Specification of signal processing programs in a pure functional language and compilation to distributed architectures

---

Master's thesis in Computer Science

by:

Per Bjesse 730921-6252 CTH

Supervisor:

Mary Sheeran

Institutionen för Datavetenskap
Chalmers Tekniska Högskola och Göteborgs Universitet
1997

Abstract

The traditional development of signal-processing software for multi-processor architectures involves writing low-level code encapsulating the underlying hardware configuration explicitly. This results in non-reusable and error-prone software.

I propose the use of a pure functional language for high-level specification of the signal-processing software, thus separating software and hardware. Program transformations could then be employed to compile specifications towards individual hardware architectures. Advantages of this approach include simplified development and easy re-use of commonly needed routines.

The thesis contains an evaluation of present development tools. An existing benchmark simulation program is taken as basis for the development of a Haskell signal-processing library and an equivalent functional benchmark. The library and simulation are evaluated, revealing pure functional languages as very suitable for signal-processing development. Symbolic evaluation is added to illustrate additional advantages of specification in a high-level language with a powerful type-system. The well-known inefficiency of compiled functional code is addressed by the introduction of a new form of deforestation suitable for signal-processing software. This, together with a new notation for inter-processor protocols and multi-processor architectures, lays a foundation for future bridges between high-level abstract specification of signal-processing programs and efficient distributed multi-processor code.

Preface

This is a Master's thesis in Computer Science done at the department of Computer Science at Chalmers Tekniska Högskola, Göteborg in the spring of 1997 by Per Bjesse. I wish to thank my supervisor Mary Sheeran, Bo Lyckegård from Ericsson Microwave Systems, John Hughes and Thomas Johnsson who were sources of ideas for many of the proposals presented in this thesis.

Institutionen för Datavetenskap
Chalmers Tekniska Högskola och Göteborgs Universitet
1997

Institutionen för Datavetenskap
Chalmers Tekniska Högskola och Göteborgs Universitet
1997

1. Introduction

During the development of the signal-processing for the ERIEYE airborne radar, engineers at Ericsson Microwave Systems encountered difficulties when mapping DSP software unto parallel architectures. The decomposition of a large program to many processors proved to be highly labour-intensive, and the resulting software was so entangled and specialized that software revisions and architectural modifications was out of the question.

This thesis is a first investigation into a possible solution based on the use of a functional language as a specification tool. Main reasons for this approach are the expressiveness of functional languages, which would allow rapid and relatively error-free prototyping, and the transformational property of functional programs, which means that they can be transformed to match an arbitrary hardware architecture. Functional programs are almost always very inefficient compared to imperative low-level programs, but the use of a transformation called deforestation might provide a bridge to high-speed implementations for the signal-processing program sub-class. These programs are generally static feed-forward process networks, which means that they are composed of a number of interconnected data-processing blocks with no conditional datapaths or backward loops.

Important issues covered in the thesis are the use and extension of deforestation, notations for protocols and hardware architectures and the results of experiments with specifications of signal-processing algorithms.

If you are less familiar with either radar, Digital Signal-Processing, parallel computations or deforestation, you might want to set out by reading about a few important concepts in appendix A.

2. Problem statement and a proposed solution

Characteristics of signal processing for radar are the very high data-rate of the stream of values arriving for processing from the receiver electronics, and the large amount of noise and uninteresting reflections contained in the data itself. As a consequence, the hardware that performs the signal-processing must execute complex algorithms under hard realtime requirements. The only feasible solution when no single processor can match the claims for computational power is therefore to distribute the load over many processors that work in parallel.

Since no good languages exists for modelling of parallel real-time applications, C or assembler is often used to implement the algorithms. As a result, it is very hard to make modifications in the hardware architecture without having to rewrite large portions of the

original code since it explicitly encapsulates the underlying hardware. Yet modification can be necessary for a number of reasons: **maintenance and upgrading, discovered performance bottle-necks,** and **algorithms changes**. Furthermore, the **re-usability of code** is very low, forcing developers to needlessly write the same kind of software over and over. Every contribution that make the process of mapping DSP programs on a processor network easier is therefore extremely valuable from an industrial standpoint.

To address these problems, I propose abstraction of the underlying hardware architecture from the software, and high-level language specification of the algorithms and a model of the architecture. This should then be the input to some transformation that as automatically as possible generated the sought low-level code for each processor. An architecture modification should then only involve some simple changes of the architecture specification result and a recompilation.

Functional languages has traditionally been judged very fit for transformational work with the aim of producing a parallel implementation on the basis that pure functional code has no internal state (no side-effects). Existing transformational techniques could perhaps be extended to allow compilation of signal-processing software in the desired manor. Transformation of sequential algorithms to parallel derivations that is overall *more effective* than the original solution is unfortunatelly generally very hard. A naive decomposition often results in a program that is overall less effective than a one-processor solution. No-one has devised a good way of doing this automatically, which means that the system constructor must aid the compiler with hints of some sort. An example of this could be a notation for the desired architecture that included the transformational goals (“Take part A, put in place B”).

Conventional functional programs tend to be very ineffective in comparison to imperative programs and require a lot of dynamical memory handling, primary memory and garbage collection. This is exactly the sort of features unavailable on signal-processors. *A general functional program is therefore not suitable for direct execution on signal-processing hardware*. The transformations must thus not only be able to split the program, but must also automatically optimize the resulting parts into **effective imperative code**. Experience shows that this is a very hard problem for general functional programs, but perhaps solvable for the streambased feed-forward process networks subclass most signal-processing applications constitute.

3. Rationale for the chosen approach

The signal-processing systems we consider have the following properties:

- Almost unlimited demands on computational power (you always want to extract more information).
- High throughput of data.
- Very hard real-time demands.
- Homogenous data.
- “Feed forward process network”-style architecture.
- Small target system in the available on-chip memory sense.
- Regular communication of MIMD type.
- Very hard to successfully partition software on different processors.
- High cost of developing software, and small production runs.

- The same signal-processing operations are performed in many systems.
- Traditional code very low-level and hence unportable.
- Restrictions on the weight of the system.

What would this imply?

- The **architecture has to be distributed** in order to cope with the throughput and real-time requirements.
- The almost unlimited demands for computational power means that the architecture rapidly becomes very complicated.
- The homogenous data suggests a **static architecture**.
- The high cost of developing software, the low-level code and the fact that the same routines are used over and over suggests that a **reusability scheme together with a high-level language** would be appropriate.
- The high cost of software development and small production runs means that it could be **cost-effective to reduce the software-development time even if it meant more (and therefore more expensive) hardware**.
- If a **functional language** is chosen in contrast to a traditional language, this could **simplify the task of program partitioning**. Unfortunately functional languages have a problem with the limited memory of signal-processors. We would therefore have to **deforest** the functional programs so they use less memory for execution.
- A certain amount of **reduced efficiency could be accepted** since we could add more processors if the resulting total cost and weight were acceptable.
- The **inter-processor communication could then also modelled using the functional language**
- The fact that a functional language is used for specification implies that high-level concepts associated with functional languages (such as formal verification etc.) was available to us.

