FORMALLY CORRECT TRANSLATION OF DSP ALGORITHMS SPECIFIED IN AN ASYNCHRONOUS APPLICATIVE LANGUAGE

Markus Freericks *Alois Knoll*

Technische Universität Berlin Sekr. FR 2-2
Franklinstr. 28-29 1000 Berlin 10
mfx@cs.tu-berlin.de

ABSTRACT

The functional programming language **ALDiSP**, which is specially tailored to the needs of DSP programming, is presented. **ALDiSP** incorporates data streams and an asynchronous control concept based on only one construct, the *suspension*. A comparison with traditional DSP languages like SILAGE is made. It is shown how **ALDiSP** programs can be translated into efficient code using the techniques of *abstract interpretation* and *partial evaluation*, in which a program is applied to symbolic input, resulting in usage information for all possible runs. This information is used to optimize the program by reconstructing it. Both the compilation of functions and the construction of a compile-time schedule make use of this approach.

1. INTRODUCTION

For a variety of reasons the development of a language for the specification and implementation of DSP algorithms is a demanding challenge. The requirements are even higher when real-time systems are to be specified.

The traditional doctrine suggests that execution speed can only be achieved in "low-level" languages, i.e., ultimately, by using the notable "C", but these have met with limited success. Special-purpose languages have also been developed. One of the more prominent ones, which will serve as our comparison, is SILAGE[1].
In 1989-90, two languages, **ImDiSP**[2], an imperative one, and **ALDiSP**[3, 5], which is based on functional concepts, were developed at TU Berlin in the context of the **CADiSP** project[4]. An **ImDiSP**-compiler for the Motorola 960000 has been written.
The **ALDiSP** compiler **ac** is under development, after a first interpretive implementation of the language made it possible to evaluate the – rather novel – concepts.
This paper describes the language and the techniques on which that compiler is based.

2. COMPARISON WITH OTHER LANGUAGES

We start with a discussion of different kinds of programming languages, organized by the list of features that a language should provide to support DSP programming.

2.1. DSP PROGRAMMING PARADIGMS

DSP programming languages can be roughly categorized by the paradigms they are based on. One major point is the distinction between synchronous and asynchronous languages.

Asynchronous languages have an event-oriented view of the world. Asynchronous programs can be understood as automata that react to input as it occurs by computing output and changing to a new state. A synchronous signal is viewed as a sequence of asynchronous events that happen to occur at equidistant points in time.

This work was in part supported by an Ernst-von-Siemens grant.

Synchronous languages are based on the premise that all input is processed in a fixed time frame. The DSP algorithm is then a loop that is executed once per sample (or per k samples, where k is a fixed constant). Some synchronous languages support multi-rate signals by providing interpolation and decimation operations to convert signals of different sampling rates. If a synchronous language supports event processing, it does so by representing it as a "busy waiting" signal: once per sample frame the signal representing the event has to be tested, an extremely inefficient way to handle i/o. Since most DSP algorithms are purely synchronous and there is an easy and obvious way to implement synchronous languages efficiently, nearly all DSP languages are based on the synchronous paradigm.

Asynchronicity typically occurs in control applications, where physical systems are coupled to sensors that initiate interrupts on system state change. In practice, DSP processors are often used for both the realization of synchronous algorithms and the processing of control events. For example, in an ISDN application, the DSP algorithm could be the compression, decompression and error checking of digitized sound or images, while the control part would be the management of user interaction and the handling of ISDN protocol.

2.2. DSP PROGRAMMING REQUIREMENTS

The following requirements should be met by a DSP language:

- Signal datatypes: DSP applications are concerned with the creation, transformation and analysis of signals. Usually, the internal representation of a signal is that of a stream of discrete equitemporally taken samples. Most algorithms are only concerned with the "current sample" and the "1... k-th previous" samples, where "k" is a small constant.

  DSP hardware often facilitates signal handling by providing circular buffers. A language should support a predefined "signal" data type that can be translated into efficient code accessing such hardware features. General-purpose languages like "C" do not offer this feature; often this is made for up by specially introduced pragmas and new data types.

  DSP specific languages are often designed around the signal datatype; in SILAGE, for example, "everything is a signal".

