more efficiently?

Synchrons are an example of a growing number of data structures that interact with garbage collection. Other examples include T's populations [RAM83], MultiScheme's pairs and finalization objects [Mil87], and Hughes's `synch` construct [Hug83, Hug84]. This trend raises a number of questions: Are there other examples of these structures waiting to be uncovered? What is an appropriate set of primitives and means of combination for constructing new ones? What are efficient techniques for implementing these structures? What are the appropriate formalisms for describing their semantics?

10.3.5 Theoretical Directions

A great deal of work can be done in formalizing the concepts introduced informally in this document. In addition to the shape calculus mentioned above, EDGAR is fertile ground for some theoretical explorations. For example:

- *Confluence*: It would be nice to know under what conditions the model is confluent (i.e., has the property that all computations from the same initial graph end at the same final graph).
- *Sliver Correctness*: Many of the sliver implementations (especially scanners) are extremely complex. EDGAR provides a formal model for their execution; can it help to prove them correct?
- *Behavioral Equivalence*: I have claimed that sliver networks are operationally equivalent to monolithic programs "modulo management operations". Classical approaches to process equivalence (like CSP's trace equivalence [Hoa85]) appear too weak to handle the crucial notion of equivalent space behavior. EDGAR seems to be a natural starting point for a richer notion process equivalence that explicitly models space consumption. The hard part is defining what "modulo management operations" means in this context. Pomset models of concurrent computation [Pra86, HA87] may be helpful for defining an EDGAR-based notion of process equivalence.

10.3.6 Pedagogy

The computational models and tools developed for this work have important pedagogical applications. The EDGAR model is an extremely rich computational model in which a wide variety of programming issues can be studied: iteration, linear and tree recursion, tail calls, first-class procedures, procedure calling mechanisms, laziness, eagerness, side effects, environment structures, continuations, concurrency, synchronization, and nondeterminism. The fact that all of these issues can be explored through the graphical animation of the DYNAMATOR makes EDGAR a natural choice for illustrating the dimensions of these issues.

The notions of shape and lock step processing presented in this dissertation suggest new ways for teaching the structure of both modular and monolithic computations. For example, the difference between linear iterations and recursions can be introduced by means of sliver diagrams annotated with shapes. The fact that elements between a *down* process and an *up* process form an explicit stack helps to convey the essence of recursion. Shapes can aid in the discussion of program organization and transformation. And the fact that slivers are mix-and-match pieces emphasizes the principles of modularity.

Slivers and shapes not only open up new teaching avenues, but they are a potentially powerful language for helping programmers express processes to each other and to their computers. It remains to be seen whether their potential will be realized.