4. Interesting questions

Is it possible to formulate the signal-processing algorithms and the connecting channels between processors in a pure functional language? Can the architecture of the target system be specified in a similar fashion by allowing channels that connects to more than one processor in each end, and supports functionality like data-overlap? From this information, supplied by the developer, could a compiler generate efficient imperative code for each processor? What efficiency improving transformations should be employed?

The functional specifications could of course be test-run on a stand-alone basis, which means that all the advantages of functional programming is at our hands when formulating new algorithms. Recognized benefits of using functional languages for specification purposes are decreased development time, increased readability and powerful type-checking. How would this affect the development cost?

What if we put the functional language Haskell to the test as a specification language for this kind of signal-processing programs by developing a signal-processing library and a simulation benchmark, and evaluated this methodology? What would the resulting characteristics of these specification be? Which operations should be chosen for implementation,

what representation should datatypes have? In what way could this specification then be used? Is it at all possible to transform applications utilizing the library to effective imperative code? What efficiency improving transformations can be used? How could the target-architecture and inter-processor protocols be specified? Could distributed code be generated?

5. Purpose of the thesis

The main goals of this thesis are to:

- Make the reader familiar with some signal-processing and Computer Science concepts, and to lay the foundation for the proposal of an inter-disciplinary solution.
- Propose a new methodology for signal-processing development.
- Define a number of important characteristics of a signal-processing development tool.
- Present an overview of other projects and tools with a somewhat similar aim.
- Implement and evaluate a signal-processing library in the functional language Haskell containing realistic operations performed in airborne radars.
- Implement and evaluate a benchmark airborne radar simulation program in the functional language Haskell.
- Present a comparison between Matlab specifications and Haskell specifications.
- Investigate the use of deforestation for signal-processing program specifications.
- Propose a notation for inter-processor protocols.
- Propose a notation for multi-processor architectures.
- Propose for how architectures could be formulated.
- Present some thoughts about how specifications could be compiled to imperative code suitable for execution on signal-processors.

6. Requirements of a good development tool for signal-processing software

In order to be attractive from an industrial viewpoint, a good development tool needs to satisfy a number of basic requirements. If the tool fail to meet these criteria, no-one will use it even if it has some interesting features unavailable in other systems. The analysis of requirements is therefore of utmost importance.

In addition to these basic requirements, some profiling features must be implemented that separates the development tool from the competition. In this case, the possibility of an way of automatically dividing code by program transformations are extremely valuable. Perhaps the Computer Science field provides more concepts that can be utilized in this Electrical Engineering context?

Basic requirements:

- Large library of pre-developed general purpose algorithms
- Good support for high-level development of new algorithms
- Abstraction from the target hardware

- Simulation capabilities
- Verification support
- Code generating abilities for a number of important architectures
- High reusability of previously developed software
- Automatic distribution of software to different hardware processor networks
- Intuitive and simple user interface

Possible advanced features:

- Automated checking of protocols, architectures and validity of load balancing
- Automated specialization of general algorithms
- Interface to formal methods

7. Overview of related projects and tools

7.1. Academical projects

7.1.1 Ptolemy

Ptolemy is a a CAD-Tool for system specification, simulation and design. It is an object-oriented framework which diverse models of computation can coexist and interact within. This framework uses hierarchy to mix heterogeneous models of computation. The result is a unified software environment for rapid prototyping of complex systems.

One of the main uses for this tool has proven to be signal-processing software development. Many similar commercial products exists, but few with the vision of tying very dissimilar tools together under the roof of one meta-tool.

Ptolemy allows system specification at different levels of abstraction,
called domains, which can be mixed within an application under certain limitations. Examples of these domains are:

- SDF: synchronous dataflow domain used for development of signal processing algorithms.
- DDF: dynamic dataflow domain. Extension of SDF, allows data dependent control flow.
- Thor: Modelling of circuits at register transfer level
- Code generation domains: synthesis of assembler code for different DSPs

Support also exists for code generation in C and the functional language Silage in addition to direct code generation for distributed architectures built around a scheduling scheme. Ptolemy is described in [PHEJ95].

7.1.2 Silage

Silage is an applicative functional language that is used to describe implementations of DSP algorithms.

Silage possesses dataflow semantics that enable a compiler to identify and exploit parallelism. The descriptions can be used as the input specification for a number of high-level hard-

ware synthesis tools for DSP applications (e.g., Hyper from U. C. Berkeley, Cathedral from IMEC and the DSP Station from Mentor Graphics). Silage descriptions can also be converted into software with a Silage to C compiler. No support exists however for compilation to a distributed architecture. More information about Silage can be found in [GHR+90].

7.1.3 The Sisal language and Paradigm compiler

The use of functional languages as specification tools for distributed signal processing has been previously explored (at least for small examples), specifically in relation to the Sisal language and the Paradigm compiler which is aimed at producing implementations by transforming functional specifications to dataflow code. This dataflow code can then be compiled and distributed to yield efficient parallel implementations. For further information, see [CFB92].

7.1.4 Lustre, Esterel et. al.

LUSTRE and ESTEREL are synchronous declarative languages for programming reactive systems. They are declarative because a description is a set of equations that always must be satisfied by the program variables. This approach has been chosen as it is close to the one adopted by designers of real-time control systems using models like systems of differential equations or synchronous operator networks. A program variable is considered to be a function of multiform time: it has an associated clock which defines the sequence of instants where the variable takes its values.

Compilation and verification of synchronous languages are well described, using Esterel or Lustre for signal-processing is a possibility even though no work has been done in this area. Lustre is described in [HCR+91], and Esterel in [BG92].

7.2. Commercial products

7.2.1 PAS and the RACEways architecture from Mercury

The RACE architecture is a commercial solution for interconnection of processors with a number of building blocks, the central being a crossbar switch. Processors need an interface chip to connect to the crossbar switch, and crossbars can be connected in a tree to allow a large number of processors to communicate.

In order to reduce the time of developing software for this architecture, a tool called PAS is available consisting of a library of functions that are called from the user's C language programs. This library contains both low and high-level functions, where the low-level functions provide facilities for data movement, synchronization and task queuing. The higher level functions are scalable parallel algorithms, that are automatically distributed over a number of processors.

