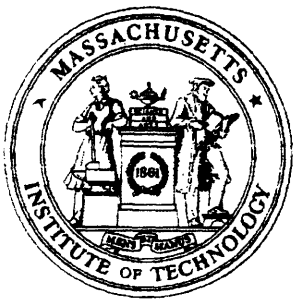


Signal Representation for Symbolic and Numerical Processing

RLE Technical Report No. 521

June 1986

Cory S. Myers

Research Laboratory of Electronics
Massachusetts Institute of Technology
Cambridge, MA 02139 USA

This work has been supported by the Advanced Research Projects Agency monitored by the ONR Under Contract N00014-81-K-0742, in part by the National Science Foundation under Grant ECS84-07285, in part by Sanders Associates Inc., and in part by an Amoco Foundation Fellowship.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Research Laboratory of Electronics, Massachusetts Institute of Technology	6b. OFFICE SYMBOL (If applicable)
6c. ADDRESS (City, State and ZIP Code): 77 Massachusetts Avenue, Cambridge, MA 02139	7a. NAME OF MONITORING ORGANIZATION: Office of Naval Research, Mathematics and Information Scien. Div.
	7b. ADDRESS (City, State and ZIP Code): 800 North Quincy Street, Arlington, Virginia 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Advanced Research Projects Agency	8b. OFFICE SYMBOL (If applicable)
	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00014-81-K-0742
8c. ADDRESS (City, State and ZIP Code): 1400 Wilson Boulevard, Arlington, Virginia 22217	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
		NR 049-506	

11. TITLE (Include Security Classification): Signal Representation for Symbolic and Numerical Processing

12. PERSONAL AUTHOR(S): Cory S. Myers

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	August 1986	239

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Traditional signal processing by computer relies on numerical methods. This thesis explores and develops the symbolic representation and manipulation of signals and demonstrates new models for the numerical representation and manipulation of signals. It is shown that combined numerical and symbolic representation provides advantages in the computer processing of signals. We describe how symbolic representation and manipulation methods may be incorporated in the numerical processing of signals and how these methods may be used to provide facilities for the design and analysis of signal processing systems.

For the numerical representation of signals a model, called the deferred array, that incorporates the functional form of a signal along with a memory and deferred evaluation, is developed. It is shown that the deferred array closely matches the way in which signals are described mathematically and

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Kyra M. Hall, RLE Contract Reports	(617)253-2569	

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

can be used to represent infinite duration signals and many signal processing operations. SPLICE, a software environment for the exploratory development of signal processing algorithms, incorporating our ideas on the numerical representation of signals, is described.

For the symbolic manipulation of signals three types of manipulations are identified as important to incorporate: manipulation of signal properties, rearrangement of signal processing expressions, and manipulation of signals of a continuous variable. Those aspects of signals and systems that must be present in a representation to allow for symbolic manipulation are given and a system for the symbolic manipulation of signals is described. Several examples of the symbolic manipulation of signals, including expression rearrangement, manipulation of continuous-frequency signals, signal property analysis, and cost analysis, are presented.

Signal Representation for Symbolic and Numerical Processing

by

Cory S. Myers

Submitted to the Department of Electrical Engineering
and Computer Science on August 13, 1986 in partial fulfillment
of the requirements for the Degree of Doctor of Philosophy
in Electrical Engineering and Computer Science

Abstract

Traditional signal processing by computer relies on numerical methods. This thesis explores and develops the symbolic representation and manipulation of signals and demonstrates new models for the numerical representation and manipulation of signals. It is shown that combined numerical and symbolic representation provides advantages in the computer processing of signals. We describe how symbolic representation and manipulation methods may be incorporated in the numerical processing of signals and how these methods may be used to provide facilities for the design and analysis of signal processing systems.

For the numerical representation of signals a model, called the *deferred array*, that incorporates the functional form of a signal along with a memory and deferred evaluation, is developed. It is shown that the deferred array closely matches the way in which signals are described mathematically and can be used to represent infinite duration signals and many signal processing operations. SPLICE, a software environment for the exploratory development of signal processing algorithms, incorporating our ideas on the numerical representation of signals, is described.

For the symbolic manipulation of signals three types of manipulations are identified as important to incorporate: manipulation of signal properties, rearrangement of signal processing expressions, and manipulation of signals of a continuous variable. Those aspects of signals and systems that must be present in a representation to allow for symbolic manipulation are given and a system for the symbolic manipulation of signals is described. Several examples of the symbolic manipulation of signals, including expression rearrangement, manipulation of continuous-frequency signals, signal property analysis, and cost analysis, are presented.

Thesis Supervisor: Alan V. Oppenheim
Title: Professor of Electrical Engineering

To Joanna

Acknowledgements

My sincere and deepest thanks go to Professor Alan Oppenheim for his guidance, encouragement, and support of this work. Working with Al has been both extremely enjoyable and has contributed greatly to my professional and personal growth.

I am also grateful to Professors Randy Davis and Victor Zue. Randy was consistently available for discussions, provided invaluable ideas to the development of this work, and diligently read drafts of this thesis. Victor also read drafts of this thesis and supplied important feedback.

Special thanks go to Webster Dove for his help and friendship. Webster started the work on SPLICE and generously let me join him in the effort. Webster has contributed much to this work, to my interest in computers, and to my sanity.

I thank all the members of the MIT Digital Signal Processing Group for their friendship and their technical interaction over the years. In particular, Evangelos Milios was an early proponent of symbolic manipulation of signals and worked with me on a preliminary software implementation. Michele Covell provided stimulating technical discussions and read and commented on many drafts of this thesis. Tae Joo and Jacek Jachner discussed with me many of the ideas presented here and gave important comments.

I gratefully acknowledge the generous financial contributions during my graduate career from the AMOCO Foundation and from Schlumberger-Doll Research. I also thank the Advanced Research Projects Agency and Sanders Associates Incorporated for sponsoring this research.

None of this would have been possible without the love, help, and encouragement of my parents. My deepest appreciation for a lifetime of support.

Finally, a special thanks to Joanna for her love, understanding, criticism, and patience. Without her none of this would matter.

Table of Contents

List of Figures

CHAPTER 1

Introduction

This thesis considers the representation and manipulation of signals for symbolic and numerical processing by computers.

The process of signal processing problem solving involves many steps and varies greatly from problem to problem. However, broadly speaking, two steps common in the problem solving process are the symbolic manipulation of signals and systems by people and the numerical processing of signals in computers. Symbolic manipulation of signals and systems is that portion of the problem solving process in which signal processing objects and descriptions (discrete signals, continuous signals, multiplication, convolution, etc.) are manipulated using knowledge about signal processing to determine useful information. Typical problems might be the design of a coding system, selection of an algorithm for convolution, or analysis of a system's frequency response. Symbolic manipulation requires the representation, usually in mathematical terms, of signals and systems and the manipulation and analysis of these representations.

Numerical processing of signals involves the representation of signals and execution of signal processing algorithms in computers. Typical problems are the implementation of a coding system, computation of the convolution of two signals, or determination, by simulation, of the response of a system to an input. It is at the numerical processing stage, and not the symbolic processing stage, that signal processing problems have traditionally been placed on the computer.

The goal of this thesis is the improved representation and manipulation of signals for both symbolic and numerical processing. On the numerical side, the goal is to provide a signal representation that is close to the way in which human signal processors think about signals and that facilitates the expression of algorithms. On the symbolic side, the goal is to show that the types of reasoning processes that people use in the design and analysis of signal processing algorithms can be captured and put to work in a computer program. Our goal here is not to discover new ways to do signal processing, but instead, to offload the burden from the human to the computer.

1.1. The Representation of Signals for Numerical Processing

The ease of representation of signals in computers for numerical signal processing is important in many signal processing programming environments. In research and development environments the dominant cost of signal processing is often program development time, hence good programming tools and representations become important. In production environments processing speed may be the dominant cost. However, good programming tools and representations are useful in the design and maintenance of a production environment. Also, with the ever increasing sophistication of modern compilers, good programming tools and representations using high level languages do not necessarily imply reduced performance.

Good programming practice dictates that the choice of a representation should be made with care because the representation we have available influences how we think about solving our problems. This representation should be as natural as possible and as clear as possible. It should allow us to express algorithms directly and should hide irrelevant details. Signal processing knowledge should be reflected directly in the representation and it should allow us to use powerful concepts from signal processing.

The representation of signals for numerical processing has been studied before [1, 2, 3, 4, 5, 6, 7, 8, 9] and modern software development techniques have been applied to the problem. However, no signal representation has been found that is generally usable and is as clear as the mathematics of signal processing.

1.2. Symbolic Manipulation of Signals

Numerical manipulation of signals involves the calculation of the values of signals. In contrast, symbolic manipulation of signals is the process of manipulating signal forms rather than signal values. With the development of specialized programming environments [10] and specialized programs for symbolic manipulation [11], the potential for a signal processing environment that combines powerful numerical signal processing capabilities along with symbolic manipulation can be realized. The transfer of the skill of symbolic manipulation for signal processing from human to computer has two potential benefits. First, offloading mechanizable tasks from the human signal processor frees the human for more important tasks. Secondly, computer systems can often be more thorough and meticulous in their analysis.

There are many potential applications of a signal processing system containing both numerical and symbolic manipulation techniques. For example, a person interested in filter design could use such a system to design an optimal filter (assuming infinite precision arithmetic), to explore different filter structures for their roundoff noise and dynamic range, and to pick an "optimal" structure that minimizes some user defined cost criteria, subject to performance constraints. In this problem symbolic reasoning would be used to determine the appropriate filter response from information about the bandwidth of the input signal and context in which the filter will be used. Numerical processing would be used in the initial filter design. Symbolic processing

would be used to determine analytically the noise performance and dynamic range of a particular filter structure. Symbolic processing and design heuristics would be used by the system to explore possible implementations to find the "optimal" one. The system might suggest different realizations of the filter according to the different ways the filter might be used. For example, if the user asked for the implementation that required the fewest multiplies, the system might, in general, suggest a frequency domain implementation. However it might suggest a time domain implementation when the filter is used in combination with downsampling.

Another application of a system that provides both numerical and symbolic manipulation of signals would be as a tool in the study of a complicated signal processing system, such as a short-time Fourier transform analysis/synthesis system. Symbolic manipulation can be used to verify that the analysis/synthesis system can be used to analyze and synthesize a restricted class of signals, e.g. bandlimited signals, without distortion. Many of the effects of noise in the coding of the analysis parameters could be determined analytically using symbolic manipulation of signals and the response of the system to non-bandlimited inputs could be approximated symbolically.

Specialized programs have been built for certain signal processing analysis problems, such as filter analysis [12] and other systems have been built for the manipulation of algebraic expressions [11]. However, no general purpose system has been built that explicitly represents signals and systems and that provides the symbolic manipulations common to signal processing.

1.3. Contributions of this Thesis

This thesis makes contributions to both the numerical and the symbolic representation and manipulation of signals. For the numerical side we provide an improved representational model and show its use. For the symbolic side we describe the representation of signals for symbolic manipulation and we present the first general purpose system for symbolic manipulation of signals.

1.3.1. Numerical Signal Representation and Manipulation

In the area of numerical signal representation and manipulation we develop a model of signals that closely matches the way in which signals are described mathematically and we show that this model can be the basis for a powerful computer representation of signals.

The model we develop for signals has two parts - an *outside view* and an *inside view*. Our outside view of signals follows the work of Kopec [8, 9] in taking a data abstraction view of signals as objects that are accessed in known ways. However, our view of signals differs from Kopec's because, while he viewed signals as finite extent objects that start at the origin, we view them as infinite in extent. We show that this view brings the program representation of signal processing operations closer to the mathematics of signal processing.

We develop our outside view of signals in a model called the *deferred array*. The deferred array incorporates a function, a memory, and deferred evaluation. It is shown to capture the idea that a discrete signal is a mapping from integer indices into signal values and to have the ability to represent both signals that do not start at the origin and infinite duration signals. By examining the deferred array, we identify those properties of signals that are important to capture in a numerical signal

representation. These properties are an explicit functional form for computing signal values, the non-zero extent of signals, immutability, deferred evaluation, and unique identity.

The inside view of signals specifies what programmers must do to build signals. We specify that signals must always be the output of systems and that the specification of the computation of signal values is the specification of how a system should compute the values for the signals it produces. We describe four models by which programmers define systems. These models are the point operator model, the array operator model, the state machine model, and the composition model. In the point operator model a system is described by a function that computes a single signal value at a time while in the array operator model a system is described by a function that computes many signal values simultaneously. In the state machine model a system is described by functions that compute the value of a signal for successive time indices. In the composition model a system is described by an interconnection of other systems. We argue that in a good representation for signal processing all these models of systems can be used while maintaining a common outside view of signals.

Our model of signals and their definition has been incorporated into a signal processing package for the Lisp Machine [10] called the Signal Processing Language and Interactive Computing Environment (SPLICE). This package incorporates our model of signals as infinite objects, the deferral of value calculation until required, and efficient numerical calculation. The signal processing package is integrated into the Lisp Machine environment and provides over 200 signal processing operations within a common framework.

1.3.2. Symbolic Signal Representation and Manipulation

In the area of symbolic signal representation and manipulation we discuss how to represent signals and systems for symbolic manipulation and we present a system for the symbolic manipulation of signals. We identify three classes of symbolic manipulations of signals - symbolic manipulation of signal properties, rearrangement of signal processing expressions, and manipulation of signals of a continuous variable - and discuss the use of these classes of symbolic manipulations in signal processing analysis.

We show the importance of explicit representation of signal type and history, signal properties, system properties, and computational costs in the symbolic manipulation of signals and we identify two extensions to our numerical signal model - signals of a continuous variable and abstract signal classes - that must be explicitly represented for the symbolic manipulation of signals.

The Extended Signal Processing Language and Interactive Computing Environment (E-SPLICE), a system for the symbolic manipulation of signals, is presented. The way in which E-SPLICE describes signal and system properties, abstract signal classes, continuous-frequency signals, and signal processing rules is described. Examples of the symbolic manipulation of signals are given, including expression simplification, generation of equivalent forms of an expression, use of continuous-frequency signals, generation of spectra by the parsing of signal descriptions, the manipulation of signal properties, the measurement of computational cost, and the selection of implementation. An example of symbolic analysis of a multi-rate network is given, showing both cost analysis and frequency analysis.

1.4. Survey of the Thesis

Chapter 2 defines, informally, the terms signal and system as they will be used in this thesis. We review those concepts from computer science that form the background for this thesis. We also briefly survey the current state of the art in signal processing software.

Chapter 3 discusses the representation of signals for numerical processing in computer programs. We first define what we mean by a signal and point out those aspects of our definition that are important for signal representation. We review previous signal representations, including arrays, streams, and signal objects. We then present the deferred array, the extended deferred array, our outside view of signals, and our inside view of signals.

Chapter 4 describes SPLICE and shows how both our outside view and our inside view are incorporated. We present several examples of the definition of systems using SPLICE and we discuss some implementation details.

Chapter 5 addresses the idea of symbolic manipulation for signal processing. We present an example of the analysis of a signal processing system by an experienced signal processor and find three classes of symbolic manipulations that are used. We discuss the requirements that symbolic manipulation of signals imposes on signal representation and we discuss extended signal objects.

Chapter 6 describes E-SPLICE and discusses how requirements for signal representation for symbolic manipulation are met. We present several examples of the symbolic manipulation of signals and show the power of our representation.

Chapter 7 summarizes the results of this thesis and discusses directions for future research.

CHAPTER 2

Background

This chapter describes those aspects of signal processing, computer science, and symbolic manipulation that form the background for this thesis. The first section contains an informal description of signals and systems as we will use these terms. The next section surveys some concepts from computer science that are important in our work. The third section examines current signal processing software and the final section describes some aspects of symbolic manipulation and signal processing.

2.1. Signals and Systems

Signals and systems are the fundamental concepts in signal processing. When we use the word signal we mean a function of one or more variables, often containing information about some physical process. Examples of signals are speech waveforms, television pictures, and seismic measurements.

When we use the word system we mean a function of zero or more inputs that produces a signal as its output. Examples of systems are signal generators, filters, multipliers (both of signals by scalars and of signals by signals), Fourier transformers, sampling rate converters, bandwidth compression systems, and parameter identification systems. We will assume that systems are deterministic, i.e. the output of a system is always the same for the same inputs. When a "system" has initial conditions that determine its output we will instead interpret the true system as taking an extra input, the initial conditions.

The activities of signal processing can be broadly characterized as being the evaluation, interpretation, and design of signals and systems. A typical evaluation problem would be determining how fast a combination of upsampling and convolution could be performed. A typical interpretation problem would be determination of the identity of an unknown signal or system from some measurements. A typical design problem would be the determination of a filter frequency response and implementation to remove some noise from a signal.

2.2. Computer Science Concepts

This section reviews some well known ideas from computer science and relates those ideas to issues in signal processing. No attempt is made to be rigorous or complete; instead, those topics that are relevant to our work are presented informally. The topics we touch on include the structuring of computer programs into procedural and data abstractions, generic operations, mutability, deferred evaluation, caching, and rule-based systems.

2.2.1. Abstraction in Programming

A theme in modern computer programming is abstraction as a tool for structured design [13]. Abstraction in computer programming is the separation of external behavior from internal implementation. Among the types of abstractions that have been identified as being useful in computer programming are procedural and data abstractions.

Procedural abstraction corresponds to the ideas of modules, subroutines, and functions. Procedures are abstractions because the user of a procedure generally is interested only in what a procedure does, not how it is done. Viewed as an abstrac-

tion, the procedure separates an abstract behavior from an implementation. Examples of procedural abstractions include logarithm functions, DFT subroutines, and linear equation solving packages.

Systems are a natural procedural abstraction mechanism for signal processing. For example, to build a frequency domain convolver a user would assemble two DFT routines, a multiply routine, and an IDFT routine, and would probably not be interested in the details of how any of these routines are implemented.

While systems and procedures are somewhat parallel, signal processing systems are more restrictive than procedures in two ways - the output of a system (as we use the term system) is always a signal and the output of a system is always the same for the same inputs. The first difference between systems and procedures, that systems always produce signals, merely restricts systems to being a subset of all procedures. The second difference merely is a reiteration of our statement that we define systems as having no hidden state. Initial conditions are instead taken to be another parameter input to the system for determination of the output signal. The notion that procedures that exhibit functional behavior are easier to use is well known in computer science [13, 14].

Data abstraction is another organizing principle in computer programs, orthogonal to procedural abstraction, and somewhat less familiar. Data abstraction refers to the idea that a user of a data object should be interested in how a data object behaves, not in how the data object is implemented. A simple data object is a floating point number. The user of a floating point number is not, in general, interested in how the floating point number is implemented, but instead is interested in how the floating point number interacts with other floating point numbers through the various arithmetic

operations. A more complicated data abstraction is that of a stack. The user of a stack need only be concerned that the operations of test for empty, push, and pop are defined for the stack, not how the stack is implemented.

In general, a data abstraction is a bundling of related pieces of information into an object and a set of operations, called *inquiry operations* [15], for manipulating objects of that data type. If objects of a particular data type can only be manipulated by operations defined for that data type, and not by operations on the underlying representation of the data object, then the behavior of the object is completely characterized by the behavior of the operations for its data type.

Designing programs using data abstractions is simpler because the external interface provided by objects of a specific data type is fixed. By providing a common interface, abstract data types make it easier to combine procedures written by different programmers. Also, abstract data types are more maintainable because the implementation of a type can change without having to change those procedures that use the type.

Signals are a natural unit of data abstraction for signal processing programs. Signals are only accessed in fixed ways, for example by examining the values of signals, and are considered as a unit when manipulated in signal processing programs.

Both procedural abstractions and data abstractions encourage well structured programming by allowing the programmer to solve the problem with those tools and objects that are natural to the problem domain. Procedures without side effects are more easily developed and maintained than those with side effects because the implementation of the procedure may be changed without having to change the uses of the procedure. Similarly, the use data abstractions that can be accessed only in fixed ways

eases development and maintenance because the implementation of the data abstraction may be changed without having to change the uses of the data abstraction.

2.2.2. Generic Operations

Generic operations are operations that perform the "same" operation on more than one data type. Examples of generic operations are the basic arithmetic operations of addition, subtraction, multiplication, and division. Many computer languages use the same operator to specify the addition of any two numbers, regardless of whether the numbers are integers, floating point numbers, or complex numbers. Use of generic operators is also called operator overloading because the "same" operator is being used to perform different functions. Operator overloading is a useful programming tool because it reduces the number of operator names that the programmer must remember and because it helps clarify the meaning of programs.

Generic operations may be implemented by either compile time or run time selection. Compile time selection involves the analysis of data types at compile time and the selection of the specific operation, e.g. integer add, floating point add, complex number add, etc., at compile time. Run time resolution of generic operations is also called data-directed programming and involves the selection of the appropriate operation, at run time, based on examination of the data type of the operands. Run time selection of generic operations requires that data objects explicitly contain their type information. When an object contains explicit representation of its data type the object is said to exhibit *manifest type*.

Run time resolution of operations is also related to object-oriented programming [16]. Object-oriented programming is a data abstraction method in which objects are instances of an abstract data type and the inquiry operations for an object are

performed by message passing. Passing a message to an object asks the object to perform the desired operation. The message passing paradigm is a type of generic operation because the sender of the message need not know how the message will be implemented for a particular data type. A display system that maintains a collection of objects of different types, for example, circles, squares, triangles, etc., might send a message to each object in the collection to display itself in a given window. The display code would be specific for each data type but the sender of the message need not know which display code will be executed for an individual object.

An important aspect to the use of generic operations is *inheritance*. The idea of inheritance is that some abstract data type can be considered as a specialization of some other more general data type, e.g. a stack of integers is a specialized type of stack, and the specialized data abstraction should perform all the operations of the more general abstraction. Thus, a stack of integers would have a push and a pop operation because it is a specialized type of stack. In addition, a stack of integers might have its own specialized operations, e.g. for determining the smallest integer on the stack.

A view of signals can be put forth in which signals are abstract objects and specialized signals are objects of a specialized data type [8]. For example, a sine wave can be considered as a specialized type of signal. In this model, the abstract data type for signal would define those operations basic to all signals, such as determining the value of a signal at an index, and specialized signal data types would perform appropriate specialized calculations. We will come back to this view of signals in later chapters.

2.2.3. Mutability

The idea that certain data objects are unchanging is called *immutability* [13]. A data object is immutable if no operation on that object can affect the object's external

behavior.[1] An array of numbers stored in a read-only memory is an example of an immutable object. No operation can change any of the values in the array. In contrast, an array of numbers stored in a typical computer program is not an immutable object because most computer languages allow the values of an array to be modified.

Signals are immutable objects because no operation that can be performed on a signal changes the values of the signal. Operations such as filtering or transforming produce new signals, they do not change the original signal. A signal may be examined or input to a system to create other signals but the signal is not changed by these operations.

2.2.4. Deferred Evaluation

Deferred evaluation, or demand-driven evaluation, is the idea that expressions are computed only as their values are needed. Specifically, the application of a function $f(x_1, x_2, \cdots, x_N)$ to some expressions $expr_1$, $expr_2$, $\cdots$, $expr_N$, $f(expr_1, expr_2, \cdots, expr_N)$, does not cause the immediate evaluation of the expressions. Instead, each expression, $expr_i$, is evaluated only if the value of its corresponding variable, x_i, is required in the body of the function f. As an example of deferred evaluation consider the function

$$f(x, y) = \text{if } x > 0 \text{ then } y \text{ else } 0 \qquad (2.1)$$

In a system with deferred evaluation the call $f(-1, g(z))$ never causes $g(z)$ to be evaluated because the value of the argument y of $f(x, y)$ is never needed.

In signal processing, deferred evaluation is the difference between defining signals and computing signal values. For example, defining a signal to be a 255 point

[1] This view of immutability does not preclude so called "benevolent" side effects that can improve the efficiency of using a data object.

Hamming window is not the same as computing all the values of the 255 point Hamming window.

2.2.5. Caching

Caching is the remembering of information for efficiency reasons. In a hardware memory system caching is the storing of recently accessed data in a fast memory in anticipation that recently accessed data will be accessed again in the near future. Caching can occur in software also. For example, in a computer language that uses deferred evaluation it might be inefficient to recompute the value of an expression each time the value is required. A better strategy is often to store the value of the expression the first time it is computed and to use the stored value rather than reevaluation when the value is again needed. Similarly, functions may store a table of inputs and outputs to avoid recomputation.

Related to the idea of caching is the notion that some operations on a data type may have "benevolent" side effects that improve efficiency. A "benevolent" side effect on a data object is something that changes the internal representation of that data object without changing the data object's observable properties. For example, a data abstraction for matrices may specify that the singular values of a matrix can be requested. The implementation of the matrix data abstraction may be such that the first time the singular values are requested they are computed and stored but later requests for the singular values simply return the stored values.

2.2.6. Rule-Based Systems

A rule-based system is a way of building a program in which the behavior of the program is specified by a discrete collection of pieces of knowledge, called rules [17].

A rule consists of some patterns and a conclusion. The patterns specify the conditions under which the conclusion is true. Rule-based systems provide a way of organizing programs in which the pieces of knowledge that are involved in solving a problem are made explicit, modular, and separate from any control structure. Rule-based systems have been built for medical diagnosis, computer configuration, signal interpretation, process control, etc.

An example of a rule from signal processing is:

> The convolution of two signals, x and h, is equal to the inverse Fourier transform of the product of the Fourier transforms of x and h.

In this rule the pattern is the convolution of any two signals. The pattern contains the variables x and h. These variables are matched to the two signals of a specific convolution. The conclusion of the rule is an assertion about the equality of two different expressions, $x * h$ and $FT^{-1}\{FT\{x\} \cdot FT\{h\}\}$. This rule may be used by a signal processing rule system to determine all the different ways in which to implement a convolution.

Rule-based systems are an active area of research. Issues of languages, e.g. LISP versus PROLOG, knowledge representations, knowledge acquisition, reasoning in the face of uncertainty, control, explanation, and learning are areas of concern.

2.3. Signal Processing Software

This section gives a brief description of signal processing software for general purpose applications. We briefly survey general purpose signal processing software, stream based software, array based software, abstract signal objects, and signal processing environments. Chapter 3 of this thesis and the work of Kopec [8] contain a more detailed analysis of some aspects of signal processing software.

2.3.1. General Purpose Signal Processing Software

Many programs for signal processing have been designed as general purpose tools or as demonstrations of signal processing theory. An outstanding example of general purpose signal processing software is the IEEE Programs for Digital Signal Processing [12]. This collection of programs contains many useful and efficient general purpose signal processing routines. However, these programs were not developed as an interconnectable set of signal processing tools. They do not use any standards for data passing between routines so interconnection of different routines, while not difficult, is not straightforward. For instance, to design an FIR filter using a standard design program and then to calculate the frequency response of the resulting filter is not simply a matter of calling a DFT routine with the results of a call to the design routine. Code must be changed, new variables declared, and data shuffled around to perform these two operations in sequence.

2.3.2. Stream Processing Languages

The earliest attempts to provide general purpose signal processing software and to standardize the interfaces used the block diagram model of signal processing [1, 2, 3, 4]. Block diagram models specify systems by interconnecting simple blocks, such as delay, scale, and sum blocks. The blocks are interconnected with "stream objects", first-in-first-out (FIFO) queues in which the output from one block loads the stream and the input to another block empties the stream. A block's behavior is specified by a pair of state and output functions. Block diagram languages provide an interface standard for signals because the output from any block can be connected as the input to any other block. Block diagram languages have been used successfully in signal processing programs since the early 1960's.

2.3.3. Array Processing Languages

We will use the term array processing languages to refer to the broad class of signal processing systems that model signals as arrays. Array processing languages are characterized by a set of computational functions that perform signal processing operations using data arrays to represent the signal. A typical operation in an array processing language would be to multiply two arrays. Many signal processing packages have been written using arrays of values as the standard interface between modules, including the Interactive Laboratory System (ILS) [18] and the Interactive Signal Processor (ISP) [19].

2.3.4. Abstract Signal Objects

Both block diagram languages and array processing languages provide models of signals. In block diagram languages the signal model is a stream and in array processing languages the signal model is an array. However, the primary organizational principle in both block diagram languages and array processing languages is the procedural abstraction of signal processing operations. The use of data abstraction and the use of more powerful data structures to represent signals in signal processing programs has also been suggested. Gethöffer's signal processing language SIPROL [5, 6, 7] and Kopec's Signal Processing Language (SPL) [8] and Signal Representation Language (SRL) [9] are examples of the use of data abstraction in signal processing software.

SIPROL was an attempt to use data abstraction in signal processing software by the specification of specialized signal processing structures. For example, the SIPROL data type **fir_system** was an abstract data type for FIR filters. SIPROL did not define an abstract data type for signals in general nor did it specify a standard interface to all signals.

Kopec's signal processing languages SRL and SPL were attempts to provide a general purpose data abstraction for signals. Kopec proposed a model of signals and developed languages in which signals could easily be defined and manipulated. We will consider Kopec's model of signals in more detail in Chapter 3.

2.3.5. Signal Processing Environments

The notion of a processing environment is that of a collection of programs and tools designed to interface with each other, providing powerful capabilities and easing the development of software. Signal processing environments have been designed for both general purpose usage and special purpose applications. ILS and ISP represent one type of signal processing environment - a set of signal processing programs and graphics routines. Another example of a general purpose signal processing environment is MITSYN [3]. MITSYN provides basic signal processing operations, including both stream based processing and array based processing, graphics, and extensibility in a single package.

Specialized signal processing environments include many image processing systems and speech processing environments. An example of a powerful specialized environment is the speech processing environment of SPIRE [20, 21, 22]. SPIRE contains both general purpose signal processing tools and specialized speech processing tools. It was designed for ease of use of the built-in operators, not extensibility.

2.4. Symbolic Manipulation for Signal Processing

Symbolic manipulation is the manipulation of symbolic, rather than numerical, data. Particularly, we will be interested in those systems for symbolic manipulation that work with algebraic expressions. One of the oldest and largest of these systems is

MACSYMA [11]. MACSYMA can symbolically differentiate or integrate expressions, factor polynomials, solve systems of equations, or compute certain transforms. As an example, MACSYMA can differentiate the expression $X^2Y^2+2X^2Y-3X^2+Y$ with respect to X to get $2XY^2+4XY-6X$ and it can factor the result as $2X(Y-1)(Y+3)$.

MACSYMA understands a large amount of mathematics, with particular expertise in the areas of basic arithmetic functions, differentiation, integration, and equation solving. Although the facilities of MACSYMA can be used to analyze signal processing systems, perform transforms, and do similar signal processing functions, MACSYMA has no special facilities for signal processing and is lacking in some standard signal processing operations.

Another aspect of symbolic manipulation that we will use in this work is the rearrangement and transformation of computations. The rearrangement and transformation of a computation is one way of viewing the operation of a compiler. Signal processing compilers for block diagram languages have been in use since the advent of block diagram languages. These compilers are used to determine the firing order of the prewritten routines for the blocks in a block diagram layout. Other signal processing software is compiled using the standard compiler for the language in which the software is written. In neither case does the compiler use any signal processing knowledge to rearrange or optimize the computation.

A different type of symbolic manipulation for signal processing is the manipulation of high level signal processing descriptions to determine a low level implementation. Bentz [23] proposes a system in which the user specifies a signal processing system using a high level block diagram editor. The system automatically collects the appropriate block descriptions, ensures that the chosen blocks are compatible, chooses

any missing design parameters, and produces low level code.

CHAPTER 3

Signal Representations for Numerical Processing

This chapter discusses the representation of signals for numerical processing in computer programs. We define what we mean by a signal and discuss what aspects of this definition are important to capture. We then review previous representations of signals including arrays, streams, and abstract signal objects, with special emphasis on the work of Kopec [8, 24, 9]. The concept of the deferred array, a simple model for describing signals in computer programs, is then presented. The deferred array is shown to provide a powerful model for thinking about signal representation. Limitations of the deferred array are discussed and extensions to the deferred array model are presented.

Drawing from our definition of a signal, previous signal representations, and our description of the deferred array, we summarize how a signal should appear to the user of signal in a computer program. We then examine how signals should appear to the programmer of signals. It is shown that the deferred array and the extended deferred array model one type of signal processing programming. Other types of signal processing programming are discussed and models for these types of programming are described.

The material presented in this chapter is intended primarily to be conceptual in nature. The result of this chapter is specification of those properties a good signal representation should possess. Chapter 4 of this thesis contains a description of an implementation of the ideas presented here.

This chapter contains two contributions. One is the presentation of the deferred array and the extended deferred array as models for signal representation. The deferred array is a model of infinite duration discrete-time signals containing an explicit representation of the functional form of the signal, a memory, and using deferred evaluation. The extended deferred array model incorporates multiple functions and explicit representation of the domains of functions into the deferred array.

The other contribution of this chapter is the formulation of a two sided view of programming for signal processing. This formulation splits the problem of programming for signal processing into an *outside view* and an *inside view*. The outside view describes a data abstraction that provides a contract on the external behavior of a signal. In this view a signal is infinite in extent, allowing access to its value at any index and its non-zero extent, immutable, and uniquely identified by the system that created it. We show that this model of signals is closer to the mathematics of signal processing than previous models and does not introduce semantic difficulties found in other models.

The inside view of signals specifies what operations a programmer is required to supply to build a signal. We recognize the importance of simply expressing signal processing algorithms while being able to build signal objects that meet the data abstraction requirements. We find that the different specifications of signals common in signal processing contain structure and we identify four models of signal processing computation - the point operator model, the array operator model, the state machine model, and the composition model. We discuss how each model specifies signal processing algorithms while maintaining a consistent outside view of signals.

3.1. What is a Signal?

To properly discuss the representation of signals for numerical processing we need to define what we mean by a signal. In this section we define signals and discuss the important aspects of our definition. Our definition of signals is similar to the definition of signals given by Kopec [24], but contains some important differences. We will discuss these differences in section 3.2.3.

Signals are functions mapping some set of indices into some range of values. A one-dimensional discrete-time signal, $x[n]$, is a function mapping the integers into the range of x, i.e.

$$x[n] = function(n) \tag{3.1a}$$

with

$$function : \{\cdots,-2,-1,0,1,2,\cdots\} \rightarrow Range(function) \tag{3.1b}$$

The range of x is most often the real numbers or the complex numbers, but may be other sets, such as voiced/unvoiced decisions, phonetic labels, etc. A one-dimensional continuous-time signal is a mapping from the real numbers into a range of values.[1] Multi-dimensional signals are defined similarly to one-dimensional signals. An example of a discrete-time signal is a cosine with period L

$$x[n] = \cos(\frac{2\pi n}{L}) \tag{3.2}$$

In the rest of this chapter we state our observations about signal representation in terms of one-dimensional discrete-time signals. Most of our comments naturally extend to multi-dimensional discrete-time signals, some extend to continuous-time signals. will come back to continuous-time signals in Chapter 5.

---

[1] We will use the term discrete-time signal for any signal defined with integer indices, regardless of whether the indices are being used as time, space, or frequency indices. Similarly, we will use the term continuous-time signal for any signal defined with real indices.

Three important aspects of our definition of signals are relevant to the representation of signals for numerical processing. First, discrete-time signals can be evaluated at any index n from $-\infty$ to ∞. This is the standard mathematical usage for signals and can be contrasted to the idea of finite length vectors in linear algebra, which are indexed only from some minimum index (typically 0 or 1) to some maximum index. The idea that discrete-time signals can be evaluated at any integer index is useful in discussing both finite duration signals that do not start at the origin, such as correlation functions, and infinite duration signals, such as the cosine of equation (3.2), unit steps, and the impulse response of IIR filters.

The second important aspect of signals for representation is that signals are immutable objects [13]. That is, the value of a signal for a particular index is always the same regardless of when or how often the value of the signal is requested or how the signal is used.

The third important aspect of our definition of signals is that all signals can be considered as the output of systems. For example, the definition of a cosine signal, in (3.2), can be considered as the definition for a system that takes as input a value for the parameter L and produces a cosine signal as output, as illustrated in Fig. 3.1. Systems may have no input parameters, such as the generator of an impulse or a unit

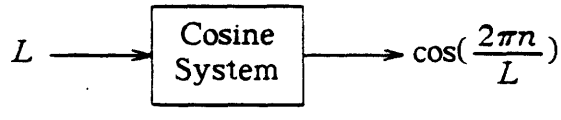


Fig. 3.1. System for producing cosine signals.

step, numerical input parameters, such as the cosine signal generator, or signals as input parameters, such as a system for convolving two signals. A signal is uniquely determined by the combination of system and parameters that generated it, i.e. the same system and the same input parameters always produce the same output signal.

3.2. Previous Signal Representations

In this section we briefly review previous signal representations, examining array based representations and stream based representations. Our work on the representation of signals for numerical processing has been greatly influenced by the work of Kopec and we review Kopec's model of an abstract signal object [8, 24, 9]. A more complete discussion of the representation of signals by arrays, streams, and abstract objects can be found in Kopec's PhD thesis [8].

3.2.1. Array Based Signal Representations

An array based signal representation refers to the use of arrays to store and manipulate signal values. Many signal processing systems are programmed using arrays. Notable examples are the Programs for Digital Signal Processing [12] and the Interactive Laboratory System (ILS) [18].

In their simplest form, arrays are indexed storage locations that can be read from or written to. Languages differ in the legal values of indices for arrays (from 0 below N, from 1 to N, from L below U, etc.) and in the layout of multi-dimensional arrays in memory. Languages also differ in bounds checking, allocation of array storage, and in facilities for determining the size, element type, and valid indices of an array.

Arrays and array based operators are used to represent discrete-time signals and systems. Signals are created by storing values in an array and are examined by read-

ing values from the array. Systems are array operators that transform arrays or create new arrays. A typical array operator, ARRAY-ADD, is given by[2]

```
(DEFINE (ARRAY-ADD X1 X2 Y)
  "Add the arrays X1 and X2 and put the result in array Y"
  (LOOP FOR I FROM 0 BELOW (ARRAY-LENGTH Y)
        DOING
    (ASET (+ (AREF X1 I) (AREF X2 I)) Y I)))
```

ARRAY-ADD takes as input three arrays, X1, X2 and Y, and fills the array Y with the element by element sum of X1 and X2. This definition assumes that arrays are indexed from 0 and that X1, X2, and Y are all the same size.

Arrays have many problems as a signal representation. Arrays are finite in extent and can only be used to model discrete-time signals. Arrays are mutable objects. While the mutability of arrays is often used to advantage in programming, e.g. overwriting a signal with its transform when only the transform is useful to keep, it can lead to many programming problems. Read-only arrays would not solve this problem either because the array must be filled with values to be used as a signal. The real problem is that arrays provide no separation between those processes that can read from the array and those processes that can write to the array.

A final problem with arrays as a signal representation is that they are restricted to finite duration signals that start at the origin. For signals that do not start at the origin, such as the autocorrelation of a signal or a centered window, the programmer must keep track of the offset from the origin and first element of the array. Arrays cannot be used to represent those signals that are infinite in extent, such as a periodic signal or the impulse response of an IIR filter. Also, because it is an error to access an array outside its bounds, arrays introduce a semantic difficulty in the definition of

[2] Examples in this chapter are given in an extended version of SCHEME [25], a dialect of LISP. The syntax of SCHEME and meaning of the operators used in our examples is given in Appendix A.

some signal processing operations. For example, ARRAY-ADD was written assuming X1, X2, and Y were all the same size. If we want to add signals of different sizes we must either pad the shorter signal out to the length of the longer or modify ARRAY-ADD to be able to work with arrays of different sizes. The first solution, of padding, is poor because it introduces an operation that is not present in the mathematical definition of adding two signals and the second solution, of changing ARRAY-ADD, imposes extra work on the programmer.[3]

Despite their problems, arrays are used successfully for signal representation. Arrays are a natural representation for finite extent discrete-time signals. Arrays provide easy access to signal values and low-level instructions are often available for array manipulation. Libraries of array operators are generally available and arrays are commonly used as the interface to high speed array processors.

3.2.2. Stream Based Signal Representations

A stream is a data structure that provides a sequence of values.[4] The sequential reading or writing of characters in a program is often formalized by providing a stream data structure. Many signal processing systems have been designed using the idea of streams to carry signals between blocks in a block diagram programming language, including such block diagram languages as Rader's language PATSI [2] and Johnson's signal processing tools [4].

Streams are data structures with two ends. One end is a source of values, the other end is a consumer of values. The source places values into the stream and the

---

[3] A third solution, of changing arrays to have them return zero when accessed outside, is a change in representation, i.e. the result would not be an array in the common usage of the term.

[4] We refer to streams in the sense of FIFO queues. Users of SCHEME will find the SCHEME concept of streams more akin to our concept of deferred arrays.

consumer reads values from the stream in the same order that they were placed on the stream. Reading from the stream has the side effect of advancing the stream, i.e. consecutive reads receive different values.

A typical stream based signal representation function is

```
(DEFINE (STREAM-ADD X1 X2 Y)
  "Add the input elements from streams X1 and X2 and place on Y"
  (PUT-STREAM (+ (GET-STREAM X1) (GET-STREAM X2)) Y))
```

STREAM-ADD takes as parameters two input streams, X1 and X2, and an output stream Y. The function STREAM-ADD reads an input value from each of the input streams, using the function GET-STREAM, and places the sum on the output stream using the function PUT-STREAM. Note that this function only adds one new value to the output stream. It does not fill the output stream with the sum of all the values of the two input streams.

Streams have been used in signal processing software in block diagram languages to carry signals from one block to another. A block in a block diagram language is a structure with stream inputs and stream outputs and with a function for computing the values of the stream output. The function STREAM-ADD is an example of a function that might be used in a block diagram language. Block diagram languages provide for the definition of new blocks and for interconnection of blocks.

An important division in block diagram languages is between those that provide a new output from each block at regular intervals and those that do not. Block diagram languages that provide outputs at regular intervals are based on a model in which each block gets one new input on each of its input streams and produces one new output on each of its output streams [2]. Block diagram languages of this type follow a model similar to those used in simulating finite difference equations, or, more generally, state

space systems of the form

$$\begin{aligned} \underline{s}[n+1] &= f\,(\underline{s}[n], \underline{x}[n]) \\ \underline{y}[n] &= g\,(\underline{s}[n], \underline{x}[n]) \\ \underline{s}[0] &= \underline{s}_0 \end{aligned} \tag{3.3}$$

Here $\underline{x}[n]$ is the vector of stream inputs at time step n, $\underline{s}[n]$ is the state of the block at time n, $\underline{s}_0$ is the initial state, $\underline{y}[n]$ is the vector of stream outputs at time step n, f is the state update function, and g is the output function. Rader referred to block diagram languages that implemented equations of this form as next-state simulation [2].

Block diagram languages that do not force blocks to produce outputs at regular intervals are based on a model of cooperating processes [4]. In this model each block is a process that reads data from its input streams and writes data to its output streams. The process is not restricted to reading or writing only a single value from its streams but can read or write as many as needed. Values must still be read in the order they are produced, however.

Streams also have problems as signal representations. Streams are a natural representation only for one-dimensional discrete-time signals. They may be finite or infinite in duration but always have an implied time origin, thus, cannot easily be used to represent signals that have values for both positive and negative time indices. It is not easy, or may not be possible, to determine the duration of a stream without reading all its values, clearly a problem for infinite duration streams. Streams cannot be used easily in algorithms that require random access to signal values.

Despite their problems, streams are used successfully as signal representations. Streams are a natural representation for simulation systems. The sequential nature of sample computation in streams, when applicable, has the advantage of eliminating the

computation devoted to index calculation.

3.2.3. Kopec's Abstract Signal Objects

Our work on signal representation draws heavily on the work of Kopec. In his PhD thesis he proposed the idea that signals should be represented in computer programs as abstract data objects [8, 13]. Specifically, all signals used in a program should be instances of some abstract signal type that specifies those operations that are legal to perform on signal objects. Thus, all signals in a program will have a standard interface by which the programmer can extract information.

In his work on signal representation Kopec considered functions of the form

$$f : [0, N_1) \times \cdots \times [0, N_m) \rightarrow Range(f) \tag{3.4}$$

where $[0,N_i)$ is the set of integers $\{0,1, \cdots, N_i - 1\}$ and $X \times Y$ is the Cartesian product of the sets X and Y, to be m-dimensional discrete-time signals. Typically, the range of f is the set of real numbers, although signals whose values are other objects, such as signals whose values are themselves signals, were also considered.