  In **ALDiSP**, signals are represented by *streams* and *pipes* (output and input driven chains of values) that are not restricted to scalar elements; it is possible to define streams of structured types, too.
- Timing specifications: in a synchronous language, timing is implicit. If multi-rate i/o is needed, one sampling frequency can be taken as a reference in relation to which all other rates are defined. Interpolation and decimation operations can then be provided to access signals not sampled with the reference frequency. SILAGE depends on this method.

  An asynchronous language has to provide either a global clock to which timing references can be made, or rely totally upon data dependencies. **ALDiSP** provides both features in its *suspension* construct (see 4.2).

0-7803-0946-4/93 $3.00

- Support of diverse numeric types: to model the restrictions of the hardware base accurately, a host of numeric types (signed/unsigned integer, biased integer, fixpoint, different floating-point formats), each of which can occur in varying widths, are needed. When these are not present, the "C" solution is adopted: a few types (int, float, fix) denote those numeric representations that the machine provides.

  While the latter approach simplifies the language, it is not appropriate when the algorithm is to be specified in a machine-independent way, or when a bit-true simulation is needed.

- Mechanisms to describe rounding and overflow behaviour: DSP algorithms need and DSP hardware provides support for many different quantization and overflow modes, most importantly saturation and different kinds of wrapping. **ALDiSP** supports overflow and rounding specifications in a novel way: it uses the concept of *exceptions*. Whenever rounding occurs, the operator concerned calls a rounding exception that returns the rounded result. The user may re-define the "current rounding mode" by providing a new rounding exception. It is even possible to define completely new rounding or overflow modes.

- Modularization features: many DSP programs are most naturally described as data-flow diagrams consisting of predefined function nodes. The development of function libraries should be aided by language features that support the creation of highly re-usable functions.

  A problem often encountered in languages like "C" or Pascal is that all functions are restricted to one type of arguments; e.g., a function that takes a 16-bit fixed-point signal may not be applicable to a 24-bit fixed-point signal (let alone a floating-point input), even if its definition does not rely in any way on the specific types of the arguments.

  ALDiSP provides *function overloading* and *polymorphic function* definitions; e.g. a function can be defined for different explicit types or in a type-unspecific way.

3. PARALLELISM IN HARDWARE AND SOFTWARE

DSP hardware tends to incorporate parallel features; the basic MAC (multiply-accumulate) architecture depends on the ability to load two values from memory, multiply them, and add the result to a register, all in one cycle. It is not unusual to have parallel data-move, integer- and floating-point units in one processor.

To utilize hardware parallelism efficiently, a language should support "parallel" operations like "add two arrays element-wise". The language should not force the programmer to sequentialize a parallel algorithm.

Single-assignment languages like SILAGE are adequate in this respect, since each variable has exactly one definition, so no "reuse" takes place: the algorithm is stateless. Problems occur when iteration/recursion is needed; the single-assignment approach is to introduce pseudo-arrays in which the nth element corresponds to the nth iteration of the loop.

Still, if the hardware provides, e.g., vector processing facilities, possible usage of such features is only extracted from "scalar" algorithms by painstaking analysis. Even when parallel operators are provided as hard-wired mechanisms in the language, it is usually not possible to extend in a portable manner when new hardware has to be addressed.

Functional programming offers the possibility of *higher order functions* (or *functionals*), i.e. functions that take other functions as arguments or return them as results. Functionals make it possible to abstract from "control flow", which helps modularization. In addition, functionals make it easy to express parallelization concepts.

Among popular functionals are *map*, which maps a functions element-wise on an array or stream, and *reduce*, which applies a binary function to vector, thereby "collapsing" it. For example, "reduce +" in **ALDiSP** is the operation that computes the sum of all elements of a vector. The difference between, e.g., a functional "map" and a built-in "map" (as defined, e.g., in FORTRAN-90) is that the first one is an ordinary function that can be defined by the user, while the second one is a "magic" mechanism built into the compiler and the language definition.

The second important property of functionals is that they can be used to write very compact code. Functional program sources are often an order of magnitude shorter than their imperative counterparts.

Single-assignment languages can be seen as functional languages *without* higher-order functions.