Unfortunately, there seems to be quite a gap between the low and high-level routines, and the programmer must be contented with the "pre-programmed" algorithms unless he is willing to spend a lot of time writing synchronization and data-movement code. More information about PAS and the RACEways architecture can be found in [IG95].

7.2.2 Rippen from Orincon

The Rippen acronym stands for Real-time Interactive, Programming and Processing ENvironment. According to ORINCON, a programmer would use the Rippen tool to “graphically design, rapidly prototype, and implement a multiprocessor real-time system for typical signal processing applications.”

A function is represented in Rippen as a tool. There is a library of existing tools that the system architect can use, and programmers can develop their own tools in addition to the standard library. All tools are written in C, but the output is binary and dependent on the Rippen library, and is thus not stand-alone functions.

If there is additional hardware hooked to the host, such as a Mercury Raceway System, the tools programmed can be mapped to a specific processor or to the host system. This mapping is done by hand, but Rippen makes it easy to change mappings and to keep track of what tools are mapped to what processor.

Rippen supports a limited number of architectures, and the generated code are target-architecture assembler containing references to a library that must be licensed if the software is sold.

7.2.3 Multiprox from Alta group

Multiprox is a multiprocessor code development option for the Alta groups Signal Processing Workstation. The aim of this system is to generate C from a block diagram of tool-interconnections, where the tools are provided by the system. In order to insert communication code into the code, the programmer draws boxes around the tools that should be located at a certain processors. Based on a table provided by the user containing the physical connections between the processors, the correct communication primitives are chosen.

7.2.4 Pegasus from Jovian

Pegasus Parallel Processing Design Environment is a tool-set for the development of multiprocessor DSP applications. A typical application is built as a block diagram, and compiled to run on the Parallel C software system for interconnected C40 processors.

Parallel C is a coherent software system built around a special operating system that supports networks of processors allowing users to build applications that run on clusters of one or more DSP processors. From a given DSP block diagram, individual modules are coded in C, assembler and DSP library calls. Any communication or process synchronisation required is performed by calling the appropriate special Parallel C functions. The source files are compiled, assembled and linked into individual object files or tasks.

A tool called the configurer binds these binary files together into a single executable application. The configurer is driven by a text file which describes the target network, the tasks and their interconnections. Since all synchronization and communication already is in the code by means of the operating system calls, no recompilation is necessary when a reconfiguration is deemed necessary.

7.3. Conclusions about related projects and tools

7.3.1 General conclusions

The tools aimed at development of software operate at a granularity level of single operations. No good way has been devised to partition a tool over many processors.

7.3.2 Academic projects

The Ptolemy project has very wide-ranging goals, including some of ours. The methodology is not very mature and fundamentally different from the one proposed here. It is furthermore low-level and does not support efficient development and verification of new tools. The Sisal language is aimed at developing code for a massively parallel architecture, which is very different from interconnected DSPs. Silage is primarily directed towards silicon compilers that produce hardware descriptions, and thus of little interest.

As for the relevance of synchronous languages like Esterel or Lustre to signal-processing, it is still regarded as an open question as these languages aren't flexible enough for practical development of signal-processing programs. A host language that supports many different datatypes and powerful constructs is needed to make development practical. The synchronous use of a functional language, implies that a rich set of datatypes and constructs are available from the start, effectively utilizing it both as a synchronous language and a host language.

7.3.3 Commercial products

All of these projects are block-oriented, and allows the user to specify new tools at a fairly low level only.

Drawbacks:

- **Tools can not be automatically distributed over more than one processor unless the partitioning has been done beforehand.**
- No support exists for high-level specification and verification of new tools, so the user is forced to rely on low-level coding anyhow.
- Many tools that support distributed implementation relies on special hardware and/or operating systems, making their solutions specialized.
- Limited set of supported architectures.

From an industrial point of view (Ericsson Microwave Systems), further arguments against these products include:

- The development tools are basically inflexible.
- All generated source-code is target system dependent.
- The tools produce to much overhead.
- Inadequate tool-supplier support.

8. Specification experiments

8.1. Origin of the simulation

To make sure that the specification experiments were realistic, a Matlab benchmark simulation developed by Ericsson Microwave Systems was taken as basis for the library and benchmark implementation. The Ericsson program is supplied to tool-developers to test the applicability of their products to a non-trivial real-life system.

8.2. Methodology

Suitable routines for implementation in a signal-processing library were identified, and a corresponding Haskell benchmark was built on top of the resulting set of operations.

The first versions of the library and simulation were very similar in style to the Matlab specifications in order to clearly show what every part in the implementation corresponded to in the original specifications. Later versions were then rewritten to utilize the full expressiveness of a functional language, substituting code generated by practices that were very natural in the original benchmark but resulted in wasteful and sub-optimal functional programs.

8.3. Requirements of the signal-processing library

During the design of the library, a number of key issues were kept in mind. Some important points were that:

- The library should be a **toolbox** for radar signal-processing.
- Tool granularity should be **suitable for general purpose application development**.
- The adapted coding style should allow the library to **be easily read and understood**.
- The **large number of advantages** of using a functional language for specification purposes should be demonstrated.

8.4. Requirements of the radar simulation

Prerequisites of the simulation were that:

- The simulation should be runnable from an interpreting version of Haskell called Hugs.
- The simulation should utilize the Haskell signal-processing library.
- The code should be written in a **functional style and illustrate the expressiveness of functional programs**.
- **Key parameters of the simulation should be easy to change**, to aid the user in understanding radar concepts and the consequence of these parameters.
- The code should be **modular and the program-flow easy to follow**.

8.5. Implementation decisions and motives

8.5.1 Why are the implementation decisions important?

Motives for the implementation decisions are important if one is to understand how the routines could be extended and modified without destroying the desired characteristics of the code.

8.5.2 General guidelines and justifications

In order to write such complicated programs as the radar simulation and the signal-processing library whilst keeping the whole program understandable a number of conventions were adopted:

- All routines were to be kept **small and modular** to increase readability.
- The operator **<$<** (indicating parenthesising with a program flow from right to left) was used to group sub-operations into a **explicit sequence of operations** where appropriate.
- The definitions of the datatypes **Matrix** and **Vector** and all the operations that used these definitions directly were separated from the other code in order to **easily be able to change the definitions of these datatypes**. Additions to the library must not directly manipulate the underlying datatypes, and should use the existing interface provided by the library.
- **Matrices and vectors represented by lists of lists should not be indexed more than absolutely necessary** since this uses up a lot of computational power. To avoid unnecessary waste, a matrix or vector should be scanned once in order to produce a new matrix or vector with elements containing the information about the neighbouring elements, as opposed to costly neighbour indexing for each and every element.