From the definition of signals in (3.4) three observations were made. Two of these observations have already been mentioned in section 3.1 in our discussion of what a signal is. First, because signals are specified by functions, signals are immutable objects. Secondly, because signals are often specified with functions containing free parameters, e.g. L is a free parameter in (3.2), signals should be organized into signal classes and individual signals should be formed by binding the free parameters to specific values. This was called the closure model. Our observation that signals are the output of systems corresponds to the closure model in that the parameters of a system are the free parameters of the functional form.

The third observation is that signals have observable properties: the signal's domain and its values. The user of a signal should be able to find out the domain of the signal and its value for any index in the domain. It is important that an abstract data type be used to represent signals since, the principles of good programming dictate that a computer program should reflect the problem and its solution, not the details of implementation. Making signals into a data type in their own right is exactly in keeping with this philosophy.

To illustrate his ideas on signal representation, Kopec developed the Signal Processing Language (SPL) [8], a CLU based signal representation language, and the Signal Representation Language (SRL) [9], a LISP based signal representation language. To build a signal in either language the programmer specifies the domain of the signal and either defines a function for computing the value of the signal at any index in its domain or defines a function for computing all the values of the signal over its entire domain simultaneously. In either language it is an error to access a signal outside its domain.

In SRL, a signal class is defined by the operator DEFSIGTYPE. For example, the signal class SUM-SIGNAL is given by

```
(DEFSIGTYPE SUM-SIGNAL
  :A-KIND-OF BASIC-SIGNAL                ; A SUM-SIGNAL is a BASIC-SIGNAL
  :PARAMETERS (X1 X2)                    ; Free parameters are X1 and X2
  :FINDER SIGNAL-SUM                     ; Use SIGNAL-SUM to build a SUM-SIGNAL
  :INIT                                  ; Set output size to shorter of inputs
    (SETQ-MY DIMENSIONS
             (MIN (SIGNAL-DIMENSIONS X1)
                  (SIGNAL-DIMENSIONS X2)))
  :FETCH ((N)                            ; Compute sample at index N
          (+ (SIGNAL-FETCH X1 N)
             (SIGNAL-FETCH X2 N))))
```

This specifies that a SUM-SIGNAL is a kind of BASIC-SIGNAL, the class has two free parameters, X1 and X2, and an particular SUM-SIGNAL can be found by applying the function SIGNAL-SUM to two signals. The domain of the resulting signal is the

shorter of the domains of X1 and X2 and the resulting signal is computed at any point in its domain by summing the corresponding points in X1 and X2. Signals may be access with the functions SIGNAL-DIMENSIONS, for finding out the domain of a signal, and SIGNAL-FETCH, for getting the value of a signal at any index in its domain.

This representation of signals overcomes some of the limitations of arrays and streams. By restricting the methods by which signals can be accessed signals are made immutable. Signals can be accessed for any index in their domain However, because the model of signals is finite duration, starting at the origin, with it being an error to access outside the domain, this representation has some of the same problems as arrays, i.e. it is not a natural representation for all signals and it introduces some semantic difficulties not present in the mathematics of signal processing. For example, the definition of SUM-SIGNAL says that the domain of the sum of two signals is equal to the shorter of the domains of the two input signals. The more natural definition is that the sum is non-zero for the longer of the two signals and that accessing the shorter signal outside its domain just returns zero.

3.3. A Simple Signal Representation - The Deferred Array

In this section we describe a simple signal representation in terms of an object called the *deferred array*. A deferred array is related to the MACSYMA idea of an array with an associated function [11] and appears to the user as an infinite array with indices extending from $-\infty$ to ∞. The deferred array can be accessed at any integer index and is an immutable object.

A deferred array contains a memory and a function. The function specifies a mapping from indices to values. For example, the deferred array whose value at an index was equal to twice the index would contain a function that mapped the index,

INDEX, into twice the index, (* 2 INDEX), as illustrated in Fig. 3.2. The function portion of a deferred array models the idea that signals are defined by functions mapping the index into values. To be properly defined, the function in a deferred array must always return the same value for the same index regardless of how often it is called.

The memory of a deferred array provides a transparent caching mechanism by recording the values of the function as they are computed. The memory is not essential to the model of signals. However, it models the idea that a signal representation should provide fast access to values. We will return to the issue of caching and speed of access in section 3.5.

We will not specify how the memory of a deferred array is implemented. The memory could be implemented using an array, an association list, a hash table or a combination of these in a single or multi-level structure. In Chapter 4, in our discussion of the Signal Processing Language and Interactive Computing Environment (SPLICE), we will describe a particular memory management scheme.

To the user, a deferred array appears as an infinite extent array that can be accessed at any index. When the user accesses the deferred array at an index the

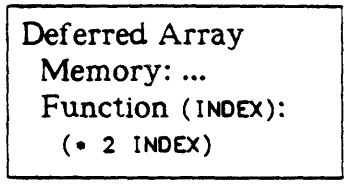


Fig. 3.2. Deferred array.

memory of the deferred array is checked to see if the value at that index has already been computed and, if so, that value is returned. Otherwise, the function is applied to the index, the value returned is stored in the memory, and is returned as the value of the deferred array at that index. At any time the deferred array may be modeled as an infinite array some of whose elements are filled and some of which are empty. Fig. 3.3 illustrates, conceptually, how a deferred array appears. Some of the slots of the deferred array are filled with values, the rest have a pointer to the function for computing the value.

The deferred array is named such because it looks like an array, providing random access over a set of integers, but the values of the array are not computed when the array is created. Instead, the computation of the values is deferred and values are computed only as they are needed. The delaying of computation until it is required is often called deferred evaluation and, in signal processing, is the idea that the definition

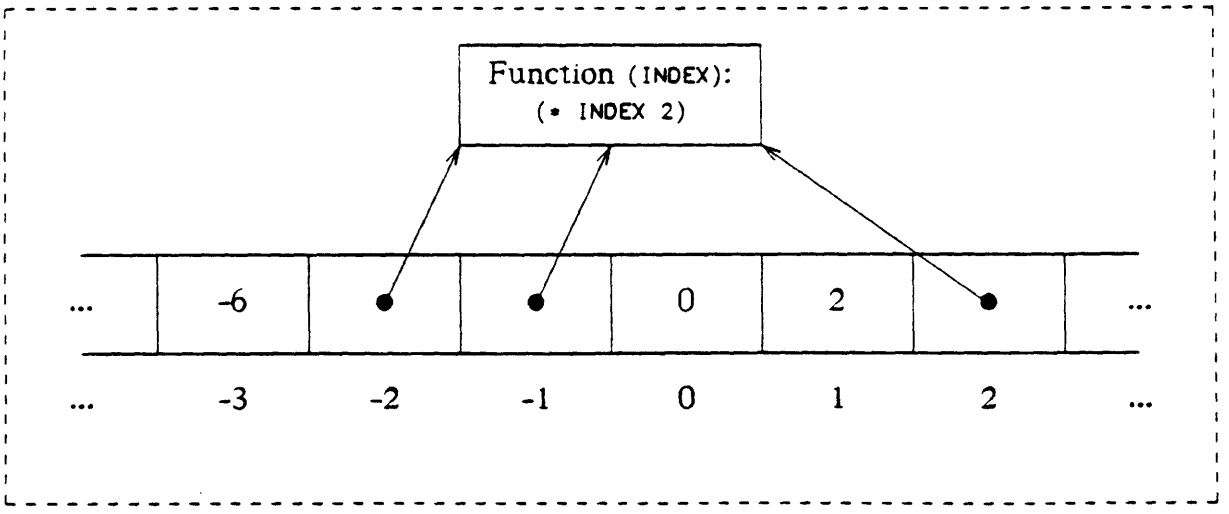


Fig. 3.3. Values, indices, and function of a deferred array.

of a function to compute the values of a signal is distinct from the computation of those values. Deferred evaluation is essential for the modeling of infinite duration signals. Stream based signal representations use deferred evaluation, i.e. all the values of the signal are not computed when the stream is created. Array based signal representations do not use deferred evaluation.

Although we have not formally defined a deferred array, our description of a deferred array is of an object whose only observable properties are its values at integer indices. Since there are no operations that modify a deferred array it is an immutable object.

3.3.1. Manipulating Deferred Arrays

We will define deferred arrays by a constructor, MAKE-DEFERRED-ARRAY, and a fetch operator, FETCH. Two different implementations for these functions are given in Appendix B. MAKE-DEFERRED-ARRAY takes as input a function of one argument, the index, that returns the value of the deferred array at that index. FETCH takes as arguments a deferred array and an index and determines the value of the deferred array at that index. For example, to define the signal

$$x[n] = \cos(\frac{2\pi n}{10}) \tag{3.5}$$

we can define the function COS-FCN as

```
(DEFINE (COS-FCN N)
  (COS (/ (* 2 PI N) 10)))
```

and we can define the signal $x[n]$ by

```
(DEFINE X (MAKE-DEFERRED-ARRAY COS-FCN))
```

Now (FETCH X *index*) returns the value of the signal $x[n]$ at n = *index*.

Fig. 3.4 shows an example of the definition and use of $x[n]$ along with some extra printing information to show the behavior of the memory. In this figure the token "FORM:" is the prompt for entering a form and the token "——>" denotes the returned value. Fig. 3.4 shows COS-FCN being defined with an extra print statement to inform the user when the function is being invoked. $x[n]$ is built from COS-FCN and fetched at index 0. Fetching $x[n]$ at 0 causes COS-FCN to be run and the value at 0 to be computed. Next $x[n]$ is fetched at -5. Again COS-FCN is run and the value is computed. Finally, $x[n]$ is fetched at 0 again. This time COS-FCN is not run. Instead the value of $x[n]$ is taken from the memory.

```
FORM: (DEFINE (COS-FCN N)                ; Define COS-FCN with printing
        (PRINT (LIST 'COMPUTING 'AT 'INDEX N))
        (COS (/ (* 2 PI N) 10)))

——> F


FORM: (DEFINE X                          ; Define a deferred array
        (MAKE-DEFERRED-ARRAY COS-FCN))

——> X


FORM: (FETCH X 0)                ; Fetch the sample at 0

(COMPUTING AT INDEX 0)           ; Print that the sample is being computed
——> 1.0                          ; Value at 0 is 1.0


FORM: (FETCH X -5)               ; Fetch the sample at -5

(COMPUTING AT INDEX -5)          ; Print that the sample is being computed
——> -1.0


FORM: (FETCH X 0)

——> 1.0                          ; No printing - value taken from memory
```

Fig. 3.4. Use of a deferred array.

3.3.2. Signals and Systems

In a way similar to our definition of the signal $x[n]$ in (3.5), we can specify the system for generating cosine signals

$$\cos(\frac{2\pi n}{L}) \tag{3.6}$$

as

```
(DEFINE (COS-SIGNAL L)
  (DEFINE (FCN N)
    (COS (/ (* 2 PI N) L)))
  (MAKE-DEFERRED-ARRAY FCN))
```

The system COS-SIGNAL takes as input the period of the desired cosine, L, and produces as output a signal. COS-SIGNAL works by defining an internal function, FCN, that is able to compute the values of a cosine signal once L is given, and by building a deferred array from this function. $x[n]$, defined in (3.5), can now be given by (COS-SIGNAL 10). The resulting deferred array structure is illustrated in Fig. 3.5, in which we make explicit the value of the parameter L is 10 for this signal.

A useful interpretation of the relationship between a system and a signal produced by the system is illustrated in Fig. 3.6. Here, the signal (COS-SIGNAL 10) has

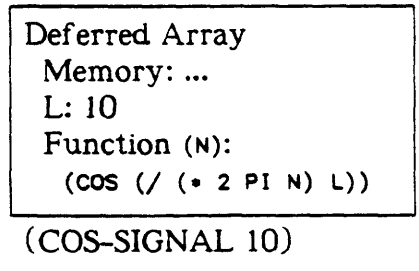


Fig. 3.5. Deferred array for (COS-SIGNAL 10).

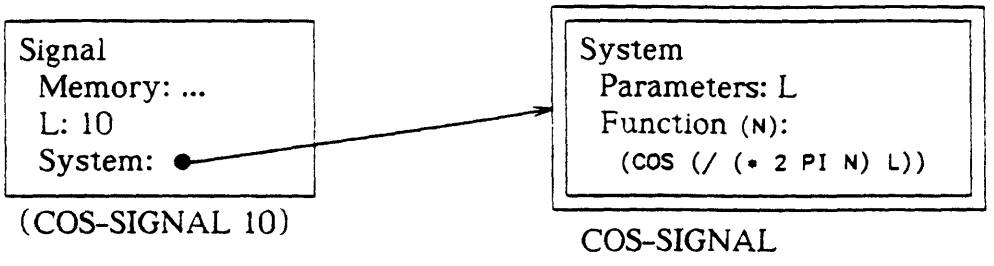


Fig. 3.6. The signal (COS-SIGNAL 10) and the system COS-SIGNAL.

been separated into a signal portion, containing a memory and the value for the parameter L, and a system portion, containing a list of parameters for the system and the function for computing signal values. To run the function associated with a signal it is necessary to look to the system that generated the signal to find the function. The function is run in an environment set up by the signal, i.e. with the values of the system parameters bound to the particular values of the signal.

It is well known that systems provide a way of organizing signal processing programs. The user of the system COS-SIGNAL need not be aware that COS-SIGNAL uses a deferred array. The user need only be concerned that COS-SIGNAL produces an object that can be fetched from. Also, systems allow the sharing of common functions among related signals. This is illustrated in Fig. 3.7, in which we show the signals (COS-SIGNAL 10) and (COS-SIGNAL 125). They are distinguished by different values of the parameter L and are related by pointing to the same system. Each different signal produced by the system COS-SIGNAL will share the body of FCN but will have a different value for the free parameter L.

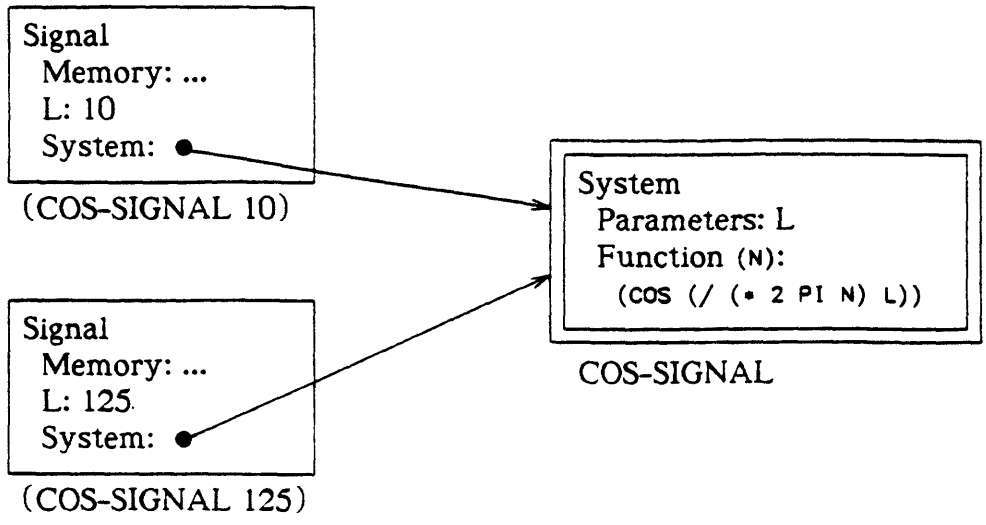


Fig. 3.7. The signals (COS-SIGNAL 10) and (COS-SIGNAL 125).

In the implementation of deferred arrays in SCHEME presented in appendix B the sharing of functions is implemented with lexical closures [25]. Two techniques for implementing this sharing include extending functions to take extra, often implied, parameters and message passing. In the rest of this chapter we will assume that systems using deferred arrays implement some form of function sharing, although we will not be explicit about it. We will also discuss exclusively the building of systems rather than the building of individual signals.

3.3.3. Applicability of the Deferred Array

The deferred array can be used to model those signal processing operations that depend on parameters and on the value of signals at their indices. It can not be used to model those signal processing operations that depend on other information about signals. In this section we will expand on those operations that are possible with the

deferred array and in the next section we will describe some of the problems with the deferred array.

We have already illustrated the use of the deferred array as a signal generator, as with the system COS-SIGNAL. Any signal whose value depends only on the index and on some parameters, such as Hamming windows, rectangular windows, etc., can be defined with the deferred array.

Using deferred arrays we can define some systems for combining signals. To multiply two signals we may use SIGNAL-MULTIPLY, defined as

```
(DEFINE (SIGNAL-MULTIPLY SIGNAL-1 SIGNAL-2)
  (DEFINE (FCN N)
    (* (FETCH SIGNAL-1 N) (FETCH SIGNAL-2 N)))
  (MAKE-DEFERRED-ARRAY FCN))
```

The product of two cosines

$$\cos(\frac{2\pi n}{10})\cos(\frac{2\pi n}{125}) \tag{3.7}$$

is now given by (SIGNAL-MULTIPLY (COS-SIGNAL 10) (COS-SIGNAL 125)).

It is instructive to examine the structure that results from the invocation of (SIGNAL-MULTIPLY (COS-SIGNAL 10) (COS-SIGNAL 125)). Fig. 3.8 illustrates the structure that is built for (SIGNAL-MULTIPLY (COS-SIGNAL 10) (COS-SIGNAL 125)). The box on the left represents the deferred array that is built by SIGNAL-MULTIPLY. The memory for this deferred array is not illustrated and we have not separated out the signal and system portions. This deferred array has a function, FCN, of an index, N. The function uses the free parameters SIGNAL-1 and SIGNAL-2 and has a body of code. SIGNAL-1 and SIGNAL-2 point to (COS-SIGNAL 10) and (COS-SIGNAL 125) respectively.

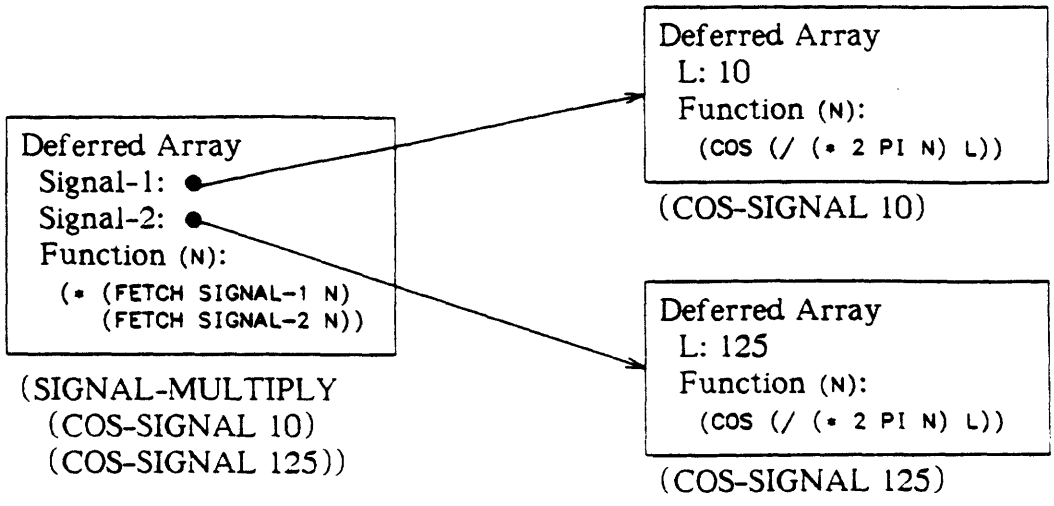


Fig. 3.8. Result of (SIGNAL-MULTIPLY (COS-SIGNAL 10) (COS-SIGNAL 125)).

When a request for a value of (SIGNAL-MULTIPLY (COS-SIGNAL 10) (COS-SIGNAL 125)) is made and the value is not in the memory the function FCN of SIGNAL-MULTIPLY is applied to the index. Executing FCN causes both (COS-SIGNAL 10) and (COS-SIGNAL 125) to be asked for their value at the index. Thus, the deferred array computes a signal value by successive requests transmitted backward, computation, and values transmitted forward. Only those values that are requested, either by the end user, or by some function in the process of answering a request posed to it, are ever computed. It seems that only the minimal number of signal values required for the final answer are computed. In Chapter 5 we shall see that this statement is not strictly true when rearrangement of computation is allowed. We will also return to the issue of minimizing computation in section 3.5.

SIGNAL-MULTIPLY illustrates the use of the deferred array for combining signals. Similarly, we can easily define systems for adding, subtracting, dividing, scaling,

or shifting signals.

We can define FIR filters and IIR filters using the deferred array. Processing a signal by an FIR filter, $H(z) = h_0 + h_1 z^{-1} + \cdots + h_{L-1} z^{-(L-1)}$, can be defined by

```
(DEFINE (FIR-FILTER X COEFFS)
  (DEFINE (FCN N)
    (LOOP FOR H IN COEFFS
          FOR K FROM 0
          SUMMING (* H (FETCH X (- N K)))))
  (MAKE-DEFERRED-ARRAY FCN))
```

Now (FIR-FILTER X (LIST $h_0 \cdots h_{L-1}$)) implements

$$y[n] = \sum_{k=0}^{L-1} h_k \, x[n-k] \tag{3.8}$$

Similarly, we can define the processing of a signal by the IIR filter $H(z) = 1/(1 - a_1 z^{-1} - \cdots - a_{L-1} z^{-(L-1)})$ as

```
(DEFINE (IIR-FILTER X COEFFS)
  (DEFINE (FCN N)
    (IF (< N 0)
        0
        (+ (FETCH X N)
           (LOOP FOR A IN COEFFS
                 FOR K FROM 1
                 SUMMING (* A (FETCH SELF (- N K)))))))
  (DEFINE SELF
    (MAKE-DEFERRED-ARRAY FCN))
  SELF)
```

Now (IIR-FILTER X (LIST $a_1 \cdots a_{L-1}$)) implements

$$y[n] = \begin{cases} x[n] + \sum_{k=1}^{L-1} a_k \, y[n-k] & n \geq 0 \\ 0 & n < 0 \end{cases} \tag{3.9}$$

Two interesting thing happen in the definition of IIR-FILTER. First, the output from IIR-FILTER, $y[n]$, is captured internally by the variable SELF and is referenced by the function that computes the values. Secondly, we have assumed that $y[n]$ is 0 for $n < 0$. This is correct only if $x[n]$ is 0 for $n < 0$ and the filter is intended as a causal filter. Since we can not find out where $x[n]$ starts we can not build the correct system. We will come back to this deficiency in the next section.

In addition to the standard signal processing systems that we have already discussed, we can also define higher level systems. A higher level system is one that models many different systems with common code and higher level systems are used in good software design. An example of a higher level system for signal processing is SIGNAL-MAP. SIGNAL-MAP is defined by

```
(DEFINE (SIGNAL-MAP OP SIGNAL)
  (DEFINE (FCN N)
    (OP (FETCH SIGNAL N)))
  (MAKE-DEFERRED-ARRAY FCN))
```

and applies the function OP to each element of the signal SIGNAL. SIGNAL-MAP is the generalization of single input memoryless systems. Using SIGNAL-MAP we can define SIGNAL-SQUARE, a system for point by point squaring of its input signal, as

```
(DEFINE (SIGNAL-SQUARE SIGNAL)
  (DEFINE (SQUARE X)
    (* X X))
  (SIGNAL-MAP SQUARE SIGNAL))
```

SIGNAL-SQUARE defines an internal function, SQUARE, and maps it over the input signal.

Another high level system is SIGNAL-COMBINE, defined by

```
(DEFINE (SIGNAL-COMBINE OP SIGNAL-1 SIGNAL-2)
  (DEFINE (FCN N)
    (OP (FETCH SIGNAL-1 N) (FETCH SIGNAL-2 N)))
  (MAKE-DEFERRED-ARRAY FCN))
```

SIGNAL-COMBINE applies the function OP to pairs of values, one from each of the signals SIGNAL-1 and SIGNAL-2. Using SIGNAL-COMBINE we can define SIGNAL-MULTIPLY as

```
(DEFINE (SIGNAL-MULTIPLY SIGNAL-1 SIGNAL-2)
  (SIGNAL-COMBINE * SIGNAL-1 SIGNAL-2))
```

We may also define higher level non-memoryless systems. For example, SIGNAL-REDUCE, defined by

```
(DEFINE (SIGNAL-REDUCE OP SIGNAL MEMORY-SIZE)
  (DEFINE (FCN N)
    (OP (LOOP FOR K FROM 0 BELOW MEMORY-SIZE
              COLLECTING (FETCH SIGNAL (- N K)))))
  (MAKE-DEFERRED-ARRAY FCN))
```

applies OP to the last MEMORY-SIZE samples of the signal SIGNAL. Using SIGNAL-REDUCE we can define FIR-FILTER as

```
(DEFINE (FIR-FILTER SIGNAL COEFFS)
  (DEFINE (OP SIGNAL-VALUES)
    (LOOP FOR H IN COEFFS
          FOR X IN SIGNAL-VALUES
          SUMMING (* H X)))
  (SIGNAL-REDUCE OP SIGNAL (LENGTH COEFFS)))
```

3.3.4. Limitations of the Deferred Array

We have defined the deferred array and shown that it can be used to model many signal processing operations. However, the deferred array has some limitations as a signal representation. As example of the primary limitation of deferred arrays, consider the convolution of two signals $x[n]$ and $h[n]$

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]\,h[n-k] \tag{3.10}$$

Using deferred arrays we could try the definition of SIGNAL-CONVOLVE

```
(DEFINE (SIGNAL-CONVOLVE X H)
  (DEFINE (FCN N)
    (SUM-OF-SIGNAL (SIGNAL-MULTIPLY X (SIGNAL-DELAY (SIGNAL-REVERSE H) N))))
  (MAKE-DEFERRED-ARRAY FCN))
```

provided that SIGNAL-REVERSE and SIGNAL-DELAY are appropriately defined. The problem arises because SUM-OF-SIGNAL, intended to sum all the elements in a signal, can not perform the summation without some information about the non-zero extent of $x[k]\,h[n-k]$.

Another signal processing operation that requires information about the non-zero extent of signals is computation of samples of the Fourier transform of a signal. To compute samples of $X(e^{j\omega})$ at a spacing of $\frac{2\pi}{L}$

$$\sum_{n=-\infty}^{\infty} x[n]\, e^{-j\frac{2\pi kn}{L}} \tag{3.11}$$

we need to know the non-zero extent of $x[n]$.[5]

Another problem with deferred arrays is the overhead they impose on the programmer to test indices. For example, in defining the system HAMMING-WINDOW

```
(DEFINE (HAMMING-WINDOW L)
  (DEFINE (FCN N)
    (IF (AND (>= N (MINUS (FLOOR (/ L 2))))
             (< N (- L (FLOOR (/ L 2)))))
        (+ .54 (* .46 (COS (/ (* 2 PI N) (- L 1)))))
        0))
  (MAKE-DEFERRED-ARRAY FCN))
```

for generating the L point Hamming window centered at the origin

$$Hamming_L[n] = \begin{cases} .54 + .46\cos(\frac{2\pi n}{L-1}) & -\lfloor L/2 \rfloor \leqslant n < L - \lfloor L/2 \rfloor \\ 0 & \text{otherwise} \end{cases} \tag{3.12}$$

the index testing is a tiresome programming chore. Notice also, that in this example, the range of indices that execute the true part of the if clause are the same range of indices that define the non-zero extent of the signal. In our description in the next section of an extension to the deferred array we will address both the non-zero extent issue and the index testing issue.

3.4. Extensions of the Deferred Array

In this section we discuss two major and one minor extensions to the deferred array. The major extensions are to allow multiple functions in the definition of a deferred array and to have the non-zero extent of a deferred array as an observable property. The minor extension is to have the output of systems uniquely defined by the input parameters.

---

[5] We will discuss in Chapter 5 situations when we can perform convolutions, Fourier transforms, and other operations without need to know the non-zero extent of signals.

3.4.1. Multiple Functions

The first major extension to the deferred array we propose is that a deferred array contain not just a memory and a function but a memory and any number of functions. The idea is illustrated in Fig. 3.9, which shows a deferred array with two functions, F1 and F2. In this example, F1 is used for negative indices and F2 is used for non-negative indices.

In general, the extended deferred array contains a memory and an association list of intervals and functions, as illustrated in Fig. 3.10. An interval is a data structure that represents a connected set of indices. We use the form (INTERVAL *start end*) to represent the set of indices $\{start, start+1, \cdots, end-1\}$. We will assume that the operators INTERVAL-START and INTERVAL-END return the start and end of an interval respectively. The intervals of an extended deferred array should be disjoint

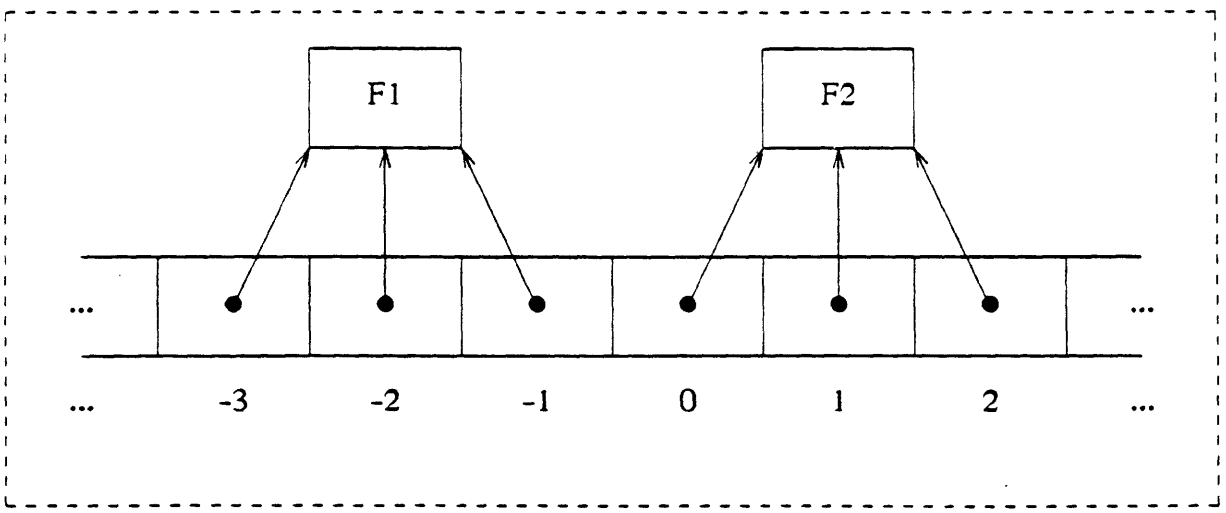


Fig. 3.9. Example of an extended deferred array.

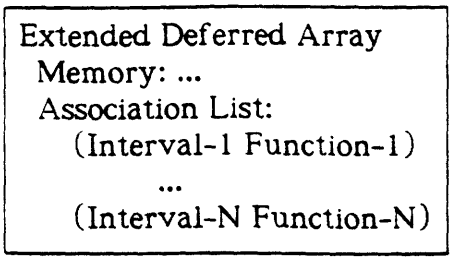


Fig. 3.10. Extended deferred array.

and the union of all the intervals should represent the set of indices from $-\infty$ to ∞. We will assume that the special objects MINF and INF represent $-\infty$ and ∞ respectively.

Before presenting examples of signal definitions using the extended deferred array we will make two more generalizations. One is that, because functions that return a constant value are so commonly used, we will assume that an entry in the association list of an extended deferred array can contain either an interval and function pair or an interval and constant pair. The other generalization is to remove the requirement that the union of all intervals cover the index set. Instead we will assume some default value or function for indices for which no interval is specified. The resulting model is now illustrated in Fig. 3.11

When the value of the extended deferred array at some index is desired, and the value is not present in memory, the extended deferred array tries to find a function or value whose associated interval contains the index. If a function is found then that function is applied to the index. If a value is found then the value is returned. If no function or value is found then the default is examined. If the default is a function

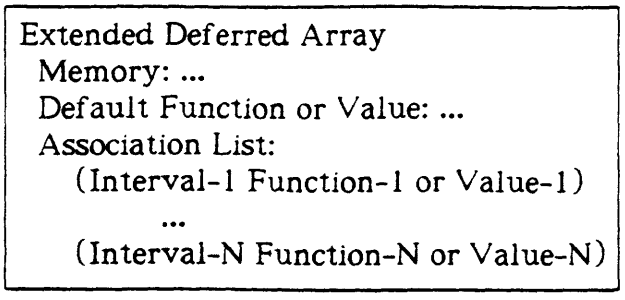


Fig. 3.11. Complete extended deferred array.

then it is applied to the index. If the default is a value it is returned.

We will define extended deferred arrays with the constructor MAKE-EXTENDED-DEFERRED-ARRAY, which takes as input a default value or function and pairs of intervals and values or functions. An implementation of extended deferred arrays is given in Appendix B and the signal representation presented in Chapter 4 is similar to the extended deferred array. The syntax of MAKE-EXTENDED-DEFERRED-ARRAY is

```
(MAKE-EXTENDED-DEFERRED-ARRAY default-function-or-value
  interval-1 function-or-value-1
  ...
  interval-N function-or-value-N)
```

As an example of the definition of a system using the extended deferred array consider

```
(DEFINE (HAMMING-WINDOW L)
  (DEFINE (FCN N)
    (+ .54 (* .46 (COS (/ (* 2 PI N) (- L 1))))))
  (MAKE-EXTENDED-DEFERRED-ARRAY 0
    (INTERVAL (MINUS (FLOOR (/ L 2))) (- L (FLOOR (/ L 2)))) FCN))
```

for defining Hamming windows of length L as in (3.12). HAMMING-WINDOW illustrates that the extended deferred array simplifies the coding of functions associated with deferred arrays because the functions do not need to test indices.

Another example of the use of the extended deferred array is

```
(DEFINE (COS-SIGNAL L)
  (DEFINE (FCN N)
    (COS (/ (* 2 PI N) L)))
  (DEFINE (DEFAULT-FCN N)
    (FETCH SELF (MOD N L)))
  (DEFINE SELF
    (MAKE-EXTENDED-DEFERRED-ARRAY DEFAULT-FCN
      (INTERVAL 0 L) FCN))
  SELF)
```

for defining cosine signals of period L as in (3.2). This definition of COS-SIGNAL uses a function, FCN, for defining the values of a cosine signal for one period and uses a default function, DEFAULT-FCN, to periodically replicate the single period.

3.4.2. Observable Non-Zero Extent

The other major extension to the deferred array has to do with the observable properties of a deferred array. We will assume that it is possible to observe the non-zero extent of an extended deferred array using the function SUPPORT. This function can examine the definition of an extended deferred array and determine an interval outside which it is guaranteed that the deferred array is zero. This can be done by forming the union over all intervals with non-zero functions and over the interval associated with the default when the default is not zero. Assuming the existence of SUPPORT, we can now define convolution as[6]

```
(DEFINE (SIGNAL-CONVOLVE X H)
  (DEFINE (FCN N)
    (SUM-OF-SIGNAL (SIGNAL-MULTIPLY X (SIGNAL-DELAY (SIGNAL-REVERSE H) N))))
  (MAKE-EXTENDED-DEFERRED-ARRAY 0
    (INTERVAL (+ (INTERVAL-START (SUPPORT X))
                 (INTERVAL-START (SUPPORT H)))
              (+ (INTERVAL-END (SUPPORT X))
                 (INTERVAL-END (SUPPORT H)) -1))
    FCN))
```

with SUM-OF-SIGNAL defined as

[6] To be absolutely correct we have to redefine + to understand both numbers and the special tokens MINF and INF because either X or H might have infinite non-zero extent.

```
(DEFINE (SUM-OF-SIGNAL SIGNAL)
  (LOOP FOR INDEX FROM (INTERVAL-START (SUPPORT SIGNAL))
                  BELOW (INTERVAL-END (SUPPORT SIGNAL))
        SUMMING (FETCH SIGNAL INDEX)))
```

This definition uses the fact that if $x[n]$ is zero outside the interval $L_x \leqslant n < U_x$ and $h[n]$ is zero outside the interval $L_h \leqslant n < U_h$ then $x[n] * h[n]$ is zero outside the interval $L_x + L_h \leqslant n < U_x + U_h - 1$. Note that we do not define SIGNAL-CONVOLVE by

```
(DEFINE (SIGNAL-CONVOLVE X H)
  (DEFINE (FCN N)
    (SUM-OF-SIGNAL (SIGNAL-MULTIPLY X (SIGNAL-DELAY (SIGNAL-REVERSE H) N))))
  (MAKE-EXTENDED-DEFERRED-ARRAY FCN))
```

because it is not easy to determine the non-zero extent of the result of convolution using this definition.

Using the SUPPORT operation it is now possible to correctly define causal IIR-filtering for input signals that begin at any time index and to properly define Fourier transform sampling.

3.4.3. Uniqueness of Signals

The minor change to the deferred array model has to do with the idea that recomputing the values of a signal is wasteful. When the user of a system invokes it twice on the same parameter values the user should get back the identical signal object, e.g. evaluating the expression (HAMMING-WINDOW 255) twice should return the same object, not two different copies. In this way previously computed signal values can be shared. An example of where this ability is useful is in the design of a large signal processing system with several components. The different components may perform the same operations, such as filtering a signal or computing a spectrum. These results should be shared. This can be done explicitly by defining global variables or implicitly

by relying on systems to return unique objects.

3.5. Implementation Tradeoffs

Our description of deferred arrays and extended deferred arrays made several design decisions in the direction of ease of programming. Particularly, we said that deferred arrays should have transparent caching of signal values, deferred evaluation, and automatic sharing of computational results. Obviously there are some costs to each of these decisions and situations in which these features should not be present.

Transparent caching has both storage and computational overhead - storage overhead from the need for a memory and computational overhead of checking the memory for previously computed values. This overhead is worthwhile when memory is cheap and signal values both are expensive to compute and are reused. Examples in which caching should not be present are when signals are easy to compute, such as constants or shifts, or when signal values are not reused, such as in stream processing. As such, caching should be optional, not required.

Deferred evaluation has the overhead of having to build signal objects that store internal functions and having to decide when to invoke these functions. For certain signals, particularly infinite duration, non-periodic signals, deferred evaluation is essential because it is not possible to compute all the signal values of a signal when the signal is created. Other signals for which deferred evaluation is particularly useful are signals for which it is not easy to predict apriori what values will be needed or signals for which computation of all samples that will be used would require too much memory. Deferred evaluation is wasteful when it is known that most or all the values of the signal will be used. As such, it should be possible to precompute some or all values of a signal at the time the signal is created.

Obviously the issues of deferred evaluation and caching are coupled. If evaluation of signal values is not deferred there must be a memory to store the values. Thus, when computation of all the signal values would require too much memory it is not useful to simultaneously use deferred evaluation and transparent caching. A worthwhile strategy would be to allow caching of only part of the signal.

The automatic sharing of results by having systems return the same signal object for the same inputs has the overhead of having systems record what inputs they have seen and checking each invocation to see if the same parameters are being used. This overhead is worthwhile when recomputation of signal values is expensive and when a large amount of sharing occurs and is wasteful when signals are easy to compute or known not to be shared.

3.6. Outside View of Signals

In this section we draw on our definition of signals, previous signal representations, and the deferred array to summarize an *outside view* of signals. The outside view of signals is the data abstraction that all signal objects must respect regardless of their underlying implementation.

The main features of our view of signals are illustrated in Fig. 3.12. Signals should have two observable properties, namely the value of the signal at any index and the non-zero extent of the signal. It is not an error to access a signal outside its non-zero extent - the value simply is zero. Our model of signals as abstract objects is an extension of Kopec's model of signals. By not imposing limits on the indices of signals we allow for infinite duration signals and we provide a more natural representation for signal processing operations. Also, we eliminate the need for the user to keep track of an implied origin and we do not introduce the semantic difficulties found in

Observable Properties

- Signal value at any index from $-\infty$ to ∞
- Non-zero extent of signal

Other Properties

- Immutability
- Deferred evaluation
- Transparent Caching
- Unique identity

Fig. 3.12. Outside view of signals.

other representations when trying to combine signals of different lengths.

While we model signals as infinite in extent, we recognize that to perform certain signal processing operations it is necessary or useful to know the non-zero extent of signals. Thus, a good signal representation should allow the user to find out an interval of indices, perhaps infinite, outside which the signal is guaranteed to be zero.

In addition to the two observable properties of signal value and non-zero extent, a good signal representation should be immutable, should support deferred evaluation, should provide caching of signal values, and should provide unique identities to signals. Immutability is in keeping with the spirit of signals as objects defined by some function. Deferred evaluation is useful for reducing unnecessary computation and is essential in representing infinite duration signals. Caching and unique identities for signals are important in providing an efficient signal representation.

3.7. Inside View of Signals

In the previous section we summarized how a good signal representation should appear to the user of a signal. A data abstraction model of signals is of no use,

however, if we do not know how to build signal objects. The deferred array and extended deferred array presented earlier implement one mechanism for specifying signal objects. It is not the only mechanism that can be used to specify signals and in this section we describe four models for the specification of signal objects - the point operator model, the array operator model, the state machine model, and the composition model.

Our viewpoint in this section is that signals are created by systems and all the signals produced by a system share common computational mechanisms, but with different parameter values. We identify four structures used in the mathematical expression of systems and discuss how these models should be translated into programming constructs. The features of the four translations that we present share the common idea that the programmer of a system should not have to specify more than the mathematics of the signal processing operation. We call the requirements imposed on a programmer by a signal representation the *inside view*. A good signal representation should support a variety of inside views, including the four specified here, while maintaining a consistent outside view of signals and should provide reasonable defaults to the programmer. In Chapter 4 we will describe a signal representation language in which it is possible to use all four of these models and give examples of system definitions using each of them.

3.7.1. Point Operator Model

The deferred array models a signal by a function (or functions) that maps an index into a value. We call this model the *point operator* model because it computes a signal on a point by point basis. We have already shown that this is a useful model for many signal processing operations, including generation of signals, addition,

multiplication, convolution, and filtering of signals.

The requirements on the programmer and the signal representation imposed by the point operator model are summarized in Fig. 3.13. The programmer must supply the function that maps indices into signal values and must specify the domain of this function. The programmer must also specify how the signal behaves outside the domain of the function. Common specifications for this are constant value, particularly zero, and periodic replication.

The representation guarantees that the point operator is invoked only for indices inside its domain. This simplifies the coding of the point operator. The representation should also supply caching and implement the behavior of the signal outside the domain of the function. Particularly, the programmer should not have to do anything to get a zero value outside the function domain and the programmer should not have to write the code for periodic replication. Instead, the programmer should only have to specify the period. Caching and implementation of signal behavior outside the function domain are required by all our inside views.

Programmer Requirements

- Define a function that maps indices into values
- Specify function domain
- Specify behavior outside function domain (constant, periodic replication)

Representation Requirements

- Insurance that function is called only over its domain
- Caching
- Implementation of behavior outside function domain

Fig. 3.13. Point operator model.

3.7.2. Array Operator Model

An example of an operation that poses a problem for the point operator model is the FFT. The FFT naturally computes many values simultaneously, rather than one value at a time. The point operator model includes a function of the index that returns a value and does not support the concept of simultaneously computing several values.

We call operators that define signals by functions mapping an interval of indices into a set of signal values *array operators*. The FFT is one example of an array operator. Other array operators are frequency domain convolutions and array processor based algorithms.

Fig. 3.14 summarizes the requirements on the programmer and on the signal representation for the use of array operators. Defining a signal with an array operator is similar to defining a signal with a point operator - in both cases the programmer needs to define a function for signal values, the domain of the function, and outside

---

Programmer Requirements

- Define a function that maps intervals of indices into values
- Specify function domain
- Specify behavior outside function domain
- Specify valid array sizes to use

Representation Requirements

- Insurance that function is called only over its domain
- Translation from single sample requests to array requests
- Storage allocation for results
- Access to interval of signal values

Fig. 3.14. Array operator model.

---

behavior. For an array operator the programmer must also specify what are legal size intervals to be used. The representation should insure that the array operator is not called for intervals of indices outside the domain of the function and that the array operator is always invoked for intervals of the proper size. The representation should allow both array operators that compute all the samples in their domain simultaneously and array operators that compute only some of the samples in their domain. Also, the representation should do whatever array management is possible, particularly by supplying an array to the array operator for results.