4. ALDiSP - AN OVERVIEW

ALDiSP is a call-by-value functional language with the following features:

- a *delay* form provides lazy evaluation on demand
- a *module* facility
- overloaded function definitions
- user-definable types; arbitrary predicates can be used as types
- exception mechanism: both aborting and returning exceptions, dynamically bound
- rounding and overflow expressible via exceptions
- automatic mapping of all functions on arrays and signals
- higher-order functions
- no restrictions due to type-checking
- i/o functions
- time and i/o expressed via suspensions

The last item will now be explained in detail. Functional programming is centered around the idea of statelessness - the whole program can be seen as a mathematical function that maps input to output; this function is composed from other functions. Computation (or better: reduction) order is only determined by data dependencies. Because of this, function applications can be substituted freely by their definitions, and new functions can be introduced wherever shared behaviour is encountered. These structural properties make it much easier to design and implement all kinds of *provably correct* program transformations.

In contrast, program transformations in imperative language are often incorrect because properties such as non-aliasing of pointers cannot be determined without actually running the program on all possible inputs, so unprovable assumptions have to be made. These properties can be subsumed under the notion of *state* – context in which expressions are evaluated. Functional languages have (nearly) no state.

Because of this lack of a state, functional languages usually do not incorporate any concept of i/o – input is most often considered to be part of the program (as in graph reduction), output is seen as "printing the result value", which is outside the scope of the language proper.

4.1. STREAMS

Recent trends in incorporating i/o control use the concept of *streams*. A stream is a (usually infinite) sequence of values produced by a function. A trivial stream is that of the natural numbers, where the nth element has the value n. Only a finite prefix of a stream must be represented extensionally, i.e. as a data structure; the infinite remainder of the stream is can be represented as a function.

For example, the stream of natural numbers can be implemented in **ALDiSP** as the recursive function

```
func CountNats(n) = n :: CountNats(n+1)
NaturalNumbers = CountNats(0)
```

The "`::`" operator constructs a stream from a first element and the tail of a stream, i.e. it prepends a value to a stream. It also *delays* the computation of its second argument to the point where it is first accessed. This prevents the function from looping indefinitely.

Streams can be used to model an output-driven system: whenever an output value is required by the printing function, a new stream element is computed. Input can be provided as a stream, too; e.g., a file can be modelled as a (finite) stream of characters.

4.2. SUSPENSIONS

Standard streams do not lend themselves to model input-driven behaviour. **ALDiSP** introduces the concept of *suspensions* for this purpose. A suspension is an expressions that is "suspended" until some condition holds. This condition relates either to other suspensions (waiting for them to evaluate) or to i/o state. Accessing the value of an as yet unevaluated suspensions suspends the accessing function itself. A predicate *isAvailable* tests for the availability of a suspension's value.

A good example for the use of suspensions is the valve controller:

```
func guard_closed_valve()=
   suspend open_valve();
           guard_open_valve();
   until current_pressure()>100.0
   within 0ms,2ms

func guard_open_valve() =
   suspend close_valve();
           guard_closed_valve()
     until current_pressure()<95.0
    within 0ms,5ms
```

These functions implement a two-state controller in a direct way. A `suspend` expression consists of three parts: the expression that is to be suspended; the condition upon which the evaluability depends, and two "duration" values. The latter determines the time frame, relative to the point where the condition becomes true, in which the expression must be evaluated. Providing a condition that is constantly "true" allows the user to request fixed timing delays.

The execution of an **ALDiSP** program is a two-level process, managed by a scheduler and an evaluator. The scheduler controls the i/o and activates suspensions; each suspension's value is then computed by the evaluator. If, during the evaluation, a side-effect is attempted, the evaluation blocks and returns a suspension. The scheduler then effects the side-effect and, later, re-awakens the suspended evaluation. Thus, evaluation is purely functional.