8.6. The symbolic evaluation

8.6.1 Purpose

The symbolic evaluation was added to illustrate the use of high-level concepts in the specifications. For an in-depth discussion, see section 13. at page 33.

8.6.2 How is it done?

The symbolic evaluation uses the Haskell class-system to overload all the arithmetic operators and a selected number of standard prelude functions. Here is part of the definition of the symbolic datatype and some selected instantiations:

```
data Expr = Var String | CVar String | OneArg Op1 Expr |
    TwoArg Op2 Expr Expr | FNum Float | INum Int |
    CNum (Float,Float) deriving (Eq,Ord)

data Op1 = Neg | Conj | Phase | Magn | Cos | Sin | Sqrt | Exp |

        Log | Round  deriving (Eq,Ord)

data Op2 =

     Add | Sub | Div | Mul | Rem | Mod | CForm | Raised
```

```
            deriving (Eq,Ord)


instance Num Expr where
    (+) (FNum f1) (FNum f2) = FNum (f1+f2)
     (+) (CNum (c1r,c1i)) (CNum (c2r,c2i)) = CNum (c1r+c2r,c1i+c2i)
     (+) s1 s2 = TwoArg Add s1 s2
     (-) (FNum f1) (FNum f2) = FNum (f1-f2)
     (-) (CNum (c1r,c1i)) (CNum (c2r,c2i)) = CNum (c1r-c2r,c1i-c2i)
     (-) s1 s2 = TwoArg Sub s1 s2
     (*) (FNum f1) (FNum f2) = FNum (f1*f2)
     (*) (CNum (a,b)) (CNum (c,d)) = CNum (a*c-b*d,c*b+d*a)
     (*) s1 s2 = TwoArg Mul s1 s2
     negate (FNum f1) = FNum (-f1)
     negate (CNum (cr,ci)) = CNum (-cr,-ci)
     negate s = OneArg Neg s
```

This means that (ideally) the results of routines that operates on the **Expr** datatype will be an symbolic expression (representing all the operations performed in the code) or numerical data depending only on the type of the input, without any special modifications to the code. Symbolical expressions could be utilized in a number of ways, ranging from simple debugging purposes to analysis tool input and formal verification.

8.6.3 The quirks

Little work has gone into making the symbolic evaluation seamless, and this (in addition to a weakness in the Haskell class-system) has resulted in some quirks, specifically:

- If the symbolic evaluation is to function without extra modifications in the code, no conditional operations can be performed on the indata. This could be avoided by overloading the **if..then..else** construction.
- Some routines must determine if indata is symbolic or numeric in order to know what format constants should have.
- The symbolic and numeric datatype had to be unified in order to avoid the type conflict induced by function like **magn** (that should return the magnitude of a complex number). The problematic functions have different signatures for different indata (**magn** for numerical values::Complex->Float, symbolic values::Symbolic->Symbolic). The Haskell type system does not allow multi-level type variables, so the only way to solve this was unfortunately to unify the types (yielding a type signature of Expr->Expr for **magn**). A certain amount of type-checking is incurred since it now is the programmers responsibility to ensure that incorrect type-mixes are banned.

8.6.4 The consequences for add-on routines

All signal-processing functions that are added to the library must use the **Expr** datatype, to fit into the library succinctly. If correct functioning of the symbolic features is to be guaranteed, the key points about constants and conditional expressions must be considered.

8.7. The signal-processing library

8.7.1 Library contents

Library routines that for signal-processing can be divided into 5 categories:

- Functions that generate the simulated data
- Filtering functions working in the time-domain and the frequency domain
- Pulse-compression functions working in the time-domain and the frequency domain
- CFAR detection functions
- Functions used for the resolving targets velocity and range

In addition to this, a number of general purpose routines for vector and matrix manipulation are included. As previously discussed, support for symbolic evaluation was embedded into all definitions.

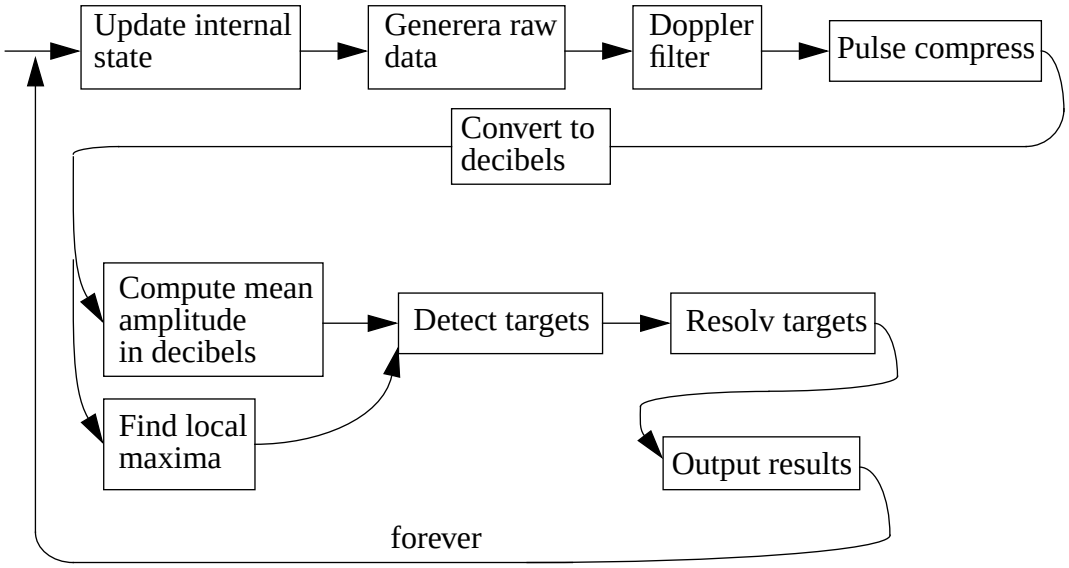
For a more in-depth treatment of the signal-processing library, refer to the program documentation section B.1. at page 60.

8.7.2 Conclusions

Since the library originated in the needs present in an existing simulation, the operations chosen have a real-life background. Some of them are specifically aimed at signal-processing for radar, others are more general in their applicability. If the implementation guidelines are followed, it is easy to add more routines that are needed for other types of signal-processing. This extends the usability of the library. The work done within the framework of this thesis has prepared the ground by implementing datatypes and fundamental operations. Future work should include more comprehensive measurements of development time for new algorithms.