As an example of the behavior of a system with an array operator, consider a system for FFTs. A request for a single sample of a signal produced by this system would be translated internally into a request to the array operator for all the signal values. The representation would allocate an array of the proper size and then invoke the array operator. The array operator would run its computation and fill the specified array with results. The representation would store these results in the local memory and, finally, the requested signal value would be returned.

Another implication of array operators is that, because array operators naturally work on many signal values at once, there should be a way to get from a signal an array of values representing the values of the signal over an interval. This operator may be as simple as one that allocates an array, fetches the value of the signal for each index in the interval, fills the array with the values and returns the array. However, such an operator could be made more efficient when the signal is naturally computed with an array operator by filling all the elements of the output array directly from the output of the array operation. In this way the local memory is examined only once and the overhead is reduced.

3.7.3. State Machine Model

Another problem with the point operator model is that it does not naturally support state equations of the form of (3.3), as can be done easily in stream based signal representations. State machines provide a more natural computational mechanism for some signal processing operations, such as causal filtering, than point operators and therefore should be supported. We have shown that it is possible to define some recursive systems using deferred arrays. However, the method works only when the state is captured in the past output signal values.

The requirements on the programmer and on the signal representation for state machines are summarized in Fig. 3.15. The programmer must specify the state update function, the output function, and the initial state. The representation must run the state machine on successive indices while preserving the outside view of random access, i.e. the user should be able to get any signal value and the representation should either find that value in memory or run the state machine up to the appropriate index. The representation should provide storage for the state.

Programmer Requirements

- Define state update function
- Define output function
- Specify initial state

Representation Requirements

- Insurance that the functions are called for successive indices
- Translation from random access to sequential computation
- Storage of state

Fig. 3.15. State machine model.

3.7.4. Composition Model

The final way in which systems can be specified is by putting several systems together as a group. For example, it should be possible to define the system for generating Hamming windows by putting together a system for generating rectangular windows, a multiplier system, and a system for generating cosine signals. Also, it should be possible to add extra information into a composition of systems that reflects our signal processing knowledge. For example, if a linear convolution is implemented by composing two FFTs, a multiplier, and an inverse FFT it should be possible to specify the non-zero extent of the convolution directly. These requirements are summarized in Fig. 3.16.

3.8. Summary

This chapter has described many aspects of the representation of signals for numerical processing. We have reviewed past signal representations and found them to be lacking in some areas. This review is summarized in Fig. 3.17. Arrays are finite extent, starting at the origin, and mutable. Streams are restricted to one dimensional signals, starting at the origin, and accessed in sequential order. Kopec's signal objects are finite extent and start at the origin.

Programmer Requirements

- Specify system in terms of old systems

Representation Requirements

- Ability to accept extra information

Fig. 3.16. Composition model.

Array Based Signal Representations

- Finite extent
- Start at origin
- Mutable objects

Stream Based Signal Representations

- Restricted to one dimensional signals
- Start at origin
- Fixed access order

Kopec's Abstract Signal Objects

- Finite extent
- Start at origin

Fig. 3.17. Summary of problems of previous signal representations.

We then presented the deferred array as a simple model for describing a large class of signal processing operations. The deferred array modeled signals by a combination of a function, a memory, and deferred evaluation. The deferred array was then extended to incorporate multiple functions, constant values, default values, and access to the non-zero extent. The deferred array and the extended deferred array are summarized in Fig. 3.18.

Our examination of previous signal representations and the deferred array led to a description of the properties a signal should have in a good signal representation. We called this the outside view of signals. These have been summarized in Fig. 3.12. To the user of a signal a signal should be able to be evaluated at any index from $-\infty$ to ∞. The user should also be able to get the non-zero extent of a signal. A signal should be an immutable object that is computed as it is needed. Signals should be uniquely identified.

Deferred Array

- Function for sample values
- Memory for caching values
- Compute values as requested

Extended Deferred Array

- Break index axis up into intervals
- Multiple functions
- Constants and default value
- Non-zero extent

Fig. 3.18. Summary of the deferred array and the extended deferred array.

We finished our discussion of signal representation by describing how signals should appear to the programmer of signals. We called this the inside view of signals. Fig. 3.19 summarizes our models for signal computation. Point operators are defined by functions that map indices into values. The deferred array is an example of a point operator model. Array operators are defined by functions that map intervals of indices into arrays of signal values. Array operators require consideration of the interval over which the calculation is going to occur. Also, array operators suggest that signals be accessed over intervals of indices as well as on a sample by sample basis. State machine models are defined by state equations and an initial state. State machine models have to be run for successive indices while maintaining the outward appearance of random access. Compositions are formed by piecing together old systems to form new ones and should allow the specification of extra information.

Point Operator Model

- Map indices into values
- Specify function domain
- Specify behavior outside function domain (constant, periodic replication)
- Deferred array model

Array Operator Model

- Map interval of indices into values
- Need to determine computation interval
- Allow access to interval of signal values

State Machine Model

- State equations
- Need initial state
- Want to maintain illusion of random access from outside

Composition Model

- Put together old systems to form new ones
- Allow specification of extra information

Fig. 3.19. Summary of inside view of signals.

CHAPTER 4

The Signal Processing Language and Interactive Computing Environment

This chapter describes an implementation of the signal representation ideas presented in Chapter 3. The Signal Processing Language and Interactive Computing Environment (SPLICE) is a carefully designed and integrated set of programs for the representation and manipulation of one-dimensional discrete-time signals. SPLICE is written in ZETALISP [26] and runs on a LISP Machine [10] and contains over 200 system definitions. In addition to SPLICE, our ideas about numerical signal representation and manipulation have been incorporated into software written in SCHEME and in C, but in this chapter we concentrate on the ZETALISP implementation to illustrate our ideas. It is not our intent to provide a complete description of SPLICE in this chapter. An introductory description of SPLICE is given in [27] and Dove [28] provides a good overview of SPLICE. Appendix C of this thesis contains a brief user's guide to SPLICE.

Briefly, signals in SPLICE are represented in the style of the extended deferred array of Chapter 3 with a single function and interval pair, and either a default value or periodic replication. Signals are unique, immutable objects, the computation of whose values is deferred until needed. Signal values may be numbers (real or complex), symbols, or other signals. The programmer can define signals using point operations, array operations, state machines, or by composing systems.

We start our discussion of SPLICE with a description of its history and the choice of ZETALISP and the LISP Machine for its implementation. We then present the outside view of signal objects in SPLICE. Next we present the inside view of signal objects. We then discuss some issues of implementation, including caching and memory management. We present an extended example illustrating the representation and numerical manipulation of discrete-time signals. Finally, we present an empirical evaluation of SPLICE as a signal representation and as a programming tool.

SPLICE not only illustrates our ideas on the numerical representation and manipulation of signals, it has also been an effective tool in our signal processing research. All our work in the area of Knowledge Based Signal Processing has been developed using SPLICE as the basis for the numerical signal processing components [29, 30, 31, 28, 32]. Also, SPLICE has been used in some strictly numerical signal processing work [33].

4.1. Background

SPLICE is a collaborative effort between this author and Dr. Webster Dove. Dr. Dove started work on SPLICE in January, 1982 and this author joined the effort in January, 1983. SPLICE was developed not only as a demonstration of our ideas about the numerical representation and manipulation of signals but also as a support tool for work in the area of Knowledge Based Signal Processing (KBSP).[1] In this section we discuss KBSP and the choice of ZETALISP and the LISP Machine.

[1] SPLICE was originally called the KBSP Package and is referred to by that name in[27, 28].

4.1.1. Knowledge Based Signal Processing

Many signal processing problems involve both mathematical models and heuristic problem solving techniques, including the tracking of ships in the ocean [34] and speech recognition [35]. The field of Knowledge Based Signal Processing is the study of the coupling between numerical and symbolic problem solving techniques in these, and other similar problems. Many proposed solutions have emphasized either numerical or symbolic processing, rather than a mix of the two and the goal of the KBSP project is to develop systems that more evenly balance and more closely couple the two processing methods. Towards this end, systems have been developed for pitch detection [28] and for the tracking of helicopters [32]. These systems show that numerical and symbolic processing techniques can interact closely in the solution of a signal processing problem.

4.1.2. LISP and the LISP Machine

SPLICE is written in ZETALISP for the LISP Machine. ZETALISP and the LISP Machine were chosen as the development environment because this was the environment that was chosen for KBSP research. The LISP Machine was a natural choice for our work on KBSP because it provides a strong base for the symbolic processing portions of our work. Additionally, both LISP and the LISP Machine are good choices for signal processing programming.

LISP is one of the oldest of computer languages and still is one of the most popular and widely used languages [36]. It is distinguished by its primary data structure, the list, is an ordered collection of atoms used to represent many data objects and functions. The representation of functions by list structure makes LISP an extendible language and facilitates the development of programs that manipulate other programs.

In addition to list structure, LISP provides flexible data structures, including arrays, strings, and structures. ZETALISP provides an object-oriented programming system called the Flavors system [26].

Variables in LISP exhibit manifest type, i.e. the type of value of a variable can be determined at run time. This simplifies programs because generic functions can be built to handle multiple data types. For example, the same sort routine can be used for an array of integers, of single precision floating point numbers, or of double precision floating point numbers.

Functions in LISP are not only represented by the basic list structure, they are first class data types. Functions may be created by other functions, passed as arguments, or returned as values. The ability to use functions as first class data structures facilitates the development of "active" data objects.

Another distinguishing feature about LISP is the way it is used. LISP comes with an interpreter for the interactive evaluation and debugging of programs. New procedures can be tested easily. Most good LISPs provide a compiler that can be used to produce efficient code.

LISP is a good language for signal processing because it contains all the tools expected in a language for signal processing, it is extendible, and it is efficient. LISP provides numerical data types, e.g. integers, floating point numbers, and complex numbers, arithmetic operators for the numerical data types, and arrays. In our discussion of SPLICE we will show how the extensibility of LISP can be used to simplify the definition of signal processing algorithms. Modern LISP compilers can be made to generate code as efficient as that of any other high level language.

The LISP Machine provides an integrated environment for the design and development of LISP programs. It provides an interactive editor, a compiler, a debugger, and a structure inspector all closely coupled, along with bit-mapped graphics, virtual memory, and large amounts of software. The LISP Machine provides specialized hardware that facilitates the execution of LISP programs, including hardware to manipulate tagged data types, to assist in garbage collection, and to perform microcode tasking. The LISP Machine has the processing power to support signal processing programs. A Symbolics 3670 LISP Machine is comparable to a Digital Equipment Corporation VAX750 for single precision arithmetic.[2]

4.2. Outside View of Signals

In Chapter 3 we developed a two sided view of signals, as illustrated in 4.1. In this view the user of a signal has some outside view that specifies those inquiries that the user can make and the programmer has an inside view that allows the programmer to define signals. In this section we describe the outside view of signals presented in SPLICE. We start with a presentation of intervals and functions for manipulating

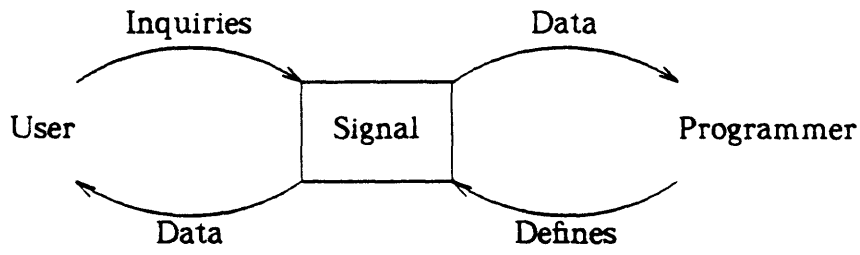


Fig. 4.1. Two sided view of signals.

[2] Based on timing measures of VAX750 with Floating Point Accelerator (FPA) single precision floating point add and multiply and Symbolics 3670 with FPA single precision floating point add and multiply.

intervals. We then describe those operations that can be used to access signal values. Next we present those operations that can access the support and period of signals. Finally we discuss other inquiry operations that can be performed on signals. We should emphasize, once again, that the outside view of signals is independent of how the signals are implemented. All signals can be accessed using the same methods.

To provide some reference for discussing the outside view of signals, Fig. 4.2 shows an annotated example of an interactive session using SPLICE. The token

```
SPLICE: (HAMMING 255)                    ; Create the 255 point Hamming
                                         ; window centered at 0
------> #<(HAMMING 255)>

SPLICE: (SUPPORT (HAMMING 255))          ; Find out the interval over
                                         ; which the window is non-zero
------> (INTERVAL -127 128)

SPLICE: (FETCH (HAMMING 255) 0)          ; Get the value of the
                                         ; Hamming window at index 0
------> 1.0

SPLICE: (FETCH (HAMMING 255) -130)       ; Get the value of the
                                         ; Hamming window at index -130
------> 0.0

SPLICE: (FETCH-INTERVAL (HAMMING 255) (INTERVAL -130 -123))
                                         ; Get an array of samples

------> #<ART-Q-7 xxxxxxx>              ; Funny printed form for an array

SPLICE: (LISTARRAY (FETCH-INTERVAL (HAMMING 255) (INTERVAL -130 -123)))
                                         ; Get an array of samples
                                         ; and make into a list

------> (0.0 0.0 0.0 0.08000001 0.08014074 0.08056289 0.801266135)

SPLICE: (PLOT (HAMMING 255))             ; Plot the 255 point Hamming
                                         ; window - see Fig. 4.3
------> #<(HAMMING 255)>
```

Fig. 4.2. Example session.

"SPLICE: " designates the user's input and the token "------> " designates the output. The first expression the user enters is (HAMMING 255). This invokes the system HAMMING on the argument 255. The system HAMMING takes as input one argument, the length, L, of the Hamming window to create, and produces a signal object as output that represents the L point Hamming window centered at the origin

$$Hamming_L[n] = \begin{cases} .54 + .46\cos(\frac{2\pi n}{L-1}) & -\lfloor L/2 \rfloor \leq n < L - \lfloor L/2 \rfloor \\ 0 & \text{otherwise} \end{cases} \quad (4.1)$$

The user first creates the 255 point Hamming window. The form #<*some description*> is the printed representation for objects that do not have a natural printed representation. The non-zero extent of the 255 point Hamming window is $-127 \leq n < 128$ and the user finds this out using the function SUPPORT. The user then fetches the value at index 0 using the function FETCH and finds that the value here is 1.0. Next, the user fetches the value at index -130 and finds the value there is 0.0. The index -130 is outside the non-zero support of the signal so the value there is zero. After using the function FETCH to determine the value of the Hamming window at an index the user uses the function FETCH-INTERVAL to get an array of values of the signal. The form (FETCH-INTERVAL (HAMMING 255) (INTERVAL -130 -123)) requests an array of seven signal values for the interval $-130 \leq n < -123$. The user first requests this array and then uses the function LISTARRAY to transform the array into a list. Finally, the user requests a plot of the Hamming window using the function PLOT. The resulting plot is shown in Fig. 4.3.

4.2.1. Intervals

SPLICE uses a signal representation similar to that of the extended deferred array presented in Chapter 3. The concept of an interval of indices is important in this

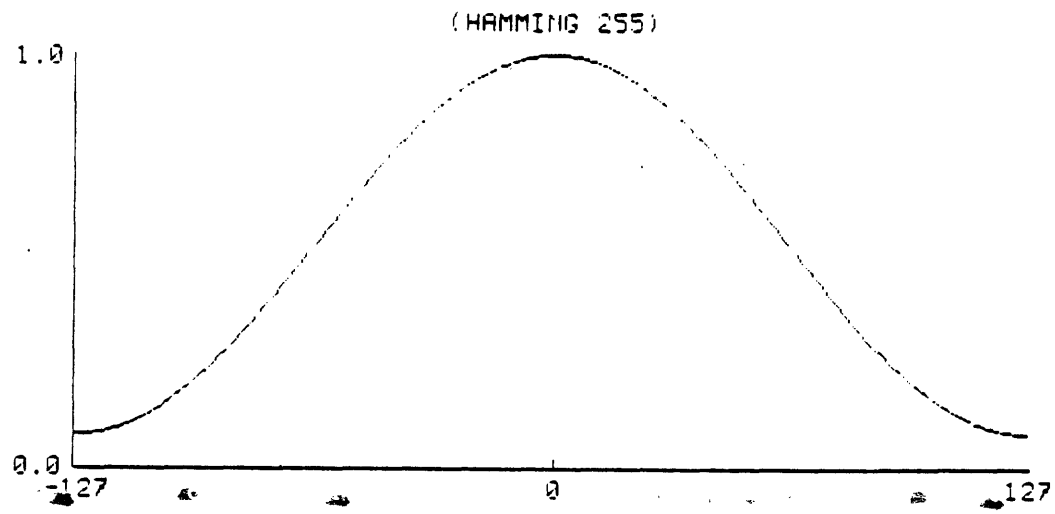


Fig. 4.3. Plot of (HAMMING 255).

representation and SPLICE provides many functions for the manipulation of intervals. The function INTERVAL is used for the creation of interval data structures. (INTERVAL *start end*) represents the set of indices $\{start, start+1, \cdots, end-1\}$. The special object, the NULL-INTERVAL, represents the empty set.

Intervals representing infinite connected sets of the integers can be created using the special objects MINF and INF. MINF and INF are the representations for $-\infty$ and ∞ and can be manipulated using the special arithmetic functions $+ (a version of + that understands both normal numbers and MINF and INF), $-, $*, and $//. The infinite interval $-\infty < n < end$ can be created by (INTERVAL MINF *end*). The infinite interval $start \leq n < \infty$ upward can be created by (INTERVAL *start* INF) and the infinite interval $-\infty < n < \infty$ can be created by (INTERVAL MINF INF). Infinite intervals are useful in describing many signals, such as the impulse response of an IIR filter.

Some of the functions that can be used to manipulate intervals are illustrated in Fig. 4.4. INTERVAL-START and INTERVAL-END provide the start and end of an interval. INTERVAL-INTERSECT can be used to find the intersection of two intervals and INTERVAL-COVER can be used to find the smallest interval that encompasses two intervals. The predicate INTERVAL-COVERS-P checks to see if one interval completely contains another and the predicate FINITE-INTERVAL-P checks to see if an interval is finite. The special LISP objects T and NIL correspond to the truth

```
SPLICE: (INTERVAL 0 10)                    ; Create an interval
------> (INTERVAL 0 10)

SPLICE: (INTERVAL-START (INTERVAL 0 10))   ; Get the start of an interval
------> 0

SPLICE: (INTERVAL-END (INTERVAL 0 10))     ; Get the end of an interval
------> 10

SPLICE: (INTERVAL-INTERSECT                ; Intersect two intervals
          (INTERVAL 0 10) (INTERVAL -10 5))
------> (INTERVAL 0 5)

SPLICE: (INTERVAL-COVER                    ; Cover two intervals
          (INTERVAL -10 -5) (INTERVAL 5 10))
------> (INTERVAL -10 10)

SPLICE: (INTERVAL-COVERS-P                 ; Test to see if one interval
          (INTERVAL 0 10) (INTERVAL 2 5))  ; covers another
------> T

SPLICE: (FINITE-INTERVAL-P                 ; Test for finite interval
          (INTERVAL 0 INF))
------> NIL
```

Fig. 4.4. Example of intervals.

values of TRUE and FALSE.

4.2.2. Signal Values

The primary feature of the model of signals presented in Chapter 3 is that signals can be evaluated at any index from $-\infty$ to ∞. Signals can either be numerically-valued (real or complex) or symbolically valued. Numerically-valued signals correspond to standard discrete-time signals. Symbolically-valued signals are useful for phonetic transcripts and voiced/unvoiced/silence decision signals. Also, signals whose values are other signals (a special case of symbolically-valued signals) are useful for short-time processing and as an approximation to multi-dimensional signals.

There are two distinct mechanisms for determining the value of signals. One method asks a signal for its value at some index. The function FETCH provides this information, as in the evaluation of (FETCH (HAMMING 255) 0) in Fig. 4.2. FETCH returns the value of a signal at the specified index (the real part of the signal value for complex-valued signals). The index that is used can be any integer, not just those integers within the non-zero support of the signal. No requirement is imposed on the user about the order in which samples are requested.

In addition to FETCH, the functions FETCH-IMAGINARY and FETCH-COMPLEX are defined. FETCH-IMAGINARY returns, for any index, the imaginary part of a complex-valued signal and FETCH-COMPLEX returns the complex value of the signal. Real-valued signals return 0.0 when asked for their imaginary part.

The other method for finding out the values of a signal allows the user to find out the values of a signal over an interval rather than at a single index. The function FETCH-INTERVAL takes a signal and a finite interval and returns an array of values. For example, (FETCH-INTERVAL (HAMMING 255) (INTERVAL -130 -123)) in Fig.

4.2 returns an array of seven elements containing the values of the 255 point Hamming window in the interval $-130 \leqslant n < -123$. As with FETCH, there is no restriction on the interval that is used in FETCH-INTERVAL. The interval may be totally contained within the non-zero support of the signal, or partially or totally outside the non-zero support of the signal. Intervals of signal values may be requested in any order. Also, the array that is returned by FETCH-INTERVAL can be used by the programmer in any way (including changing the values of any elements in the array) without affecting the signal object. As with the use of FETCH, FETCH-IMAGINARY, and FETCH-COMPLEX, complex-valued signals can be examined over an interval for their real-part, their imaginary-part or their complex-value using FETCH-INTERVAL, FETCH-IMAGINARY-INTERVAL, and FETCH-COMPLEX-INTERVAL.

4.2.3. Support, Period, and Default Value

In terms of the model presented in Chapter 3, signals in SPLICE are represented with a single function and interval pair, and either a default value or periodic replication. In addition to being able to determine the values of a signal, the user of a signal may determine the non-zero support of a signal, the signal's period, and the signal's default value. The default value of signal is either 0 for numerically-valued signals or NIL for symbolically-valued signals and the function DEFAULT-VALUE returns the default value of a signal. The function SUPPORT, as illustrated by the expression (SUPPORT (HAMMING 255)) in Fig. 4.2, returns the non-zero extent of a numerical-valued signal and the non-NIL extent for a symbolically-valued signal. The function PERIOD returns the period of a signal. The period of a signal is either an integer (since we are working in discrete-time) or INF, for non-periodic signals.

4.2.4. Other Inquiry Operations

In addition to finding out signal values and signal support, SPLICE defines several other inquiry operations that are useful in signal processing programming. Among these operations are ATOMIC-TYPE, RANGE, and PLOT.

SPLICE makes no basic distinction between signals that are purely real-valued and those that are complex-valued. All numerically-valued signals can be considered as if they were complex-valued. However, the distinction is often an important one in signal processing operations. The function ATOMIC-TYPE returns :REAL when applied to a purely real-valued signal and returns :COMPLEX when applied to a complex-valued signal. The determination of the atomic-type of a signal is made by examining the definition of the signal, not by examining all the values of a signal.

The function RANGE applied to a signal returns the maximum and minimum amplitude of the real part of the signal. Similarly, IMAGINARY-RANGE returns the maximum and minimum of the imaginary part of a signal.

The function PLOT is not really an inquiry operation because it does not return information about the signal. However, it plots the signal in a graphics window and is useful for studying signals. PLOT can be used to plot real-valued signals, complex-valued signals, and signals whose values are themselves other signals.

4.3. Inside View of Signals

The major programming activity in SPLICE is the defining of signals. Signals are created by systems and the programmer defines new types of signals by defining new systems. Systems may be generators of signals, such as a Hamming window generator, or systems may form new signals from old ones, such as convolving two signals to

produce a third. The programmer may define systems using point operators, array operators, state machines, or system composition. This section describes the different methods of defining systems.

4.3.1. Signal Classes

Signals are organized into sets, called *signal classes* [24]. The fundamental signal class is the signal class SEQUENCE, referring to all one dimensional discrete-time signals.

The signal classes NUMERIC-SEQUENCE and SEQUENCE-OF-SEQUENCES are specializations of the signal class SEQUENCE. The signal class NUMERIC-SEQUENCE refers to all signals that are numerically-valued (either real or complex). All standard one dimensional signal processing operators produce signals that are in the signal class NUMERIC-SEQUENCE.

The signal class SEQUENCE-OF-SEQUENCES describes all signals whose values are themselves signals. Signals of this class include sequences of spectral slices and sequences of windowed sections of a sequence. Signals of the class SEQUENCE-OF-SEQUENCES are used extensively in short-time signal processing operators.

Signals are also grouped into signal classes according to the system that produces them. For example, Hamming windows are produced by the system HAMMING and the signal class HAMMING is a specialization of the signal class NUMERIC-SEQUENCE. Fig. 4.5 illustrates the relationship among the signal classes SEQUENCE, NUMERIC-SEQUENCE, SEQUENCE-OF-SEQUENCES, and HAMMING.

Signal classes not only provide a way of organizing different collections of signals, they also are used as abstract data types and define an inheritance of signal properties.

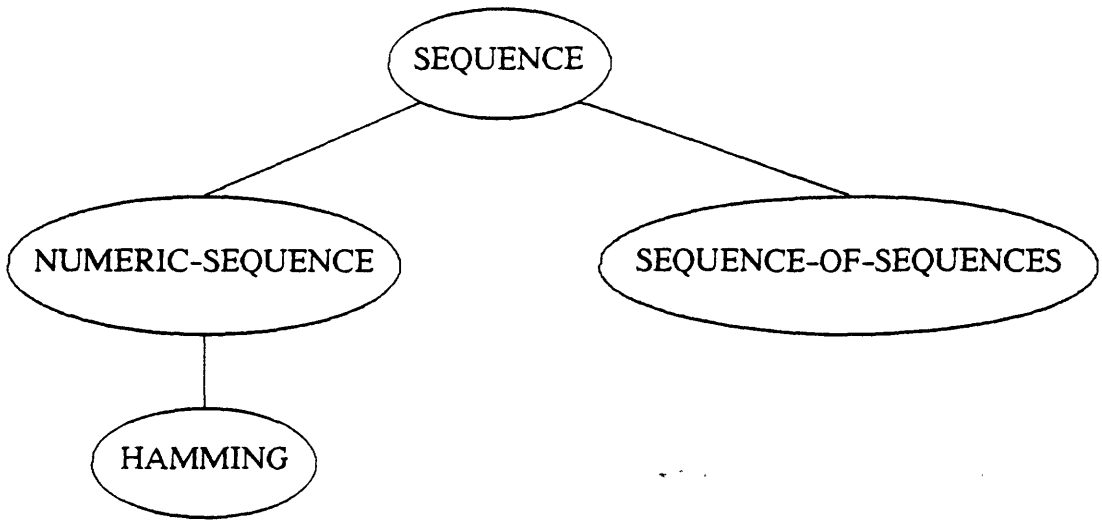


Fig. 4.5. Hierarchy of signal classes.

Associated with each signal class are functions that are used for all members of the signal class. Unless explicitly overridden, a specialized signal class inherits all the functions from its parent signal class. For example, all signals of the signal class NUMERIC-SEQUENCE share a common function for computing the maximum and minimum value of the real part of the signal. This function looks at all the signal values in the non-zero support of the signal to determine the range on the real part. Certain specializations of the signal class NUMERIC-SEQUENCE specify their own function for determining this property. For example, the class of all complex exponentials specifies that the range on the real part is from -1.0 to 1.0.

4.3.2. DEFINE-SYSTEM

The primary method for defining systems is the operator DEFINE-SYSTEM, which can be used to define systems using either point operators or array operators.

An example of a point operator system definition is the definition for the system HAMMING

```
(DEFINE-SYSTEM HAMMING (L)
               (NUMERIC-SEQUENCE)
  "Hamming window of size L, centered on the origin"
  (SUPPORT ()
    (INTERVAL (MINUS (FLOOR (// L 2)))
              (- L (FLOOR (// L 2)))))
  (SAMPLE-VALUE (N)
    (+ .54 (* .46 (COS (// (* 2 PI N) (- L 1)))))))
```

The system HAMMING takes as input a parameter called L, the number of points in the Hamming window, and produces a numerically valued signal as its output. The signal produced from the HAMMING system for a particular value of L is given by (4.1). A particular Hamming window is generated by applying the system HAMMING to a length value, e.g. (HAMMING 255), as in Fig. 4.2, to create the 255 point Hamming window.

The general syntax of DEFINE-SYSTEM is

```
(DEFINE-SYSTEM name parameter-list
               (parent-signal-class)
  [documentation-string]
  shared-functions)
```

where a *shared-function* is specified by

```
(function-name function-parameter-list
  body )
```

DEFINE-SYSTEM defines the system *name* as taking as inputs the parameters named in *parameter-list*. The new signal class *name* is a specialization of the signal class *parent-signal-class*. For example, the system HAMMING was defined to take one input parameter, L, and produces a NUMERIC-SEQUENCE as output.

All signals produced by the system *name* can use the functions inherited from the parent signal class as well as those functions described by the *shared-functions*. Each of the shared functions is described by its *function-name*, a list of parameters to

that function and a body of code to be executed by that function. For example, the system HAMMING specifies a SUPPORT function, of no arguments, and a SAMPLE-VALUE function, of one argument, that are specific to HAMMING signals.

Within the body of the shared functions the user may reference the variables in the parameter list of the system. When a signal object is created these parameters are bound and when a shared function is called for a particular signal the variables will have the values specific to that signal. For example, when the SUPPORT function for the signal (HAMMING 255) is run the parameter L will have the value 255.

4.3.3. Defining Systems by Point Operations

In Chapter 3, in our discussion of the deferred array and the extended deferred array, we demonstrated that many signal processing operations could be defined using functions that map indices into signal values. We called this model of signals the point operator model. Signals in SPLICE can be defined using the point operator model. The definition of HAMMING

```
(DEFINE-SYSTEM HAMMING (L)
               (NUMERIC-SEQUENCE)
  "Hamming window of size L, centered on the origin"
  (SUPPORT ()
    (INTERVAL (MINUS (FLOOR (// L 2)))
              (- L (FLOOR (// L 2)))))
  (SAMPLE-VALUE (N)
    (+ .54 (* .46 (COS (// (* 2 PI N) (- L 1)))))))
```

is an example of defining a system by a point operator.

In general, to define a system according to a point operation, the programmer must specify three things: a function that maps indices into values, the interval over which the function is to be applied, and the default behavior of the signal outside this interval.

To specify the function that maps indices into values the programmer can specify a SAMPLE-VALUE clause, as was done in the definition of HAMMING.[3] Requests, from the outside, for values of a signal, using the FETCH function or the FETCH-INTERVAL function, will be translated into invocations of the SAMPLE-VALUE clause for the signal. The SAMPLE-VALUE clause will be passed a single parameter, the index at which the signal value is desired, and the body of the SAMPLE-VALUE clause should return the value of the signal at that index. SPLICE guarantees that the index passed to the SAMPLE-VALUE clause will be within the interval specified as appropriate for the point operator. SPLICE makes no guarantee about the order in which signal values will be requested (especially, no assumption of left to right processing is made).

For systems built from the signal class NUMERIC-SEQUENCE the SAMPLE-VALUE clause is used for the real-part of the value of a signal and a SAMPLE-VALUE-IMAGINARY clause can be used for the imaginary part of a signal. For example,

```
(DEFINE-SYSTEM COMPLEX-EXPONENTIAL (OMEGA)
               (NUMERIC-SEQUENCE)
  "Complex exponential exp {jωn}"
  (SAMPLE-VALUE (INDEX)
    (COS (* OMEGA INDEX)))
  (SAMPLE-VALUE-IMAGINARY (INDEX)
    (SIN (* OMEGA INDEX))))
```

defines the complex exponential $e^{j\omega n}$. If no imaginary part is specified the signal is assumed to be strictly real valued. The programmer may also specify a SAMPLE-VALUE-COMPLEX clause in place of both the SAMPLE-VALUE and the SAMPLE-VALUE-IMAGINARY clauses. The SAMPLE-VALUE-COMPLEX clause must specify how to compute both the real and the imaginary parts of a signal's value. For

[3] We will use the term clause to describe the text in a DEFINE-SYSTEM expression that defines a shared function.

example,

```
(DEFINE-SYSTEM COMPLEX-EXPONENTIAL (OMEGA)
               (NUMERIC-SEQUENCE)
  "Complex exponential exp {jωn}"
  (SAMPLE-VALUE-COMPLEX (INDEX)
    (LET ((PHASE (* OMEGA INDEX)))
      (VALUES (COS PHASE) (SIN PHASE)))))
```

also defines the complex exponential $e^{j\omega n}$. The purpose of providing SAMPLE-VALUE, SAMPLE-VALUE-IMAGINARY, and SAMPLE-VALUE-COMPLEX separately, rather than a single SAMPLE-VALUE that can return either real values or complex values, is to allow the system to be more efficient. By separating the real and imaginary parts, all arithmetic can be performed using real operations rather than complex operations.[4]

For non-periodic signals the programmer can specify the domain of the point operator by giving a SUPPORT clause, as was done in the definition of the system HAMMING. If no SUPPORT clause is given the default of $-\infty$ to ∞ is used. Outside the domain of the point operator non-periodic signals take on the default value.

For periodic signals the programmer specifies the period, using a PERIOD clause, and the domain of the point operator, using a COMPUTE-DOMAIN clause. If no PERIOD clause is specified or the value of the PERIOD clause is INF then the signal is non-periodic. If the period is not INF then the period must be an integer. If no COMPUTE-DOMAIN clause is specified and a PERIOD clause is specified then the domain of the point operator is assumed to be $0 \leqslant n < period$. For periodic signals, the point operator will get called only for indices within this interval and signal values outside this interval are determined by periodic replication. An example periodic sig-

[4] The LISP Machine is at least an order of magnitude slower performing complex arithmetic than it is performing real arithmetic. This limitation seems to be due to the choice of functions to put in microcode rather than due to any fundamental hardware limitation.

nal is

```
(DEFINE-SYSTEM DFT-SAMPLES (SEQ L)
               (NUMERIC-SEQUENCE)
  "Compute L uniformly spaced samples of the DFT of SEQ"
  (PERIOD ()
    L)
  (SAMPLE-VALUE-COMPLEX (K)
    (SUM-OF-SEQ
      (SEQ-MULTIPLY
        SEQ
        (COMPLEX-EXPONENTIAL (// (* -2 PI K) L))))))
```

which defines the periodic signal

$$X(k) = \sum_n x[n]\, e^{-j\,2\pi kn/L} \tag{4.2}$$

The SAMPLE-VALUE-COMPLEX clause will only be invoked for $0 \leqslant k < L$.

4.3.4. Defining Systems by Array Operations

Another model for signal definition is the array operator model. In this model signals are computed by functions that map intervals of indices into arrays of values. SPLICE supports the definition of systems according to array operators. To define an array operator the user must specify four things: a function mapping intervals into arrays of values, the domain of the function, the default behavior outside this interval, and specification for the types of intervals that should be used with the function. The first three requirements are similar to the requirements on point operators.

An example of a system defined by an array operator is COMPLEX-FFT

```
(DEFINE-SYSTEM COMPLEX-FFT (SEQUENCE N)
               (NUMERIC-SEQUENCE)
  "The N point FFT of SEQUENCE"
  (SUPPORT ()
    (INTERVAL 0 N))
  (COMPUTE-INTERVAL (INTERVAL)
    (IGNORE INTERVAL)
    (INTERVAL 0 N))
  (INTERVAL-VALUES-COMPLEX (INTERVAL REAL-OUTPUT-ARRAY IMAG-OUTPUT-ARRAY)
    (ARRAY-COMPLEX-FFT
      (FETCH-INTERVAL SEQUENCE (INTERVAL 0 N))
      (FETCH-IMAGINARY-INTERVAL SEQUENCE (INTERVAL 0 N))
      REAL-OUTPUT-ARRAY
      IMAG-OUTPUT-ARRAY)))
```

COMPLEX-FFT is a simplified version of a system for taking the N point FFT of a signal.[5] The SUPPORT clause of COMPLEX-FFT specifies that the output of COMPLEX-FFT has support $0 \leqslant n < N$. The computation of signal values for COMPLEX-FFT is defined using the INTERVAL-VALUES-COMPLEX clause. The INTERVAL-VALUES-COMPLEX clause is passed the interval over which the signal values are desired and two arrays. Both of the arrays are of size equal to the length of the interval and the INTERVAL-VALUES-COMPLEX clause must fill the two arrays with the real and the imaginary parts of the signal values for the indices within the interval. For COMPLEX-FFT this is done by fetching the real and imaginary parts of the input sequence and then calling the array operator ARRAY-COMPLEX-FFT, the details of which we are not concerned about here.

In general, to specify the function that maps intervals into arrays of values the programmer may specify a INTERVAL-VALUES clause, a INTERVAL-VALUES clause and a INTERVAL-VALUES-IMAGINARY clause, or a INTERVAL-VALUES-COMPLEX clause. The INTERVAL-VALUES and INTERVAL-VALUES-IMAGINARY clauses have syntax that is the obvious analog to INTERVAL-VALUES-COMPLEX. SPLICE ensures that the INTERVAL-VALUES clauses will not be called with an interval that is not entirely within the domain of the array operator. Also, SPLICE always provides arrays of the proper size for returning signal values. This feature eliminates the need for the programmer to provide bounds checks or to think about storage allocation.

To define the domain of the array operator the programmer uses the SUPPORT, PERIOD, and COMPUTE-DOMAIN clauses as they were used for point operators.

---

[5] Simplified because COMPLEX-FFT assumes that N is a power of two and that the input sequence is zero outside the interval $0 \leqslant n < N$.

SPLICE allows a signal to be fetched over any interval. However, some operators, such as COMPLEX-FFT, restrict the intervals for which it is natural to perform computation. The COMPUTE-INTERVAL clause in the definition of the COMPLEX-FFT system controls those intervals that computation will be performed over. When an interval of values is requested from a signal the COMPUTE-INTERVAL clause is passed the interval that is desired. The COMPUTE-INTERVAL clause should return the interval that really should be used for computation. The interval passed to the COMPUTE-INTERVAL clause is guaranteed to be totally within the domain of the array operator and the COMPUTE-INTERVAL clause must return either the interval passed to it or a larger one. In the definition of COMPLEX-FFT the COMPUTE-INTERVAL clause ignores the interval passed to it and returns the interval $0 \leqslant n < N$ (the entire domain of the array operator).

Array operators are useful not only for signals that are naturally defined by arrays but also for signals that are more naturally defined by point operators. For example, the system SEQ-MULTIPLY may be defined by a point operator as

```
(DEFINE-SYSTEM SEQ-MULTIPLY (SEQ-1 SEQ-2)
               (NUMERIC-SEQUENCE)
  "Point by point multiplication of SEQ-1 and SEQ-2"
  (SUPPORT ()
    (INTERVAL-INTERSECT (SUPPORT SEQ-1) (SUPPORT SEQ-2)))
  (SAMPLE-VALUE (INDEX)
    (* (FETCH SEQ-1 INDEX) (FETCH SEQ-2 INDEX))))
```

or by an array operator as

```
(DEFINE-SYSTEM SEQ-MULTIPLY (SEQ-1 SEQ-2)
               (NUMERIC-SEQUENCE)
  "Point by point multiplication of SEQ-1 and SEQ-2"
  (SUPPORT ()
    (INTERVAL-INTERSECT (SUPPORT SEQ-1) (SUPPORT SEQ-2)))
  (INTERVAL-VALUES (INTERVAL OUTPUT-ARRAY)
    (ARRAY-MULTIPLY
      (FETCH-INTERVAL SEQ-1 INTERVAL)
      (FETCH-INTERVAL SEQ-2 INTERVAL))))
```

The first definition is clearer but the second can be more efficient. The efficiency arises

because signals are often examined over intervals. For example, in taking the FFT of a signal all the values of the signal are needed simultaneously. To get an interval of signal values using the point operator model requires some overhead per index. For the array operator model there is some overhead per interval. For sufficiently large intervals, the total overhead can be significantly less. As such, most operators in SPLICE are coded as array operators.

4.3.5. Defining Systems by State Machines

A third mechanism for defining systems is to define the computation of signal values using a state machine. To define a state machine the user must specify how to initialize the state, how to update the state, and how to compute output from the state and inputs. State machines are defined using the form DEFINE-SM-SYSTEM, a specialized version of DEFINE-SYSTEM.

As an example,

```
(DEFINE-SM-SYSTEM FIRST-ORDER-FEEDBACK (INPUT ALPHA)
                  (NUMERIC-SEQUENCE)
  "First order feedback system y[n] = x[n] + α y[n-1]"
  (SM-START ()
    (START INPUT))
  (INITIAL-STATE (INDEX)
    (IGNORE INDEX)
    0)
  (CURRENT-VALUE (CURRENT-STATE INDEX)
    (+ (FETCH INPUT INDEX) (* ALPHA CURRENT-STATE)))
  (NEXT-STATE (CURRENT-STATE INDEX)
    (+ (FETCH INPUT INDEX) (* ALPHA CURRENT-STATE))))
```

defines the system FIRST-ORDER-FEEDBACK. This system takes as inputs an input signal, INPUT, and a feedback coefficient, ALPHA, and produces as output a filtered version of the input signal according to the state equations (for input x, output y, state s, and feedback coefficient α)

$$\begin{aligned} s[n+1] &= x[n] + \alpha s[n] \\ y[n] &= x[n] + \alpha s[n] \end{aligned} \tag{4.3}$$

To define the initial conditions for a state machine the programmer uses the clauses SM-START and INITIAL-STATE. The SM-START clause must return the index at which the state machine is initialized. The initial index is assumed to be zero if it is not specified. For the system FIRST-ORDER-FEEDBACK the initial index is the start of the input, not zero. The support of a signal defined by a state machine is from the initial index to ∞.

The initial state for a state machine is defined by the INITIAL-STATE clause. The INITIAL-STATE clause is passed the index for the initial state and must return the value of the initial state. The state may be represented in whatever form is convenient. For the FIRST-ORDER-FEEDBACK system the initial state is zero.

The clauses CURRENT-VALUE and NEXT-STATE describe the evolution of the system. The CURRENT-VALUE clause is also passed the current state and the current index and must return the current output value. The NEXT-STATE clause is passed the current state and the current index and must return the next state. The current value clause is called before the next state clause. In the FIRST-ORDER-FEEDBACK system both the next state and the current output have the same value.

Unlike systems defined by point operations or by array operations, SPLICE guarantees that the state update clauses of a system built from a state machine will be invoked with successive indices starting from the initial index specified by the signal. This guarantee is made without restricting the outside access of a state machine signal to a fixed ordering and is done by internally buffering the values of a state machine signal.

Although state machines are generally considered to have separate state update and output equations, as in (4.3), it is often more computationally efficient to simultaneously compute the next state and the output. For example,

```
(DEFINE-SM-SYSTEM FIRST-ORDER-FEEDBACK (INPUT ALPHA)
                  (NUMERIC-SEQUENCE)
  "First order feedback system y[n] = x[n] + α y[n-1]"
  (SM-START ()
    (START INPUT))
  (INITIAL-STATE (INDEX)
    (IGNORE INDEX)
    0)
  (SM-UPDATE (CURRENT-STATE INDEX)
    (LET ((OUTPUT-VALUE
            (+ (FETCH INPUT INDEX) (* ALPHA CURRENT-STATE))))
      (VALUES OUTPUT-VALUE OUTPUT-VALUE))))
```

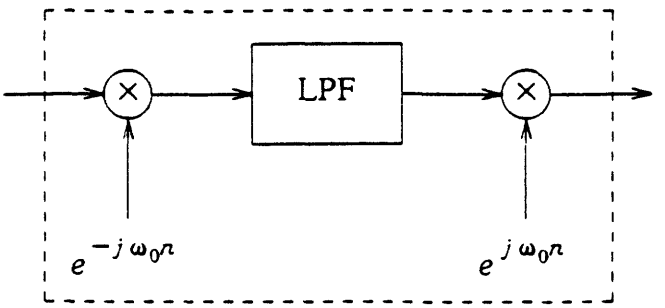
illustrates an alternative version of the system FIRST-ORDER-FEEDBACK. In this definition the two clauses CURRENT-VALUE and NEXT-STATE have been replaced with the single clause SM-UPDATE. The SM-UPDATE clause is passed two arguments, the current state and the current index, and must return both the current value and the next state.