4.3. A COMPLETE PROGRAM

A suspension-based analogon to streams, dubbed *pipes*, provides for input-driven i/o. The following is a complete **ALDiSP** program that reads an input at a fixed rate, applies a simple second-order FIR filter to it, and writes it out again:

```
func ReadFromRegister(Rate,Reg) =
  read(Reg) ::
     suspend readFromRegister(Rate,Reg)
       until true within Rate,Rate

func WriteToRegister(Reg,OutputPipe) =
  (write(Reg,head(OutputPipe));
   suspend WriteToRegister(Reg,tail(OutputPipe))
     until isAvailable(tail(Reg))
    within 0ms, 0.1ms)

func FIR(a0,a1,a2)(inp) =
  let s0 = 0::inp
      s1 = 0::s0
      s2 = 0::s1
  in
      inp + a0*s0 + a1*s1 + a2*s2
  end

filter1 = FIR(0.98132343,-0.3225834,0.12465574)

net
 SamplingRate = 1 sec / 44000
 Input = ReadFromRegister(SamplingRate,StdIn)
in
 WriteToRegister(StdOut,filter1(FilteredInput))
```

The `ReadFromRegister` function samples a given input register (e.g., an ADC) at a frequency of 44kHz. The resulting pipe is then filtered and written to an output register (e.g., a DAC).

5. COMPILATION

There are two traditional compilation techniques for functional programming languages, combinator-based compilation and stack machines. The first is based on the translation of the program and its input into a graph which is then reduced in a rather arbitrary order. Stack machines have a more conventional machine model in mind; they are mainly used in the implementation of strict and impure functional languages. The Lisp machines were descendants of this approach.

Both techniques are not well matched to **ALDiSP**, mostly because they rely on a tagged memory heap providing dynamic storage and garbage collection facilities.

With the advent of CPS-based code generation[6], a technique has emerged that lowers the level of lambda expressions (the intermediate form common to most functional language compilers and interpreters) to that of machine code. CPS is based on the idea that the program is rewritten to a form in which all control flow is explicit. In particular, a function never "returns" (to some call site which is statically unknown), but calls a "continuation" function that represents the "rest of the program". This property makes it possible to view a function call as a "jump+rename", i.e., to abolish the need for a return stack. In this view, function parameters correspond to machine registers.

After CPS conversion, a heap is still needed to store the activation records of non-linear function calls. A further optimization tries to prove that the program is restricted to a nested control flow pattern. If this is possible, the continuation frames can be implemented on a stack.

CPS code still needs a garbage-collected heap as long as non-scalar data objects are used and higher-order functions employed: array-like objects cannot in general be updated in-place, and higher-order functions – at least those that have function-valued results – must allocate environment closures on the stack.

Here, we apply two new methods: an *abstract interpretation* (AI) of a program is a "simulated execution" on *abstract input values*. For example, an operation that reads an input register will return a symbolic value that denotes "some 16-bit integer". The abstract interpretation is monitored; after it has been run, it is known which function has been called with what types of arguments. Using this knowledge, a process of *partial evaluation* (PE) re-writes the program by introducing *specialized* functions.

Abstract interpretation and partial evaluation are a generalization of traditional optimization techniques based on data-flow analysis, such as constant-propagation, inlining, and dead code removal.

If a language has a formal semantics, it is possible to prove that some abstract interpretation of this language is "correct". There are a variety of possible abstract interpretation schemes for a give language, depending on

- what abstract values are needed: a simple abstract values would be "some number", a highly sophisticated one would be "a vector of 32 floating-point values, none of which is negative or zero, which should be held in cache". During an abstract interpretation, all kind of extra information can be lugged around in the abstract values.
- how recursion is treated: the abstract interpretation of an `if` construct needs to follow both arms of the conditional. In the context of a recursive function, this leads to nontermination. By

caching the calls to functions, these can be avoided, but it is known that the cost of finding a fixed point for a recursive function can be exponential in the height of the abstract data domain – that is, quite expensive.

- How "context information" is used: when the arms of an `if` expression are evaluated, a context is assumed in which the condition of the `if` is either true or false. This information can be carried along and used to steer the progress and, later, the specialization process.

These, among others, are active areas of research[7, 8].

5.1. STATIC CONTROL FLOW

DSP algorithms, especially those of the real-time persuasion, tend to have a very regular (or static) control flow. This gives reason to expect that a program that specifies such an algorithm will behave very well under abstract interpretation, even if the language is a very "dynamic" one. Typical **ALDiSP** programs are reduced indeed to the expected core functionality by the partial evaluation process. In analogy to the well-known "syntactic sugar" that denotes all syntactic niceties of a language that can be stripped away in a parser/preprocessor without any deeper understanding of the program, we would like to introduce the term "semantic sugar" to denote all mechanisms – higher-order functions, overloading, type checking, automatic mapping, exceptions, etc. – that can be removed by a partial evaluator.