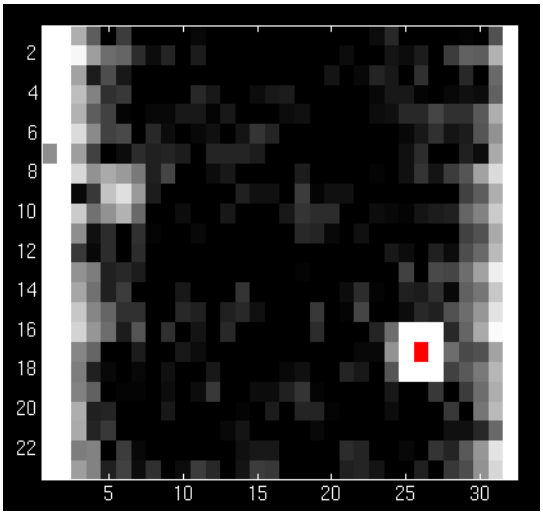
8.8. The airborne radar simulation

8.8.1 Functionality

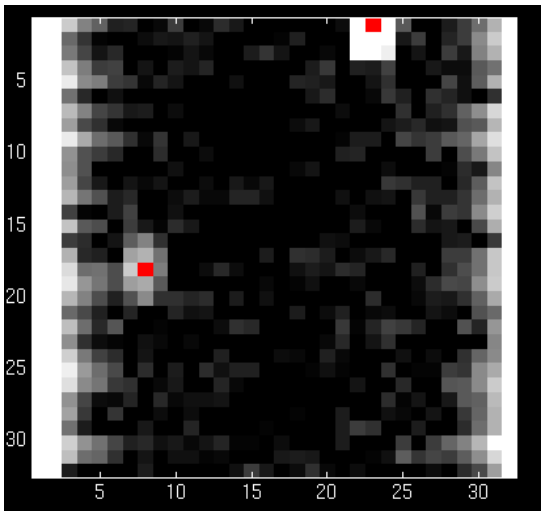


The airborne radar simulation utilizes the Haskell signal-processing library to do all the signal-processing operations needed in a standard application. To keep the operations chosen for implementation realistic and avoid cutting corners, an industrial benchmark Matlab program was used as an input.

The simulation generates simulated digital data internally, to keep the simulation self-contained. Critical parameters for each iteration include the dimensions of the data-matrix (the number of doppler-channels and range-bins used) and the noise-generation seeds. An example doppler filtered data-matrix looks like this:



Observe the white clutter present at the left and right edges of the matrix, the background noise and the two targets. A different number of doppler-channels and range-bins results in a completely different-looking data-matrix:



The fact that **folding** seems to move the targets like this is the basis of range and velocity resolving. If the dimensional parameters were the same for each iteration, no additional information could be gained about targets that was approximately stationary (targets that didn't have the speed to move a substantial distance between integration intervals). The resolving would be thwarted since no movement could be observed in the data-matrix. The simulation thus needs an internal state that allows it to calculate the new dimensions and critical parameters for each iteration.

Once the dimensions and other important parameters are fixed, doppler filtering is performed on the data matrices to separate the data according to their velocity. Clutter is thus offset in the highest and lowest channels, with a relatively clear area in-between. All data are then pulse-compressed by the application of an matched filter making the returned energy-pulses narrower and higher.

The next step is to logarithm all amplitudes to be able to substitute additions for multiplications. Strictly speaking, this may or may not be necessary depending on the hardware used. Some processors perform addition much more cheaply than multiplication, some don't.

Mean values and local maxima are sought in order to provide the detection algorithm with the information it needs to compute some threshold value for each element that the amplitudes can be compared to. An element amplitude larger than the local threshold indicates a present target.

The identified locations of the targets in the data-matrix are fed into the resolving algorithms, which tries to find the unambigous ranges and velocities. It is of great importance that the resolve functions are initialized with their state each iteration in order for them to "remember" old unresolved targets. As the algorithms exploit the so called "m out of n" criteria, a number of simulation iterations might be necessary before any targets are identified.

Targets that are resolved are written to the output, and internal states are collected for the next iteration, which promptly is started. The output to the screen looks like this:

```
Dimensions:(23,32)

Num detections:1

List:[(16,25)]
```

The *dimensions* tuple contains **(number of range-bins, number of doppler-channels)**. Every iteration has different dimensional parameters for the benefit of the resolving routines. The *list* tuples contains the unresolved data-matrix positions of the detected targets.

```
Dimensions:(29,32)
Num detections:2
List:[(12,6), (25,23)]
```

After three iterations, enough data has been provided to detect one of the two targets present unambiguously:

```
Dimensions:(32,32)
Num detections:2
List:[(0,22), (17,7)]
Range: 99.95 Vel: 388.748

Dimensions:(34,32)
Num detections:2
List:[(0,7), (0,21)]

Dimensions:(36,32)
Num detections:2
List:[(2,21), (20,8)]
Range: 50.0499 Vel: -352.819
```

Some iterations later, the number of doppler-channels are increased to 64.

```
Dimensions:(23,64)
Num detections:2
List:[(8,10), (16,50)]
```

And so on.....

For a more thorough explanation of the simulation functionality, refer to the program documentation appendix section B.2. at page 80.

8.8.2 Conclusions

The simulation is a very valuable input for further experiments with specification and transformation. “Real-life” problems are often both hard to find and time-consuming to convert to a suitable form for experimentation, so the Haskell implementations are important results of the thesis.

8.9. Haskell compared to Matlab as a specification tool

A relatively inexperienced Haskell programmer took 9 weeks to write:

- A complete radar benchmark simulation with the same functionality as the Matlab specification
- A comprehensive signal-processing library

- Matrix, vector primitives

This included time spent on developing an understanding of the algorithms and problem-domain. First versions of the simulation were deliberately implemented closely following the detailed processing in the Matlab original. Four additional weeks were then spent on cleaning up the program and rewriting it in a functional style. Considering that the whole code was written from scratch (all matrix operations, FFT, ..), this must be considered **rapid**.

Surprisingly, the length of code of the Haskell specification (including the definitions of all primitive datatypes and operations) was in all **shorter** than the Matlab specification (Haskell 1100 loc. , Matlab 1500 loc.). Granted, Haskell is a higher level language, but Matlab is specifically aimed at numeric calculations such as signal-processing, and thus has many primitive operations dealing with matrices and elementary filtering built-in. Consequently, this result was not clearly foreseen at the beginning of the investigation into the characteristics of a rewritten simulation.