4.3.6. Defining Systems by Composition

The final way of defining systems is by composing existing systems to form new ones. The system composition function is DEFINE-COMPOSITION. For example, the system BPF-FROM-LPF, defined by

```
(DEFINE-COMPOSITION BPF-FROM-LPF (SEQUENCE LPF CENTER-FREQ)
  "Create a bandpass filter from a lowpass filter"
  (SEQ-MULTIPLY
    (COMPLEX-EXPONENTIAL CENTER-FREQ)
    (SEQ-CONVOLVE (SEQ-MULTIPLY SEQUENCE
                                (COMPLEX-EXPONENTIAL (MINUS CENTER-FREQ)))
                  LPF)))
```

implements a bandpass filter given an input signal, a lowpass filter impulse response, and a center frequency. Fig. 4.6 illustrates the block diagram for this system.



BPF-FROM-LPF

Fig. 4.6. Bandpass filter from lowpass filter.

The general syntax for DEFINE-COMPOSITION is

```
(DEFINE-COMPOSITION name  parameter-list
  [documentation ]
  equivalent-expression
  shared-functions)
```

DEFINE-COMPOSITION defines the system *name*, taking as inputs the parameters named in the *parameter-list*. The *equivalent-expression* defines how to implement *name*. Evaluation of the equivalent expression must return a signal object.

Shared functions for DEFINE-COMPOSITION provide a way of adding extra information to a system composition. For example,

```
(DEFINE-COMPOSITION COS-SIGNAL (FREQ)
  "Cosine wave of specified frequency"
  (SEQ-SCALE .5 (SEQ-ADD (COMPLEX-EXPONENTIAL FREQ)
                         (COMPLEX-EXPONENTIAL (MINUS FREQ))))
  (ATOMIC-TYPE ()
    :REAL))
```

defines the system COS-SIGNAL for generating cosine waves of a specified frequency according to

$$\cos(\omega n) = \frac{e^{j\omega n} + e^{-j\omega n}}{2} \qquad (4.4)$$

and adds the additional information that the output is real not complex.

In addition to providing a simple mechanism for extending the number of available systems, system composition provides a mechanism for the development of *generic* systems. A generic system is a system that can accept more than one type of input signal and will produce an output signal appropriate to its input signals. For example,

```
(DEFINE-COMPOSITION FFT (SEQUENCE N)
  "Perform the N point FFT of the input SEQUENCE"
  (SELECTQ (ATOMIC-TYPE SEQUENCE)
    (:COMPLEX (COMPLEX-FFT SEQUENCE N))
    (:REAL (REAL-FFT SEQUENCE N))))
```

defines the system FFT, a system for taking the N point FFT of its input signal, SEQUENCE. The system FFT is built assuming there already exist two other systems, COMPLEX-FFT, for taking the FFT of a complex-valued signal, and REAL-FFT, for taking the FFT of a real-valued signal. The FFT system determines if the input signal is real-valued or complex-valued and chooses the implementation appropriate to the input signal.

Generic systems, such as FFT, provide an abstraction mechanism by which the user of a generalized system is insulated from the underlying implementation. This buffering of the user from the true implementation allows different algorithms to be used in different cases according to efficiency needs. It also allows a more elegant handling of special cases. Furthermore, as we will discuss in Chapter 5, the idea of a generic system that can examine its inputs to determine the proper implementation can be extended to the idea of symbolic manipulation of signal processing expressions.

4.4. Signal Object Behavior

We have described the outside view of signals as objects and the inside view for defining signals. In this section we describe the behavior of signal objects in terms of immutability, deferred evaluation, and caching.

4.4.1. Immutability

Signal objects in SPLICE are immutable. None of the operations we have described for manipulating a signal or for examining a signal change any observable properties of the signal.

4.4.2. Deferred Evaluation

The model of signals presented in Chapter 3 and in the previous sections of this chapter all describe signals by some form of computational function. We argued in Chapter 3 that the definition of a signal is different than the computation of the signal's values. Signals exhibit this property of deferred evaluation.

The values of signals are not computed at the time signals are created; they are calculated only as they are required. A request for a signal value causes its calculation. This calculation may require values from other signals and will cause these other values also to be calculated as needed.

Signal values may also be calculated as the side effect of some operations. For example, requesting any one sample of an FFT causes calculation of all the samples. Similarly, requesting any sample of the output from a state machine causes the state machine to calculate all its values from the starting index up to the required index. Also, signal values may be calculated so that caching can be simplified. We describe this in section 4.5.2.

Not only is the calculation of signal values deferred, SPLICE defers the calculation of some signal properties until they are needed. The support, period, and atomic-type of a signal are not determined unless they are explicitly requested. Similarly, the equivalent expression in a system defined by composition is not evaluated until it is required. Not only does deferral of these calculations potentially save computation, as we will discuss in Chapters 5 and 6, it allows us to take advantage of powerful inference techniques in reasoning about signal processing signals and systems.

4.4.3. Caching

A signal remembers those values of the signal that have been computed. When the value of signal at some index is requested the signal first checks if it has already computed the value at that index. If so, then the stored value is returned. Otherwise, the value at that index is computed, stored in signal memory, and returned.

Signal values are only stored for the portion of the signal that is computed by some function. Those portions of the signal that are determined by a default value are not cached. Also, caching can be disabled on an individual signal or on an entire signal class.

In addition to caching the values of signals, SPLICE caches the values of systems, i.e signal objects. When a system is invoked with a set of parameter values the system first checks if it has already been invoked with those same parameter values. If so, then the system returns the same signal object that it had previously created. Thus, when the system HAMMING is first invoked with its parameter, L, set to 255, by (HAMMING 255), the HAMMING system creates a new signal object and remembers that signal along with the parameter value of 255. All future times that the HAMMING system is invoked with its parameter set to 255 the HAMMING system returns

the same signal object.

Caching of signal objects gives signals unique identities and reduces the computational burden because as values of a signal are calculated and cached the values become accessible in all places that use the signal.

4.5. Implementation Considerations

This section discusses those issues of implementation of a numerical signal representation that are specific to SPLICE. In particular, we discuss efficiency, caching of signal values and signal objects, and array management.

4.5.1. Computational Efficiency

To provide a good numerical signal processing environment, SPLICE attempts to maintain a high degree of computational efficiency. The primary mechanism by which computational efficiency is maintained is extensive use of arrays and array operators. Arrays can be accessed quickly and the use of arrays encourages simultaneous computation of many signal values, which reduces the overhead of the signal representation. Typical overhead for fetching an array of samples of a signal is between 20% and 100%. This overhead involves checking for previously computed values, doing some bounds checking, allocating some storage, and copying some values. A simple signal, such as the COMPLEX-FFT, which implicitly windows its input signal for $0 \leqslant n < N$, has only as small amount of overhead, but a more complicated FFT would have more overhead.

Caching compensates for some of the overhead of the signal representation. Since signals have unique identities and remember previously computed values, duplicate calculations are eliminated.

Array operators are, of course, coded to be as efficient as possible. It is worth noting that programming in ZETALISP on the LISP Machine does not put a penalty on the execution speed of signal processing operators compared with more conventional languages, such as FORTRAN or C. For example, an FFT array operator on the LISP Machine requires 120 milliseconds to do a 1024 point complex single precision FFT, as compared to 210 milliseconds for the MIT Digital Signal Processing Group's best FFT routine on a VAX750.[6]

4.5.2. Caching

The caching of signal values is done by associating with every signal an array and an interval. The array stores the previously computed values of the signal and the interval records the range of indices for which values are stored. SPLICE uses the strategy that all signal values should be remembered as they are computed and signals should be cached over a single connected interval. The remembering of all signal values ensures fast access to all previously computed signal values. The use of a single array to cache signal values over a single connected interval reduces the overhead of looking for previously computed signal values to simple interval calculations. Also, the strategy of caching over a single connected interval fits well with typical patterns of access to signal values. However, signals that are sampled sparsely do not fit this model well and the result is that some signal values that are not explicitly requested are computed. For signals of this type, the user can disable the caching of signal values.

---

[6] Symbolics 3670 LISP Machine with FPA (no instruction fetch unit) and 1M word of memory and a DEC VAX750 with FPA and 6M byte memory. VAX routine written in C and compiled for single precision (no conversion to double).

The caching of signal objects as the output of systems is implemented by associating a hash table with each system. When a set of parameter values is passed to a system, the system uses a list made from these parameter values as the key for hash table lookup. If a signal is found in the hash table the signal is returned by the system. Otherwise, a new signal object is created, entered into the hash table with the appropriate key, and returned by the system.

4.5.3. Array Management

SPLICE makes extensive use of arrays for caching signal values and for the extraction of signal values and, as such, the allocation and deallocation of storage for arrays becomes a major issue. The LISP Machine environment provides facilities for garbage collection but arrays are allocated and deallocated at such a high rate and in such a regular fashion that the general purpose garbage collection algorithm is inefficient. Therefore, facilities for the explicit allocation and deallocation of arrays are provided.

Array storage is maintained by providing a resource from which arrays are allocated, and to which arrays are returned. The resource for allocating and saving arrays maintains an internal data structure containing all the unallocated arrays. When an array is requested either an array of the correct size is taken from the resource or a new one is constructed.

To simplify the user's view of array storage, special forms for using arrays are provided. These special forms are USING-ARRAY, WITH-SEQ-ARRAY, WITH-IMAGINARY-SEQ-ARRAY, and WITH-COMPLEX-SEQ-ARRAY. These forms provide scoped usage of arrays with automatic allocation and deallocation. The syntax for USING-ARRAY is

(USING-ARRAY (*array-name array-size*)
body)

USING-ARRAY allocates an array of size *array-size* and binds the variable *array-name* to the array. The array can be used within the body and is automatically deallocated after.

The WITH-SEQ-ARRAY special form provide a mechanism for accessing an array of signal values. The syntax of WITH-SEQ-ARRAY is

(WITH-SEQ-ARRAY (*array-name signal interval*)
body)

In the body the named array contains the values of the signal over the specified interval (by (FETCH-INTERVAL *signal interval*)). The array can be used within the body, including being written into, and is automatically deallocated after. Using the complex version of this form, WITH-COMPLEX-SEQ-ARRAY, we can create a version of COMPLEX-FFT that does not use up storage

```
(DEFINE-SYSTEM COMPLEX-FFT (SEQUENCE N)
               (NUMERIC-SEQUENCE)
  "The N point FFT of SEQUENCE"
  (SUPPORT ()
    (INTERVAL 0 N))
  (COMPUTE-INTERVAL (INTERVAL)
    (IGNORE INTERVAL)
    (INTERVAL 0 N))
  (INTERVAL-VALUES-COMPLEX (INTERVAL REAL-OUTPUT-ARRAY IMAG-OUTPUT-ARRAY)
    (WITH-COMPLEX-SEQ-ARRAY (REAL-INPUT-ARRAY IMAG-INPUT-ARRAY
                             SEQUENCE (INTERVAL 0 N))
      (ARRAY-COMPLEX-FFT REAL-INPUT-ARRAY IMAG-INPUT-ARRAY
                         REAL-OUTPUT-ARRAY IMAG-OUTPUT-ARRAY))))
```

Caching of signal values also uses arrays. The allocation and deallocation of arrays for the caching of signal values is automatically controlled. However, the user may use the function UNCACHE to remove the cache from a signal. Also, the user may use the special form WITH-UNCACHING, as (WITH-UNCACHING *body*), to execute the body and to automatically uncache all signals created in the body when the body is exited.

4.6. Extended Examples

In this section we illustrate the power of SPLICE by showing how to define overlap-add and overlap-save convolution.

Overlap-add and overlap-save convolution are examples of block convolution algorithms. These algorithms break an input signal up into sections, circularly convolve each section with a convolution kernel, and reassemble the output. Given a signal $x[n]$ and an impulse response $h[n]$, overlap-add convolution forms $y[n] = x[n] * h[n]$ by breaking the signal $x[n]$ into non-overlapping sections of length L

$$x_k[n] = \begin{cases} x[n + k \cdot L] & 0 \leqslant n < L \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

Each section $x_k[n]$ is convolved with $h[n]$, $y_k[n] = x_k[n] * h[n]$, and $y[n]$ is formed by

$$y[n] = \sum_k y_k[n - k \cdot L] \tag{4.6}$$

To program overlap-add convolution we need a routine for circular convolution of two signals. One possible routine is CIRCULAR-CONVOLVE, defined as

```
(DEFINE-COMPOSITION CIRCULAR-CONVOLVE (X-SECTION H N)
  "Perform circular convolution, of length N, of X-SECTION and H.
  (IDFT (SEQ-MULTIPLY (DFT X-SECTION N) (DFT H N)) N))
```

Given CIRCULAR-CONVOLVE, we can define OVERLAP-ADD-CONVOLVE as

```
(DEFINE-SYSTEM OVERLAP-ADD-CONVOLVE (X H N L)
               (NUMERIC-SEQUENCE)
  "Overlap-add convolution of X and H using N point circular
convolutions and L point sections.  Assumes H starts at 0 and N is
bigger than L and bigger than the length of H."
  (SUPPORT ()
    (INTERVAL (START X)
              (+ (END X) (END H) -1)))
  (SAMPLE-VALUE (INDEX)
    (LET ((BLOCK (FLOOR (// INDEX L)))
          (OFFSET (MOD INDEX L)))
      (LOOP FOR K FROM BLOCK BY -1
            FOR OFF FROM OFFSET BY L
            WHILE (< OFF N)
            SUMMING
        (FETCH (CIRCULAR-CONVOLVE (SECTION X (* K L) L) H N) OFF)))))
```

where SECTION is

```
(DEFINE-SYSTEM SECTION (X START SIZE)
               (NUMERIC-SEQUENCE)
  "Section out SIZE points of X starting at START"
  (SUPPORT ()
    (INTERVAL 0 SIZE))
  (SAMPLE-VALUE (INDEX)
    (FETCH X (+ START INDEX))))
```

The point operator for OVERLAP-ADD-CONVOLVE takes the index that is requested and sums the appropriate samples from the appropriate partial convolutions. The code is written assuming nothing about the number of sections that will be required to be examined.

Overlap-save convolution can be defined similarly to overlap-add convolution. The differences are that the input is broken into overlapping sections, rather than non-overlapping sections, and the output is formed by taking pieces from the partial convolutions, rather than by summing. OVERLAP-SAVE-CONVOLVE is given by

```
(DEFINE-SYSTEM OVERLAP-SAVE-CONVOLVE (X H N L)
               (NUMERIC-SEQUENCE)
  "Overlap-save convolution of X and H using N point circular
convolutions and L point shift between sections.  Assumes H starts at
0 and N is bigger than L and bigger than the length of H."
  (SUPPORT ()
    (INTERVAL (START X)
              (+ (END X) (END H) -1)))
  (SAMPLE-VALUE (INDEX)
    (LET ((OVERLAP (- N L)))
      (LET ((BLOCK (FLOOR (// (- INDEX OVERLAP) L)))
            (OFFSET (MOD (- INDEX OVERLAP) L)))
        (FETCH (CIRCULAR-CONVOLVE (SECTION X (* BLOCK L) N) H N)
               (+ OFFSET OVERLAP))))))
```

The definitions of OVERLAP-ADD-CONVOLVE and OVERLAP-SAVE-CONVOLVE show that complicated signal processing operations can be defined easily in SPLICE.

4.7. Evaluation of SPLICE

The goals of SPLICE were to develop a numerical signal representation that reflected the way we think about signal processing and that supported our work in KBSP. SPLICE currently contains nearly 750,000 bytes of ZETALISP code and has over 200 systems defined for general use.

4.7.1. As a Signal Representation

SPLICE attempts to provide a numerical signal representation that is close to the model proposed in Chapter 3. In many ways SPLICE achieves this goal. Signal objects have consistent observable properties, corresponding to those observable properties that are necessary for a good signal representation, and signal objects can be defined using common computational models.

The main observable properties of signal objects, and the functions for observing these properties, are summarized in Fig. 4.7. Signals may be examined at any index, using FETCH, or over any interval, using FETCH-INTERVAL. The non-zero extent, default value, period, and atomic-type of a signal can be determined. Because signal objects, regardless of their origin, can be accessed uniformly the programming process

Signal Values - at any index from $-\infty$ to ∞

- FETCH - return signal value at an index
  also FETCH-IMAGINARY and FETCH-COMPLEX
- FETCH-INTERVAL -return an array of signal values for an interval
  also FETCH-IMAGINARY-INTERVAL and FETCH-COMPLEX-INTERVAL

Signal Properties

- SUPPORT - interval of non-zero extent
- DEFAULT-VALUE - value in regions not specified by a function
- PERIOD - period of signal
- ATOMIC-TYPE - real or complex valued

Fig. 4.7. Observable properties of signal objects.

is greatly simplified.

The methods for defining systems are summarized in Fig. 4.8. Systems may be defined using point operators, array operators, state machines, or by system composition. Signals defined by any of these methods have the correct outside view. The ability to choose among different methods for defining systems make programming signal processing operations easier. Also, because the outside view operators are well matched to the inside views of defining signals, particularly the ability to get both single sample values and arrays of sample values, the building of systems using the appropriate model is simplified.

In addition to providing a consistent outside view of signals and many inside views of signals, SPLICE provides other useful features for signals. These feature are immutability, deferred evaluation, and caching and are summarized in Fig. 4.9.

Finally, SPLICE makes some implementation decisions that help to make it a more useful tool. These decisions are summarized in Fig. 4.10. Most systems are

Point Operators - Compute Sample at an Index - Use DEFINE-SYSTEM

- SUPPORT - specify non-zero extent
- PERIOD - specify signal period
- COMPUTE-DOMAIN - specify period to be replicated
- SAMPLE-VALUE - specify how to compute signal value at an index
  also SAMPLE-VALUE-IMAGINARY and SAMPLE-VALUE-COMPLEX

Array Operators - Compute Samples over an Interval - Use DEFINE-SYSTEM

- SUPPORT - specify non-zero extent
- PERIOD - specify signal period
- COMPUTE-DOMAIN - specify period to be replicated
- COMPUTE-INTERVAL - specify intervals to use for array operator
- INTERVAL-VALUES how to compute signal values over an interval
  also INTERVAL-VALUES-IMAGINARY and INTERVAL-VALUES-COMPLEX

State Machine - Compute Samples Sequentially - Use DEFINE-SM-SYSTEM

- SM-START - specify initial index for state machine
- INITIAL-STATE - specify initial state
- CURRENT-VALUE - output equation
- NEXT-STATE - state update equation
  also SM-UPDATE

System Composition - Piece together Systems - Use DEFINE-COMPOSITION

- Define an expression
- Add extra information

Fig. 4.8. Methods of defining systems.

Immutability

- No operations may modify signals

Deferred Evaluation

- Allows infinite extent signals
- Potential reduction in work

Caching

- Saving signal values reduces work
- Saving signal objects provides unique identities to signals

Fig. 4.9. Other important properties of signals.

Computational Efficiency

- Most systems are coded with array operators

Caching

- Single buffer
- Cache over a connected interval

Array Management

- Array management resource
- Scoped forms - USING-ARRAY, WITH-SEQ-ARRAY, and WITH-UNCACHING

Fig. 4.10. Implementation decisions.

coded with array operators for efficiency. Caching is performed in a simple, but powerful, method, and arrays storage is explicitly managed.

SPLICE has some shortcomings as a signal representation mechanism. Signals are restricted to being one dimensional, discrete-time signals. SEQUENCES-OF-SEQUENCES provide some facility with multiple dimensions but a true representation

for multi-dimensional signals would be useful. Also, extension to continuous-time signals would be useful. The signal representations that are the most natural in SPLICE, point operations, array operations, state machines, and system composition, are most useful for signal processing problems involving algorithm design and testing. SPLICE is not well suited to hardware simulation of signal processing algorithms or as a tool for studying parallel processing in signal processing.

The use of a single interval for computation and either a default value or periodic replication outside this interval is not the most general possible representation. More general methods, such as those described in Chapter 3, would be useful.

4.7.2. As a Tool for Signal Processing

Our experience with SPLICE is that it provides a flexible and powerful tool for signal processing in a symbolic processing environment. There are over 200 signal processing systems available in basic SPLICE and over a hundred more developed by individual users for particular problem domains. A list of some of the systems defined in SPLICE is given in Fig. 4.11. These systems were designed by different users without much discussion among the users about the interface between systems. This was made possible by a signal representation that provided a consistent interface to all signals.

All the numerical signal processing in our Knowledge Based Signal Processing work is run using software developed from SPLICE. Also, some strictly numerical signal processing has been done with SPLICE. It has been our observation that program development time is from two to seven times faster using SPLICE than using the Digital Signal Processing Group's VAX/UNIX programming environment.

Sequence Generators

- Hamming, Hanning, Bartlett, Rectangular Windows
- Impulse, Unit step, Complex Exponential, Cosine, Sine, Constant
- White noise
- Read from file

Basic Single Sequence Operations

- Scale, Shift, Reverse, Alias, Upsample, Downsample
- Real part, Imaginary part, Conjugate, Magnitude, Phase
- FIR filtering, IIR filtering, Autocorrelation
- Smooth, Median filter
- FFT, IFFT

Basic Multiple Sequence Operations

- Add, Subtract, Multiply, Divide
- Convolve, Correlate

Composed Systems

- Cepstrum
- Overlap-save, Overlap-add

Special Purpose Systems

- FIR filter design
- LPC spectrum, Inverse filter, LPC synthesis
- Formant track, Autocorrelation pitch detector
- Periodogram spectral analysis
- MLM spectral estimation
- Spectrogram, LPC spectrogram

Fig. 4.11. Some systems available in SPLICE.

CHAPTER 5

Signal Representations for Symbolic Processing

This chapter discusses the representation of signals for symbolic processing in computer programs. Symbolic processing of signals, in contrast to numerical processing of signals, is the manipulation of signal forms and signal representations and the determination of signal properties rather than the computation of signal values. The goal of symbolic manipulation of signals is the performance of signal processing tasks normally performed by humans.

We begin our discussion with an example of the manipulation of signals as performed by an experienced signal processor. The purpose of this section is not the exposition of new signal manipulation techniques but a description of some of the signal manipulations that humans perform that we would like to be able to capture in a system for the symbolic manipulation of signals.

From our example we identify three classes of symbolic manipulations of signals. These are the symbolic analysis of signal properties, such as support, bandwidth, and symmetry, the rearrangement of signal processing expressions, such as transforming a time domain convolution into a frequency domain convolution, and the manipulation of signals of a continuous variable, such as continuous-frequency signals.

How our representation of signals for numerical processing must be modified to incorporate symbolic manipulation is then discussed. We first describe those features of signals and systems that must be represented to support the symbolic manipulation of signals. These features include the explicit representation of signal type and

history, signal usage context, signal properties, such as non-zero extent, periodicity, and symmetry, system properties, such as linearity, shift-invariance, and memory-lessness, and computational costs.

Next we describe two extensions to our model of signal objects that must be explicitly represented for the symbolic manipulation of signals. These are signals of a continuous variable and abstract signal classes. Signals of a continuous variable include continuous-time and continuous-frequency signals and are used in signal expression rearrangement. Abstract signal classes describe collections of signals rather than individual signals and are used in the symbolic manipulation of signal properties.

This chapter makes two contributions - identification of three classes of symbolic manipulations that can be performed on signals and description of the requirements symbolic manipulation of signals imposes on the representation of signals. The material presented in this chapter is conceptual in nature. Chapter 6 will describe the Extended Signal Processing Language and Interactive Computing Environment (E-SPLICE), an extended version of SPLICE that incorporates our ideas about the symbolic manipulation of signals and has the ability to perform the manipulations described here.

5.1. An Example of Signal Manipulation

In this section we give an extended example of the analysis and rearrangement of signals as performed by an experienced signal processor. The purpose of this section is not the exposition of new signal analysis methods but the explanation of a signal analysis example to show the information and manipulations used. The manipulations presented here are of the type that we would like to be able to capture in a computer system for the symbolic manipulation of signals. To emphasize the idea that these

manipulations can be expressed in a computer we will use LISP notation in addition to standard mathematical notation.

The system we will manipulate is shown in Fig. 5.1. An input signal, $x[n]$, is upsampled by a factor L by insertion of L-1 zeros between each of its samples and the result is passed through a linear, shift-invariant filter with impulse response $h[n]$ to get the signal $y[n]$. The output of the overall system in Fig. 5.1 is $Y(e^{j\omega})$. Mathematically,

$$x_L[n] = \begin{cases} x\left[\frac{n}{L}\right] & n \bmod L = 0 \\ 0 & n \bmod L \neq 0 \end{cases} \tag{5.1a}$$

$$y[n] = x_L[n] * h[n] \tag{5.1b}$$

$$Y(e^{j\omega}) = \sum_n y[n]\, e^{-j\omega n} \tag{5.1c}$$

or, in the language of the signal representation of Chapter 3,

```
(DEFINE XL (UPSAMPLE X L))
(DEFINE Y (CONVOLVE XL H))
(DEFINE Y-FT (FT Y))
```

One type of manipulation that can be performed on the system of Fig. 5.1 is the analysis of the properties of the output signal $Y(e^{j\omega})$ and the intermediate signals $x_L[n]$ and $y[n]$ from the properties of the input signals $x[n]$ and $h[n]$. For example,

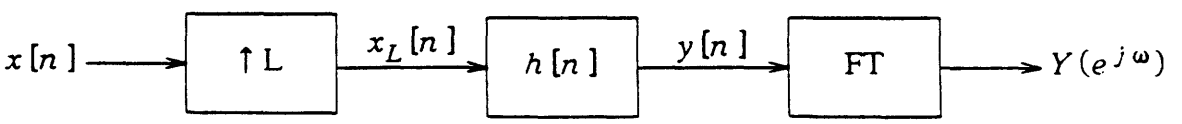


Fig. 5.1. Signal processing system for manipulation.

if $x[n]$ is zero outside the interval $S_x \leqslant n < E_x$ and $h[n]$ is zero outside the interval $S_h \leqslant n < E_h$ we can use our knowledge of the properties of upsampling and convolution to determine that $x_L[n]$ is zero outside the interval $LS_x \leqslant n < L(E_x - 1) + 1$ and $y[n]$ is zero outside the interval $LS_x + S_h \leqslant n < L(E_x - 1) + E_h$. The rules used in this analysis can be written in a LISP-like rule syntax as[1]

```
(START (UPSAMPLE ?X ?L) = (* ?L (START ?X))
(END (UPSAMPLE ?X ?L)) = (+ (* ?L (- (END ?X) 1)) 1)

(START (CONVOLVE ?X ?H)) = (+ (START ?X) (START ?H))
(END (CONVOLVE ?X ?H)) = (+ (END ?X) (END ?H) -1)
```

Similarly, if $x[n]$ is real-valued and $h[n]$ is real-valued, then we can determine that $x_L[n]$ is real-valued, $y[n]$ is real-valued, and $Y(e^{j\omega})$ is conjugate symmetric

```
(REAL-OR-COMPLEX (UPSAMPLE ?X ?L)) = (REAL-OR-COMPLEX ?X)

(IF (AND (EQ (REAL-OR-COMPLEX ?X) :REAL)
         (EQ (REAL-OR-COMPLEX ?H) :REAL))
    (REAL-OR-COMPLEX (CONVOLVE ?X ?H)) = :REAL
    (REAL-OR-COMPLEX (CONVOLVE ?X ?H)) = :COMPLEX)

(IF (EQ (REAL-OR-COMPLEX ?X) :REAL)
    (SYMMETRY (FT ?X)) = :CONJUGATE-SYMMETRIC)
```

Another type of manipulation that can be performed on the system of Fig. 5.1 is rearrangement of the system to alternative forms, i.e. ones that compute the same output using different operations. For example, the rule that the Fourier transform of a convolution is the product of Fourier transforms

```
(FT (CONVOLVE ?X ?H))  <==>  (MULTIPLY (FT ?X) (FT ?H))
```

can be used to determine that

$$Y(e^{j\omega}) = X_L(e^{j\omega})H(e^{j\omega}) \tag{5.2}$$

as illustrated in Fig. 5.2.

---

[1] We will not formally define the LISP-like rule language we use here. We only note that *?variable* is used to denote a variable in a rule and we embellish LISP syntax to include the infix operators =, ==>, and <==>, representing equal to, implies, and equivalence. A more complete rule syntax is found in Chapter 6.

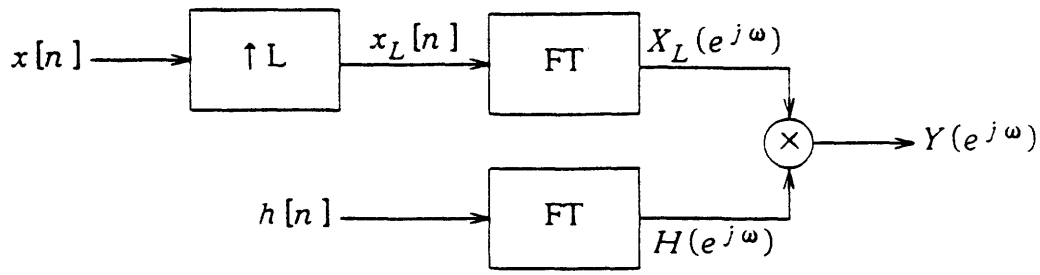


Fig. 5.2. Rearrangement for computation of Fourier transform.

In a similar way, we can rearrange the calculation of the Fourier transform of $x_L[n]$ using the rule that the Fourier transform of an upsampled signal is the Fourier transform of the original signal with a change of scale

$$X_L(e^{j\omega}) = X(e^{j\omega L}) \tag{5.3}$$

or

(FT (UPSAMPLE ?X ?L)) <==> (SCALE-AXIS (FT ?X) ?L)

as illustrated in Fig. 5.3

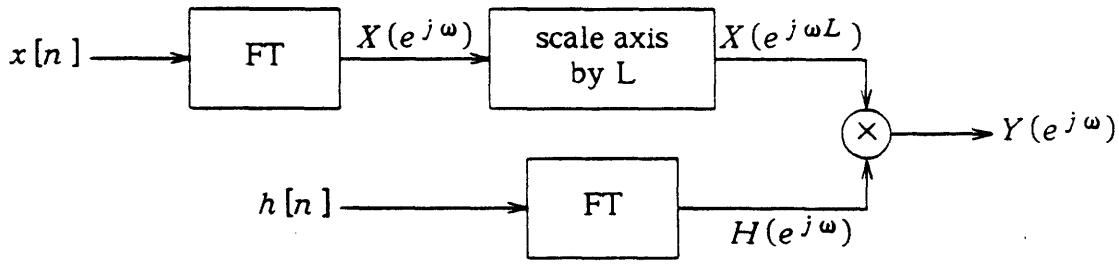


Fig. 5.3. Another rearrangement for computation of Fourier transform.

Another rearrangement of the system of Fig. 5.1 is into the polyphase network [37] shown in Fig. 5.4. As an experienced signal processor we recognize that upsampling followed by filtering can be turned into the polyphase structure.

(CONVOLVE (UPSAMPLE ?X ?L) ?H) <==> (POLYPHASE-UPSAMPLE ?X ?H ?L)

Alternatively, we would be able to find this rearrangement by breaking the signal $h[n]$ into L interleaved pieces

$$h[n] = \sum_{i=0}^{L-1} h_i[n-i] \tag{5.4a}$$

with

$$h_i[n] = \begin{cases} h[n+i] & n \bmod L = 0 \\ 0 & n \bmod L \neq 0 \end{cases} \tag{5.4b}$$

then writing $y[n]$ as L interleaved convolutions, and simplifying the result.

A final manipulation that we will consider will be analysis of the networks of Fig. 5.1 and Fig. 5.4 to determine which is a more efficient method for computing the

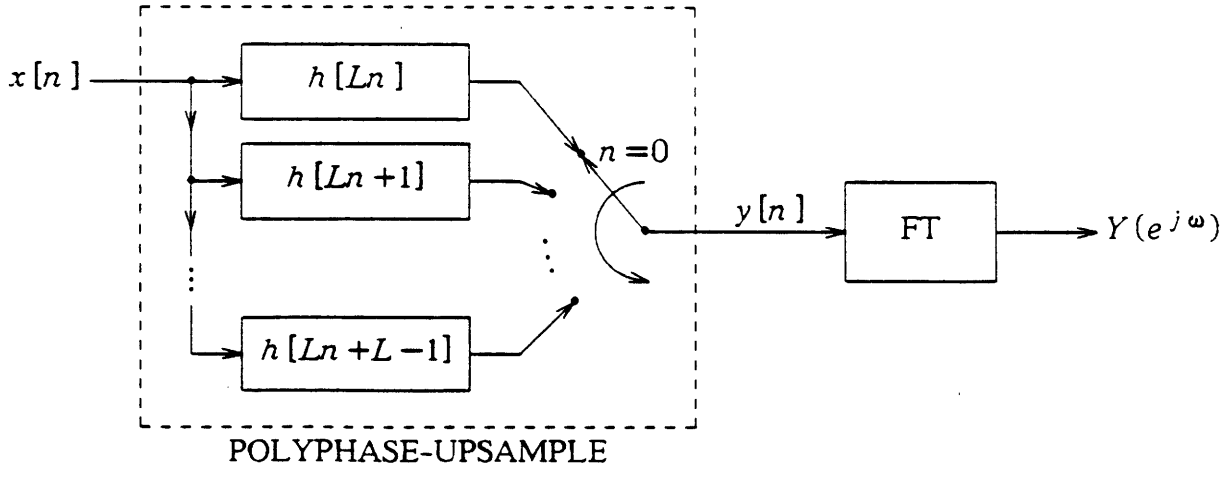


Fig. 5.4. Rearrangement to use polyphase structure.

signal $y[n]$. If we assume that the filter $h[n]$ of Fig. 5.1 is FIR, implemented in a direct form, and is N_h points long then we can use our knowledge of the implementation of FIR filters to determine that computation of $y[n]$, in this network, uses N_h multiplies and $N_h - 1$ additions per sample.

```
(MULTIPLIES-PER-SAMPLE (CONVOLVE ?X ?H)) = (LENGTH ?H))
(ADDITIONS-PER-SAMPLE (CONVOLVE ?X ?H)) = (- (LENGTH ?H) 1)
```

The cost of computing a single sample of $y[n]$ using the network of Fig. 5.4 depends on which sample of $y[n]$ is requested. However, we can determine the average cost by assuming each of the filters $h[Ln+i]$ is implemented in a direct form, determining the cost of computing one sample from each of these filters, and averaging these L costs.

```
(MULTIPLIES-PER-SAMPLE (POLYPHASE-UPSAMPLE ?X ?H ?L)) =
  (/ (LOOP FOR I FROM 0 BELOW ?L
           SUMMING
        (MULTIPLES-PER-SAMPLE (CONVOLVE ?X (DOWNSAMPLE (ADVANCE ?H ?I) ?L))))
     ?L)
```

Doing this we find that the network of Fig. 5.4 costs, on average, N_h / L multiplies and $(N_h / L) - 1$ additions per sample of $y[n]$.

Some comments should be made about the types of manipulations illustrated in this example. First, the manipulations shown here do not use information about the numerical values of the signals. The information that is being manipulated is descriptive information about signals. Secondly, the properties used to describe signals are interrelated. For example, the symmetry of $Y(e^{j\omega})$ depends not on the symmetry of the input signals but on the real or complex-valuedness of the input signals. Thirdly, we used continuous-frequency signals along with discrete-time signals. Finally, the manipulations shown here use multiple steps. Determination of the symmetry of $Y(e^{j\omega})$ involved two intermediate signals, and determination of the computational

cost in the polyphase network involved the averaging of costs of subpieces of the network.

5.2. Symbolic Manipulation of Signals

Symbolic manipulation of signals is the representation and manipulation of signals and systems using symbolic information rather than numerical signal values. In this section we identify three classes of manipulation of signals that are important to be able to do. These are the analysis of signal properties, the rearrangement of signal processing expressions, and the manipulation of signals of a continuous variable.

Symbolic manipulation of signals is somewhat analogous to, but distinctly different from, algebraic manipulation, as typified by the MACSYMA system [11]. Algebraic manipulation systems emphasize the analytic form of expressions while symbolic manipulation of signals concentrates more on the properties of signals and systems. Typical problems solved by MACSYMA are evaluation of integrals, such as

$$\int \frac{ae^x}{ae^x + b} dx = \log(ae^x + b) \tag{5.5}$$

or solution of equations, such as

$$\frac{kx}{y} = \frac{k}{h} \tag{5.6}$$

for y to get

$$y = xh \tag{5.7}$$

MACSYMA has many capabilities and contains much expertise on the manipulation of expressions but has no knowledge specific to signal processing. For example, MACSYMA does not have a Fourier transform operator or a convolve operator, nor does it understand discrete-time signals. Also, MACSYMA does not process properties of the objects it manipulates. For example, if x is an even integer and h is an integer

in (5.7) MACSYMA can not determine that y is an even integer. In contrast, as we will show, symbolic manipulation of signals would take equation (5.7), in which y, x, and h, were signals, and would allow us to determine the symmetry of y in terms of the symmetries of x and h.

5.2.1. Manipulation of Signal Properties

The computation and manipulation of the properties of signals without numerical calculation on signal values is a primary aspect to the symbolic manipulation of signals. For instance, in the example of section 5.1, we analyzed the signal $y[n]$ to determine some of its properties from properties of the input signals $x[n]$ and $h[n]$. We determined the non-zero extent of $y[n]$ from the non-zero extents of $x[n]$ and $h[n]$ and the real or complex-valuedness of $y[n]$ from the real or complex-valuedness of $x[n]$ and $h[n]$. These are examples of symbolic manipulations, even though analysis of non-zero extent involve numbers and even though non-zero extent was used in our numerical representation, because the information that is being examined and propagated does not involve numerical calculation on signal values.

Fig. 5.5 lists some well known signal properties that are manipulated in the analysis of signals. Not all these properties are useful to determine for all signals in all situations but all these properties provide useful information about signals. For example, our numerical representation of signals used the properties of non-zero extent, periodicity, and real or complex-valuedness. The non-zero extent of a signal was used to limit the range of function applicability, the period of signal was used to reduce computation, and the real or complex-valuedness was used in algorithm selection, e.g. choice of an FFT for complex signals or one for real signals.

- Non-zero extent
- Period
- Real or Complex
- Symmetric, Antisymmetric, Conjugate symmetric, Conjugate antisymmetric
- Linear Phase
- Sampling rate, Bandwidth
- Minimum Phase, Maximum Phase
- All-pole, All-zero
- Stable, Causal, Anticausal
- Signal to noise ratio

Fig. 5.5. Signal Properties.

5.2.2. Rearrangement of Signal Processing Expressions

The rearrangement of signal processing expressions to determine equivalent ways of expressing the same computation is a fundamental symbolic manipulation. For example, in the example of section 5.1 we manipulated the original system of Fig. 5.1, into the three different forms of Figs. 5.2, 5.3, and 5.4 using rules about the Fourier transform of a convolution,

```
(FT (CONVOLVE ?X ?H))  <==>  (MULTIPLY (FT ?X) (FT ?H))
```

the Fourier transform of an upsampled signal,

```
(FT (UPSAMPLE ?X ?L))  <==>  (SCALE-AXIS (FT ?X) ?L)
```

and polyphase networks,

```
(CONVOLVE (UPSAMPLE ?X ?L) ?H)  <==>  (POLYPHASE-UPSAMPLE ?X ?H ?L)
```

Within the domain of expression rearrangement we can identify a special category of rearrangements, called simplifications. Although the simplified form of an expression is a notoriously hard concept to capture [38], and clearly depends on how the form is to be used in later processing, certain rules about signal processing expressions

always reduce the size of a signal processing expression without introducing new signals. For example, the sum of a signal and zero is the signal

```
(ADD ?X (CONSTANT 0)) ==> ?X
```

or the convolution of a signal and an impulse is the signal

```
(CONVOLVE ?X (IMPULSE)) ==> ?X
```

are simplifications. It is almost always a good idea to use a simplification rule when it applies. The exception occurs when simplifying prevents further useful manipulations. For example, in the example of section 5.1, if we decompose $h[n]$ into L interleaved pieces so that we can derive the polyphase structure we should not "simplify" the L interleaved signals back to $h[n]$. Also, use of simplification rules alone may not lead to the most desirable form; it may be necessary to expand an expression before the best form can be found. For example, derivation of the polyphase structure requires expansion of a signal into pieces followed by simplification.

The rearrangement of signal processing expressions is used in symbolic manipulation problems, such as property analysis and cost analysis. Rearrangement is used in property analysis by changing the form of a signal processing expression from one in which the desired property is not clear into a form in which it is clear. For example, if there were no rules for the symmetry of a signal formed by convolving two other signals, it would be possible to determine symmetry using rules about the symmetry of inverse Fourier transforms and a rearrangement rule to change the time domain convolution into a frequency domain multiplication.

Conversely, signal properties can be used to determine the applicability of rearrangements. For example, the real part of a real-valued signal can be simplified to the original signal or the convolution of a real-valued signal and a complex-valued signal

can be transformed into two convolutions of real-valued signals.

Rearrangement of signal processing expressions is also used in the search for low cost implementations of signal processing operations. In the example of section 5.1, we changed the original system to a polyphase version and found that the polyphase version used fewer multiplies and additions. In general, we would like to be able to automatically search through potential rearrangements of a signal processing expression looking for the one with a desired property, such as low computational cost, low delay, low noise, etc.

5.2.3. Manipulation of Signals of a Continuous Variable

The explicit representation and manipulation of signals of a continuous variable, such as continuous-time or continuous-frequency signals, is another fundamental symbolic manipulation of signals. For example, in our manipulations of the system in section 5.1 we used continuous-frequency signals to express signal processing rules, we rearranged expressions involving continuous-frequency signals, and we manipulated signal properties for continuous-frequency signals.

In general, it is not possible to represent signals of a continuous variable with a finite set of samples. However, symbolic manipulation of signals is less concerned with the values of signals, and more concerned with properties and expressions. As such, we will be able to develop a representation for signals of a continuous variable for symbolic processing. This will be discussed in section 5.4.1.

5.3. Representation for Symbolic Manipulation of Signals

In this section we describe features of signals and systems that must be represented for the symbolic processing of signals and describe how these features are

used. Our goal is to extend our numerical signal representation to incorporate the symbolic manipulation of signals. The features that we find must be represented are signal types and histories, signal usage contexts, signal properties, system properties, and computational costs. While none of the features described here represent new ideas about signals, a key point is that these features and objects must be present in a representation for the symbolic manipulation of signals.

5.3.1. Signal Types and Histories

The fundamental requirement for a signal representation for symbolic manipulation is the explicit representation of the type and history of a signal. Signal type is the name of the system that created the signal and signal history is a description of all the parameters that were passed to the system to create the signal. For example, a signal created by the FFT system is of type FFT and the history of the signal is the description of both the signal that was passed to the FFT system and the size of the FFT taken. The explicit representation of the type and history of a signal is fundamental because it is necessary to know what a signal is before symbolic manipulation rules can be applied to it.

One natural representation for the type and history of a signal object is by the expression that generates it. For example, the object that represents the signal

$$y[n] = x_L[n] * h[n] \tag{5.8}$$

from the example of section 5.1 can be generated by

```
(DEFINE Y (CONVOLVE (UPSAMPLE X L) H))
```

The expression "(CONVOLVE (UPSAMPLE X L) H)" contains the type of $y[n]$, CONVOLVE, and a description of the two signals that are being convolved, (UPSAMPLE X L) and H. Explicit representation of the type and history of signals means that we can

examine the resulting signal object Y and determine this information. Although we have not explicitly stated the inquiry operations that extract the type and history of a signal, clearly the signal representation proposed in Chapter 3 contains all the necessary information. In this representation signal objects contain explicit pointers to the system that created them and an explicit record of the parameters passed to the system. This should be contrasted to the use of arrays and streams for signal representation, neither of which records the type or history of a signal object.

5.3.2. Signal Usage Contexts

The type and history of a signal object describes what a signal is and what it depends on. A complementary feature that must be present in a representation for the symbolic manipulation of signals is a description of how a signal is used by other signals. We call this the usage context for a signal. For example, we have argued that the signal (CONVOLVE (UPSAMPLE X L) H) should contain the information that it depends on the signals (UPSAMPLE X L) and H. Similarly, the signals (UPSAMPLE X L) and H should record that they are used by the signal (CONVOLVE (UPSAMPLE X L) H).