5.2. STATE ANALYSIS

When partial evaluation is completed, the functions that constitute the program are simplified as far as possible. Still, the other part of the run-time model has to be handled: the scheduler.

Here, too, the abstract interpretation approach is applied. When running the program, there are two main *states*: The system is *resting* when all pending suspensions are either waiting for input to occur (or time to elapse), or are dependent upon suspensions that are waiting for input. As time passes, input takes place and suspensions are activated. The evaluation of these input-dependent suspensions enables other suspensions to run, and so on. Eventually, the system comes to rest again.

The above outlined process can be "abstracted" by providing abstract input whenever the system is resting. If the control behaviour of the program is independent of the actual input values, the only thing to model is the absence or presence of input. Otherwise, all possible input values must be simulated. At the moment, our compiler does not do the latter; the overhead would be too high.

Since a program may be waiting for a number k of input sources simultaneously, all possible combination of input events must be simulated. For each such combination, a new abstract state is reached after the system comes to rest.

Two abstract states are said to be "similar" if all suspensions they contain are waiting for the same conditions, and only their particular variable bindings (modelling the state) differ.

If the abstract interpretation of the scheduler finds that the set of possible states at time k (denoted ${S_k}^2$) is similar $Sk + c^2$, it has found a static schedule. In a purely synchronous program, ${S_k}^2$ will be similar to S^2_{k+1}, i.e., the set of possible states is the same at all points in time.

Once a stable schedule is found, compilation can follow. The scheduler can be executed "in the compiler's mind", and no run-time scheduling is needed any more.

6. CONCLUSION AND OPEN PROBLEMS

We have shown that it is possible to compile a language equipped with a number of semantic frills targeted at a demanding realm such as DSP programming, in a correct way into efficient code. While the compilation process is a costly undertaking, we hope that it is more than made up by the gained ease of programming.

Our compiler is now able to run the first phases of abstract interpretation and partial evaluation; the state analysis technique outlined in the last section is being implemented.

As an added bonus, the abstract interpreter, being a superset of an "actual" interpreter, can be used to simulate programs (albeit slowly – the abstract interpreter is burdened by a lot of bookkeeping chores.)

For now, no specific target architecture is envisioned. The output of the compiler is a pseudo-code similar to "C" or FORTRAN.

The previous section shows one major practical problem: In a program with two sampling frequencies, the constant c will be the lowest common denominator of the frequencies – quite a large number, and a real problem for the compiler. In addition, even if the number of input sources is small (say, 4 or 5), there are 2^k possible combinations. Heuristics are needed to speed up the simulation process.

A second problem is that of *code explosion*: a naive partial evaluator will unroll all loops and over-specialize many functions. Here, too, heuristic guides are needed to control the expansion process.

7. REFERENCES

[1] *Silage User's and Reference Manual*, prepared by Mentor Graphics/EDC, June 1991 (describes version 2.0)

[2] Volker Kruckemeyer, Alois Knoll: *Eine imperative Sprache zur Programmierung digitaler Signalprozessoren*, Forschungsberichte des Fachbereichs Informatik Nr. 1990/10, TU Berlin

[3] Alois Knoll, Markus Freericks: *An applicative real-time language for DSP-programming supporting asynchronous data-flow concepts*, in: Microprocessing and Microprogramming, Vol. 32, No. 1-5 (August 1991 - Proceedings Euromicro '91), pp. 541-548

[4] Alois Knoll, Rupert Nieberle: *CADiSP - a Graphical Compiler for the Programming of DSP in a Completely Symbolic Way*, Proc. IEEE-ICASSP'90, pp. 1077-1080

[5] M. Freericks, A. Knoll, L. Dooley: *The Real-Time Programming Language ALDiSP-0: Informal Introduction and Formal Semantics*, Forschungsberichte des Fachbereichs Informatik Nr. 92-26

[6] Andrew W. Appel: *Compiling with Continuations*, Cambridge University Press, 1992

[7] D. Bjœrner, A. P. Ershov, N. D. Jones: *Partial Evaluation and Mixed Computation*, Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation, North-Holland, 1988

[8] *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Yale Univ., June 17-19, 1991, published as: SIGPLAN Notices, Vol.26, No.9, Sept. 1991