The finished functional program was rated as **more readable** by a number of skilled imperative programmers as it was shorter and higher level than the original specification. A programmer with an basic understanding of functional programs could immediately grasp what each routine was supposed to do just by reading the code, which is something that one couldn't honestly say about the Matlab specification.

In addition to the same functionality as the original benchmark program, the Haskell program was implemented containing elementary support for symbolic evaluation. This illustrates the **added benefits** of using a functional language for specification of signal-processing software.

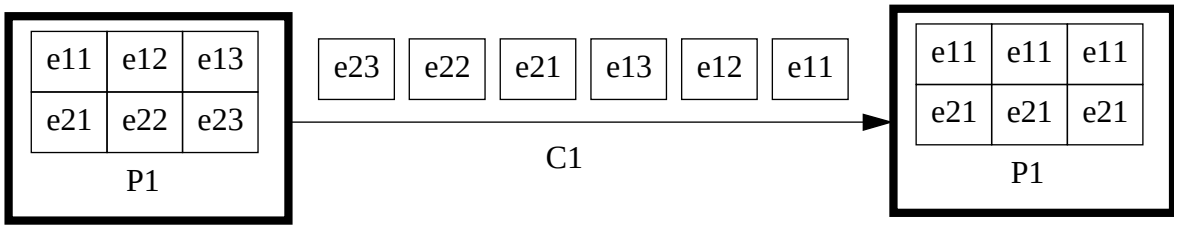
The big *drawback* of the rewritten simulation is the comparatively **low speed of execution**. Even small dimensions of data results in large amounts of time spent processing. This was of course to be expected since an interpreting version of Haskell was used and Matlab uses precompiled routines that are painstakingly optimized for numerical computations. The use of a Haskell compiler such as *hbc* might speed things up as much as ten times.

Speed of execution is not is an important feature, however, since Haskell is supposed to be used as an specification language only. The inefficiencies are accordingly of no great consequence since we expect to compile these specifications to a better form. Even if the overall efficiency are worse on the individual processors, we could gain enough on the increased ease of partitioning the program to many processors to be able to compensate the loss by adding more hardware. This methodology is of course only valid if production runs are suitably small.

9. Specification of protocols

Many of the signal-processing functions in the library operate on vectors or matrices. These datatypes must be communicated one element at a time through the one-dimensional streams that flow over the inter-processor connections. The elements in a complex datastructure, such as an two-dimensional matrix, can be sent in an arbitrary order. This order, or **protocol**, must be well-specified between any two connected processors, so as to avoid erroneous assumptions of dimensions and order of arrival.

Here is an simple example of a transmission protocol:



The processor P1 works internally on matrixes with dimensions 2*3, and is connected by means of C1 to processor P2. Since both processors assume that the protocol states that the matrix should be sent in the order e11..e13, e21..e23, the matrix arrives safely.

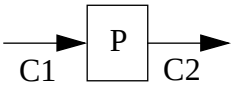
It is very important that protocol mismatches are automatically detected. How could this be done?

If we use **types** to specify the protocol for the connections between the processors, *we could just type-check the hardware specifications* to see that it doesn't contain protocol mismatches. One example of a protocol mismatch is the result of a number of interconnected processors erroneously assuming that the incoming one-dimensional data-stream represents a vector when the others assume it represents a matrix. By type-checking the architecture, this important error could be detected immediately.

In order to capture the protocols, **two functions that converts between the one-dimensional streams of elements that flows over the incoming and outgoing processor interconnection networks and the internal data that the functions operate on, can be included as the type of the processor blocks**.

An example of such a function is the standard-prelude function **concat** which collapses a matrix represented as a list of lists to a single list, which could be converted to a stream. The inverse function is of course interesting, since it would convert a list to a matrix. In order for **concat** to be invertible, the matrix dimensions must be known. A special variant of concat called **concat m n** needs therefore to be introduced. The inverse of this function is then notated **(concat m n)$^{-1}$**. Other invertible functions can be defined that collapse data in different ways, even splitting it over many connections (which will be utilised in the next section).

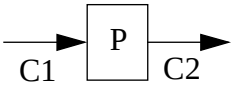
A simple processor block **P** that reads and writes matrices of dimensions m*n from the connections C1 and C2 by collapsing and rebuilding the list of lists is specified as:



P:p:concat m n->(concat m n)$^{-1}$

The symbol **:p:** should be read *has the protocol*. The type information provides as complete a specification of the protocols used as needed.

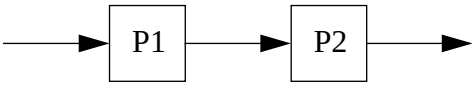
Note that different protocols can be used on C1 and C2, as in this example, where the function operating on **P** halve the number of matrix rows:



P:p:concat m n->(concat m/2 n)$^{-1}$

10. Specification of architectures

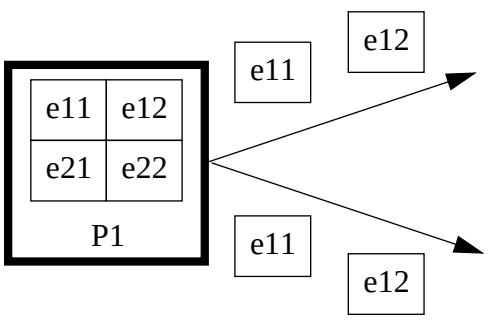
By using *combinators* to notate which processors should be interconnected, together with the specifications of the processor protocols, complete architectures could be defined. Combinators has been previously used to specify message-processing networks in [CH93], and to notate hardware composition [JS90]. A very simple two-processor network that relayed m*n matrices could be connected using the serial connection combinator >->:



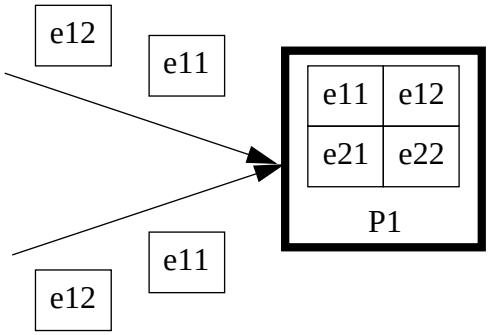

```
P1:p:concat m n->(concat m n)-1
P2:p:concat m n->(concat m n)-1
A = P1 >-> P2
```

This two-processor network is not a very exciting vehicle for signal-processing implementation. To be able to distribute load over many processors, a way of notating architectures that supports data-sharing is needed.

If we allowed protocols that distributed data to more than one connection, we could use this to split datastructures such as matrixes to many processors, thus laying ground for specifications of more interesting architectures. Protocols must then be defined that **split** high-level datastructures to two or more connections, and **join**ed elements from two or more connections to form higher level datastructures.