The usage context for a signal can provide information useful in guiding symbolic manipulation. For example, in manipulating the signal (CONVOLVE (UPSAMPLE X L) H) to determine an efficient implementation it would be useful to know if the Fourier transform of H, (FT H), has already been computed. This can be checked easily if all the signals that use the signal object H can be found by examining H.

5.3.3. Signal Properties

A fundamental type of symbolic manipulation of signals is the symbolic analysis of signal properties. To be able to manipulate properties of signals they must be represented. We have listed in section 5.2.1 some signal properties that are used in the manipulation of signals and we will describe a system that can manipulate a subset of these properties in Chapter 6.

The signal representation described in Chapter 3 and used in SPLICE contains explicit representation of a signal's non-zero extent by defining for each system a method of computing the non-zero extent of the signals output from the system in terms of the non-zero extent of the signals input to the system. This method can be extended for use in specifying all signal properties by associating with each system rules for each signal property. For example, the system for taking the Fourier transform of a discrete-time signal should contain the rule that specifies that the Fourier transform of a real signal is conjugate symmetric

```
(IF (EQ (REAL-OR-COMPLEX ?X) :REAL)
    (SYMMETRY (FT ?X)) = :CONJUGATE-SYMMETRIC)
```

Any Fourier transform signal would be able to check if it were conjugate symmetric by using the rule shared by all Fourier transform signals.

A representation for symbolic manipulation of signals must also allow for the combination of multiple rules about a signal property. For example, both the rule that a sum is complex if either argument is complex

```
(IF (OR (EQ (REAL-OR-COMPLEX ?X) :COMPLEX)
        (EQ (REAL-OR-COMPLEX ?Y) :COMPLEX))
    (REAL-OR-COMPLEX (ADD ?X ?Y)) = :COMPLEX)
```

and the rule that the sum of a signal and its conjugate is real

```
(REAL-OR-COMPLEX (ADD ?X (CONJUGATE ?X))) = :REAL
```

are applicable to finding out if $e^{j\omega} + e^{-j\omega}$ is real-valued or complex-valued. Extra information is needed to choose among applicable rules or to combine the outputs of multiple rules. Signal processing rules for signal properties, however, do not require the general purpose methods of modern rule systems because the rules do not involve inexact reasoning. Most signal processing properties have a natural ordering that can be used to choose among rule outputs. For example, when determining if a signal is real or complex, if one rule says complex and another says real then the signal is real.

5.3.4. System Properties

The symbolic manipulation of signals also requires the representation of the properties of signal processing systems. The terms listed in Fig. 5.6 are among those used to describe systems. For example, a system that performs FIR filtering is linear and shift-invariant and an adder is associative and commutative.

System properties are useful for specifying many signal processing rules. Some system properties, such as linearity and shift-invariance, allow the output of a system, for certain inputs, to be determined from the output of the system from other inputs. For example, for additive systems we can write

---

- Additive, Homogeneous, Linear
- Shift-Invariant
- Causal, Anticausal, Memoryless
- Associative, Commutative
- Stable
- Invertible

Fig. 5.6. System properties.

```
(IF (ADDITIVE ?SYSTEM)
    (?SYSTEM (ADD ?X ?Y))  <==>  (ADD (?SYSTEM ?X) (?SYSTEM ?Y)))
```

and for shift-invariant systems we can write

```
(IF (SHIFT-INVARIANT ?SYSTEM)
    (?SYSTEM (SHIFT ?X ?AMOUNT))  <==>  (SHIFT (?SYSTEM ?X) ?AMOUNT))
```

All the system properties listed in Fig. 5.6 can be used to determine relationships among signal processing expressions.

In addition to their use in determining the relationships among signal expressions, system properties can be used to determine signal properties. For example, the output signal from a causal system never starts before the start of the input signal

```
(IF (CAUSAL ?SYSTEM)
    (START (?SYSTEM ?X)) = (START ?X))
```

Also, the period of the output from a memoryless system is the same as the period of the input

```
(IF (MEMORYLESS ?SYSTEM)
    (PERIOD (?SYSTEM ?SIGNAL)) = (PERIOD ?SIGNAL))
```

Similar rules can be written for other combinations of signal and system properties. A collection of rules using system properties is given in Appendix D in the summary of the rules contained in E-SPLICE.

5.3.5. Computational Costs

Another property of systems that must be present in a representation for symbolic manipulation of signals is computational cost. A system not only specifies what a signal is, it also specifies how to compute the values of the signal. The expense of computing signal values should be explicitly available so that it is possible to compare the relative costs of different signal processing implementations. For example, the number of multiplies per output point for a Fourier transform is equal to the number

of points in the non-zero extent of the input signal

```
(MULTIPLIES-PER-SAMPLE (FT ?X)) = (LENGTH ?X)
```

Different measures of cost are possible. Useful measures include the number of multiplies and additions, but it may also be important to know the number of memory reads or memory writes, the delay, the noise introduced, etc. It is not reasonable to define a single cost metric for all uses. A better approach is to make explicit all cost components. Thus, the cost per point of a Fourier transform might be a vector of costs

```
(COST-PER-SAMPLE (FT ?X)) = ((MULTIPLIES (LENGTH ?X))
                             (ADDITIONS (- (LENGTH ?X) 1))
                             (MEMORY-READS (LENGTH ?X))
                             (MEMORY-WRITES 1))
```

In addition to the choice of an appropriate measure of cost, there are other difficulties in expressing the cost of a signal processing algorithm. Cost per sample is a reasonable measure for systems based on point operations but it is not reasonable for systems based on array operations. For example, saying that the cost per sample of an FFT is proportional to the log of the length of the FFT is misleading because all the points of the FFT are calculated even if only a few are used. For systems based on array operations a more reasonable measure might be a cost that depends on what interval of signal values is required.

The Fourier transform example illustrates another issue in the measurement of costs. In computing a sample of the Fourier transform not only are there multiplies and additions for the Fourier transform calculation, there are also requests for sample values of the input signal. These requests can cause computation and these nested computations must also be measured. Also, because signal values can be cached, costs should not be duplicated. For example, in computing two samples of a Fourier

transform we do not need to compute the input signal twice.

We will not present a complete description of cost properties for systems now. In Chapter 6 we will present one simple and preliminary model of costs that has been incorporated into a system for the symbolic manipulation of signals.

5.4. Extended Signal Objects

The symbolic manipulation of signals requires two extensions to the model of signals presented in Chapter 3. These are the introduction of signals of a continuous variable and abstract signal classes. This section describes the two extensions and discusses their use in the symbolic manipulation of signals.

5.4.1. Signals of a Continuous Variable

Signals of a continuous variable, such as continuous-time or continuous-frequency signals, and are used in the mathematics of signal processing. As such, we have identified the manipulation of signals of a continuous variable as a fundamental symbolic manipulation and thus, these signals must be explicitly represented.

To represent signals of a continuous variable we can use a model similar to the deferred array of Chapter 3. Particularly, we can define a signal object by defining a point operation that maps any index (from $-\infty$ to ∞, not necessarily an integer) into a signal value. The value of a signal at any index is determined by applying its function to the index. For example, we could define the Fourier transform of a signal as

```
(DEFINE (FT X)
  (DEFINE (FCN OMEGA)
    (SUM-OF-SIGNAL (SIGNAL-MULTIPLY X (COMPLEX-EXPONENTIAL (MINUS OMEGA)))))
  (MAKE-SIGNAL FCN))
```

where MAKE-SIGNAL is a function that creates a signal of a continuous variable from a point operator, just as MAKE-DEFERRED-ARRAY created a signal of a discrete

variable from a point operator. Exactly as with the deferred array of Chapter 3, we can associate a memory with a signal for caching signal values[2], we can define multiple functions, and we can record the non-zero extent of signals. We will show a particular representation of signals of a continuous variable in Chapter 6.

Clearly we can define many operations that can be applied to signals of a continuous variable, but not all operations that can be defined mathematically can be implemented directly in a computer. Particularly, we cannot numerically perform those operations that involve integration. For example, the inverse Fourier transform

$$x[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(e^{j\omega})e^{j\omega n}\, d\omega \tag{5.9}$$

can not always be evaluated numerically. These types of operations either must be evaluated symbolically or symbolic information must be used to guide the choice of a numerical approximation technique.

5.4.2. Abstract Signal Classes

In manipulating signals and analyzing signal processing signals it is important to be able to manipulate entire classes of signals simultaneously. For example, in the system of Fig. 5.1 we said that we could determine that $Y(e^{j\omega})$ was conjugate-symmetric if we knew that $x[n]$ and $h[n]$ were real. If all we know about $x[n]$ is that it is real-valued we cannot define a particular signal object to represent it. Instead we must be able to define, in the computer, a signal object X that captures the information that $x[n]$ is real-valued.

We call a representation for an entire class of signals, such as all real signals, an abstract signal class. The expression

[2] This memory must be implemented as a general associative store rather than as an array if we want to cache all signal values and maintain random access.

```
(DEFINE X (ABSTRACT-SIGNAL-CLASS :REAL-OR-COMPLEX :REAL))
```

specifies that the object X represents the class of all real signals. X can be used as if it were a signal, i.e. it can be used as input to a system, but the only information that can be extracted from it is that it is real-valued. It would be an error to attempt to get a signal value from X. For the symbolic manipulation of signals a representation should allow the specification of abstract signal classes for any legal combination of signal properties.

5.5. Summary

In this chapter we identified three classes of symbolic manipulations of signals. These are the manipulation of signal properties, the rearrangement of signal processing expressions, and the manipulation of signals of a continuous variable. They are summarized in Fig. 5.7.

Manipulation of Signal Properties

- Manipulation of descriptions, not values
- Used in rearrangement of expressions

Rearrangement of Signal Processing Expressions

- Change the form of expressions
- Simplify expressions
- Used to search for desired form

Manipulation of Signals of a Continuous Variable

- Natural expression of signal processing rules
- Property manipulation and expression manipulation

Fig. 5.7. Summary of classes of symbolic manipulation of signals.

We have described the requirements on a representation for the symbolic manipulation of signals. These are summarized in Fig. 5.8. We argued that the explicit representation of the type and history of a signal is fundamental for the symbolic manipulation of signals and that the expression that generated a signal is a natural record of the signal. We discussed the idea of keeping records of how a signal was used in addition to keeping a record of what the signal uses. The need for explicit representation of signal and system properties was discussed. Particular attention was

---

Signal Type and History

- System that created signal and parameters
- Necessary for symbolic manipulation
- Signal expression is a natural record

Signal Usage Contexts

- How signal is used by other signals
- Useful for finding related signals

Signal Properties

- Non-zero extent, period, symmetry, computational cost, etc.
- Can be manipulated without signal values
- Used in algorithm selection
- Used to determine rule applicability

System Properties

- Linearity, shift-invariance, commutativity, etc.
- Used to determine rule applicability
- Used to determine signal properties

Computational Costs

- Additions, multiplies, memory reads, memory writes, etc.
- Want cost vector rather than number

Fig. 5.8. Summary of signal and system features.

---

paid to the idea of representing the computational cost of a signal.

Finally, we presented two extensions to the model of signals presented in Chapter 3, signals of a continuous variable and the abstract signal classes, that are used in the symbolic manipulation of signals. These are summarized in Fig. 5.9. Signals of a continuous variable map the real numbers into signal values and are natural to use for the expression of many signal processing rules. An abstract signal class represent the set of all signals with some common properties and abstract signal classes are useful in analyzing signal processing systems.

Signals of a Continuous Variable

- Continuous-frequency signals and continuous-time signals
- Can be computed at any index (not restricted to integer)
- Useful in expressing signal processing knowledge
- May be manipulated using signal processing rules

Abstract Signal Classes

- Define a set of signals with common properties
- Useful in analysis of signal processing systems

Fig. 5.9. Summary of extended signal objects.

CHAPTER 6

Symbolic Manipulation Examples

This chapter presents several examples of the symbolic manipulation of signals by computer using an extended version of the Signal Processing Language and Interactive Computing Environment (SPLICE). We begin our discussion with an overview of the features of the Extended Signal Processing Language and Interactive Computing Environment (E-SPLICE). We describe extended signal objects, including continuous-frequency signals and abstract signal classes. We then describe signal processing rules and give examples of the manipulation of signal processing expressions. Next we discuss signal features and give examples of the automatic manipulation of signal properties. We discuss manipulation of signals to measure costs and to determine implementations and we give examples of the analysis of signal processing algorithms. Finally, we give an example of the manipulation of a reasonably complex signal processing system.

Our description of E-SPLICE differs in spirit from our description of SPLICE given Chapter 4. SPLICE was designed both as a useful collection of signal processing algorithms and as a user extendible signal programming environment. In contrast, E-SPLICE was designed primarily as a demonstration of the use of symbolic processing and not as a user extendible environment. As such, our description of E-SPLICE emphasizes usage and examples rather than definition.

The primary contribution of this chapter is the presentation of examples of the symbolic manipulation of signals. We present a variety of manipulations that have

been performed by computer and show that symbolic manipulation can be used in signal processing. We also describe E-SPLICE, a system for the symbolic manipulation of signals incorporated into a numerical signal processing environment. We show how our ideas on the representation of signals for symbolic manipulation, presented in Chapter 5, are used and how the system can be used to analyze complicated signal processing systems.

6.1. Overview

E-SPLICE provides a rule-based system for the symbolic manipulation of signals and incorporates extended signal objects and extended signal and system properties into the representation used in SPLICE. E-SPLICE can be used for the manipulation of signal processing expressions, the manipulation of continuous-frequency signals, the analysis of signal properties, and the analysis of computational costs.

6.1.1. Rule-Based System

A rule-based system was chosen as the inference mechanism because it can be extended incrementally and because the rules could be expressed more clearly in a rule-based language than in ZETALISP. The system we developed is not substantially different from standard rule-based systems, although it is tailored to interface easily with our signal objects. The interested reader is referred to Waterman and Hayes-Roth [17] for a general description of rule-based systems or to Dove [28] for a description of a specialized rule system for signal processing.

Fig. 6.1 shows the components of the system. Central to the system is a database of assertions called the blackboard. Assertions are facts about objects, e.g.

assertion: (HAMMING 255) is real-valued,

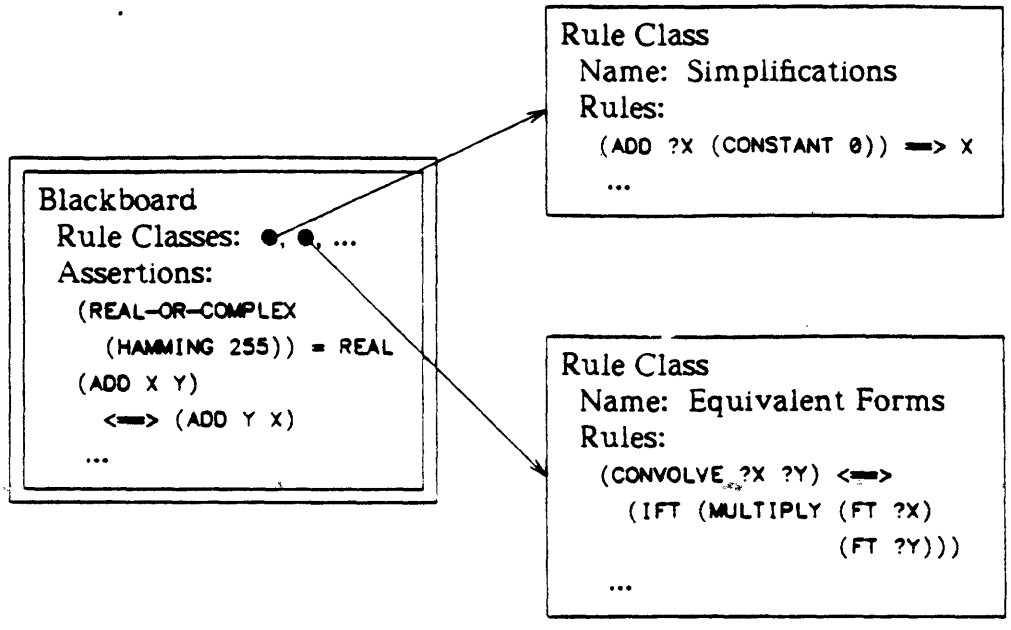


Fig. 6.1. The rule-based system.

and relationships among objects, e.g.

assertion: (ADD X Y) and (ADD Y X) are equivalent.

Associated with the blackboard are collections of rules called rule classes. A typical rule class is the set of all rules for reducing signals to simpler forms. Rule classes may be active or inactive and provide a convenient structuring mechanism for rules and some efficiency because only those rule classes applicable to a problem need be activated.

A typical rule is the simplification rule

```
(DEFINE-SIMPLIFICATION-RULE SHIFT-MULTIPLE
  "Multiple shifts are cumulative"
  (:FORM
   (SHIFT (SHIFT ?X ?SHIFT-1) ?SHIFT-2))
  (:RESULT
   (SHIFT X (+ SHIFT-1 SHIFT-2))))
```

This rule, SHIFT-MULTIPLE, states that two successive shifting operations can be combined into a single shifting operation with a shift amount equal to the sum of the two original shift amounts. This rule uses the system SHIFT, which takes as input a signal and an amount by which it is to be shifted. The :FORM portion of this rule specifies to what signals this rule applies and the :RESULT portion specifies the simplification that results when this rule is run. The expressions ?X, ?SHIFT-1, and ?SHIFT-2 represent rule variables that are bound to values during the firing of this rule. For example, if this rule were applied to the signal (SHIFT (SHIFT (HAMMING 255) 10) 15) then the bindings would be

X: (HAMMING 255)
SHIFT−1: 10
SHIFT−2: 15

and the result of firing this rule would be (SHIFT (HAMMING 255) 25).

In addition to being able to specify forms and results in rules, the rule language also supports test clauses, local variables, triggers, and action clauses. A more complete description of the rule language is beyond the scope of this thesis but the interested reader can find descriptions of rule languages in [17].

Activation of rule classes and solving of problems is controlled by a collection of programs called knowledge sources. Each knowledge source is responsible for solving a particular problem. There are knowledge sources for simplifying signal processing expressions, for determining equivalent forms, for determining signal properties, and for searching for efficient implementations. The knowledge sources will be discussed as we describe the manipulations that can be performed.

6.1.2. Systems

E-SPLICE extends the idea of a system, as used in SPLICE, by providing system properties. Fig. 6.2 summarizes those system properties that are used. They include the algebraic concepts of associativity and commutativity and the signal processing concepts of linearity, shift-invariance, and memorylessness. The operator IS-A can be

---

Associative (ASSOCIATIVE-SYSTEM)

- Can change grouping of inputs
- Examples: Add, multiply, convolve

Commutative (COMMUTATIVE-SYSTEM)

- Can change order of inputs
- Examples: Add, multiply, convolve

Additive (ADDITIVE-SYSTEM)

- Distributes over addition
- Examples: Conjugate, real-part, imaginary-part

Homogeneous (HOMOGENEOUS-SYSTEM)

- Scaled input becomes scaled output

Linear (LINEAR-SYSTEM)

- Additive and homogeneous
- Examples: Fourier transform, multiplication, convolution, shift, scale

Shift Invariant (SHIFT-INVARIANT-SYSTEM)

- Shifting input shifts output
- Examples: Convolution, shift, scale

Memoryless (MEMORYLESS-SYSTEM)

- Maps a single input sample into a single output sample
- Examples: Scale, conjugate, real-part, imaginary-part

Fig. 6.2. System properties.

---

used to determine if a system has a certain property. For example, (IS-A 'CONVOLVE 'ASSOCIATIVE-SYSTEM) returns true and (IS-A 'REAL-PART 'LINEAR-SYSTEM) returns false. Examples of the use of system properties will be given when we discuss expression rearrangement.

The systems that can be manipulated are listed in Fig. 6.3. These systems include the full range of system types, i.e. systems for generating signals, systems for modifying signals, systems for combining signals, and systems for transforming signals.

Generators of Basic Signals

- Impulse, Unit step, Complex exponential, Constant

Generators of Signals - built from simple signals

- Cosine, Sine, Hamming window, Rectangular window

Single Input, Single Output Systems

- Scale, Shift, Reverse
- Real part, Imaginary part, Conjugate
- FIR filter, IIR filter
- Upsample, Downsample
- Alias, Scale Frequency Axis

Multiple Input, Single Output Systems

- Add, Subtract, Multiply, Divide, Convolve

Transformations of Signals

- Fourier transform, Inverse Fourier transform

Fig. 6.3. Systems manipulated in E-SPLICE.

6.2. Extended Signal Objects

This section describes continuous-frequency signals and abstract signal classes. Both continuous-frequency signals and abstract signal classes are extensions of the numerical signal objects used in SPLICE.

6.2.1. Continuous-Frequency Signals

In Chapter 5 we argued that the representation of signals of a continuous variable was an important part of the symbolic manipulation of signals. To demonstrate this manipulation, E-SPLICE implements operations on a particular class of signals of a continuous variable, continuous-frequency signals, i.e. signals used to represent the Fourier transform of discrete-time signals. Continuous-frequency signals are specified similarly to the way in which point operators are specified in SPLICE. DEFINE-SYSTEM is used to define a system that creates a continuous-frequency signal, specifying that the system output type is SIGNAL and specifying a function for computing the value of the signal at any frequency. For example, the definition of the system FT for computing the Fourier transform of a discrete-time signal

$$X(e^{j\omega}) = \sum_{n} x[n]\, e^{-j\omega n} \tag{6.1}$$

is

```
(DEFINE-SYSTEM FT (SEQUENCE)
                  (SIGNAL)
  "Fourier transform of SEQUENCE"
  (SIGNAL-VALUE-COMPLEX (OMEGA)
    (SUM-OF-SEQ
      (MULTIPLY SEQUENCE (COMPLEX-EXPONENTIAL-SEQ (MINUS OMEGA))))))
```

Continuous-frequency signals are used to represent the transform of discrete-time signals and, as such, are defined from $-\infty$ to ∞ and have a period of 2π. The computation function for a continuous-frequency signal is defined either by a

SIGNAL-VALUE-COMPLEX clause, specifying the complex portion of the value of the continuous-frequency signal at any frequency, or by SIGNAL-VALUE-REAL and SIGNAL-VALUE-IMAG clauses, specifying respectively the real and imaginary parts of the value of the signal at any frequency.

Most systems in E-SPLICE can be used with either discrete-time or continuous-frequency signals. For example, CONVOLVE and MULTIPLY can take as input either two discrete-time or two continuous-frequency signals and will produce the appropriate output signal in each case. E-SPLICE also provides symbolic manipulation facilities for the rearrangement of continuous-frequency signal expressions into forms that can be implemented numerically. For example, (CONVOLVE (FT X) (FT Y)) can be rearranged into (FT (MULTIPLY X Y)). Examples are given in section 6.3.3.

6.2.2. Abstract Signal Classes

Another extension to the numerical signal objects used in SPLICE are abstract signal classes. Abstract signal classes are specifications of sets of signals sharing common properties. To define an abstract signal class the user uses the function ABSTRACT-SIGNAL-CLASS, the syntax of which is

(ABSTRACT-SIGNAL-CLASS *domain-type*
property-keyword-1 property-value-1
...
property-keyword-N property-value-N)

The domain type may be either :DISCRETE-TIME or :CONTINUOUS-FREQUENCY. ABSTRACT-SIGNAL-CLASS produces an object that has the specified properties as output. For example, (ABSTRACT-SIGNAL-CLASS :DISCRETE-TIME :SYMMETRY :SYMMETRIC) defines the class of all discrete-time symmetric signals. Objects produced by ABSTRACT-SIGNAL-CLASS have the most unrestricted values for

unspecified properties. For example, if no support is specified, it is taken to be $-\infty < n < \infty$. Abstract signal classes can be manipulated in the same way as all other signal objects through systems, e.g. (FT (ABSTRACT-SIGNAL-CLASS :DISCRETE-TIME :SYMMETRY :SYMMETRIC)), as long as no attempt is made to ask for a signal value from an abstract signal class. Abstract signal objects will be used in the presentation of examples in the rest of this chapter.

6.3. Signal Expression Manipulation

This section describes the symbolic manipulation of signal processing expressions. We describe rules for expression manipulation and give examples of the manipulation of expressions and of the manipulation of continuous-frequency signals.

6.3.1. Rules

Rules for the manipulation of signal processing expressions provide a major component of E-SPLICE. These rules are divided into three categories - equivalent forms, simplifications, and rearrangements. E-SPLICE contains about 150 rules for the manipulation of signal processing expressions. These rules are collected in Appendix D.

Equivalent form rules express the equivalence between two expressions. For example, the equivalent form rule

```
(DEFINE-EQUIVALENT-FORM-RULE CONVOLVE-AS-PRODUCT
  "Convolution of two signals can be written as the product
   of Fourier transforms"
  (:FORM
   (CONVOLVE ?X ?Y))
  (:RESULT
   (IFT (MULTIPLY (FT X) (FT Y)))))
```

states that the convolution of two signals can be written as the inverse Fourier transform of the product of Fourier transforms of the signals. Equivalent form rules do not judge which form is preferable.

A simplification rule is a special type of equivalent form rule that is marked as always being useful to apply. For example, the simplification rule

```
(DEFINE-SIMPLIFICATION-RULE CONVOLVE-WITH-IMPULSE
  "Convolving a signal with an impulse returns the signal"
  (:FORM
   (CONVOLVE ?X (IMPULSE)))
  (:RESULT
   X))
```

states that the convolution of a signal with an impulse is the same as the original signal. Simplification rules assume that the resulting form is desirable.

A rearrangement rule is also a special type of equivalent form rule. Rearrangements are those rules that rearrange the terms in an expression without changing the operations involved. For example, the rearrangement rule

```
(DEFINE-REARRANGEMENT-RULE COMMUTATIVE-SYSTEM-ORDER
  "Can change the order of arguments to a commutative system"
  (:FORM
   (?OP ?X ?Y))
  (:TEST
   (IS-A OP 'COMMUTATIVE-SYSTEM))
  (:RESULT
   (OP Y X)))
```

states that the order of the two inputs to a commutative system can be changed. Rearrangement rules do not judge which form is preferable and are used in the simplification process.

Control over the simplification of signal processing expressions is performed by a *simplification knowledge source*. It takes a signal to be simplified, activates the rule classes for simplifications and rearrangements, recursively attempts to simplify the pieces of the expression, looks for any rule that can be used to simplify the resulting expression, applies the rule, and recursively tries to simplify the result. Both rearrangement and simplification rules must be used in this process so that all the ways in which a simplification rule can be used are checked. It should be emphasized that the simplifier is a pragmatic solution to a difficult problem. It keeps trying to simplify the

expression until it has no more rules to run. It does not use context in guiding the simplification process.

The knowledge source for determining equivalent forms of signal expressions is like the simplifier. It takes the signal for which equivalent forms are required, first simplifies it, then recursively attempts to find equivalent forms for the pieces of the expression. Next, the knowledge source looks for any rule that can be used to find an equivalent form for the resulting expressions, applies that rule, and if a new equivalent form is generated tries to generate equivalent forms for the new form. In this way all the possible equivalent forms of a signal expression are generated.

6.3.2. Examples of Simplification and Rearrangement

Fig. 6.4 presents three examples of the simplification of signal processing expressions. The first form entered by the user uses the operator ABSTRACT-SIGNAL-CLASS to define signal classes and the operator SEQ-SETQ to name signals. The user defines the signals $x[n]$ and $h[n]$ to be discrete time signals about which no other information is known. The user then requests the simplified form, using the operator SIMPLIFIED-FORM, of the addition of $x[n]$ to the sequence that has constant value of zero. The system reports that it is trying to simplify the expression, that it found a rule (ADD-ZERO) which was applicable, and that the result of applying this rule was $x[n]$.

The next expression the user asks to be simplified is $g_1(x[n] * g_2\delta[n])$. This expression is input using the systems SCALE, for scaling a signal by a constant, and CONVOLVE, for convolving two signals. The system tries to simplify the expression by first trying to simplify the subexpression $(x[n] * g_2\delta[n])$. It finds that it can simplify this using a rule for homogeneous systems applied to scaled signals. The system

```
SPLICE: (LIST                                              ; Define the discrete time
                                                           ; Signals x[n] and h[n]
          (SEQ-SETQ X (ABSTRACT-SIGNAL-CLASS :DISCRETE-TIME))
          (SEQ-SETQ H (ABSTRACT-SIGNAL-CLASS :DISCRETE-TIME)))

------> (X H)


SPLICE: (SIMPLIFIED-FORM (ADD X (CONSTANT-SEQ 0)))         ; x[n] + 0

 Trying to simplify (ADD X (CONSTANT-SEQ 0))               ; Messages about
                                                           ; simplification process
 Found a simplification to X by rule ADD-ZERO

------> X                                                  ; x[n]


SPLICE: (SIMPLIFIED-FORM                                   ; g1(x[n] * g2 δ[n])
          (SCALE
            (CONVOLVE X (SCALE (IMPULSE-SEQ) 'G2)) 'G1))

 Trying to simplify (SCALE (CONVOLVE X (SCALE (IMPULSE-SEQ) G2)) G1)

  Trying to simplify (CONVOLVE X (SCALE (IMPULSE-SEQ) G2))

  Found simplification to (SCALE (CONVOLVE X (IMPULSE-SEQ)) G2)
   by rule HOMOGENEOUS-SYSTEM-SCALE

   Trying to simplify (CONVOLVE X (IMPULSE-SEQ))

   Found simplification to X by rule CONVOLVE-WITH-IMPULSE

 Trying to simplify (SCALE (SCALE X G2) G1)

 Found simplification to (SCALE X (* G1 G2))
  by rule SCALE-OF-SCALE

------> (SCALE X (* G1 G2))                               ; g1 g2 x[n]


SPLICE: (SIMPLIFIED-FORM
          (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 2)
                                (SHIFT (UPSAMPLE H 2) 1)) 2))

  ... (Many messages) ...

  ; Intermediate forms:
  ;  (DOWNSAMPLE (SHIFT (CONVOLVE (UPSAMPLE X 2) (UPSAMPLE H 2)) 1) 2)
  ;  (DOWNSAMPLE (SHIFT (UPSAMPLE (CONVOLVE X H) 2) 1) 2)

------> (CONSTANT-SEQ 0)
```

Fig. 6.4. Examples of expression simplification.

now tries to simplify $(x[n] * \delta[n])$ and finds that this can be simplified to $x[n]$. Finally, the system finds that it can combine the two scaling operations into a single

operation. This example demonstrates the use of system properties because $(x[n] * g_2 \delta[n])$ was simplified to $g_2(x[n] * \delta[n])$ not by a rule about convolutions but by a more general rule about homogeneous systems. Also, the example shows that in E-SPLICE some inputs to systems, such as the two gain factors G1 and G2, can be symbols rather than numbers or signals.

In the third example the user asks for the simplified form of (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 2) (SHIFT (UPSAMPLE H 2) 1)) 2) which is illustrated in Fig. 6.5. The system finds, after several steps, that this expression is equivalent to the constant zero.

Another important type of signal expression manipulation is the generation of equivalent forms for an expression. Fig. 6.6 gives two examples of the generation of equivalent forms. In the first example the user requests all ways of expressing the convolution of $x[n]$ and $h[n]$. The system runs the rule that turns time domain convolution into frequency domain multiplication and returns the time domain and fre-

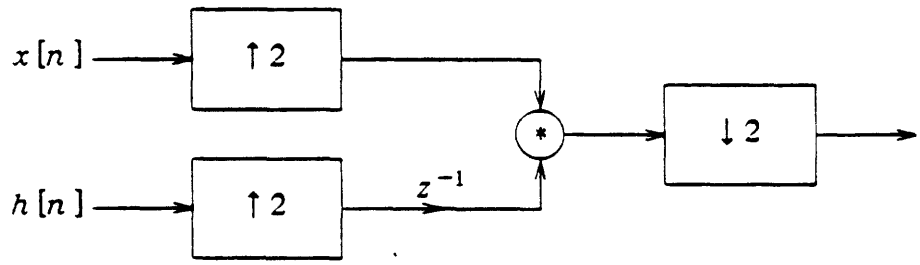


Fig. 6.5. The expression (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 2) (SHIFT (UPSAMPLE H 2) 1)) 2).

```
SPLICE: (ALL-EQUIVALENT-FORMS (CONVOLVE X H))          ; x[n]* h[n]

 Found an equivalent form (INVERSE-FT (MULTIPLY (FT X) (FT Y)))
  by rule CONVOLUTION-AS-TRANSFORM-PRODUCT

------> ((CONVOLVE X H)                               ; x[n]* h[n]

         (INVERSE-FT (MULTIPLY (FT X) (FT Y))))       ; FT^-1{X(e^jw)·H(e^jw)}


SPLICE: (ALL-EQUIVALENT-FORMS (DOWNSAMPLE (CONVOLVE X H) 2))

------> ((DOWNSAMPLE (CONVOLVE X H) 2)                ; Original

                                                      ; Frequency domain version
         (DOWNSAMPLE                                  ; of original
           (INVERSE-FT (MULTIPLY (FT X) (FT Y))) 2)


         (DOWNSAMPLE                                  ; Break x[n] and h[n]
           (CONVOLVE                                  ; into pieces
             (ADD (UPSAMPLE (DOWNSAMPLE X 2) 2)
                  (SHIFT (UPSAMPLE (DOWNSAMPLE (SHIFT X 1) 2) 2) -1))
             (ADD (UPSAMPLE (DOWNSAMPLE H 2) 2)
                  (SHIFT (UPSAMPLE (DOWNSAMPLE (SHIFT H 1) 2) 2) -1)))
           2)


         (ADD (CONVOLVE (DOWNSAMPLE X 2)              ; Polyphase structure
                        (DOWNSAMPLE H 2))
              (CONVOLVE (DOWNSAMPLE (SHIFT X 1) 2)
                        (DOWNSAMPLE (SHIFT H -1) 2)))

                                                      ; Frequency domain version
         (INVERSE-FT                                  ; of polyphase structure
           (ADD (MULTIPLY (FT (DOWNSAMPLE X 2))
                          (FT (DOWNSAMPLE H 2)))
                (MULTIPLY (FT (DOWNSAMPLE (SHIFT X 1) 2))
                          (FT (DOWNSAMPLE (SHIFT H -1) 2)))))

                                                      ; Alternative frequency
         (INVERSE-FT                                  ; domain version of
           (ADD (MULTIPLY (ALIAS (FT X) 2)            ; polyphase structure
                          (ALIAS (FT H) 2))
                (MULTIPLY (ALIAS (FT (SHIFT X 1)) 2)
                          (ALIAS (FT (SHIFT H -1)) 2))))
         ...)
```

Fig. 6.6. Examples of equivalent forms.

quency domain forms of the convolution.[1]

[1] These certainly are not "all ways" of expressing $x[n]* h[n]$. Missing are reorderings, e.g. $h[n]* x[n]$, invertible operations, e.g. $x[n]* FT^{-1}\{FT\{h[n]\}\}$, addition of zero, etc. What is here are all the equivalent forms that are generated by the system using equivalent form rules and always simplifying the result.

In the second example the user requests all the equivalent forms of the result of downsampling a convolution. The system responds with over a hundred equivalent forms, not all of which are interesting. Among the interesting ones are a frequency domain version of the original convolution, a polyphase structure, and two frequency domain versions of the polyphase structure. Fig. 6.7 illustrates the original system, the polyphase structure, and the first frequency domain version of the polyphase structure. It should be emphasized that the system did not directly use a rule to determine the polyphase structure. Instead the system used a rule to break up $x[n]$ and $h[n]$ into pieces and then found the polyphase structure by manipulating the resulting expression.

6.3.3. Examples of Manipulation of Continuous-Frequency Signals

The previous set of examples showed the manipulation of signal processing expressions for simplification and the determination of equivalent forms of expressions. In this section we show the manipulation of continuous-frequency signals. Fig. 6.8 shows an annotated example of the use of continuous-frequency signals. The first expression that the user enters is (FT (HAMMING 15)). This invokes the system FT on the 15 point Hamming window, centered at the origin, and creates the signal

$$X(e^{j\omega}) = \sum_{n=-7}^{7} (.54 + .46\cos(\frac{2\pi n}{14}))\, e^{-j\omega n} \qquad (6.2)$$

The user finds out that the non-zero extent of this signal is from $-\infty$ to ∞, it is periodic with a period of 2π, and it is both symmetric and conjugate-symmetric. Finally the user uses the inquiry operator FETCH to determine the value of the transform at 0.0 and π.

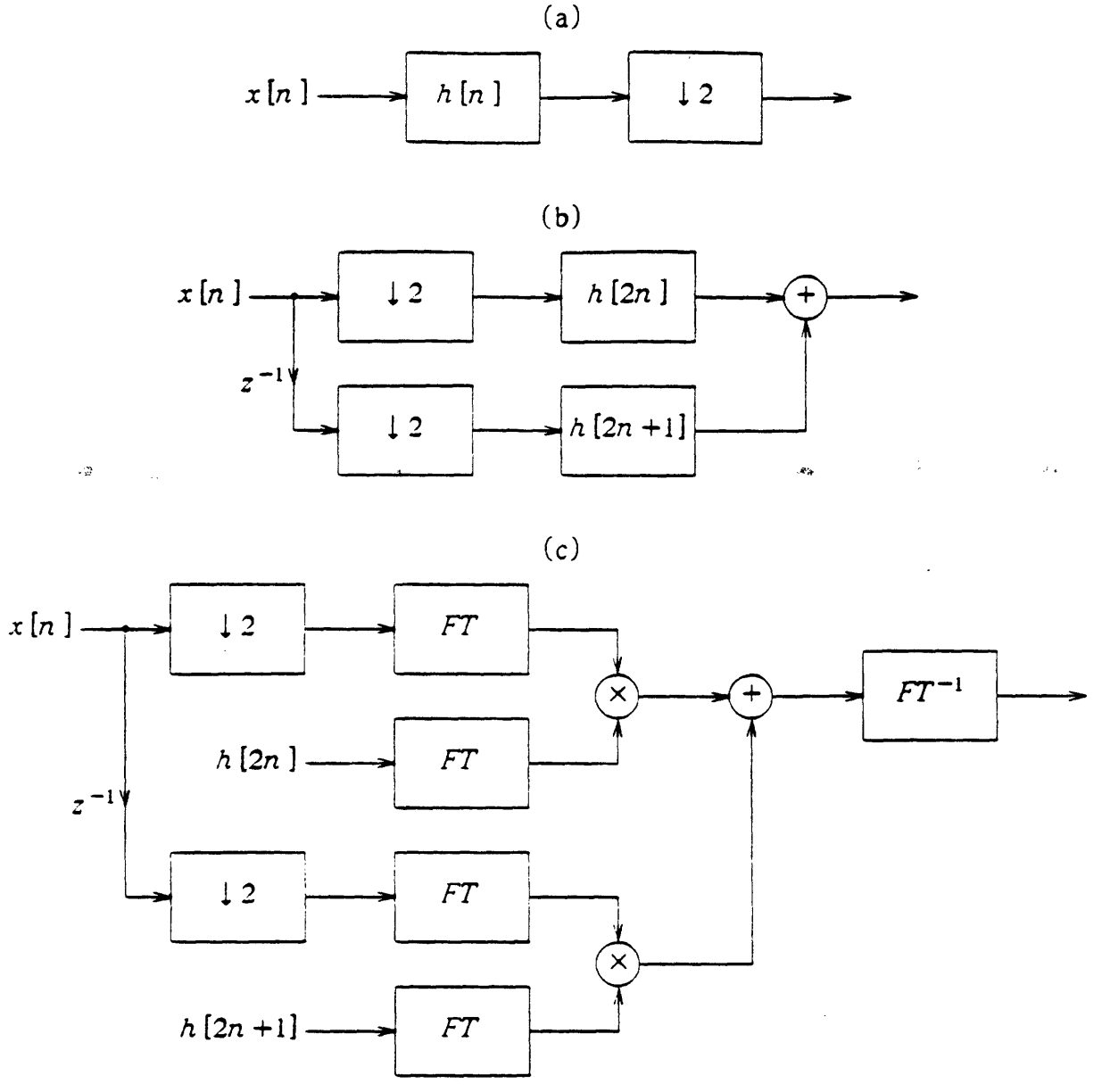


Fig. 6.7. Diagrams for equivalent forms.
(a) Original.
(b) Polyphase structure.
(c) Frequency domain version of polyphase structure.

```
SPLICE: (FT (HAMMING 15))                ; Create the Fourier transform of the
                                         ; 15 point Hamming window centered at 0
------> (FT (HAMMING 15))

SPLICE: (SUPPORT (FT (HAMMING 15)))      ; Find out the interval over which
                                         ; the transform is non-zero
------> (INTERVAL *MINUS-INFINITY* *INFINITY*)

SPLICE: (PERIOD (FT (HAMMING 15)))       ; Transform is periodic with
                                         ; period 2π
------> 6.2831855

SPLICE: (SYMMETRY (FT (HAMMING 15)))     ; Transform is both symmetric
                                         ; and conjugate-symmetric
------> (:SYMMETRIC :CONJUGATE-SYMMETRIC)

SPLICE: (FETCH (FT (HAMMING 15)) 0.0)    ; Value of transform at 0.0
------> 7.64

SPLICE: (FETCH (FT (HAMMING 15)) PI)     ; Value of transform at π
------> -0.08
```

Fig. 6.8. Use of continuous-frequency signals.

Symbolic manipulation of signals is used in generating frequency domain representations of signals. Fig. 6.9 shows how Fourier transforms can be generated in the system. The first example, the Fourier transform of a constant, illustrates the ability of the system to recognize special Fourier transform pairs. The second example, the Fourier transform of a Hamming window, illustrates the ability of the system to decompose signals into parts and to take the transform of these parts. The signal (SINC-SIG L) is the transform of an L point rectangular window, L odd, centered at zero

$$sinc_L(\omega) = \sum_{n=-M}^{n=M} e^{j\omega n} = \frac{\sin(\frac{\omega L}{2})}{\sin(\frac{\omega}{2})} \quad L = 2M+1 \tag{6.3}$$

```
SPLICE: (SIMPLIFIED-FORM (FT (CONSTANT-SEQ 'VAL)))        ; FT{val}

------> (SCALE (IMPULSE-SIG) (* 2PI VAL))                ; 2π val · δ(ω)

SPLICE: (SIMPLIFIED-FORM (FT (HAMMING-WINDOW 'L)))        ; FT{Hamming_L[n]}

 Trying to simplify (FT (HAMMING-WINDOW L))

 Trying to simplify (FT (MULTIPLY                         ; Decompose Hamming
                          (RECTANGULAR-WINDOW L)
                          (ADD (CONSTANT-SEQ .54)
                               (SCALE (COSINE-SEQ (/ 2PI (- L 1))) .46))))

  ... (Many messages) ...

------> (ADD (SCALE (SINC-SIG L) .54)
             (SCALE (SHIFT (SINC-SIG L) (/ 2PI (- L 1))) .23)
             (SCALE (SHIFT (SINC-SIG L) (/ (* -1 2PI) (- L 1))) .23))
          ; .54 sinc_L(ω) + .23 sinc_L(ω − 2π/(L−1)) + .23 sinc_L(ω + 2π/(L−1))

SPLICE: (SIMPLIFIED-FORM                                  ; FT{e^{jω0n}(h[n] * x[n] e^{−jω0n})}
          (FT (MULTIPLY (COMPLEX-EXPONENTIAL-SEQ 'OMEGA-0)
                        (CONVOLVE H (MULTIPLY X (COMPLEX-EXPONENTIAL-SEQ
                                                  (MINUS 'OMEGA-0)))))))

  ... (Many messages) ...

  ; Intermediate forms:
  ;   δ(ω−ω0) * FT{(h[n] * x[n] e^{−jω0n})}
  ;   δ(ω−ω0) * (H(e^{jω}) · FT{x[n] e^{−jω0n}})
  ;   δ(ω−ω0) * (H(e^{jω}) · (FT{x[n]} * δ(ω+ω0)))
  ;   δ(ω−ω0) * (H(e^{jω}) · (X(e^{jω}) * δ(ω+ω0)))
  ;   δ(ω−ω0) * (H(e^{jω}) · X(e^{j(ω+ω0)}))
  ;   H(e^{j(ω−ω0)}) · X(e^{j(ω+ω0−ω0)})

------> (MULTIPLY (SHIFT (FT H) OMEGA-0) (FT H))        ; H(e^{j(ω−ω0)}) · X(e^{jω})
```

Fig. 6.9. Generation of Fourier transforms by parsing signal representation.