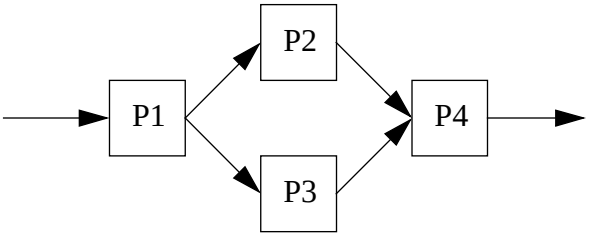


A split-functionality protocol



A join-functionality protocol

Here is an example four processor architecture:



Processors P1..P4 have the types (protocols):

```
P1:p:concat m n->split [m/2,m/2] n
P2:p:concat (m/2) n->(concat (m/2) n)-1
```

```
P3:p:concat (m/2) n->(concat (m/4) n)-1
P4:p:join [m/2,m/4] n->(concat 3m/4 n)-1
```

The *join* and *split* functions splits and joins a matrix according to the number of rows in the provided argument lists. Note that the function running on processor P3 reduces the number of rows in the matrices that flow through the architecture.

In order to specify which processors are interconnected, the combinators >-> (serial connection) and >+> (parallel connection) are used:

```
A = P1 >-> (P2 >+> P3) >-> P4
```

This is a short and elegant architecture notation! A correctly written type-checker for this enriched type-system should not detect any protocol-conflict in this particular architecture. If the types were given as below, on the other hand, an error would be present on account of the mismatch between P1 and P2. The type-checker would immediately detect this and signal an interconnection mismatch.

```
P1:p:concat m n->split [m/3,m/2] n
P2:p:concat (m/2) n->(concat (m/2) n)-1
P3:p:concat (m/2) n->(concat (m/4) n)-1
P4:p:join [m/2,m/4] n->(concat 3m/4 n)-1
A = P1 >-> (P2 >+> P3) >-> P4
```

Good ways of formulating interconnections and protocols are important tools for notation of transformation goals.

11. Optimization of functional code

11.1. Background

Even if we had a compiler that generated correct machine instructions or C code from a functional program, we could still not expect to run the produced code on a signal-processor. Conventional functional compilers expect support for dynamically allocated memory and garbage-collection. Signal-processing programs, on the other hand, operate on data of a pre-determined size, and have little use for dynamic memory allocation. Garbage collection is therefore unnecessary for the class of programs considered in this thesis, and should be avoided.

A traditional functional programming style utilizes a large amount of intermediate datastructures that are removed before the final output is presented. This is the primary reason for the need of run-time memory management and the large memory demands of executable functional programs. As signal-processors have a severely limited amount of on-chip memory, only a subclass of all (functional) programs are suitable for these processors. If we want use our specifications, we must use alternative ways of preparing the output that avoids these wasteful datastructures.

Practice has unfortunately showed us that programs with minimal dynamic memory utilization tend to be very hard to read, understand and maintain. The cure for this apparent conflict is **deforestation**. A short introduction to this transformation can be found in section A.5. at page 55.

11.2. A new notation

In order to successfully deforest a functional program, we must make sure that the number of fully simplified different results of applying the transformational rules are finite. To ascertain this, we must use a notation for certain constructs that groups an infinite number of derivations into one.

Traditional deforestation works well for functional programs that produces single element output from input elements. This is unfortunately not good enough, since signal-processing programs produce multidimensional data. Simple experiments with code that operate on matrices represented as lists of lists has showed that the transformation of programs that produces more than single elements as output often result in sequences of derivations with more and more free elements, containing a residue that can't be completely removed. This is illustrated in this simple example of deforestation applied to the expression **map f data** where **map f x:xs = (f x):(map f xs)** and **data** is assumed to be of infinite length:

1. **map f data**
2. **(f x1):(map f rest) {data = x1 : rest}**
3. **y1:(map f rest) {y1=f x1}**
4. **y1:(f x2):(map f rest2) {rest = x2 : rest2}**
5. **y1:y2:(map f rest 2) {y2=f x2}**
6. **etc. forever**

Here each line contain a derivation, with sketchy imperative state-code inside the curly braces. **data = x1 : rest** means that the head of data is bound to **x1** and that the remaining list is bound to **rest**. Each state points to the next, if no explicit goto-statement is present within the curly braces. The transformation evidently results in an infinite amount of derivations since the length of **data** is unknown, and no opportunity for knot-tying can be found. This important problem can be solved by the introduction of a notation and some new transformations that allows the knot-tying to work properly, thus reducing the number of derivations.

$$[X_i]_{i=a}^{b} = [X_a, X_{a+1} .. X_b]$$

T_1: $x:(f\ xs) = [x_i]_{i=a}^{b}$ ++ (f xs) (a :=1,b:=1)

T_2: $[x_i]_{i=a}^{b}$ ++ $[y_i]_{i=c}^{d} = [x_i]_{i=a}^{b+(c-d+1)}$ $\{x_{b+1}..x_{b+c-d+1} = y_c..y_d\}$

The first rule T_1 simply states that a single element can be notated as an list of length one. The second rule T_2 expresses the validity of concatenating two lists by copying the elements of the second into elements that are appended at the end of the first.

If this new notation and the transformation rules are used appropriately, knot-tying can be used in the normal way, substituting the infinitely descending arm for a loop by applying the generalization rule and going back to previous states. As a consequence, a large subclass of previously untransformable programs can now be deforested.

The previous example would look like this, if the new notation and rules were used:

1. **map f data**
2. **(f x1):(map f rest) {data = x1 : rest}**
3. **y1:(map f rest) {y1=f x1}**

4. **$[x_i]_{i=a}^{b}$ ++ (map f rest) {a = 1,b = 1, x_1 = x1}**
5. **$[x_i]_{i=a}^{b}$ ++ ((f x2):(map f rest2)) {rest = x2 : rest2}**
6. **$[x_i]_{i=a}^{b}$ ++ (y2:(map f rest2)) {y2= f x2}**
7. **$[x_i]_{i=a}^{b}$ ++ $[y_i]_{i=c}^{d}$++(map f rest2) {c = 1,d = 1,y_1=y2}**
8. **$[x_i]_{i=a}^{b+(c-d+1)}$++(map f rest2) {$x_{b+1}..x_{b+c-d+1} = y_c..y_d$}**
9. **$[x_i]_{i=a}^{b}$++(map f rest) {b = b+c-d+1,rest = rest2, goto 5}**

As can be seen, the result in derivation 9 is identical to derivation 4 which means that knot-tying by generating a goto-link from state 9 to state 5 can be employed. Since D_1-D_9 covers all possible derivations, the deforestation is successful.