In the third example of generation of Fourier transforms the user enters $FT\{e^{j\omega_0 n}(h[n] * x[n]e^{-j\omega_0 n})\}$.[2] The system reports that this can be simplified to $H(e^{j(\omega-\omega_0)}) \cdot X(e^{j\omega})$ after several steps.

[2] This is the expression for implementing a bandpass filter from a lowpass filter and was used in Chapter 4 in the system BPF-FROM-LPF.

6.4. Signal Property Manipulation

Those signal properties that are manipulated are listed in Fig. 6.10. In addition to the signal properties of support, period, and atomic-type from SPLICE, E-SPLICE manipulates symmetry and bandwidth. In this section we describe the representation of symmetry and bandwidth, we discuss rules for signal properties, and we give examples of the manipulation of signal properties.

6.4.1. Extended Signal Properties

E-SPLICE manipulates information about the symmetry of signals, by keeping track of the type of symmetry and the center of symmetry. The symmetry type of a complex-valued signal may be either :SYMMETRIC, :CONJUGATE-SYMMETRIC, :ANTISYMMETRIC, or :CONJUGATE-ANTISYMMETRIC and the symmetry type of a real-valued signal may be either both :SYMMETRIC and :CONJUGATE-SYMMETRIC, or both :ANTISYMMETRIC and :CONJUGATE-ANTISYMMETRIC. Discrete-time signals may have a center of symmetry that is any integer or any integer plus one half. Continuous-frequency signals may have any value for a center of sym-

---

Properties used previously in SPLICE

- Support - non-zero extent of signal
- Period
- Atomic-type - real or complex-valued

Properties added in E-SPLICE

- Symmetry - location and type of symmetry
- Bandwidth - non-zero extent of Fourier transform

Fig. 6.10. Signal properties.

---

metry.

The bandwidth of a signal is the non-zero extent of the signal's frequency domain representation and can be requested with the inquiry operator BANDWIDTH. The default bandwidth of signals is the full band $-\pi \leq \omega < \pi$.

6.4.2. Signal Property Rules

Rules about signal properties are associated with each system listed in Fig. 6.3. For example, the system CONVOLVE contains rules about the symmetry of a convolution, such as that the convolution of two symmetric signals is symmetric and the convolution of two real-valued signals is real. The signal property rules used are summarized in Appendix D. Also defined for the different signal properties are functions for combining different rule results.[3] These combining functions are listed in Fig. 6.11. For example, if in determining the non-zero extent of a signal two rules give different answers then the two answers are combined by taking the intersection of the two.

Support - Take intersection of the two supports

Period - Take greatest common divisor of the two periods

Atomic-type - Real if either is real

Symmetry - Keep both

Bandwidth - Take intersection of the two bandwidths

Fig. 6.11. Combining functions for signal properties.

---

[3] We assume all rules are "correct" but may not be as restrictive as possible.

The *signal properties knowledge source* controls the rules about signal properties. To find out the value of some signal property of a signal it first runs the signal property rules associated with the system that created the signal. Then it requests the simplified form of the signal and invokes the property computation function associated with the simplified form. The two results are combined. If requested, the knowledge source for property computation will also examine equivalent forms of the signal for the computation of a signal property.

6.4.3. Examples

Fig. 6.12 shows some examples of the analysis of signal symmetry. The first example shows the user requesting a description of all the signal properties of $\cos(\frac{2\pi n}{10})$. The system tells the user that the signal is infinite in extent, periodic with a period of 10, real-valued, symmetric about the index 0, and has a bandwidth of from $-2\pi/10$ to $2\pi/10$. The user then asks about the signal properties of the result of shifting a 255 point Hamming window by 127 points. This signal is not a fundamental signal but the system is able to determine the properties of the signal by examining the structure of the signal and using its rules about multiplication, addition, rectangular windows, and complex exponentials.

Next, the user tells the system that $x[n]$ will be used to represent the class of real-valued discrete-time signals whose bandwidth is from $-\pi/4$ to $\pi/4$. The user then asks for the properties of $FT\{x[n]\}$. The system responds that the transform is bandlimited, complex-valued, and conjugate-symmetric about zero. Finally, the user requests information about $FT\{x^2[n]e^{j\pi n/4}\}$. The system responds that this transform is non-zero for $-\pi/4 \leqslant \omega \leqslant 3\pi/4$, is complex-valued, and is conjugate-symmetric about the frequency $\pi/4$.

```
SPLICE: (SIGNAL-PROPERTIES (COSINE-SEQ (/ 2PI 10)))      ; cos(2πn/10)

  Signal (COSINE-SEQ (/ 2PI 10)) is a discrete-time signal with:
    Support = (INTERVAL *MINUS-INFINITY *INFINITY*)
    Period = 10
    Real-or-Complex = REAL
    Symmetry = SYMMETRIC and CONJUGATE-SYMMETRIC about the index 0
    Bandwidth = (INTERVAL (/ -2PI 10) (/ 2PI 10))

------> (COSINE-SEQ (/ 2PI 10.))


SPLICE: (SIGNAL-PROPERTIES (SHIFT (HAMMING-WINDOW 255) 127))

  Signal (SHIFT (HAMMING-WINDOW 255) 127) is a discrete-time signal with:
    Support = (INTERVAL 0 255)
    Period = *INFINITY*
    Real-or-Complex = REAL
    Symmetry = SYMMETRIC and CONJUGATE-SYMMETRIC about the index 127
    Bandwidth = (INTERVAL -PI PI)

------> (SHIFT (HAMMING-WINDOW 255) 127)


SPLICE: (SEQ-SETQ X (ABSTRACT-SIGNAL-CLASS :DISCRETE-TIME        ; x[n]
                      :REAL-OR-COMPLEX :REAL
                      :BANDWIDTH (INTERVAL (/ -PI 4) (/ PI 4))))

------> X


SPLICE: (SIGNAL-PROPERTIES (FT X))                               ; FT{x[n]}

  Signal (FT X) is a continuous-frequency signal with:
   Support = (INTERVAL (/ -PI 4) (/ PI 4))
   Real-or-Complex = COMPLEX
   Symmetry = CONJUGATE-SYMMETRIC about the frequency 0

------> (FT X)


SPLICE: (SIGNAL-PROPERTIES                                       ; FT{x^2[n] e^{jπn/4}}
          (FT (MULTIPLY X X (COMPLEX-EXPONENTIAL-SEQ (/ PI 4)))))

  Signal (FT (MULTIPLY X X (COMPLEX-EXPONENTIAL-SEQ (/ PI 4))))
   is a continuous-frequency signal with:
    Support = (INTERVAL (/ -PI 4) (/ (* 3 PI) 4))
    Real-or-Complex = COMPLEX
    Symmetry = CONJUGATE-SYMMETRIC about the frequency (/ PI 4)

------> (FT (MULTIPLY X X (COMPLEX-EXPONENTIAL-SEQ (/ PI 4))))
```

Fig. 6.12. Examples of signal properties.

6.5. Implementations and Cost Analysis

This section describes manipulation of signal expressions to measure costs and to find efficient implementations. We describe the cost measure used, rules about implementations, and give examples of cost analysis and rearrangement according to cost criteria.

6.5.1. Cost Measures

The measurement of the computational cost of various signal processing operations is a feature that is provided in a rough form by E-SPLICE. Cost is measured on an average cost per output sample basis using a cost vector of real adds, complex adds, real multiplies, and complex multiplies and is found with the inquiry operation COST-PER-SAMPLE. This cost measure is not intended to be a complete description of all the costs of running an algorithm on any particular architecture but it is representative of possible cost measures. A more complete cost measure would include memory requirements, delay, etc.

With the cost measure provided, we find (COST-PER-SAMPLE (CONVOLVE X (HAMMING-WINDOW 255))) returns (:REAL-MULTIPLIES 255 :REAL-ADDS 254) when X is real valued. This cost measurement assumes that the system CONVOLVE is implemented in a straightforward way and does not take advantage of the symmetry of the Hamming window. We will describe how signal properties such as symmetry are used in cost analysis in the next section.

The cost per sample of computing a signal, such as (CONVOLVE X (HAMMING-WINDOW 255)) depends not only on the cost of the system, i.e. CONVOLVE, but also on the cost of computing the inputs to the system. The example of (COST-PER-SAMPLE (CONVOLVE X (HAMMING-WINDOW 255))) assumed that it was free to

get each sample of X and (HAMMING-WINDOW 255). If computation is required to get the input samples then this cost will be added on. For example, (COST-PER-SAMPLE (CONVOLVE (CONVOLVE X (HAMMING-WINDOW 15)) (HAMMING-WINDOW 255))) would return (:REAL-MULTIPLIES 270 :REAL-ADDS 268) because the internal convolution, (CONVOLVE X (HAMMING 15)), requires 15 multiplies and 14 adds per sample. A complete description of the cost measures for various systems is given in Appendix D.

6.5.2. Implementation Rules and Cost Analysis

E-SPLICE provides a limited set of rules for the implementation of general purpose signal processing systems in specific forms according to the types of inputs. For example, the system MULTIPLY can be specialized from its general form (which assumes its two inputs are complex valued) according to whether the two signals that are being multiplied are both real valued or one real valued and the other complex valued. The systems for which implementation rules exist are listed in Fig. 6.13.

Implementation rules provide one way to explore the space of possible implementations. These rules are always applicable to finding a good implementation, e.g. using the specialized system for multiplying two real valued signals is better than using the general purpose system for multiplying two complex valued signals. However, good implementations may also be found by rearranging signal expressions. For example, a frequency domain implementation of convolution can often be more efficient than a time domain implementation. An *implementation knowledge source* provides the facility to explore the space of implementations using equivalent forms. It uses implementation rules and equivalent form rules to generate a list of potential implementations. It then removes any forms that can not be implemented, e.g. no Fourier

Multiply Special Cases

- Both inputs real
- One input real and one complex

FIR Filter Special Cases

- Symmetric impulse response
- Real input and real coefficients
- Complex input and real coefficients

FFT and Inverse FFT Special Cases

- Real input
- Conjugate symmetric input

Fig. 6.13. Special cases for implementation of systems.

transforms of infinite duration signals. The resulting list of forms is evaluated according to the cost analysis function. The costs of the various forms are compared and any form whose cost is clearly worse than another form is eliminated.[4] The resulting list of forms can be compared with a user supplied cost function and the best selected, or the entire list of remaining forms can be returned by the knowledge source.

6.5.3. Examples

Fig. 6.14 shows some examples of cost measurement. The user first defines the signals $x[n]$ and $h[n]$ where both are complex and $h[n]$ is 63 points long. Next the user finds that convolving $x[n]$ and $h[n]$ in the time domain costs 63 complex multiplies and 62 complex adds per sample. To examine the cost of frequency domain con-

[4] For one form to be clearly worse than another form its cost vector must have uniformly larger entries. We do not assume a fixed weighting of the individual elements of the cost vector.

```
SPLICE: (LIST (SEQ-SETQ X (ABSTRACT-SIGNAL-CLASS :DISCRETE-TIME))
              (SEQ-SETQ H (ABSTRACT-SIGNAL-CLASS :DISCRETE-TIME
                            :SUPPORT (INTERVAL 0 63))))

------> (X H)

SPLICE: (COST-PER-SAMPLE (CONVOLVE X H))

------> (:COMPLEX-MULTIPLIES 63 :COMPLEX-ADDS 62)

SPLICE: (COST-PER-SAMPLE
          (OVERLAP-SAVE-CONVOLVE (IFFT (MULTIPLY (FFT X 512) (FFT H 512)) 512)))

------> (:COMPLEX-MULTIPLES 11.38 :COMPLEX-ADDS 20.48)

SPLICE: (COST-PER-SAMPLE (DOWNSAMPLE (CONVOLVE X H) 10))

------> (:COMPLEX-MULTIPLIES 63 :COMPLEX-ADDS 62)

SPLICE: (COST-PER-SAMPLE
          (DOWNSAMPLE
            (OVERLAP-SAVE-CONVOLVE (IFFT (MULTIPLY (FFT X 512) (FFT H 512)) 512))
            10))

------> (:COMPLEX-MULTIPLES 113.8 :COMPLEX-ADDS 204.8)
```

Fig. 6.14. Examples of cost measurement.

volution the user uses the operator OVERLAP-SAVE-CONVOLVE, which takes as input a form that specifies how to perform circular convolution. Here the user specifies that circular convolution should be performed using 512 point FFTs. The user finds that convolving $x[n]$ and $h[n]$ using 512 point FFTs requires 11.38 complex multiplies and 20.48 complex adds per sample.[5] Next the user finds the cost per sample of computing the downsampled convolution of $x[n]$ and $h[n]$. The system responds that the cost per sample is the same as the original convolution. However, the user finds that the cost of downsampling the overlap-save convolution increases the cost per sample by a factor equal to the downsampling rate. This cost

[5] These costs are not integer amounts because our cost measure is an average per sample and overlap-save convolution computes several samples simultaneously.

measurement assumes that the time domain convolution can be run only for those samples that are needed, i.e demand driven, but overlap-save convolution must compute the intermediate samples even when they are not required.

Fig. 6.15 illustrates implementation analysis. The user tells the system that $x[n]$ is a real valued signal and $h[n]$ is real valued, symmetric, and 63 points long. The user then asks the system to find efficient implementations of downsampling the

```
SPLICE: (LIST (SEQ-SETQ X (ABSTRACT-SIGNAL-CLASS :DISCRETE-TIME
                           :REAL-OR-COMPLEX :REAL))
              (SEQ-SETQ H (ABSTRACT-SIGNAL-CLASS :DISCRETE-TIME
                           :REAL-OR-COMPLEX :REAL :SUPPORT (INTERVAL 0 63)
                           :SYMMETRY :SYMMETRIC)))

------> (X H)

SPLICE: (EFFICIENT-IMPLEMENTATIONS (DOWNSAMPLE (CONVOLVE X H) 2))

------> (((DOWNSAMPLE (CONVOLVE-REAL-REAL-SYMMETRIC X H) 2)      ; Original
          (:REAL-ADDS 62 :REAL-MULTIPLIES 32))

                                                                  ; Frequency domain
         ((DOWNSAMPLE                                             ; version of original
            (OVERLAP-SAVE-CONVOLVE
              (IFFT-SYMMETRIC (MULTIPLY (FFT-REAL X 512) (FFT-REAL H 512)) 512))
            2)
          (:COMPLEX-ADDS 25.03 :COMPLEX-MULTIPLES 13.65))


         ((ADD (CONVOLVE-REAL-REAL-SYMMETRIC                      ; Polyphase structure
                 (DOWNSAMPLE X H) (DOWNSAMPLE H 2))
               (CONVOLVE-REAL-REAL-SYMMETRIC
                 (DOWNSAMPLE (SHIFT X 1) 2) (DOWNSAMPLE (SHIFT H -1) 2)))
          (:REAL-ADDS 62 :REAL-MULTIPLIES 32))

                                                                  ; Frequency domain version
         ((ADD (OVERLAP-SAVE-CONVOLVE                             ; of polyphase structure
                 (IFFT-SYMMETRIC
                   (MULTIPLY (FFT-REAL (DOWNSAMPLE X 2) 256)
                             (FFT-REAL (DOWNSAMPLE H 2) 256)) 256))
               (OVERLAP-SAVE-CONVOLVE
                 (IFFT-SYMMETRIC
                   (MULTIPLY (FFT-REAL (DOWNSAMPLE (SHIFT X 1) 2) 256)
                             (FFT-REAL (DOWNSAMPLE (SHIFT H -1) 2) 256)) 256)))
          (:COMPLEX-ADDS 22.71 :COMPLEX-MULTIPLES 12.49 :REAL-ADDS 1))
         ...)
```

Fig. 6.15. Implementation examples.

convolution of $x[n]$ and $h[n]$, illustrated in Fig. 6.7(a), using the function EFFICIENT-IMPLEMENTATIONS. This function invokes the knowledge source for cost analysis and responds with a list of impiementations and their associated costs. In the implementations the system uses special purpose systems, such as FFT-REAL for the FFT of a real signal and CONVOLVE-REAL-REAL-SYMMETRIC to convolve a real signal with a real, symmetric kernel.

6.6. Extended Example

In this section we present an extended example of the analysis of a signal processing expression incorporating symbolic analysis and manipulation. The expression we will examine is (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2) and is illustrated in Fig. 6.16. Fig. 6.17 shows several example manipulations of the expression. The user first finds the expression for the Fourier transform of the expression in terms of the Fourier transforms of $x[n]$ and $h[n]$. The system responds that the Fourier transform is an aliased version of the product of the Fourier transform of $h[n]$ and a compressed version of the Fourier transform of $x[n]$. Then, by setting $x[n]$ to a complex exponential and $h[n]$ to an impulse the user finds that putting a single complex exponential as input produces three complex exponentials as output. Finally, the user specifies that $x[n]$ is a real valued, periodic signal with period three and is symmetric

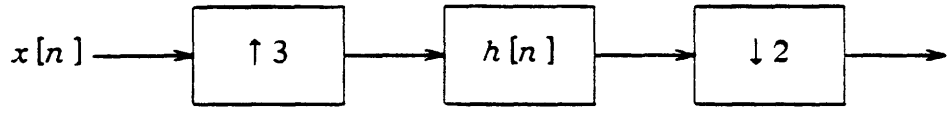


Fig. 6.16. The signal (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2).

```
SPLICE: (SIMPLIFIED-FORM (FT (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2)))
------> (ALIAS (MULTIPLY (SCALE-AXIS (FT X) 3) (FT H)) 2)
                                                  ; Alias_2{X(e^{j3ω})H(e^{jω})}

SPLICE: (LIST (SEQ-SETQ X (COMPLEX-EXPONENTIAL-SEQ 'OMEGA))      ; x[n] = e^{jω}
              (SEQ-SETQ H (IMPULSE-SEQ)))                        ; h[n] = δ[n]
------> (X H)

SPLICE: (SIMPLIFIED-FORM (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2))
------> (SCALE (ADD (COMPLEX-EXPONENTIAL-SEQ (* (/ OMEGA 3) 2))
                    (COMPLEX-EXPONENTIAL-SEQ (* (/ (+ OMEGA 2PI) 3) 2))
                    (COMPLEX-EXPONENTIAL-SEQ (* (/ (+ OMEGA 4PI) 3) 2)))
               (/ 1 3))
                                  ; (e^{2j ω/3} + e^{2j (ω+2π)/3} + e^{2j (ω+4π)/3})/3

SPLICE: (LIST (SEQ-SETQ X (ABSTRACT-SIGNAL-CLASS :DISCRETE-TIME
                          :REAL-OR-COMPLEX :REAL :SYMMETRY :SYMMETRIC
                          :CENTER-OF-SYMMETRY 1 :PERIOD 3))
              (SEQ-SETQ H (ABSTRACT-SIGNAL-CLASS :DISCRETE-TIME
                          :REAL-OR-COMPLEX :REAL :SYMMETRY :SYMMETRIC
                          :BANDWIDTH (INTERVAL (/ -2PI 8) (/ 2PI 8))))))
------> (X H)

SPLICE: (SIGNAL-PROPERTIES (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2))

  Signal (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2)
   is a discrete-time signal with:
    Support = (INTERVAL *MINUS-INFINITY* *INFINITY*)
    Period = 9
    Real-or-Complex = REAL
    Symmetry = SYMMETRIC and CONJUGATE-SYMMETRIC about the index 1.5
    Bandwidth = (INTERVAL (/ -2PI 4) (/ 2PI 4))

------> (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2)
```

Fig. 6.17. Manipulations of (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2).

about the index one and that $h[n]$ is a real, symmetric, bandlimited signal and finds that for these classes of input signals the resulting signal is real, periodic, symmetric, and bandlimited with a bandwidth twice the bandwidth of $h[n]$.

Fig. 6.18 shows some of the different ways of rearranging the expression (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2). The first three equivalent forms shown correspond to the original, a polyphase version of the upsample and filter subsection followed by downsampling, and a polyphase version of the filter and

```
SPLICE: (ALL-EQUIVALENT-FORMS (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2))

------> ((DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2)      ; Original

                                                ; Polyphase upsample and filter
         (DOWNSAMPLE                            ; followed by downsampling
           (INTERLEAVE (CONVOLVE X (DOWNSAMPLE H 3))
                       (CONVOLVE X (DOWNSAMPLE (SHIFT H -1) 3))
                       (CONVOLVE X (DOWNSAMPLE (SHIFT H -2) 3)))
           2)

                                                ; Polyphase filter and downsample
         (ADD                                   ; of upsampled signal
           (CONVOLVE (DOWNSAMPLE (UPSAMPLE X 3) 2)
                     (DOWNSAMPLE H 2))
           (CONVOLVE (DOWNSAMPLE (SHIFT (UPSAMPLE X 3) 1) 2)
                     (DOWNSAMPLE (SHIFT H -1) 2)))

                                                ; Unusual version
         (ADD
           (INTERLEAVE
             (CONVOLVE (DOWNSAMPLE X 2) (DOWNSAMPLE H 6))
             (CONVOLVE (DOWNSAMPLE X 2) (DOWNSAMPLE (SHIFT H -2) 6))
             (CONVOLVE (DOWNSAMPLE X 2) (DOWNSAMPLE (SHIFT H -4) 6)))
           (INTERLEAVE
             (CONVOLVE (DOWNSAMPLE (SHIFT X 1) 2) (DOWNSAMPLE (SHIFT H -3) 6))
             (CONVOLVE (DOWNSAMPLE (SHIFT X 1) 2) (DOWNSAMPLE (SHIFT H -5) 6))
             (CONVOLVE (DOWNSAMPLE (SHIFT X 1) 2) (DOWNSAMPLE (SHIFT H -7) 6))))
         ...)
```

Fig. 6.18. Equivalent forms of (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2).

downsample section using an upsampled input. The system INTERLEAVE, used in these examples, takes L signals as input and combines them by taking one sample from the first, then one sample from the second, etc. This is an efficient way of upsampling L signals by L, shifting them all by different amounts, and summing the results.

The fourth structure of Fig. 6.18 is an unusual one and is shown in Fig. 6.19. This structure is unusual because it uses many intermediate filters and is not obvious from the original specification or from the polyphase versions.

The user then analyzes the implementation costs of the system of Fig. 6.16 when $x[n]$ is real and $h[n]$ is real, symmetric, and 127 points long. This is shown in Fig.

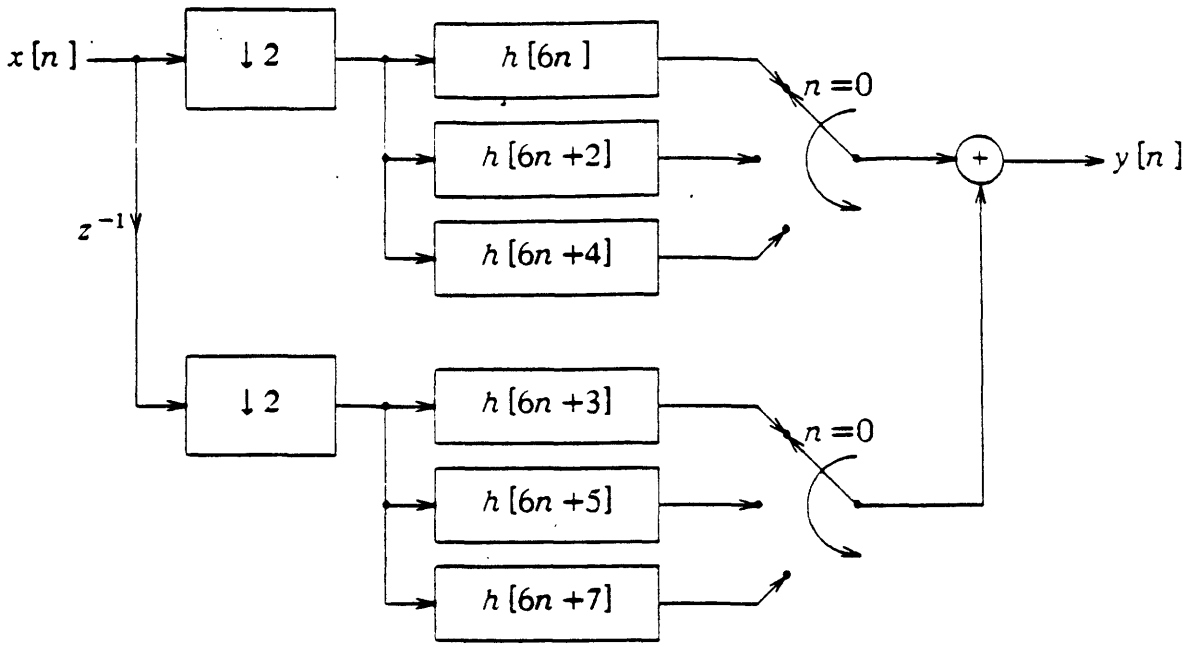


Fig. 6.19. A structure for (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2).

```
SPLICE: (LIST (SEQ-SETQ X (ABSTRACT-SIGNAL-CLASS :DISCRETE-TIME
                          :REAL-OR-COMPLEX :REAL))
              (SEQ-SETQ H (ABSTRACT-SIGNAL-CLASS :DISCRETE-TIME
                          :REAL-OR-COMPLEX :REAL :SUPPORT (INTERVAL 0 127)
                          :SYMMETRY :SYMMETRIC)))

------> (X H)


SPLICE: (IMPLEMENTATION (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2))

------> (DOWNSAMPLE (CONVOLVE-REAL-REAL-SYMMETRIC (UPSAMPLE X 3) H) 2)


SPLICE: (COST-PER-SAMPLE
          (IMPLEMENTATION (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2)))

------> (:REAL-MULTIPLIES 64 :REAL-ADDS 126)


SPLICE: (EFFICIENT-IMPLEMENTATIONS (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2))

------> (((DOWNSAMPLE                              ; Polyphase upsample and filter
            (INTERLEAVE                            ; followed by downsampling
              (CONVOLVE-REAL-REAL-SYMMETRIC X (DOWNSAMPLE H 3))
              (CONVOLVE-REAL-REAL X (DOWNSAMPLE (SHIFT H -1) 3))
              (CONVOLVE-REAL-REAL X (DOWNSAMPLE (SHIFT H -2) 3)))
            2)
          (:REAL-MULTIPLIES 35.33 :REAL-ADDS 41.33))


         ((ADD                                     ; Unusual structure
            (INTERLEAVE
              (CONVOLVE-REAL-REAL-SYMMETRIC (DOWNSAMPLE X 2) (DOWNSAMPLE H 6))
              (CONVOLVE-REAL-REAL (DOWNSAMPLE X 2) (DOWNSAMPLE (SHIFT H -2) 6))
              (CONVOLVE-REAL-REAL (DOWNSAMPLE X 2) (DOWNSAMPLE (SHIFT H -4) 6)))
            (INTERLEAVE
              (CONVOLVE-REAL-REAL-SYMMETRIC (DOWNSAMPLE (SHIFT X 1) 2)
                                            (DOWNSAMPLE (SHIFT H -3) 6))
              (CONVOLVE-REAL-REAL (DOWNSAMPLE (SHIFT X 1) 2)
                                  (DOWNSAMPLE (SHIFT H -5) 6))
              (CONVOLVE-REAL-REAL (DOWNSAMPLE (SHIFT X 1) 2)
                                  (DOWNSAMPLE (SHIFT H -7) 6))))
          (:REAL-MULTIPLIES 35.33 :REAL-ADDS 41.33))
         ...)
```

Fig. 6.20. Efficient implementations of (DOWNSAMPLE (CONVOLVE (UPSAMPLE X 3) H) 2).

6.20. First the user asks for the implementation of the original system and finds that the convolution can be specialized because both inputs are real and one is symmetric. Then the user asks for the cost per sample of the original implementation and finds that it to be 64 real multiplies and 126 real adds per sample. The user then asks for

efficient implementation and the system returns a list of implementations and their associated costs. The system has eliminated the original form and the polyphase downsample and filter of an upsampled signal as too expensive. The system has also replaced the general-purpose CONVOLVE operator with special purpose convolve operators for real signals and real signals with one being symmetric. The system finds that among the efficient implementations are the polyphase version of upsample and filter followed by downsampling and the unusual structure. It should be noted that if downsampling a convolution cannot be run in a demand driven form, i.e. the convolution can not be done sparsely, then the new structure is more efficient than the polyphase version of the upsample and filter followed by downsampling.

6.7. Summary

We have described E-SPLICE and given several examples of the symbolic manipulation of signals. The main features of E-SPLICE are summarized in Fig. 6.21. It provides extended signal objects, signal expression manipulation facilities, signal properties, and implementation analysis.

The examples we presented show the power of the system. It is able to analyze signal processing systems, derive important signal processing structures, and manipulate complicated signal processing expressions.

Extended Signal Objects

- Continuous-frequency signals
- Abstract signal classes

Signal Expression Manipulation

- Simplifications
- Equivalent Forms
- Manipulation of continuous-frequency signals

Signal Property Manipulation

- Support, Period, Real or Complex, Symmetry, Bandwidth
- Used in expression manipulation
- Used in implementation analysis

Implementation and Cost Analysis

- Simple cost measure
- Automatic analysis and selection of implementation

Fig. 6.21. Summary of E-SPLICE.

CHAPTER 7

Summary and Suggestions for Future Research

In this chapter we summarize the results of this thesis and suggest directions for future research.

7.1. Summary

This thesis has made several contributions towards the representation and manipulation of signals in computers. We have extended the representation of signals for numerical processing by both the development of a powerful signal model and the study of methods of defining systems. We have developed representations for signals and systems for symbolic processing and we have presented a system for the symbolic manipulation of signals.

7.1.1. Signal Representation for Numerical Processing

For the numerical representation and manipulation of signals, the deferred array has been proposed as a model. The deferred array combines an explicit functional description of a signal with a memory and deferred evaluation. We have shown that this model of signals closely matches the way in which signals are described mathematically and we have given examples of the use of the deferred array to define signals, including infinite duration signals, and signal processing operations.

The deferred array model was extended to incorporate multiple functions, constant values, default values, periodicity, and explicit representation of a signal's non-zero extent. Together, the deferred array and the extended deferred array define a

data abstraction model of signals, called the *outside view* of signals. In this view a signal is an object whose observable properties are its value at any index, from $-\infty$ to ∞, and its non-zero extent. Also, a signal is an immutable object that is uniquely defined by the system and the inputs to the system that created the signal. A signal can be defined without having to compute any of its sample values.

In addition to defining a data abstraction for signal objects we also identified four methods for building systems, called *inside views*. Systems specify signals by binding free parameters along with function definitions to create signal objects. The four models are the point operator model, the array operator model, the state machine model, and the composition model. In the point operator model a signal is defined by a function that maps an index into a signal value. The point operator model was used in the description of the deferred array. In the array operator model a signal is defined by a function that maps intervals of indices into arrays of values. The array operator model also requires a mechanism for the specification of the valid intervals for computation. In the state machine model a signal is defined by a pair of functions that map the current state and input into the next state and the current output. The state machine model also requires specification of an initial state. In the composition model a system is defined by the composition of other systems. We argued that a good signal representation provides facilities for the definition of systems using all four inside views while maintaining a common outside view.

We described the Signal Processing Language and Interactive Computing Environment (SPLICE), an environment for the exploratory development of signal processing algorithms, incorporating our signal representation ideas into an interactive environment. We discussed the outside view of signal objects and described how our different

models for building systems are used. We presented several examples of system definition and we discussed some implementation issues important to the development of SPLICE.

7.1.2. Signal Representation for Symbolic Processing

For the symbolic manipulation of signals, we have discussed the representation of signals and systems for symbolic manipulation. We have shown that symbolic manipulation of signals can be done on computers and that it provides useful capabilities. We have presented three categories of symbolic manipulations of signals - symbolic analysis of signal properties, rearrangement of signal processing expressions, and manipulation of signals of a continuous variable - and shown their use.

We discussed extensions to our signal representation for numerical processing to allow us to incorporate symbolic processing. We showed that signal types and histories, signal usage context, signal properties, and system properties were all important to capture in a signal representation for symbolic manipulation.

We extended our model of signal objects to incorporate more than discrete-time signals. We showed that it was important for the symbolic manipulation of signals to be able to represent signals of a continuous variable, such as continuous-time or continuous-frequency signals. Finally, we showed that it is important not to be constrained to manipulate individual signals but to be able to manipulate entire classes of signals simultaneously.

We described the Extended Signal Processing Language and Interactive Computing Environment (E-SPLICE), a system for the symbolic manipulation of signals. We discussed extended signal objects and those signal and system properties that are represented in the system. We presented several examples of the symbolic

manipulation of signals, including signal expression rearrangement, signal property analysis, manipulation of continuous-frequency signals, and cost analysis. We demonstrated the ability of the system to do the three classes of symbolic manipulations that we had described and we showed that symbolic manipulation of signals could be used in interesting signal processing problems.

7.2. Future Research

Both the areas of numerical and symbolic representation and manipulation of signals can be developed further. In this section we propose some directions for future research in each of these areas.

7.2.1. Signal Representation for Numerical Processing

One potential extension to our numerical signal representation is in the area of multi-dimensional signals. The idea that the value of a signal can be found at any index naturally extends to multiple dimensions. However, the idea of an interval of non-zero extent for one-dimensional signals has several potential extensions to multiple dimensions, including "rectangular" regions, interval of intervals, and parametrically described regions. Also important in the extension to multi-dimensional signals is the identification of computational models particular to multiple dimensions.

We have shown in our numerical signal representation the usefulness of deferred evaluation of signal values. Another area in which the relationship between deferred evaluation and signal representation can be studied is in the area of deferred evaluation of signal expressions. Our numerical signal representation assumes that when a system is invoked it looks at its input signals and produces an output signal. Another approach would be for the system to be given expressions for each of its input signals

and the system would produce an expression for its output signal. These expressions need not be evaluated until the signal object was required. This approach could be useful in the expression of some signal processing algorithms, particularly those that use recursive interconnection of signal objects.

In our work on the numerical representation of signals we identified four models of systems. Further work should be directed toward the identification of other models, such as parallel computation models, that can be incorporated into the uniform outside view of signals.

A major research project related to both our numerical representation and our symbolic manipulation of signals would be the development of a "signal processing compiler". Potential areas of study include the translation of less efficient computational models, such as point operators, into more efficient models, such as array operators, removal of the overhead of index checking and cache lookup, storage management, algorithm rearrangement, and translation from high level representation into low level instructions. Analysis of signal processing expressions to predict what samples of a signal will be required so that they can be computed as soon as possible, i.e. eager evaluation, is another potential area of research.

7.2.2. Signal Representation for Symbolic Processing

An obvious place for further work in the area of symbolic manipulation of signals is the development of a more complete system. E-SPLICE provides a set of tools that show the potential of symbolic manipulation but these tools must be extended and refined before a production quality system is achieved. Particularly, the number of signal processing systems that can be manipulated by the system needs to be expanded, the cost measures used by the system needs to be made more realistic

(memory usage, data transfers, processors required, delay, parallel structure, etc.), and other useful signal properties (signal to noise ratio, sampling rate, etc.) need to be incorporated. The system needs more information about the relationships between signals of a continuous variable and sampled versions of these signals. Also, the cost analysis procedure must be made capable of recognizing shared results.

Further work is also required in controlling symbolic manipulations. For example, the simplification process used in E-SPLICE is a pragmatic solution to a difficult problem that requires more study. The difference between simplification and rearrangement needs to be explored further and methods for goal driven simplification need to be examined.

Extensions to the notion of abstract signal classes also deserve further exploration. Removal of the distinction between numerically-valued signals and abstract signal classes by specifying all signals are really abstract signal classes, i.e. they have properties that can be found out and properties that can not be found out, could provide a common structure for both numerical and symbolic processing.

Another area for research is the specification of new classes of symbolic manipulations of signals and the associated signal representations that are required. For example, reasoning about parallel structures for signal processing would require extraction of the relevant signal processing rules and development of the representations for parallel structures. Symbolic manipulation of signals in specialized fields of signal processing, such as filter design, spectral estimation, etc., is another possibility.

A different approach to the development of a system for the symbolic manipulation of signals would be the development of "smart" signal objects that can perform certain operations on themselves. For example, certain signals could "know" their

Fourier transform and the process of generating Fourier transforms would become a combination of rule processing and questioning of signal objects. One type of "smart" signal object has already appeared in the context of an acoustical data analysis system [32].

Symbolic manipulation of signals could be useful in the development of a design assistant for signal processing. Such a system could be used for automatic error checking at a high level, such as checking that when a linear convolution is implemented in the frequency domain the FFT size specified by the user is large enough, or checking that the filter specified in a filter and downsample structure provides enough rejection to prevent aliasing. A design assistant could be used to pick algorithm parameter values based on formulas and propagation of constraints [39].

In the long range, symbolic manipulation of signals should be coupled with the design of signal processing systems from high level descriptions. Ideally, it should be possible for a user to tell a system that the input data is in a particular form with certain properties and tell the system what the desired information is and have the system design and optimize a signal processing structure to do the required data analysis.

APPENDIX A

SCHEME Syntax

This appendix describes the syntax of SCHEME and defines the operators used in this thesis. This description is intended only to provide enough background for the reader to follow the material in this thesis. The definitive description of SCHEME is given in [25] and definitions of the operations that we use from ZETALISP are given in [26].

A.1. Expressions

Expressions in SCHEME may either be atomic or compound. Atomic expressions are numbers, such as 12, 45.67, or 2.3e5, strings, such as "hello world", or symbols, such as HELLO, +, or X. Compound expressions are lists of expressions, such as (ELEMENT-1 ELEMENT-2), (A B C D), or (+ 3 4). The list of no expressions is (). Since compound expressions are themselves expressions, compound expressions may be nested, as ((HELLO)) or (+ (* 3 4) (* 4 5)).

All expressions in SCHEME can be evaluated. Numbers and strings evaluate to themselves, e.g. the value of the expression 5 is 5. Symbols evaluate to the value of the symbol in the current environment. For example, the symbol + evaluates to the function that adds numbers. Compound expressions begin evaluation by evaluating the first element in the list. The result must either be a function or a special form. If it is a function it is applied to the result of evaluating the remaining elements in the list. Thus, (+ 3 4) evaluates to 7. Since evaluation is recursive, nested compound

expressions, such as (+ (* 3 4) (* 5 6)), evaluate correctly. If the first element in the list evaluates to a special form then the special form is applied to the remaining elements in the list, without evaluation of the remaining elements. We will give an example of a special form in the next section.

A.2. Defines

The special form DEFINE is the method by which symbols are given values in SCHEME. DEFINE may be used to give a symbol a value, as

(DEFINE *symbol value*)

Thus, (DEFINE X 3) assigns to the symbol X the value 3. After this assignment, evaluating the symbol X would return the value 3 and so, (DEFINE Y (+ X 4)) would assign the symbol Y the value 7. DEFINE is a special form because its first argument is not evaluated.

DEFINE can also be used to define functions. For purposes of defining functions, the syntax of DEFINE is

(DEFINE (*function-name parameters*)
  body)

For example

```
(DEFINE (PLUS X Y)
  (+ X Y))
```

defines the symbol PLUS to have as its value a function of two arguments, X and Y, that adds X and Y. After this definition, the value of the expression (PLUS 3 4) is 7.

The body of a function definition may be any number of expressions. When a function is applied to some arguments the body is evaluated in an environment in which the parameters of the function are bound to the actual values. The forms in the

body are evaluated and the result is the value of the last form. The process may be pictured as one in which the actual values for the parameters of the function are substituted for the formal parameter names in the body and then the body is evaluated.

Two features of SCHEME are important in our presentation.[1] One is that SCHEME is a lexically scoped language. In a lexically scoped language the environment in which a free variable is looked up is determined by the environment in which the function is defined. A free variable is one that is not a formal parameter of the current function. For example, in

```
(DEFINE (ADD-TO-Y X)
  (+ X Y))
```

the variable Y is a free variable and its value will be found in the environment in which ADD-TO-Y is defined.

The other important feature of SCHEME is it allows functions to manipulated as data. Functions can be passed as arguments or returned as values. Functions may also be defined as part of body of some other function. For example,

```
(DEFINE (MAKE-ADDER X)
  (DEFINE (INTERNAL-ADDER Y)
    (+ Y X))
  INTERNAL-ADDER)
```

defines a function MAKE-ADDER of one argument, X, that internally defines a function INTERNAL-ADDER and returns as its value the internal function. In particular, the value of (MAKE-ADDER 3) is a function of one argument, Y, that adds 3 to the argument. Thus the value of ((MAKE-ADDER 3) 4) is 7. Remember, the evaluation rule is to evaluate the first element of the compound expression, (MAKE-ADDER 3),

[1] Most dialects of LISP, and many other programming languages, provide features similar to these and neither feature is critical to our signal representation. They simply make it easier to talk about the structure of the model.

and apply the resulting function to the result of evaluating the remaining arguments.

A.3. Operations

Most dialects of LISP have several hundred functions available. The operators defined in SCHEME that we use in this thesis are the following:

LIST arg_1 arg_2 ... arg_N *function*
Create a list containing arg_1, arg_2, ..., arg_N as elements.

+, -, *, / x y *functions*
Do the appropriate arithmetic operation on x and y.

MINUS x *function*
Negate x.

FLOOR x *function*
Integer less than or equal to x.

MOD x y *function*
x modulo y.

COS, SIN x *functions*
Trigonometric functions of x.

>, >=, <, <=, = x y *functions*
Arithmetic comparison of x and y.

PRINT *arg* *function*
Print *arg*.

IF *conditional then-part else-part* *special form*
If the *conditional* evaluates to non-FALSE the value is the value of the *then-part*, otherwise it the value of the *else-part*. Only the appropriate part is evaluated.

The following operators manipulate arrays and were taken from ZETALISP. Arrays in ZETALISP start at index zero.

ARRAY-LENGTH *array* *function*
Return the number of elements in *array*.

AREF *array index* *function*

Return the value of *array* at position *index*. It is an error to access *array* for *index* less than zero or greater than or equal to the length of *array*.

ASET *value array index* *function*

Modify *array* so that its value at position *index* is *value*. It is an error to access *array* for *index* less than zero or greater than or equal to the length of *array*.

The LOOP operator is a ZETALISP special form for iteration. We will define only the portion of the LOOP syntax that we use in this thesis.

LOOP *iteration-clauses action-clauses* *special form*

LOOP performs iteration. The *iteration-clauses* index a variable through a range of values and are either of the form

FOR *var* **FROM** *start* [**BELOW** *end*] [**BY** *increment*] *clause*

for numerically indexing *var* from *start* below *end* by *increment*

or of the form

FOR *var* **IN** *list* *clause*

for indexing *var* through a list.

The *action-clauses* perform action in the iteration and are of the form

DOING *action* *clause*

for performing the action at each iteration

or

SUMMING *value* *clause*

for summing *value* during the iteration

or

COLLECTING *value* *clause*

for collecting *value* into a list during the iteration

The following two operators are not defined in either SCHEME or ZETALISP but were used to illustrate the behavior of streams. We assume that a stream object is a first-in-first-out (FIFO) queue. The operations that can be performed on a stream are:

GET-STREAM *stream* *function*

Get the next value from *stream*. Modifies *stream* by removing that value.

PUT-STREAM *value stream* *function*

Put *value* into *stream*. When *value* will be removed from the stream depends on how full the stream currently is.

The following operations are used in our description of deferred arrays and extended deferred arrays. SCHEME implementations of these operations can be found in Appendix B.

MAKE-DEFERRED-ARRAY *function* *function*

Create a deferred array using *function*. The value of the resulting deferred array at any index is determined by applying *function* to the index.

FETCH *deferred-array index* *function*

Get the value of *deferred-array* at *index*. Also works for extended deferred arrays.

MAKE-EXTENDED-DEFERRED-ARRAY *default-function-or-value interval*$_1$ *function-or-value*$_1$ ... *interval*$_N$ *function-or-value*$_N$ *function*

Create an extended deferred array with default *default-function-or-value* and breaking the index axis into intervals and associated functions or values using *interval*$_1$ *function-or-value*$_1$ ... *interval*$_N$ *function-or-value*$_N$.

SUPPORT *extended-deferred-array* *function*

Return an interval outside which *extended-deferred-array* is guaranteed to be zero.

INTERVAL *start end* *function*

Create an object to represent the interval of indices *start* $\leqslant n <$ *end*.

INTERVAL-START *interval* *function*

Return the start of *interval*.

INTERVAL-END *interval* *function*

Return the end of *interval*.

APPENDIX B

Implementation of

Deferred Arrays in SCHEME

This appendix presents two implementations of deferred arrays in SCHEME and one implementation of extended deferred arrays.

B.1. Implementation of Deferred Arrays

```
;;;
;;; Deferred Arrays - SCHEME implementation
;;;
;;; Deferred Array Structure is a list of the first index in the
;;; memory (start), one beyond the last index in the memory (end), the
;;; memory, and the function.
;;;
;;; This code assumes that the value of a deferred array is never nil.
;;; Would need either a special token for "empty", a separate vector
;;; to record full or empty, or extra code to make sure the vector is
;;; always full.
;;;

;;;
;;; Create a Deferred Array
;;;
;;; Memory is initially empty
;;;

(DEFINE (MAKE-DEFERRED-ARRAY FCN)
  (LIST 0 0 (VECTOR-CONS 0 NIL) FCN))

;;;
;;; Access Portions of the Structure
;;;
;;; Structure consists of start index, end index (one beyond last
;;; index), memory, and the function.
;;;

(DEFINE (DA-MEMORY-START DEFERRED-ARRAY)
  (FIRST DEFERRED-ARRAY))

(DEFINE (SET-DA-MEMORY-START! DEFERRED-ARRAY NEW-START)
  (SET-CAR! DEFERRED-ARRAY NEW-START))

(DEFINE (DA-MEMORY-END DEFERRED-ARRAY)
  (SECOND DEFERRED-ARRAY))

(DEFINE (SET-DA-MEMORY-END! DEFERRED-ARRAY NEW-END)
  (SET-CAR! (CDR DEFERRED-ARRAY) NEW-END))
```

```
(DEFINE (DA-MEMORY DEFERRED-ARRAY)
  (THIRD DEFERRED-ARRAY))

(DEFINE (SET-DA-MEMORY-MEMORY! DEFERRED-ARRAY NEW-MEMORY)
  (SET-CAR! (CDDR DEFERRED-ARRAY) NEW-MEMORY))

(DEFINE (DA-FUNCTION DEFERRED-ARRAY)
  (FOURTH DEFERRED-ARRAY))

;;;
;;; Get a sample - look it up or compute it and remember it
;;;

(DEFINE (FETCH DEFERRED-ARRAY INDEX)
  (LET ((OLD (LOOKUP DEFERRED-ARRAY INDEX)))
    (IF OLD OLD
        (REMEMBER ((DA-FUNCTION DEFERRED-ARRAY) INDEX)
                  DEFERRED-ARRAY INDEX))))

;;;
;;; Look up a sample in the memory
;;;

(DEFINE (LOOKUP DEFERRED-ARRAY INDEX)
  (LET ((START (DA-MEMORY-START DEFERRED-ARRAY))
        (END (DA-MEMORY-END DEFERRED-ARRAY))
        (MEMORY (DA-MEMORY DEFERRED-ARRAY)))
    (IF (AND (<= START INDEX) (> END INDEX))
        (VECTOR-REF MEMORY (- INDEX START)))))

;;;
;;; Store a sample in the memory
;;;
;;; Create a new memory if needed and copy the old values
;;;

(DEFINE (REMEMBER VALUE DEFERRED-ARRAY INDEX)
  (LET ((START (DA-MEMORY-START DEFERRED-ARRAY))
        (END (DA-MEMORY-END DEFERRED-ARRAY))
        (MEMORY (DA-MEMORY DEFERRED-ARRAY)))
    (COND ((AND (<= START INDEX) (< INDEX END))
           (VECTOR-SET! MEMORY (- INDEX START) VALUE))
          ((<= START INDEX)
           (LET ((NEW-MEMORY (VECTOR-CONS (1+ (- INDEX START)) NIL)))
             (COPY-VECTOR-PORTION MEMORY 0 (- END START)
                                  NEW-MEMORY 0 (- END START))
             (SET-DA-MEMORY-END! DEFERRED-ARRAY (1+ INDEX))
             (SET-DA-MEMORY-MEMORY! NEW-MEMORY)
             (VECTOR-SET! NEW-MEMORY (- INDEX START) VALUE)))
          (T
           (LET ((NEW-MEMORY (VECTOR-CONS (- END INDEX) NIL)))
             (COPY-VECTOR-PORTION MEMORY 0 (- END START)
                                  NEW-MEMORY
                                  (- START INDEX)
                                  (- END INDEX))
             (SET-DA-MEMORY-START! DEFERRED-ARRAY INDEX)
             (SET-DA-MEMORY-MEMORY! NEW-MEMORY)
             (VECTOR-SET! NEW-MEMORY 0 VALUE)))))
  VALUE)

;;;
;;; Copy one vector into another
;;;

(DEFINE (COPY-VECTOR-PORTION FROM FROM-START FROM-END TO TO-START TO-END)
  (DEFINE (ITER I J)
    (IF (AND (< I FROM-END) (< J TO-END))
```

```
        (SEQUENCE
          (VECTOR-SET! TO J (VECTOR-REF FROM I))
          (ITER (1+ I) (1+ J)))))
  (ITER FROM-START TO-START))

;;;
;;; Examples
;;;

;;;
;;; Cos-signal - cosine wave with specified period
;;;

(DEFINE (COS-SIGNAL PERIOD)
  (DEFINE (FCN N) (COS (/ (* 2 PI N) PERIOD)))
  (MAKE-DEFERRED-ARRAY FCN))

;;;
;;; Add-signals - add two signals
;;;

(DEFINE (ADD-SIGNALS X Y)
  (DEFINE (FCN N) (+ (FETCH X N) (FETCH Y N)))
  (MAKE-DEFERRED-ARRAY FCN))
```

B.2. Another Implementation of Deferred Arrays

```
;;;
;;; Deferred Arrays - SCHEME Implementation
;;;
;;; Deferred arrays are functions that can be applied to indices to
;;; get values.
;;;
;;; Deferred Array Structure is an environment containing the first
;;; index in the memory (start), one beyond the last index in the
;;; memory (end), and the memory.
;;;
;;; This code assumes that the value of a deferred array is never nil.
;;; Would need either a special token for "empty", a separate vector
;;; to record full or empty, or extra code to make sure the vector is
;;; always full.
;;;

;;;
;;; Create a Deferred Array
;;;

(DEFINE (MAKE-DEFERRED-ARRAY FUNCTION)
  (LET ((START 0)
        (END 0)
        (MEMORY (VECTOR-CONS 0 NIL)))
    (DEFINE (LOOKUP INDEX)
      (IF (AND (<= START INDEX) (> END INDEX))
          (VECTOR-REF MEMORY (- INDEX START))))
    (DEFINE (REMEMBER VALUE INDEX)
      (COND ((AND (<= START INDEX) (> END INDEX))
             (VECTOR-SET! MEMORY (- INDEX START) VALUE))
            ((<= START INDEX)
             (LET ((NEW-MEMORY (VECTOR-CONS (1+ (- INDEX START)) NIL)))
               (COPY-VECTOR-PORTION MEMORY 0 (- END START)
                                    NEW-MEMORY 0 (- END START))
               (SET! END (1+ INDEX))
               (SET! MEMORY NEW-MEMORY)
               (VECTOR-SET! NEW-MEMORY (- INDEX START) VALUE)))
```

```
            (T
             (LET ((NEW-MEMORY (VECTOR-CONS (- END INDEX) NIL)))
               (COPY-VECTOR-PORTION MEMORY 0 (- END START)
                                    NEW-MEMORY
                                    (- START INDEX)
                                    (- END INDEX))
               (SET! START INDEX)
               (SET! MEMORY NEW-MEMORY)
               (VECTOR-SET! NEW-MEMORY 0 VALUE))))
      VALUE)
    (DEFINE (DEFERRED-ARRAY INDEX)
      (LET ((OLD (LOOKUP INDEX)))
        (IF OLD OLD (REMEMBER (FUNCTION INDEX) INDEX))))
    DEFERRED-ARRAY))

;;;
;;; Get a sample - apply the Deferred Array to the index
;;;

(DEFINE (FETCH DEFERRED-ARRAY INDEX)
  (DEFERRED-ARRAY INDEX))

;;;
;;; Copy one vector into another
;;;

(DEFINE (COPY-VECTOR-PORTION FROM FROM-START FROM-END TO TO-START TO-END)
  (DEFINE (ITER I J)
    (IF (AND (< I FROM-END) (< J TO-END))
        (SEQUENCE
          (VECTOR-SET! TO J (VECTOR-REF FROM I))
          (ITER (1+ I) (1+ J)))))
  (ITER FROM-START TO-START))

;;;
;;; Examples
;;;

;;;
;;; Cos-signal - cosine wave with specified period
;;;

(DEFINE (COS-SIGNAL PERIOD)
  (DEFINE (FCN N) (COS (/ (* 2 PI N) PERIOD)))
  (MAKE-DEFERRED-ARRAY FCN))

;;;
;;; Add-signals - add two signals
;;;

(DEFINE (ADD-SIGNALS X Y)
  (DEFINE (FCN N) (+ (X N) (Y N)))
  (MAKE-DEFERRED-ARRAY FCN))
```

B.3. Implementation of Extended Deferred Arrays

```
;;;
;;; Extended Deferred Arrays - SCHEME implementation
;;;
;;; Extended Deferred Array is an environment of a list of eda-portions,
;;; a default eda-portion, and the support
;;;
;;; This code assumes that the value of an extended deferred array is
;;; never nil.  Would need either a special token for "empty", a
```

```
;;; separate vector to record full or empty, or extra code to make sure
;;; the vector is always full.
;;;
;;; This code is wasteful of space if the default is broken up by many
;;; other functions and requires a memory because its memory can
;;; overlap the memory of the other functions.
;;;

;;;
;;; Create an Extended Deferred Array
;;;

(DEFINE (MAKE-EXTENDED-DEFERRED-ARRAY DEFAULT-FCN-OR-VALUE . PAIRS)
  (DEFINE (MAKE-EDA-PORTION-LIST LIST)
    (IF LIST
        (CONS (MAKE-EDA-PORTION (FIRST LIST) (SECOND LIST))
              (MAKE-EDA-PORTION-LIST (CDDR LIST)))))
  (DEFINE (SUPPORT-LIST LIST)
    (IF LIST
        (LET ((SUPPORT (IF (EQ? (SECOND LIST) 0)
                           NULL-INTERVAL
                           (FIRST LIST))))
          (COVER-INTERVALS SUPPORT (SUPPORT-LIST (CDDR LIST))))
        NULL-INTERVAL))
  (LET ((EDA-PORTION-LIST (MAKE-EDA-PORTION-LIST PAIRS))
        (DEFAULT-EDA-PORTION
          (MAKE-EDA-PORTION COMPLETE-INTERVAL DEFAULT-FCN-OR-VALUE))
        (SUPPORT (COVER-INTERVALS (IF (EQ? DEFAULT-FCN-OR-VALUE 0)
                                      NULL-INTERVAL
                                      COMPLETE-INTERVAL)
                                  (SUPPORT-LIST PAIRS))))
    (DEFINE (LOOKUP-LIST LIST INDEX)
      (IF LIST
          (LET ((VAL ((FIRST LIST) INDEX)))
            (IF VAL VAL (LOOKUP-LIST (CDR LIST) INDEX)))))
    (DEFINE (EXTENDED-DEFERRED-ARRAY MESSAGE)
      (COND ((NUMBER? MESSAGE)
             (LET ((VAL (LOOKUP-LIST EDA-PORTION-LIST MESSAGE)))
               (IF VAL VAL (DEFAULT-EDA-PORTION MESSAGE))))
            ((EQ? MESSAGE 'SUPPORT) SUPPORT)))
    EXTENDED-DEFERRED-ARRAY))

;;;
;;; Get a sample
;;;

(DEFINE (FETCH EXTENDED-DEFERRED-ARRAY INDEX)
  (EXTENDED-DEFERRED-ARRAY INDEX))

;;;
;;; Get the non-zero support
;;;

(DEFINE (SUPPORT EXTENDED-DEFERRED-ARRAY)
  (EXTENDED-DEFERRED-ARRAY 'SUPPORT))

;;;
;;; Interval creation and manipulation - Interval is an active data object.
;;;

(DEFINE (INTERVAL START END)
  (DEFINE (INTERVAL-OBJECT MESSAGE . ARGS)
    (COND ((EQ? MESSAGE 'START) START)
          ((EQ? MESSAGE 'END) END)
          ((EQ? MESSAGE 'COVERS?)
           (LET ((INDEX (FIRST ARGS)))
             (AND (IF (EQ? START MINF) T (<= START INDEX))
```

```
                (IF (EQ? END INF) T (> END INDEX)))))))
  INTERVAL-OBJECT)

(DEFINE (INTERVAL-START INTERVAL)
  (INTERVAL 'START))

(DEFINE (INTERVAL-END INTERVAL)
  (INTERVAL 'END))

(DEFINE (INTERVAL-COVERS? INTERVAL INDEX)
  (INTERVAL 'COVERS? INDEX))

(DEFINE (COVER-INTERVALS INTERVAL-1 INTERVAL-2)
  (DEFINE ($MIN X Y)
    (COND ((EQ? X INF) Y)
          ((EQ? Y INF) X)
          ((OR (EQ? X MINF) (EQ? Y MINF)) MINF)
          (T (MIN X Y))))
  (DEFINE ($MAX X Y)
    (COND ((EQ? X MINF) Y)
          ((EQ? Y MINF) X)
          ((OR (EQ? X INF) (EQ? Y INF)) INF)
          (T (MAX X Y))))
  (INTERVAL ($MIN (INTERVAL-START INTERVAL-1) (INTERVAL-START INTERVAL-2))
            ($MAX (INTERVAL-END INTERVAL-1) (INTERVAL-END INTERVAL-2)))))

(DEFINE MINF '*MINUS-INFINITY*)

(DEFINE INF '*INFINITY*)

(DEFINE NULL-INTERVAL (INTERVAL INF MINF))

(DEFINE COMPLETE-INTERVAL (INTERVAL MINF INF))

;;;
;;; EDA-Portion - a portion of an extended deferred array
;;;
;;; Implemented like a deferred array except:
;;;
;;; 1. Checks interval before fetching
;;;
;;; 2. No memory is used if value is a constant
;;;

(DEFINE (MAKE-EDA-PORTION INTERVAL FCN-OR-VALUE)
  (LET ((START 0)
        (END 0)
        (MEMORY (VECTOR-CONS 0 NIL)))
    (DEFINE (LOOKUP INDEX)
      (IF (AND (<= START INDEX) (> END INDEX))
          (VECTOR-REF MEMORY (- INDEX START))))
    (DEFINE (REMEMBER VALUE INDEX)
      (COND ((AND (<= START INDEX) (> END INDEX))
             (VECTOR-SET! MEMORY (- INDEX START) VALUE))
            ((<= START INDEX)
             (LET ((NEW-MEMORY (VECTOR-CONS (1+ (- INDEX START)) NIL)))
               (COPY-VECTOR-PORTION MEMORY 0 (- END START)
                                    NEW-MEMORY 0 (- END START))
               (SET! END (1+ INDEX))
               (SET! MEMORY NEW-MEMORY)
               (VECTOR-SET! NEW-MEMORY (- INDEX START) VALUE)))
            (T
             (LET ((NEW-MEMORY (VECTOR-CONS (- END INDEX) NIL)))
               (COPY-VECTOR-PORTION MEMORY 0 (- END START)
                                    NEW-MEMORY
                                    (- START INDEX)
                                    (- END INDEX))
```

```
            (SET! START INDEX)
            (SET! MEMORY NEW-MEMORY)
            (VECTOR-SET! NEW-MEMORY 0 VALUE))))
      VALUE)
    (DEFINE (EDA-PORTION INDEX)
      (IF (INTERVAL-COVERS? INTERVAL INDEX)
          (IF (NUMBER? FCN-OR-VALUE)
              FCN-OR-VALUE
              (LET ((OLD (LOOKUP INDEX)))
                (IF OLD OLD
                    (REMEMBER (FCN-OR-VALUE INDEX) INDEX))))))
    EDA-PORTION))

;;;
;;; Copy one vector into another
;;;

(DEFINE (COPY-VECTOR-PORTION FROM FROM-START FROM-END TO TO-START TO-END)
  (DEFINE (ITER I J)
    (IF (AND (< I FROM-END) (< J TO-END))
        (SEQUENCE
          (VECTOR-SET! TO J (VECTOR-REF FROM I))
          (ITER (1+ I) (1+ J)))))
  (ITER FROM-START TO-START))

;;;
;;; Examples
;;;

;;;
;;; Hamming window
;;;

(DEFINE (HAMMING-WINDOW L)
  (DEFINE (FCN N)
    (+ .54 (* .46 (COS (/ (* 2 PI N) (- L 1))))))
  (MAKE-EXTENDED-DEFERRED-ARRAY 0
    (INTERVAL (MINUS (FLOOR (/ L 2))) (- L (FLOOR (/ L 2)))) FCN))

;;;
;;; Cos-signal - cosine wave with specified period
;;;

(DEFINE (COS-SIGNAL L)
  (DEFINE (FCN N)
    (COS (/ (* 2 PI N) L)))
  (DEFINE (DEFAULT-FCN N)
    (FETCH SELF (MOD N L)))
  (DEFINE SELF
    (MAKE-EXTENDED-DEFERRED-ARRAY DEFAULT-FCN
      (INTERVAL 0 L) FCN))
  SELF)

;;;
;;; Add-signals - add two signals
;;;

(DEFINE (ADD-SIGNALS X Y)
  (DEFINE (FCN N) (+ (X N) (Y N)))
  (MAKE-EXTENDED-DEFERRED-ARRAY 0
    (COVER-INTERVALS (SUPPORT X) (SUPPORT Y)) FCN))
```

APPENDIX C

The Signal Processing Language and Interactive Computing Environment

This appendix provides a brief description of some features of the Signal Processing Language and Interactive Computing Environment (SPLICE). The intent of this appendix is show the wide range of tools available in SPLICE. A more complete, although somewhat dated, description can be found in [27].

C.1. Extended Numbers and Intervals

Extended numbers are the set of integers, reals, and $\pm\infty$. SPLICE provides facilities for manipulation of $\pm\infty$. Generally, functions that manipulate extended numbers have a "$" prepended to the name of the function that manipulates normal numbers. Included are the following:

MINF, INF *constants*
Minus infinity and infinity.

$>, $>=, $<, $<=, $= *x y* *functions*
Arithmetic comparison of extended numbers *x* and *y*.

$+, $-, $*, $// *x y* *functions*
Arithmetic operations of add, subtract, multiply, and divide for extended numbers *x* and *y*.

$MINUS *x* *function*
Negation of extended number *x*.

$MIN, $MAX *x y* *functions*
Minimum and maximum of extended numbers *x* and *y*.

$ *x y* *function*
Extended number remainder of *x* divided by *y*.

Intervals represent connected sets of indices. Operations for creating and examining intervals are the following:

INTERVAL *start end* *function*
Create an object to represent the interval of indices *start* $\leqslant n <$ *end*.

INTERVAL-P *thing* *function*
Predicate to test if *thing* is an interval.

INTERVAL-START *interval* *function*
Return the start of *interval*.

INTERVAL-END *interval* *function*
Return the end of *interval*.

NULL-INTERVAL *constant*
The empty interval.

NULL-INTERVAL-P *interval* *function*
Predicate to test if interval is the empty interval.

INTERVAL-LENGTH *interval* *function*
Size of *interval*.

FINITE-INTERVAL-P, INFINITE-INTERVAL-P *interval* *functions*
Predicates to check if *interval* is finite or infinite.

INTERVAL-INTERSECT-P $interval_1$ ... $interval_N$ *function*
Predicate to test if the intervals $interval_1$ ... $interval_N$ all intersect.

INTERVAL-COVERS-P *interval x* *function*
Predicate to check if *interval* completely contains *x*. *x* may either be an interval or an extended number.

INTERVAL-EQ *x y* *function*
Predicate to check if *x* and *y* are the same interval.

The following operations take intervals as input and produce an interval as output:

INTERVAL-INTERSECT $interval_1$ ... $interval_N$ *function*
Intersection of the intervals $interval_1$ ... $interval_N$.

INTERVAL-COVER *interval*$_1$ ... *interval*$_N$ *function*
Interval that completely contains the intervals *interval*$_1$ ... *interval*$_N$.

INTERVAL-ADVANCE, INTERVAL-RETARD *interval amount* *functions*
New interval advanced to the right or retarded to the left of *interval* by *amount*.

INTERVAL-REVERSE *interval* *function*
Time reversed interval from *interval*.

The following generic functions work for both intervals and sequences:

$START *object* *function*
Start of *object*. First index in *object* if it is an interval, or first index in the support of *object* if it is a sequence.

$END *object* *function*
End of *object*.

$LENGTH *object* *function*
Length of *object*. Size of interval if *object* is an interval, and size of non-zero support of *object* if it is a sequence.

C.2. Getting Information From Sequences

The following functions can be used to get the value of a sequence at any index:

FETCH, FETCH-IMAGINARY *sequence index* *functions*
Gets the real part or imaginary part of the value of *sequence* at *index*.

FETCH-COMPLEX *sequence index* *function*
Gets both the real part and the imaginary part of *sequence* at *index*. Real and imaginary parts are returned as multiple values.

The following functions can be used to get an interval of sequence values:

FETCH-INTERVAL *sequence interval [array] [cached]* *function*
Return an array of the real part of sequence values for *sequence* over *interval*. If *array* is given then use that array for the results. If *cached* is specified as :NO then the sequence does not remember it has computed the values. The default is to cache sequence values.

FETCH-IMAGINARY-INTERVAL *sequence interval [array] [cached]* *function*
Return an array of the imaginary part of sequence values for *sequence* over *interval*.

FETCH-COMPLEX-INTERVAL *sequence interval [real-array] [imag-array] [cached]* *function*

Return two arrays, one for the real part and one for the imaginary part of *sequence* over *interval*.

The following functions return information about sequences:

SEQUENCEP *object* *function*

Predicate to test if *object* is a sequence.

SUPPORT *sequence* *function*

Return an interval outside which *sequence* is guaranteed to be zero.

PERIOD *sequence* *function*

Return the period of *sequence*. INF if *sequence* is not periodic.

ATOMIC-TYPE *sequence* *function*

Return the basic type of *sequence*. :REAL for real-valued sequences, :COMPLEX for complex-valued sequences, and :UNKNOWN for others.

RANGE *sequence [interval]* *function*

Interval representing the range on the real part of *sequence* over *interval*. If *interval* is not specified it defaults to the non-zero extent of *sequence*.

IMAGINARY-RANGE *sequence [interval]* *function*

Interval representing the range on the imaginary part of *sequence* over *interval*.

SHOW *sequence* *function*

Return a list that describes how *sequence* was created.

SHOWR *sequence [recursion-levels]* *function*

Return a list that describes how *sequence* was created and how the parts that compose *sequence* were created. Continue describing the parts down *recursion-level* levels. The default is to keep describing recursively until objects that cannot be decomposed are encountered.

SEQ-PUTPROP *sequence property value* *function*

Assign the value of the property *property* of *sequence* to be *value*.

SEQ-GET *sequence property* *function*

Find the value of the property *property* of *sequence*.

SEQ-TYPEP *sequence [type]* *function*

If *type* is given check if *sequence* is of type *type*. If *type* is not given then return the type of *sequence*.

The following functions deal with the association between symbols and sequences:

SEQ-SETQ *symbol sequence* *special form*
Assign *symbol* to have as its value *sequence* and name *sequence* as *symbol*. *symbol* is not evaluated.

UNNAME *sequence* *function*
Remove a name from *sequence*.

SEQ-NAME *sequence* *function*
Name by which *sequence* can be referred. Either the name of a symbol whose value is *sequence* or a description of *sequence*.

The following functions get numerical information from sequences:

SUM-OF-SEQ *sequence [interval]* *function*
Sum values of *sequence* over *interval*. Default for *interval* is the non-zero support of *sequence*. Returns two values, one for the real part and one for the imaginary part.

MEAN-OF-SEQ *sequence [interval]* *function*
Average value of *sequence* over *interval*.

SUM-SQUARE-OF-SEQUENCE *sequence [interval]* *function*
Sum of the magnitude squared of the values of *sequence* over *interval*.

MEAN-SQUARE-OF-SEQUENCE *sequence [interval]* *function*
Average magnitude squared of the values of *sequence* over *interval*.

VARIANCE-OF-SEQUENCE *sequence [interval]* *function*
Average variance of *sequence* over *interval*.

The following functions are used for plotting purposes:

PLOT *sequence [window]* *function*
Plot *sequence*. If *window* is specified then use it. If *window* is NIL then ask user for a window. Otherwise, pick one for the user.

OVERLAY-PLOT *sequence [window]* *function*
Plot *sequence* without clearing *window*.

H-SPAN *sequence* *function*
The interval for *sequence* that will be used as the horizontal axis for plotting.

V-SPAN *sequence [interval]* *function*

The interval for *sequence* that will be used as the vertical axis for plotting the real part when plotting over *interval*. Default *interval* is the non-zero support of *sequence*.

IMAG-V-SPAN *sequence [interval]* *function*

The interval for *sequence* that will be used as the vertical axis for plotting the imaginary part when plotting over *interval*. Default *interval* is the non-zero support of *sequence*.

C.3. Defining Systems

The primary programming activity in SPLICE is defining systems for creating signals and the primary method of defining systems is with the special form DEFINE-SYSTEM. The syntax for DEFINE-SYSTEM is

```
(DEFINE-SYSTEM name parameter-list
               (parent-signal-class)
  [documentation-string]
  shared-functions)
```

DEFINE-SYSTEM defines both a function, *name*, and the signal class *name*. The function *name* takes as input the parameters given in *parameter-list* and creates a signal as output. The signal class *name* is a specialization of *parent-signal-class*.

Signal classes are implemented using ZETALISP flavors [26]. The following are the basic signal classes in SPLICE:

BASIC-SEQUENCE *signal class*

The fundamental signal class for discrete-time sequences. All other signal classes are built on this. This class is almost never used directly.

SEQUENCE *signal class*

Built from BASIC-SEQUENCE, this signal class provides buffering of signal values. This class is used for symbolically-valued signals.

NUMERIC-SEQUENCE *signal class*

Built from SEQUENCE, this signal class provides the functionality for numerically-valued sequences. This class provides default behavior for the atomic-type, range, imaginary range, and plotting. It also provides buffering of the real and imaginary parts and a default value of zero.

SEQUENCE-OF-SEQUENCES *signal class*

Built from SEQUENCE, this signal class provides the functionality for sequences whose values are themselves sequences. This class provides default behavior for range and plotting.

Shared functions of a signal class are specified using DEFINE-SYSTEM with the syntax

(function-name function-parameter-list
body)

A typical system definition is

```
(DEFINE-SYSTEM HAMMING (L)
               (NUMERIC-SEQUENCE)
  "Hamming window of size L, centered on the origin"
  (SUPPORT ()
    (INTERVAL (MINUS (FLOOR (// L 2)))
              (- L (FLOOR (// L 2)))))
  (SAMPLE-VALUE (N)
    (+ .54 (* .46 (COS (// (* 2 PI N) (- L 1)))))))
```

for the system HAMMING. This defines two shared functions, SUPPORT and SAMPLE-VALUE. The SUPPORT shared function of the signal class HAMMING interacts with the SUPPORT function described earlier. When the SUPPORT function is invoked on a HAMMING signal the SUPPORT shared function is executed.

The following clauses can be used to define shared functions for determining the non-zero extent of signals:

SUPPORT *clause*

Defines the non-zero support of the signals. Default is $-\infty < n < \infty$.

PERIOD *clause*

Defines the period of a signal. Default is ∞. Specifying a non-infinite period implies that the non-zero support of the signal is $-\infty < n < \infty$.

COMPUTE-DOMAIN *clause*

For a periodic signal, define the primary interval of the signal. Outside this interval the signal is computed by periodic replication. The default for this is the interval $0 \leqslant n < period$.

The following clauses can be used to specify information about signals:

ATOMIC-TYPE *clause*
Specifies the atomic-type of a signal. Default is :REAL for real-valued signals, :COMPLEX for complex-valued signals, and :UNKNOWN for others.

DEFAULT-H-SPAN *clause*
Specifies the interval over which to plot the signal. If not specified then the signal is plotted over its non-zero support.

DEFAULT-V-SPAN *clause*
Specifies the vertical interval to use when plotting the signal. If not specified then the range of the signal is used.

DEFAULT-IMAG-V-SPAN *clause*
Specifies the vertical interval to use when plotting the imaginary part of the signal. If not specified then the imaginary range of the signal is used.

The following clauses can be used to specify actions to be performed when creating a signal:

CHECK-ARGS *clause*
This clause will be run when the signal is created. It can be used to signal errors if the parameters of the signal are incorrectly specified.

INIT-FORMS *clause*
This clause will be run when the signal is created and after argument checking. It can be used to do any required initializations of the signal.

With DEFINE-SYSTEM, the computation of signal values may be specified using either point operations or array operations. We will describe how to compute signal values for numerically-valued signals. Symbolically-valued signals used the methods specified for computing the real part of numerically-valued signals. The following clauses are used to define the computation of signal values for point operations:

SAMPLE-VALUE *index* *clause*
Compute the real part of the signal at $n = index$.

SAMPLE-VALUE-IMAGINARY *index* *clause*
Compute the imaginary part of the signal at $n = index$. If this clause is not specified, and no SAMPLE-VALUE-COMPLEX clause is specified, the imaginary part is assumed to be zero.

SAMPLE-VALUE-COMPLEX *index* *clause*

Compute the real and imaginary parts of the signal at $n = index$. Cannot have this clause along with either a SAMPLE-VALUE or SAMPLE-VALUE-IMAGINARY clause.

Point operators are never invoked outside the non-zero extent of a signal, nor outside the primary interval for periodic signals.

The following clauses can be used to define the computation of signal values using array operations:

INTERVAL-VALUES *interval array* *clause*

Compute the values of the real part of the signal over *interval* and place the results in *array*. *Array* will be the same size as *interval*.

INTERVAL-VALUES-IMAGINARY *interval array* *clause*

Compute the values of the imaginary part of the signal over *interval* and place the results in *array*. *Array* will be the same size as *interval*. If this clause is not specified, and no INTERVAL-VALUES-COMPLEX clause is specified, the imaginary part is assumed to be zero.

INTERVAL-VALUES-COMPLEX *interval real-array imag-array* *clause*

Compute both the real and imaginary parts of the signal over *interval* and place the results in *real-array* and *imag-array*. Cannot have this clause along with either an INTERVAL-VALUES or INTERVAL-VALUES-IMAGINARY clause.

COMPUTE-INTERVAL *interval* *clause*

Before any invocation of an array operator this clause will be passed an interval that specifies the minimum interval over which the computation must be performed. This clause must return the same or larger interval to specify the interval over which it is legal to do the computation. The default, if this clause is not specified, is that any interval that is required is legal.

The special form DEFINE-SM-SYSTEM defines systems based on state machines. Its syntax is the same as DEFINE-SYSTEM and the clauses that can be used are the following:

SM-START *clause*

Specify the initial index for running the state machine. The non-zero support of the state machine is from this index to ∞. If this clause is not specified the default is to start the state machine at zero.

INITIAL-STATE *index* *clause*

Specify the real part of the initial state given that the state machine will start at *index*. If this clause is not specified the default starting state is the empty list.

INITIAL-STATE-IMAGINARY *index* *clause*

Specify the imaginary part of the initial state given that the state machine will start at *index*. If this clause is not specified the default starting state is the empty list.

INITIAL-STATE-COMPLEX *index* *clause*

Specify the real and imaginary parts of the initial state given that the state machine will start at *index*. Cannot have this clause and either an INITIAL-STATE or INITIAL-STATE-IMAGINARY clause.

CURRENT-VALUE *current-state index* *clause*

Compute the real part of the output value at *index* given *current-state*.

CURRENT-VALUE-IMAG *current-state-imaginary index* *clause*

Compute the imaginary part of the output value at *index* given *current-state-imaginary*.

CURRENT-VALUE-COMPLEX *current-state-real current-state-imaginary index* *clause*

Compute the real and imaginary parts of the output value at *index* given *current-state-real* and *current-state-imaginary*. Cannot have this clause and either a CURRENT-VALUE or CURRENT-VALUE-IMAG clause.

NEXT-STATE *current-state index* *clause*

Compute the real part of the next state given *current-state* at *index*.

NEXT-STATE-IMAG *current-state-imaginary index* *clause*

Compute the imaginary part of the next state given *current-state-imaginary* at *index*.

NEXT-STATE-COMPLEX *current-state-real current-state-imaginary index* *clause*

Compute the real and imaginary parts of the next state given *current-state-real* and *current-state-imaginary* at *index*. Cannot have this clause and either a NEXT-STATE or NEXT-STATE-IMAG clause.

SM-UPDATE *current-state index* *clause*

Compute both the real part of the output value and the next state given *current-state* at *index*. This clause is used in place of both the CURRENT-VALUE and NEXT-STATE clauses.

SM-UPDATE-IMAG *current-state-imaginary index* *clause*
Compute both the imaginary part of the output value and the next state given *current-state-imaginary* at *index*. This clause is used in place of both the CURRENT-VALUE-IMAG and NEXT-STATE-IMAG clauses.

SM-UPDATE-COMPLEX *current-state-real current-state-imaginary index* *clause*
Compute the real and imaginary parts of the output value and the next state given *current-state-real* and *current-state-imaginary* at *index*. This clause is used in place of both the CURRENT-VALUE-COMPLEX and NEXT-STATE-COMPLEX clauses. Cannot have this clause and either a SM-UPDATE or SM-UPDATE-IMAG clause.

The special form DEFINE-COMPOSITION defines new systems from old ones. The general syntax of DEFINE-COMPOSITION is

```
(DEFINE-COMPOSITION name  parameter-list
  [documentation ]
  equivalent-expression
  shared-functions)
```

The shared functions can be specified using the syntax of DEFINE-SYSTEM. The *equivalent-expression* may use the variables in the *parameter-list* but must evaluate to a signal. The result of evaluating the *equivalent-expression* is used as the implementation. For example

```
(DEFINE-COMPOSITION COS-SIGNAL (FREQ)
  "Cosine wave of specified frequency"
  (SEQ-SCALE .5 (SEQ-ADD (COMPLEX-EXPONENTIAL FREQ)
                         (COMPLEX-EXPONENTIAL (MINUS FREQ))))
  (ATOMIC-TYPE ()
    :REAL))
```

defines the system COS-SIGNAL for generating cosine waves of a specified frequency according to

$$\cos(\omega n) = \frac{e^{j\omega n} + e^{-j\omega n}}{2} \qquad (C.1)$$

and adds the additional information that the output is real, not complex.

C.4. Systems

SPLICE contains over 200 systems for the creation of signals. In this section we describe some of the available systems.

C.4.1. Systems for Generating Signals

The following are some of the systems for generating basic signals:

SEQ-CONSTANT *real-part [imag-part]* *system*
Constant valued sequence. Default for *imag-part* is zero.

SEQ-COS *period* *system*
Cosine signal with period *period* .

SEQ-SIN *period* *system*
Sine signal with period *period* .

SEQ-COMPLEX-EXP *period* *system*
Complex exponential with period *period* .

SEQ-EXPONENTIAL *alpha* *system*
Real exponential $alpha^n$.

IMPULSE *system*
An impulse.

UNIT-STEP *system*
Unit step.

SEQ-RAMP *[slope]* *system*
Ramp signal with *slope* defaulting to one.

RECTANGULAR *length [centered]* *system*
Length point rectangular window. If *centered* is specified as NIL then the window starts at zero. Otherwise the window is centered about zero.

BARTLETT-WINDOW *length [centered]* *system*
Length point triangular window.

HAMMING *length [centered]* *system*
Length point Hamming window.

HANNING *length [centered]* *system*
Length point Hanning window.

GAUSSIAN-WINDOW *sigma* *system*
Gaussian window with standard deviation *sigma.*

WHITE-NOISE *support [sigma] [mean] [seed]* *system*
Real-valued white Gaussian noise with non-zero support specified. Default *sigma* is one and default *mean* is zero.

COMPLEX-WHITE-NOISE *support [sigma] [real-mean] [imag-mean] [seed]* *system*
Complex-valued white Gaussian noise with non-zero support specified. Default variance is $sigma^2$ equals one and default means are zero.

SEQ-FUNCTION *function domain* $arg_1 \ldots arg_N$ *system*
Generate a sequence by applying *function* to the current index and $arg_1 \ldots arg_N$. Non-zero support is the *domain* of *function*.

C.4.2. Systems for Modifying Signals

The following systems take one sequence as input and produce a modified signal as output:

SEQ-SHIFT *sequence shift* *system*
Shift *sequence* to the left by *shift*.

SEQ-SCALE *sequence real-gain [imag-gain] [real-offset] [imag-offset]* *system*
Scale *sequence* by a scale factor. Subtract offset first. Default offsets are zero and default *imag-gain* is zero.

SEQ-RECIPROCAL *sequence* *system*
Point by point reciprocal of *sequence* over its non-zero support only.

SEQ-NEGATE *sequence* *system*
Point by point negation of *sequence*.

SEQ-REAL-PART *sequence* *system*
Real part of *sequence*.

SEQ-IMAG-PART *sequence* *system*
Imaginary part of *sequence*.

SEQ-CONJUGATE *sequence* *system*
Conjugate of *sequence* .

SEQ-POLAR *sequence* *system*
New sequence whose real part is the magnitude of *sequence* and whose imaginary part is the phase, modulo 2π, of *sequence* .

SEQ-RECTANGULAR *sequence* *system*
Inverse of the operation SEQ-POLAR.

SEQ-MAG *sequence* *system*
Point by point magnitude of *sequence* .

SEQ-MAG-SQUARE *sequence* *system*
Point by point magnitude squared of *sequence* .

SEQ-PHASE *sequence* *system*
Point by point phase, modulo 2π, of *sequence* .

SEQ-LOG-POLAR *sequence* *system*
New sequence whose real part is the log magnitude ($20\log_{10}\{\cdot\}$) of *sequence* and whose imaginary part is the phase, modulo 2π, of *sequence* .

SEQ-LOG-MAG *sequence* *system*
Point by point log magnitude of *sequence* .

LOG-MAG *sequence* *system*
Point by point log magnitude of *sequence* over the first half of the non-zero support of *sequence* . Used in plotting spectra.

SEQ-REVERSE *sequence* *system*
Time reverse *sequence* .

SEQ-ALIAS *sequence period* *system*
Time domain aliasing of *sequence* with specified *period* .

SEQ-AUTOCOR *sequence* *system*
Autocorrelation of *sequence* .

SEQ-ENERGY *sequence [window] [shift]* *system*
Short-time energy in *sequence* . *Window* defaults to a 255 point Hamming window and *shift* to 100 samples. Each output sample is the energy in a windowed section of *sequence* .

SEQ-CLIP *sequence min max* *system*
Clip values of *sequence* to between min and max.

SEQ-NORMALIZE *sequence* *system*
Normalize *sequence* to zero mean and unit variance over its non-zero extent.

SEQ-PREEMPHASIZE *sequence [filter]* *system*
Preemphasize *sequence*. Default *filter* is a first order difference.

SEQ-DEEMPHASIZE *sequence [filter]* *system*
Deemphasize *sequence*. Default *filter* is the inverse of a first order difference.

SEQ-UNIT-AREA *sequence* *system*
Normalize *sequence* to have unit area.

SEQ-UNIT-ENERGY *sequence* *system*
Normalize *sequence* to have unit energy.

ADD-NOISE *sequence snr-in-dbs* *system*
Add white noise to *sequence* so its *snr-in-dbs* over its non-zero extent is as specified.

SEQ-WINDOW *sequence window shift* *system*
Multiply *sequence*, shifted to the left by *shift*, by *window*.

SEQ-GATE *sequence interval* *system*
Gate out a section of *sequence* over *interval*.

SEQ-SECTION *sequence interval* *system*
Gate out a section of *sequence* over *interval* and shift to the origin.

SEQ-APPLY *function sequence* $arg_1 \ldots arg_N$ *system*
Apply *function* point by point to *sequence* with $arg_1 \ldots arg_N$.

C.4.3. Systems for Combining Signals

The following systems combine sequences to create new ones:

SEQ-ADD $seq_1 \ldots seq_N$ *system*
Point by point addition of $seq_1 \ldots seq_N$.

SEQ-SUBTRACT $seq_1\ seq_2 \ldots seq_N$ *system*
Point by point subtraction of $seq_2 \ldots seq_N$ from seq_1.

SEQ-MULTIPLY, SEQ-* $seq_1 \ldots seq_N$ *systems*
Point by point multiplication of $seq_1 \ldots seq_N$.

SEQ-DIVIDE seq_1 seq_2 ... seq_N *system*
Point by point division of seq_1 by seq_2 ... seq_N.

SEQ-MAP *function* seq_1 ... seq_N *system*
Point by point application of *function* to seq_1 ... seq_N.

C.4.4. Systems for Transforms

The following systems generate transforms:

FFT *sequence [size]* *system*
Take the *size* point FFT of *sequence*. If *size* is not specified take the next power of two greater than or equal to the size of the non-zero support of *sequence*. FFT generates samples of the Fourier transform of *sequence*, regardless of the location of the non-zero support of *sequence*, by doing time-domain aliasing to generate the *size* array to use.

IFFT *sequence [size]* *system*
Size point inverse FFT.

IFFT-REAL *sequence [size]* *system*
Size point inverse FFT when the output is known to be real-valued.

DFT *sequence [size]* *system*
Size point DFT of *sequence*. Default for *size* is the length of *sequence*.

IDFT *sequence [size]* *system*
Size point inverse DFT of *sequence*. Default for *size* is the size of *sequence*.

SEQ-CEPSTRUM *sequences [fft-size]* *system*
Real cepstrum (no phase unwrapping) of *sequence*.

C.4.5. Systems for Convolution

The following systems perform convolutions:

SEQ-CONVOLVE seq_1 ... seq_N *system*
Convolve the sequences seq_1 ... seq_N.

SEQ-CORRELATE x h *system*
Correlation of x and h.

SEQ-FFT-CONVOLVE *x h* *system*
Linear convolution of *x* and *h* using FFTs.

OVCONV *x h* *system*
Overlap-save convolution of *x* and *h* .

C.4.6. Systems for Reading and Writing Data

The following two operations deal with files of data:

FILE *pathname* *system*
Read data in from *pathname* and create a sequence. Format of the data is beyond the scope of this document.

SEQ-DUMP-TO-FILE *sequence pathname* *function*
Dump *sequence* to *pathname* . Can be read back with FILE.

C.4.7. Systems for Spectral Estimation

The following operators perform spectral estimation:

PERIODOGRAM *sequence [fft-size]* *system*
Periodogram of *sequence* . *fft-size* defaults to the next power of two larger than the length of *sequence* .

WINDOWED-PERIODOGRAM *sequence window [fft-size]* *system*
Windowed, in the correlation domain, periodogram of *sequence* .

WELCH-PERIODOGRAM *sequence window shift [fft-size]* *system*
Welch's modified periodogram by using windowed sections of data.

BARTLETT-PERIODOGRAM *sequence section-size [fft-size]* *system*
Bartlett's periodogram by using *section-size* pieces of data.

SEQ-MLM *sequence order [fft-size]* *system*
MLM spectral estimate of *sequence* using *order* analysis.

C.4.8. Systems for Filtering

The following operators deal with filtering sequences:

FIR $h_0 \ldots h_{N-1}$ *system*
Sequence that represents the impulse response of an FIR filter with coefficients $h_0 \ldots h_{N-1}$.

FIR-FILTER *sequence fir-filter-response* *system*
Filter *sequence* through a filter with impulse response *fir-filter-response*, which is specified using FIR.

IIR $a_1 \ldots a_N$ *system*
Sequence that represents the impulse response of an all-pole IIR filter with feedback coefficients $-a_1 \ldots -a_N$.

IIR-FILTER *sequence iir-filter-response* *system*
Filter *sequence* through a filter with impulse response *iir-filter-response*, which is specified using IIR.

C.4.9. Systems for LPC Analysis

The following operations deal with LPC analysis and synthesis:

LPC-SEQ *sequence order* *system*
Sequence that represents the LPC coefficients up to *order* in the analysis of *sequence*.

ERROR-SEQ *sequence order* *system*
Inverse filtering of *sequence* using *order* LPC analysis.

LPC-SPECTRUM *sequence order* *system*
Smoothed spectrum of *sequence* using *order* LPC analysis.

FILTER-FROM-LPC *excitation lpc-coefficients* *system*
Run filtering operation with input *excitation* and IIR filter from *lpc-coefficients*.

C.4.10. Systems that Manipulate Sequences of Sequences

The following systems manipulate sequences of sequences for various purposes:

SLICES *sequence window shift* *system*
Output is a sequence of sequences representing windowed versions of *sequence* at successive shifts of *window* by *shift*.

SPECTROGRAM *sequence [window] [shift] [fft-size]* *system*
Spectrogram of *sequence*. Default *window* is appropriate to wideband spectrogram.

LPC-SPECTROGRAM *sequence order [window] [shift] [fft-size]* *system*
Spectrogram of *sequence* using *order* LPC analysis.

LPC-SLICES *sequence order [window] [shift]* *system*
Sequence of sequences representing *order* LPC analysis of *sequence* at successive shifts of *window* by *shift* .

LPC-RECONSTRUCTION *lpc-slices lpc-shift pitch-estimates pitch-shift* *system*
LPC resynthesis using *lpc-slices* as the LPC analysis results, at successive shifts of *lpc-shift*, and pitch estimates from *pitch-estimates*, at successive shifts of *pitch-shift*.

LPC-RESYNTHESIS *lpc-slices lpc-shift excitation* *system*
LPC resynthesis using *lpc-slices* for filter responses and *excitation* as input.

STFT *sequence window shift [fft-size]* *system*
Short-time Fourier transform of *sequence* using *window* successively shifted by *shift* .

STFT-MAG *sequence window shift [fft-size]* *system*
Short-time Fourier transform magnitude of *sequence* using *window* successively shifted by *shift* .

ISTFT *stft window shift [fft-size]* *system*
Inverse short-time Fourier transform.

C.4.11. Other Systems

Other systems that are defined include the following:

SPECTRAL-SUBTRACTION *sequence alpha snr [fft-size]* *system*
Spectral subtraction of *alpha* times the noise power from *sequence* .

DESIGN-FILTER *size type frequency-samples frequency-domain band-spec*$_1$ ... *band-spec*$_N$ *system*
Design an equiripple FIR filter. Format of the arguments is beyond the scope of this document.

DESIGN-DIFFERENTIATOR *size frequency-samples max-frequency [frequency-domain]* *system*
Design an FIR differentiator.

REMEZ-FROM-SEQUENCE *size type values weighting* *function*
General Remez exchange for filter design.

Other systems have been built for pitch detection, time-scale modification, linear programmed spectral estimation, and formant tracking.

C.5. Miscellaneous

In this section we describe miscellaneous functions and special forms in SPLICE.

The following deal with the usage of arrays:

USING-ARRAY *(name size) body* — *special form*
Execute *body* with *name* bound to an array of size *size*.

USING-ARRAYS *((name$_1$ size$_1$) ... (name$_N$ size$_N$))* body — *special form*
Execute *body* with *name$_i$* bound to an array of size *size$_i$*.

WITH-SEQ-ARRAY *(array-name sequence interval) body* — *special form*
Execute *body* with *array-name* bound to the array of values of the real part of *sequence* over *interval*.

WITH-IMAGINARY-SEQ-ARRAY *(array-name sequence interval) body* — *special form*
Execute *body* with *array-name* bound to the array of values of the imaginary part of *sequence* over *interval*.

WITH-COMPLEX-SEQ-ARRAY *(real-array-name imag-array-name sequence interval) body* — *special form*
Execute *body* with *real-array-name* and *imag-array-name* bound to arrays of the real part and the imaginary part of *sequence* over *interval*.

ALLOC-ARRAY *size* — *function*
Allocate an array of size *size*.

DEALLOC-ARRAY *array* — *function*
Free *array*.

The following deal with the caching of signal values:

WITH-UNCACHING *body* — *special form*
Execute *body* and then remove cache buffers from any sequences that acquire cache buffers while *body* is executed.

UNCACHE-NEWLY-CACHED-SEQUENCES — *function*
Remove cache buffers from sequences that have been recently cached.

UNCACHE *thing* — *function*
If *thing* is a sequence remove cache buffers from it. If *thing* is a system name remove cache buffers from all the sequences created by it.

UNCACHE-ALL-SYSTEMS *function*

Remove cache buffers from all sequences created by systems.

CACHE *sequence* *function*

Remove *sequence* from the list of newly cached sequences so it will not get uncached by the next execution of UNCACHE-NEWLY-CACHED-SEQUENCES.

BFR *sequence* *function*

CACHE *sequence* and force *sequence* to have a cache buffer even if it would not normally have one.

C.6. What has not Been Discussed

SPLICE contains many other features that have not been discussed here. Included among these are a subsystem for complicated plots, mouse interaction, methods for specifying new plotting algorithms, methods for specifying options, methods for specifying sparsely sampled signals, methods for specifying the default caching behavior of sequences, array operations, debugging facilities, audio facilities, and hardcopy facilities.

APPENDIX D

Rules in the Extended Signal Processing Language and Interactive Computing Environment

This appendix describes many of the rules used in the Extended Signal Processing Language and Interactive Computing Environment (E-SPLICE). We give the rules for expression rearrangement, signal property manipulation, and cost analysis.

D.1. Notation and Definitions

In this section we describe the notation used in this appendix and define the terms used. In general, x and y are signals. We will not be specific about discrete-time or continuous-frequency unless the distinction is important to the rule. We will use c for a constant-valued signal and g for a constant.

The following terms are used to describe systems:

ASSOCIATIVE *system property*
Can change grouping of inputs.

COMMUTATIVE *system property*
Can change order of inputs.

ADDITIVE *system property*
Distributes over addition.

HOMOGENEOUS *system property*
Scaled input give scaled output.

LINEAR *system property*
Additive and homogeneous.

SHIFT-INVARIANT *system property*

Shifted input gives shifted output.

MEMORYLESS *system property*

Memoryless system. If single input then SHIFT-INVARIANT.

The following signals are used in addition to standard signals (impulses, constants, exponentials):

$FIR(h_0, \cdots, h_{N-1})$ - Impulse response of an FIR filter with coefficients h_0 ... h_{N-1}.

$IIR(a_0, \cdots, a_N)$ - Impulse response of an IIR filter with feedback coefficients $-a_1$ ... $-a_N$.

$Rect_L[n]$ - L point rectangular window.

$Sinc_L(\omega)$ - Transform of $Rect_L[n]$, $sin(\frac{\omega L}{2})/sin(\frac{\omega}{2})$.

The following systems are used in rules:

$x + y$ - Add x and y. MEMORYLESS system. Infix operators, such as +, may also be written in prefix notation, such as $+(x, y)$, so that rules of the form $T(x, y)$ apply also to infix operators.

$Alias_M(x)$ - Alias x, a continuous-frequency signal, with a period of $\frac{2\pi}{M}$ and divide by M. LINEAR system.

$Conj(x)$ - Conjugate of x. ADDITIVE and MEMORYLESS system.

$x * y$ - Convolve x and y. Normal definition for discrete-time signals and "circular convolution" for continuous-frequency signals. LINEAR and SHIFT-INVARIANT system in either x or y.

x / y - Divide x by y. MEMORYLESS system.

$\downarrow_M(x)$ - Downsample x, a discrete-time signal, by a factor of M. LINEAR system.

$FT(x)$ - Fourier transform of x, a discrete time signal. LINEAR system.

$IFT(x)$ - Inverse Fourier transform of x, a continuous-frequency signal. LINEAR system.

$Im(x)$ - Imaginary part of x. ADDITIVE and MEMORYLESS system.

$Interleave(x_0, \cdots, x_{L-1})$ - Interleave samples of x_0 ... x_{L-1}. Only for discrete-time signals.

$x \cdot y$ - Multiply x and y. LINEAR, in either x or y, and MEMORYLESS system.

$Re(x)$ - Real part of x. ADDITIVE and MEMORYLESS system.

$1/(x)$ - Point by point reciprocal of x. MEMORYLESS system.

$Rev(x)$ - Reverse x. LINEAR system.

$Scale\text{-}axis_L(x)$ - Compress the frequency scale for x, a continuous-frequency signal, by a factor of L. LINEAR system.

$z^{-l}(x)$ - Shift x by l. LINEAR and SHIFT-INVARIANT system.

$x - y$ - Subtract y from x. MEMORYLESS system.

$\uparrow_L(x)$ - Upsample x, a discrete-time signal, by a factor of L by inserting $L-1$ zeros. LINEAR system.

D.2. Signal Expression Rearrangement

This section lists some of the rules for expression rearrangement used in E-SPLICE. Because of the limitations of the rule system many of the rules listed here appear in E-SPLICE in different forms, or in multiple ways. We list here only one form of the rule. The following rules apply to abstract descriptions of systems:

Shift invariant operator applied to shift
If T is a shift-invariant system then $T(z^{-l}(x)) = z^{-l}(T(x))$.

Memoryless operator applied to shifts
If T is a memoryless system then $T(z^{-l}(x_1), \cdots, z^{-l}(x_N)) = z^{-l}(T(x_1, \cdots, x_N))$.

Memoryless operator applied to downsample
If T is a memoryless system then $T(\downarrow_M(x_1), \cdots, \downarrow_M(x_N)) = \downarrow_M(T(x_1, \cdots, x_N))$.

Memoryless operator applied to upsample
If T is a memoryless system then $T(\uparrow_L(x_1), \cdots, \uparrow_L(x_N)) = \uparrow_L(T(x_1, \cdots, x_N))$.

Commutative system reorder
If T is a commutative system then $T(x, y) = T(y, x)$.

Associative system reorder
If T is an associative system then $T(x, T(y, w)) = T(T(x, y), w)$.

Additive system and zero input
If T is an additive system then $T(0) = 0$.

Additive system and sum input
If T is an additive system then $T(x + y) = T(x) + T(y)$.

Homogeneous system and scaled input
If T is a homogeneous system then $T(g \cdot x) = g \cdot T(x)$.

The following expression rearrangement rules deal with basic signals:

Constant as complex exponential
$c = c \cdot e^{j 0n}$.

Rectangular window of length one
$Rect_1[n] = \delta[n]$.

The following rules rearrange expressions involving shifts:

Shift by zero
$z^0(x) = x$.

Shift a constant
$z^{-l}(c) = c$.

Shift multiple
$z^{-l_1}(z^{-l_2}(x)) = z^{-l_1-l_2}(x)$.

Shift of downsample
$z^{-l}(\downarrow_M(x)) = \downarrow_M(z^{-l \cdot M}(x))$.

The following rules rearrange expressions involving scaled signals:

Scale by zero
$0 \cdot x = 0$.

Scale by one
$1 \cdot x = x$.

Scale a scaled signal
$g_1 \cdot (g_2 \cdot x) = (g_1 \cdot g_2) \cdot x$.

The following rules deal with real part, imaginary part, conjugation:

Conjugate of real signal
If x is real-valued then $Conj(x) = x$.

Conjugate of conjugate
$Conj(Conj(x)) = x$.

Conjugate of complex exponential
$Conj(e^{j\alpha n}) = e^{-j\alpha n}$.

Conjugate of a product
$Conj(x \cdot y) = Conj(x) \cdot Conj(y)$.

Real part of real signal
If x is real-valued then $Re(x) = x$.

Real part of a product
$Re(x \cdot y) = Re(x) \cdot Re(y) - Im(x) \cdot Im(y)$.

Imaginary part of real signal
If x is real-valued then $Im(x) = 0$.

Imaginary part of a product
$Im(x \cdot y) = Re(x) \cdot Im(y) + Im(x) \cdot Re(y)$.

The following rules deal with upsampled and downsampled signals:

Upsample by one
$\uparrow_1(x) = x$.

Upsample of upsample
$\uparrow_{L_1}(\uparrow_{L_2}(x)) = \uparrow_{L_1 \cdot L_2}(x)$.

Upsample impulse
$\uparrow_L(\delta[n]) = \delta[n]$.

Upsample constant
$$\uparrow_L(c) = \frac{c}{L} \cdot \sum_{i=0}^{L-1} e^{j\frac{2\pi i}{L}n}.$$

Upsample complex exponential
$$\uparrow_L(e^{j\alpha n}) = \frac{1}{L} \cdot \sum_{i=0}^{L-1} e^{j\frac{\alpha+2\pi i}{L}n}.$$

Upsample of shift

$$\uparrow_L (z^{-l}(x)) = z^{l \cdot L}(\uparrow_L (x)).$$

Signal as sum of upsampled pieces

$$x = \sum_{i=0}^{L-1} z^i (\uparrow_L (\downarrow_L (z^{-i}(x)))).$$

Downsample of downsample

$$\downarrow_{M_1}(\downarrow_{M_2}(x)) = \downarrow_{M_1 \cdot M_2}(x).$$

Downsample impulse

$$\downarrow_M (\delta[n]) = \delta[n].$$

Downsample constant

$$\downarrow_M (c) = c.$$

Downsample complex exponential

$$\downarrow_M (e^{j \alpha n}) = e^{j \alpha M n}.$$

Downsample of upsample (L multiple of M)

If $L \bmod M = 0$ then $\downarrow_M (\uparrow_L (x)) = \uparrow_{L/M} (x)$.

Downsample of upsample (M multiple of L)

If $M \bmod L = 0$ then $\downarrow_M (\uparrow_L (x)) = \downarrow_{M/L} (x)$.

Downsample of upsample (M and L relatively prime)

If $GCD(L, M) = 1$ then $\downarrow_M (\uparrow_L (x)) = \uparrow_L (\downarrow_M (x))$.

Downsample of shifted upsample

If $l \bmod L = 0$ then $\downarrow_L (z^{-l} (\uparrow_L (x))) = z^{l/L} (x)$.

If $l \bmod L \neq 0$ then $\downarrow_L (z^{-l} (\uparrow_L (x))) = 0$.

Downsample of shifted upsample (M and L relatively prime)

If $GCD(L, M) = 1$ then $\downarrow_M (z^{-l} (\uparrow_L (x))) = z^{-l_1} (\uparrow_L (\downarrow_M (z^{-l_2} (x))))$ for appropriate choices of l_1 and l_2.

Interleave one signal

$$Interleave(x) = x.$$

Interleave as sum of upsampled signals

$$\sum_{i=0}^{L-1} z^{-i} (\uparrow_L (x_i)) = Interleave(x_0, \cdots, x_{L-1}).$$

Downsample of interleave (M and L relatively prime)

If $GCD(L, M) = 1$ then $\downarrow_M (Interleave(x_0, \cdots, x_{L-1})) =$
$Interleave(\cdots, \downarrow_M (z^{-\lfloor (i \cdot M)/L \rfloor} (x_{(i \cdot M) \bmod L})), \cdots)$.

The following rules deal with aliased and scale-compressed continuous-frequency signals:

Alias by one

$Alias_1(x) = x$.

Multiple aliases

$Alias_{M_1}(Alias_{M_2}(x)) = Alias_{M_1 \cdot M_2}(x)$.

Alias constant

$Alias(c) = c$.

Multiple scale-axis

$Scale\text{-}axis_{L_1}(Scale\text{-}axis_{L_2}(x)) = Scale\text{-}axis_{L_1 \cdot L_2}(x)$.

Scale-axis constant

$Scale\text{-}axis(c) = c$.

Scale-axis complex exponential

$Scale\text{-}axis_L(e^{j\alpha\omega}) = e^{j\alpha L\omega}$.

The following rules deal with reversed signals:

Reverse of reverse

$Rev(Rev(x)) = x$.

Reverse of constant

$Rev(c) = c$.

Reverse of complex exponential

$Rev(e^{j\alpha n}) = e^{-j\alpha n}$.

The following rules deal with the sum of signals:

Add zero

$x + 0 = x$.

Add signal to itself

$x + x = 2 \cdot x$.

Add signal to scaled version of itself

$$x + g \cdot x = (g+1) \cdot x .$$

Add scaled versions of signal

$$g_1 \cdot x + g_2 \cdot x = (g_1 + g_2) \cdot x .$$

Common scale factor to add

$$g \cdot x + g \cdot y = g \cdot (x + y).$$

The following rules deal with the product of signals:

Multiply by impulse

$$x[n] \cdot \delta[n - n_0] = x[n_0] \cdot \delta[n].$$

Multiply complex exponentials

$$e^{j\alpha_1 n} \cdot e^{j\alpha_2 n} = e^{j(\alpha_1 + \alpha_2) n} .$$

The following rules deal with the convolution of signals:

Convolve with impulse

$$x * \delta[n] = x .$$

Convolve with zero

$$x * 0 = 0.$$

Convolve of upsampled signals

$$\uparrow_L (x) * \uparrow_L (y) = \uparrow_L (x * y).$$

The following rules deal with the reciprocal of signals:

Reciprocal of complex exponential

$$1/(e^{j\alpha n}) = e^{-j\alpha n} .$$

Reciprocal of reciprocal

$$1/(1/x) = x.$$

Reciprocal of product

$$1/(x \cdot y) = (1/x) \cdot (1/y).$$

The following rules deal with Fourier transforms:

Fourier transform of inverse Fourier transform

$$FT(IFT(x)) = x .$$

Fourier transform of impulse

$$FT(\delta[n]) = 1.$$

Fourier transform of unit step

$$FT(u_{-1}[n]) = \frac{1}{1 - e^{-j\omega}}.$$

Fourier transform of constant

$$FT(c) = 2\pi c \cdot \delta(\omega).$$

Fourier transform of complex exponential

$$FT(e^{j\alpha n}) = 2\pi \cdot \delta(\omega - \alpha).$$

Fourier transform of rectangular window

$$FT(Rect_L[n]) = Sinc_L(\omega).$$

Fourier transform of convolution

$$FT(x * y) = FT(x) \cdot FT(y).$$

Fourier transform of product

$$FT(x \cdot y) = FT(x) * FT(y).$$

Fourier transform of shifted signal

$$FT(z^{-i}(x)) = FT(x) \cdot e^{-j\omega i}.$$

Fourier transform of FIR

$$FT(FIR(h_0, \cdots, h_{N-1})) = \sum_{i=0}^{N-1} h_i \cdot e^{-j\omega i}.$$

Fourier transform of IIR

$$FT(IIR(a_1, \cdots, a_N)) = 1/(1 + \sum_{i=1}^{N} a_i \cdot e^{-j\omega i}).$$

Fourier transform of conjugate

$$FT(Conj(x)) = Conj(Rev(FT(x))).$$

Fourier transform of reverse

$$FT(Rev(x)) = Rev(FT(x)).$$

Fourier transform of upsample

$$FT(\uparrow_L(x)) = Scale\text{-}axis_L(FT(x)).$$

Fourier transform of downsample

$$FT(\downarrow_M(x)) = Alias_M(FT(x)).$$

D.3. Signal Property Manipulation

This section describes the rules for the manipulation of signal properties. We will not describe the properties of basic signals, such as impulses, constants, etc., but instead will concentrate on the properties of other signals generated by systems that take signals as input, such as UPSAMPLE, FT, etc. We will use the following notation for signal properties:

Bandwidth (x) - Non-zero extent of frequency domain representation of x.

Center-of-symmetry(x) - Center of symmetry of x.

End (x) - End of non-zero extent of x. For discrete-time signals this is the index one beyond the last non-zero sample.

End-BW(x) - End of bandwidth of x.

Period (x) - Period of x.

Real-or-complex(x) - Is x REAL or COMPLEX?

Start (x) - Start of non-zero extent of x.

Start-BW(x) - Start of bandwidth of x.

Support (x) - Non-zero extent of x.

Symmetry (x) - Type of symmetry of x. We use the terms CONJUGATE-SYMMETRIC, CONJUGATE-ANTISYMMETRIC, SYMMETRIC, and ANTISYMMETRIC. *Symmetry* is multi-valued, i.e. for real valued signals the value of *Symmetry* may be both CONJUGATE-SYMMETRIC and SYMMETRIC or both CONJUGATE-ANTISYMMETRIC and ANTISYMMETRIC.

The support of a signal is an interval of indices outside which the signal is guaranteed to be zero. The default support of signals is from $-\infty$ to ∞. The following rules determine the non-zero support of signals:

Support of memoryless, additive system

If T is a memoryless, additive system then *Support* $(T(x)) =$ *Support* (x).

Support of shifted signal

Start $(z^{-l}(x)) =$ *Start* $(x) + l$ and *End* $(z^{-l}(x)) =$ *End* $(x) + l$.

Support of upsampled signal

$Start(\uparrow_L(x)) = L \cdot Start(x)$ and $End(\uparrow_L(x)) = L \cdot (End(x) - 1) + 1$.

Support of downsampled signal

$$Start(\downarrow_M(x)) = \left\lfloor \frac{Start(x)}{M} \right\rfloor \text{ and } End(\downarrow_M(x)) = \left\lfloor \frac{End(x) - 1}{M} \right\rfloor + 1.$$

Support of interleaved signal

$Start(Interleave(x_0, \cdots, x_{L-1})) = min(\cdots, L \cdot Start(x_i) + i, \cdots)$ and
$End(Interleave(x_0, \cdots, x_{L-1})) = max(\cdots, L \cdot (End(x_i) - 1) + 1 + i, \cdots)$.

Support of alias

$Start(Alias_M(x)) = M \cdot Start(x)$ and $End(Alias_M(x)) = M \cdot End(x)$ with appropriate testing for aliasing.

Support of reverse

$Start(Rev(x)) = -End(x)$ and $End(Rev(x)) = -Start(x)$.

Support of sum

$Start(x + y) = min(Start(x), Start(y))$ and $End(x + y) = max(End(x), End(y))$.

Support of product

$Start(x \cdot y) = max(Start(x), Start(y))$ and $End(x \cdot y) = min(End(x), End(y))$.

Support of convolution

$Start(x * y) = Start(x) + Start(y)$ and $End(x * y) = End(x) + End(y) - 1$.

Support of Fourier transform

$Support(FT(x)) = Bandwidth(x)$.

The period of a signal refers to the standard signal processing definition. The default period of discrete-time signals is ∞ (non-periodic) and the default period of continuous-frequency signals is 2π. The following rules determine the period of signals:

Period of memoryless system

If T is a memoryless system then $Period(T(x_1, \cdots, x_{N-1})) = LCM(Period(x_0), \cdots, Period(x_{N-1}))$.

Period of shift-invariant system

If T is a shift-invariant system then $Period(T(x)) = Period(x)$.

Period of upsample

$$Period\,(\uparrow_L (x)) = L \cdot Period\,(x).$$

Period of interleave

$$Period\,(Interleave\,(x_0, \cdots, x_{L-1})) = LCM\,(Period\,(x_0), \cdots Period\,(x_{L-1})).$$

Period of downsample

$$Period\,(\downarrow_M (x)) = \frac{LCM\,(Period\,(x), M)}{M}.$$

Period of convolution

If $Period\,(x) = \infty$ then $Period\,(x * y) = Period\,(y)$.

Numerically-valued signals may either be real or complex-valued. The default real-or-complex of a signal is COMPLEX. The following rules determine if a signal is real or complex:

Real or complex of shift

$$Real\text{-}or\text{-}complex(z^{-l}(x)) = Real\text{-}or\text{-}complex(x).$$

Real or complex of conjugate

$$Real\text{-}or\text{-}complex(Conj\,(x)) = Real\text{-}or\text{-}complex(x).$$

Real or complex of real part

$$Real\text{-}or\text{-}complex(Re\,(x)) = \text{REAL}.$$

Real or complex of imaginary part

$$Real\text{-}or\text{-}complex(Im\,(x)) = \text{REAL}.$$

Real or complex of upsample

$$Real\text{-}or\text{-}complex(\uparrow_L (x)) = Real\text{-}or\text{-}complex(x).$$

Real or complex of interleave

If $Real\text{-}or\text{-}complex(x_i) = \text{REAL}$ for all $0 \leqslant i < L$ then $Real\text{-}or\text{-}complex(Interleave\,(x_0, \cdots x_{L-1})) = \text{REAL}$.

Real or complex of downsample

$$Real\text{-}or\text{-}complex(\downarrow_M (x)) = Real\text{-}or\text{-}complex(x).$$

Real or complex of alias

$$Real\text{-}or\text{-}complex(Alias_M (x)) = Real\text{-}or\text{-}complex(x).$$

Real or complex of scale-axis

$$Real\text{-}or\text{-}complex(Scale\text{-}axis_L (x)) = Real\text{-}or\text{-}complex(x).$$

Real or complex of reverse
Real-or-complex(*Rev* (x)) = *Real-or-complex*(x).

Real or complex of sum
If *Real-or-complex*(x) = REAL and *Real-or-complex*(y) = REAL then *Real-or-complex*($x + y$) = REAL.

Real or complex of product
If *Real-or-complex*(x) = REAL and *Real-or-complex*(y) = REAL then *Real-or-complex*($x \cdot y$) = REAL.

Real or complex of convolve
If *Real-or-complex*(x) = REAL and *Real-or-complex*(y) = REAL then *Real-or-complex*($x * y$) = REAL.

Real or complex of reciprocal
Real-or-complex($1/x$) = *Real-or-complex*(x).

Real or complex of Fourier transform
If *Symmetry* (x) = CONJUGATE-SYMMETRIC then *Real-or-complex*(*FT* (x)) = REAL.

Real or complex of inverse Fourier transform
If *Symmetry* (x) = CONJUGATE-SYMMETRIC then *Real-or-complex*(*IFT* (x)) = REAL.

The bandwidth of a signal is an interval of frequencies outside which the frequency domain representation of the signal is guaranteed to be zero.[1] The default bandwidth is the interval from $-\pi$ to π. The following rules deal with the bandwidth of signals:

Bandwidth of linear, shift-invariant system
If T is a linear, shift-invariant system then *Bandwidth* (T (x)) = *Bandwidth* (x).

Bandwidth of conjugate
Start-BW(*Conj* (x)) = $-$*End-BW*(x) and *End-BW*(*Conj* (x)) = $-$*Start-BW*(x).

Bandwidth of real part
Start-BW(*Re* (x)) = *min* (*Start-BW*(x), $-$*End-BW*(x)) and *End-BW*(*Re* (x)) = max(*End-BW*(x), $-$*Start-BW*(x)).

[1] For discrete-time signals we are only interested in the interval $-\pi$ to π because spectra of discrete-time signals are periodic with period 2π.

Bandwidth of imaginary part

Start-BW($Im\,(x)$) = *min* (*Start-BW*(x), $-$*End-BW*(x)) and *End-BW*($Im\,(x)$) = max(*End-BW*(x), $-$*Start-BW*(x)).

Bandwidth of downsample

Start-BW($\downarrow_M (x)$) = $M \cdot$*Start-BW*(x) and *End-BW*($\downarrow_M (x)$) = $M \cdot$*End-BW*(x) with appropriate testing for aliasing.

Bandwidth of reverse

Start-BW(*Rev* (x)) = $-$*End-BW*(x) and *End-BW*(*Rev* (x)) = $-$*Start-BW*(x).

Bandwidth of sum

Start-BW($x + y$) = *min* (*Start-BW*(x), *Start-BW*(y)) and *End-BW*($x + y$) = *max* (*End-BW*(x), *End-BW*(y)).

Bandwidth of product

Start-BW($x \cdot y$) = *Start-BW*(x) + *Start-BW*(y) and *End-BW*($x \cdot y$) = *End-BW*(x) + *End-BW*(y).

Bandwidth of convolution

Start-BW($x * y$) = *max* (*Start-BW*(x), *Start-BW*(y)) and *End-BW*($x * y$) = *min* (*End-BW*(x), *End-BW*(y)).

Bandwidth of inverse Fourier transform

Bandwidth (*IFT* (x)) = *Support* (x).

Symmetry in E-SPLICE is recorded using the descriptive terms CONJUGATE-SYMMETRIC, CONJUGATE-ANTISYMMETRIC, SYMMETRIC, and ANTISYMMETRIC and by recording the center of symmetry. E-SPLICE also internally records linear phase and constant phase terms. The following rules deal with symmetry type and center of symmetry:

Symmetry of memoryless system

If T is a memoryless system, *Symmetry* (x_i) = SYMMETRIC for $0 \leqslant i < N$, and if *Center-of-symmetry*(x_i) = *Center-of-symmetry*(x_j) for $0 \leqslant i, j < N$ then

Symmetry $(T(x_0, \cdots, x_{N-1}))$ = SYMMETRIC and

Center-of-symmetry($T(x_0, \cdots, x_{N-1})$) = *Center-of-symmetry*(x_0).

Symmetry of shift

Symmetry $(z^{-l}(x))$ = *Symmetry* (x) and *Center-of-symmetry*($z^{-l}(x)$) = *Center-of-symmetry*(x) + l.

Symmetry of sum

If *Symmetry* (x) = *Symmetry* (y) and *Center-of-symmetry*(x) = *Center-of-symmetry*(y) then *Symmetry* ($x + y$) = *Symmetry* (x) and *Center-of-symmetry*($x + y$) = *Center-of-symmetry*(x).

Symmetry of product

If *Center-of-symmetry*(x) = *Center-of-symmetry*(y) then

If *Symmetry* (x) = CONJUGATE-SYMMETRIC and *Symmetry* (y) = CONJUGATE-SYMMETRIC then *Symmetry* ($x \cdot y$) = CONJUGATE-SYMMETRIC.

If *Symmetry* (x) = CONJUGATE-SYMMETRIC and *Symmetry* (y) = CONJUGATE-ANTISYMMETRIC then *Symmetry* ($x \cdot y$) = CONJUGATE-ANTISYMMETRIC.

If *Symmetry* (x) = CONJUGATE-ANTISYMMETRIC and *Symmetry* (y) = CONJUGATE-SYMMETRIC then *Symmetry* ($x \cdot y$) = CONJUGATE-ANTISYMMETRIC.

If *Symmetry* (x) = CONJUGATE-ANTISYMMETRIC and *Symmetry* (y) = CONJUGATE-ANTISYMMETRIC then *Symmetry* ($x \cdot y$) = CONJUGATE-SYMMETRIC.

If *Symmetry* (x) = SYMMETRIC and *Symmetry* (y) = SYMMETRIC then *Symmetry* ($x \cdot y$) = SYMMETRIC.

If *Symmetry* (x) = SYMMETRIC and *Symmetry* (y) = ANTISYMMETRIC then *Symmetry* ($x \cdot y$) = ANTISYMMETRIC.

If *Symmetry* (x) = ANTISYMMETRIC and *Symmetry* (y) = SYMMETRIC then *Symmetry* ($x \cdot y$) = ANTISYMMETRIC.

If *Symmetry* (x) = ANTISYMMETRIC and *Symmetry* (y) = ANTISYMMETRIC then *Symmetry* ($x \cdot y$) = SYMMETRIC.

Symmetry of convolution

Use the rules for product to determine type of symmetry. Also *Center-of-symmetry*(x * y) = *Center-of-symmetry*(x) + *Center-of-symmetry*(y).

Symmetry of Fourier transform

If *Real-or-complex*(x) = REAL then *Symmetry* (*FT* (x)) = CONJUGATE-SYMMETRIC.

If *Real-or-complex*(x) = COMPLEX and *Symmetry* (x) = SYMMETRIC then *Symmetry* (*FT* (x)) = SYMMETRIC.

If *Real-or-complex*(x) = COMPLEX and *Symmetry* (x) = ANTISYMMETRIC then *Symmetry* (*FT* (x)) = ANTISYMMETRIC.

Symmetry of inverse Fourier transform

Same rules as for Fourier transform symmetries.

Symmetry of real part

If *Symmetry* (x) = CONJUGATE-SYMMETRIC or *Symmetry* (x) = SYMMETRIC then *Symmetry* (*Re* (x)) = SYMMETRIC and *Center-of-symmetry*(*Re* (x)) = *Center-of-symmetry*(x).

If *Symmetry* (x) = CONJUGATE-ANTISYMMETRIC or *Symmetry* (x) = ANTISYMMETRIC then *Symmetry* (*Re* (x)) = ANTISYMMETRIC and *Center-of-symmetry*(*Re* (x)) = *Center-of-symmetry*(x).

Symmetry of imaginary part

If *Symmetry* (x) = CONJUGATE-SYMMETRIC or *Symmetry* (x) = ANTISYMMETRIC then *Symmetry* (*Im* (x)) = ANTISYMMETRIC and *Center-of-symmetry*(*Im* (x)) = *Center-of-symmetry*(x).

If *Symmetry* (x) = CONJUGATE-ANTISYMMETRIC or *Symmetry* (x) = SYMMETRIC then *Symmetry* (*Im* (x)) = SYMMETRIC and *Center-of-symmetry*(*Im* (x)) = *Center-of-symmetry*(x).

Symmetry of conjugate

Symmetry (*Conj* (x)) = *Symmetry* (x) and *Center-of-symmetry*(*Conj* (x)) = *Center-of-symmetry*(x).

Symmetry of reverse

Symmetry (*Rev* (x)) = *Symmetry* (x) and *Center-of-symmetry*(*Rev* (x)) = −*Center-of-symmetry*(x).

Symmetry of alias

Symmetry ($Alias_M$ (x)) = *Symmetry* (x) and *Center-of-symmetry*($Alias_M$ (x)) = M ·*Center-of-symmetry*(x).

Symmetry of downsample

If $2 \cdot$*Center-of-symmetry*(x) mod $M = 0$ then *Symmetry* ($\downarrow_M$ (x)) = *Symmetry* (x) and *Center-of-symmetry*($\downarrow_M$ (x)) = $\dfrac{\textit{Center-of-symmetry}(x)}{M}$.

Symmetry of upsample

Symmetry ($\uparrow_L$ (x)) = *Symmetry* (x) and *Center-of-symmetry*($\uparrow_L$ (x)) = $L \cdot$*Center-of-symmetry*(x).

D.4. Cost Measurement and Implementation Analysis

In this section we describe how cost is measured in E-SPLICE and give implementation rules. In addition to the systems discussed earlier, the following special cases are handled:

Convolution - Both inputs real. Both inputs real, one symmetric. One input

real, the other complex.

Multiplication - Both inputs real. One input real, the other complex.

FFT - Input real. Input conjugate-symmetric.

IFFT - Input real. Input conjugate-symmetric.

Computational costs are measured in terms of multiplications and additions with separate entries kept for real and complex operations. For purposes of our discussion, however, we will not distinguish between real and complex operations. The following systems have non-trivial costs:

Scale - Scaling a signal costs one multiply per output sample.

Add - Adding two signals costs one add per output sample.

Multiply - Multiplying two signals costs one multiply per output sample.

Convolve (General) - Let h be a finite duration signal with length N. Then, convolving h with x costs N multiplies and $N-1$ adds per output sample.

Convolve (Symmetric Impulse Response) - Let h be a symmetric, finite duration signal with length N. Then, convolving h with x costs $N/2$ multiplies and $N-1$ adds per output sample. Cost assumes that output can be downsampled without computing unnecessary samples.

FFT, IFFT (General, Radix 2, Size N) - Costs $0.5\log_2(N/2)$ multiplies and $\log_2(N)$ adds per output sample. Cannot be downsampled without computing unnecessary samples.

FFT, IFFT (Real or Conjugate-symmetric, Radix 2, Size N) - Costs $0.5+0.25\log_2(N/4)$ multiplies and $1.5+0.5\log_2(N/2)$ adds per output sample. Cannot be downsampled without computing unnecessary samples.

Overlap-save-convolution - Scales costs of circular convolution by $N/(N-overlap)$. Cannot be downsampled without computing unnecessary samples.

Also important to the measurement of cost is how often samples are required. Costs in E-SPLICE are scaled by the sampling rate and the following systems have non-trivial influence on the sampling rate:

Upsample - An upsampled signal requires input samples at $1/L$ of the rate at which output samples are required.

Interleave - Interleaving L signals averages the cost among the L signals, each at a rate of $1/L$ of the output rate.

Downsample - A downsampled signal requires input samples spaced M apart.

Bibliography

[1] J. L. Kelly, C. L. Lochbaum, and V. A. Vyssotsky, "A Block Diagram Compiler," *Bell System Technical Journal*, vol. 40, pp. 669-676, May, 1961.

[2] B. Gold and C. Radar, *Digital Processing of Signals*. New York: McGraw-Hill Book Company, 1969.

[3] W. Henke, "MITSYN - An Interactive Dialogue Language for Time Signal Processing," MIT RLE Technical Report 1, February, 1975.

[4] D. Johnson, "Signal Processing Software Tools," in *Proceedings International Conference on Acoustics, Speech, and Signal Processing*, San Diego, CA, April, 1984, pp. 8.6.1-8.6.3.

[5] H. Gethöffer, A. Lacroix, and R. Reiss, "A Unique Hardware and Software Approach for Digital Signal Processing," in *Proceedings International Conference on Acoustics, Speech, and Signal Processing*, Hartford, CT, May, 1977, pp. 151-154.

[6] H. Gethöffer, K. Hoffmann, A. Lenzer, N. Roeth, and H. Waldschmidt, "A Design And Computing System for Signal Processing," in *Proceedings International Conference on Acoustics, Speech, and Signal Processing*, Washington, D.C., April, 1979, pp. 688-691.

[7] H. Gethöffer, "SIPROL: A High Level Language for Digital Signal Processing," in *Proceedings International Conference on Acoustics, Speech, and Signal Processing*, Denver, CO, April, 1980, pp. 1056-1059.

[8] G. Kopec, "The Representation of Discrete-Time Signals and Systems in Programs," MIT PhD Thesis, Cambridge, MA, May, 1980.

[9] G. Kopec, "The Signal Representation Language SRL," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-33, no. 4, pp. 921-932, August, 1985.

[10] R. D. Greenblatt, T. F. Knight, Jr., J. Holloway, D. A. Moon, and D. L. Weinreb, "The LISP Machine," in *Interactive Programming Environments*, ed. D. R. Barstow, H. E. Shrobe, and E. Sandewall, New York: McGraw-Hill Book Company, 1984, pp. 326-352.

[11] The Mathlab Group, "MACSYMA Reference Manual," Laboratory for Computer Science, MIT, 1983.

[12] Digital Signal Processing Committee, IEEE Acoustics, Speech, and Signal Processing Society, *Programs for Digital Signal Processing*. New York: IEEE Press, 1979.

[13] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM*, vol. 20, no. 8, pp. 564-576, August, 1977.

[14] P. Henderson, *Functional Programming: Application and Implementation*. Englewood Cliffs, NJ: Prentice Hall Internation, 1980.

[15] B. Liskov and V. Berzins, "An Appraisal of Program Specifications," in *Research Directions in Software Technology*, Cambridge, MA: MIT Press, 1979.

[16] M. Stefik and D. Bobrow, "Object-Oriented Programming: Themes and Variations," *AI Magazine*, vol. 6, no. 4, pp. 40-62, Winter, 1986.

[17] D. A. Waterman and F. Hayes-Roth, *Pattern-Directed Inference Systems*. Orlando, FL: Academic Press, Inc., 1978.

[18] Signal Technology Incorporated, *Interactive Laboratory System (ILS)*.

[19] Bedford Research, *Interactive Signal Processor (ISP)*.

[20] D. S. Cyphers, "Spire: A Speech Research Tool," MIT SM Thesis, Cambridge, MA, May, 1985.

[21] D. W. Shipman, "Development of Speech Research Software on the MIT Lisp Machine," in *The Journal of the Acoustical Society of America*, Chicago, IL, April, 1982, pp. 26-30.

[22] V. W. Zue, D. S. Cyphers, R. H. Kassel, D. H. Kaufman, H. C. Leung, M. Randolph,. S. Seneff, J. E. Unverferth, III, and T. Wilson, "The Development of the MIT Lisp-Machine Based Speech Research Workstation," in *Proceedings International Conference on Acoustic, Speech, and Signal*.

[23] B. Bentz, "An Automatic Programming System for Signal Processing Applications," *Pattern Recognition*, vol. 18, no. 6, pp. 491-495, 1985.

[24] G. Kopec, "An Overview of Signal Representation in Programs," in *VLSI and Modern Signal Processing*, ed. S. Y. Kung, H. J. Whitehouse, and T. Kailath, Englewoods Cliffs, NJ: Prentice-Hall, Inc., 1985, pp. 241-256.

[25] H. Abelson and G. Sussman, *Structure and Interpretation of Computer Programs*. Cambridge, MA: MIT Press, 1985.

[26] D. Moon, R. Stallman, and D. Weinreb, "LISP Machine Manual, Fifth Edition," MIT Artificial Intelligence Laboratory, January, 1983.

[27] W. Dove, C. Myers, and E. E. Milios, "An Object-Oriented Signal Processing Environment: The Knowledge-Based Signal Processing Package," MIT RLE Technical Report 502, October, 1984.

[28] W. Dove, "Knowledge-Based Pitch Detection," MIT PhD Thesis, Cambridge, MA, May, 1986.

[29] W. Dove, C. Myers, A. Oppenheim, R. Davis, and G. Kopec, "Knowledge Based Pitch Detection," in *Proceedings International Conference on Acoustics, Speech, and Signal Processing*, Boston, MA, April, 1983, pp. 1348-1351.

[30] C. Myers, A. Oppenheim, R. Davis, and W. Dove, "Knowledge Based Speech Analysis and Enhancement," in *Proceedings International Conference on Acoustics, Speech, and Signal Processing*, San Diego, CA, April, 1984, pp. 39A.4.1-39A.4.4.

[31] E. E. Milios and S. H. Nawab, "Interpretation-Guided Signal Processing via Protocol Analysis," in *Proceedings International Conference on Acoustics, Speech, and Signal Processing*, Tampa, FL, March, 1985, pp. 42.16.1-42.16.4.

[32] E. E. Milios, "Signal Processing and Interpretation using Multilevel Signal Abstractions," MIT PhD Thesis, Cambridge, MA, April, 1986.

[33] C. Myers, "On the Use of Linear Programming for Spectral Estimation," in *Proceedings International Conference on Acoustics, Speech, and Signal Processing*, Tokyo, Japan, April, 1986, pp. 189-192.

[34] H. P. Nii, E. A. Feigenbaum, J. J. Anton, and A. J. Rockmore, "Signal-to-Symbol Transformation: HASP/SIAP Case Study," *AI Magazine*, vol. 3, no. 1, pp. 23-35, Spring, 1982.

[35] L. Erman, R. Hayes-Roth, V. R. Lesser, and D. R. Reddy, "The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty," *Computing Surveys*, vol. 12, no. 2, pp. 213-254, June, 1980.

[36] G. L. Steele Jr., *Common Lisp: The Language*. Billerica, MA: Digital Press, 1984.

[37] R. E. Crochiere and L. R. Rabiner, *Multirate Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

[38] J. Moses, "Algebraic Simplification: A Guide for the Perplexed," *Communications of the ACM*, vol. 14, no. 8, pp. 527-537, August, 1974.

[39] G. J. Sussman and G. L. Steele, Jr., "Constraints - A Language for Expressing Almost-Hierarchical Descriptions," *AI Journal*, vol. 14, pp. 1-39, 1980.

DISTRIBUTION LIST

	DODAAD Code	
Director Defense Advanced Research Project Agency 1400 Wilson Boulevard Arlington, Virginia 22209 Attn: Program Management	HX1241	(1)
Head Mathematical Sciences Division Office of Naval Research 800 North Quincy Street Arlington, Virginia 22217	N00014	(1)
Administrative Contracting Officer E19-628 Massachusetts Institute of Technology Cambridge, Massachusett 02139	N66017	(1)
Director Naval Research Laboratory Attn: Code 2627 Washington, D.C. 20375	N00173	(6)
Defense Technical Information Center Bldg 5, Cameron Station Alexandria, Virginia 22314	S47031	(12)
Dr. Judith Daly DARPA / TTO 1400 Wilson Boulevard Arlington, Virginia 22209		(1)