11.3. Application of the new notation to a representative problem

To further clarify the use of the new notation and the related rules, we will try our hand on an example that has proven to be a simple, yet challenging problem. The function **transp** transposes a matrix described by a list of lists that arrive one element at a time (row-wise) on a one-dimensional stream, by turning each row into a column and writing the matrix one element at a time (row-wise) on the output stream. One of the troublesome properties of **transp** is that it needs to read almost the whole matrix before it can output a single row. This is the most demanding situation that can arise when reshuffling data in a matrix, which makes this simple problem a good test-case for the proposed notation.

11.3.1 General

In order to simplify the process, only the derivations and some very sketchy state-code will be shown. The full state-code is simple to deduce. Note that an tail-recursive variant of **transp** is used here, to minimize the memory that is used on the stack for call-frames. Trivial tests and simplifications are not explicitly written out as individual states for the benefit of clarity.

11.3.2 Definitions of used functions

```
transp m = transp' [] m
transp' m' m@(r:rs) =
  if r == [] then
     m'
  else
     transp' (m'++[map head m]) (map tail m)

head (x:_) = x
tail (_:xs) = xs
map _ [] = []
```

```
map f (x:xs) = (f x):(map f xs)
```

11.3.3 The type information (channel connection)

In order to specify the how the matrix is fitted into the one-dimensional stream, we use the following protocol definition for the **transp** function operating on the **proc1** processor (for a discussion on the notation of protocols, see section 9. at page 23):

```
transp::[[a]]->[[a]]
proc1:p:concat^r_c->(concat^r'_c')^-1 (r' = c,c'=r)
```

11.3.4 Transformation rules

Careful application of the following rules are the key to success in this transformation:

- Replace with definition
- Pattern match:

Tabell 1:

Pattern	What to do
x:xs	read one element (x) of data from channel (with proper size) if data is unknown and data is available.
x:xs	If all data is read: fail
x:xs	If data is known, bind variables
[]	If unknown data, check if no more data is available
[]	If known data, check if empty list

- Specialisation of if-statement
- Knot-tie by renaming of variables and ‘goto’ to some previous state
- []++x = x++[] = x
- x:xs = [x]++xs = $[x_i]_{i=a}^{b}$ {a :=1,b:=1}
- $[x_i]_{i=a}^{b}$ ++ $[y_i]_{i=c}^{d}$ = $[x_i]_{i=a}^{b+(c-d+1)}$ $\{x_{b+1}..x_{b+c-d+1} = y_c..y_d\}$
- $[[x_i]_{i=a}^{b}] = [([x_i]_{i=a}^{b})_j]_{j=c}^{d}$
- $[([x_i]_{i=a}^{b})_j]_{j=c}^{d}$ ++ $([y_i]_{i=e}^{f})_j]_{j=g}^{h}$ = $[([x_i]_{i=a}^{b})_j]_{j=c}^{d+h-g+1}$
  $\{([x_i]_{i=a}^{b})_{b+1}..([x_i]_{i=a}^{b})_{b+c-d+1} = ([x_i]_{i=a}^{b})_c..([x_i]_{i=a}^{b})_d\}$
  iff a-b = e-f
- Testing if $[x_i]_{i=a}^{b}$ = [] is equivalent to checking if a>b

11.3.5 The informal transformation

1. **transp m**
2. **transp’ [] m**

3. **if head m == []**
   []
   else
   transp' ([]++[map head m]) (map tail m)
4. **{m=r:rs}**
5. **{if r==[] then goto 6 else goto 7}**
6. **[] {output [],stop}**
7. **transp' ([]++[map head m]) (map tail m)**
8. **{if m==[] then goto 9 else goto 11}**
9. **transp' ([]++[[]]) []**
10. **[[]] {output [[]],stop}**
11. **transp' ([]++[(head r):(map head rs)]) ((tail r): (map tail rs))**
12. **{r=e:es}**
13. **transp' ([]++[e:(map head rs)]) (es:(map tail rs))**
14. **{e = $[e_i]_{i=a}^{b}$, a:=1, b:=1, es = $[es_i]_{i=c}^{d}$, c:=1, d:= 1}**
15. **transp' ([]++[$[e_i]_{i=a}^{b}$++(map head rs)]) ([$es_i]_{i=c}^{d}$++(map tail rs))**
16. **transp' [$[e_i]_{i=a}^{b}$++(map head rs)] ([$es_i]_{i=c}^{d}$++(map tail rs))**
17. **{if rs==[] then goto 25 else goto 18}**
18. **{rs=r':rs'}**
19. **transp' [$[e_i]_{i=a}^{b}$++((head r'):(map head rs'))]**
    [$es_i]_{i=c}^{d}$++((tail r'):(map tail rs'))
20. **{r'=e':es'}**
21. **transp' [$[e_i]_{i=a}^{b}$++(e':(map head rs'))]**
    [$es_i]_{i=c}^{d}$++(es':(map tail rs'))
22. **{e_{b+1} := e',es_{d+1} :=es'}**
23. **transp' [$[e_i]_{i=a}^{b+1}$++(map head rs')]**
    [$es_i]_{i=c}^{d+1}$++(map tail rs')
24. **{b := b+1,d := d+1, rs=rs',goto 16}**
25. **transp' [$[e_i]_{i=a}^{b}$] $[es_i]_{i=c}^{d}$**
26. **{e:=1,f:=1}**
27. **transp' [$([e_i]_{i=a}^{b})_j]_{j=e}^{f}$ $[es_i]_{i=c}^{d}$**
28. **if head $[es_i]_{i=c}^{d}$== []**
    $[([e_i]_{i=a}^{b})_j]_{j=e}^{f}$
    else
    transp' ($[([e_i]_{i=a}^{b})_j]_{j=e}^{f}$++[map head $[es_i]_{i=c}^{d}$]) (map tail $[es_i]_{i=c}^{d}$)
29. **{if c>d then 30 else 31}**
30. **$[([e_i]_{i=a}^{b})_j]_{j=e}^{f}$ {output $[([e_i]_{i=a}^{b})_j]_{j=e}^{f}$,stop}**
31. **transp' ($[([e_i]_{i=a}^{b})_j]_{j=e}^{f}$++[(head es_c):(map head $[es_i]_{i=c+1}^{d}$)])**
    ((tail es_c):(map tail $[es_i]_{i=c+1}^{d}$))

32. $\{es_c = y:ys\}$

33. $transp'\ ([([e_i]_{i=a}{}^{b})_j]_{j=e}{}^{f}{+}{+}[y:(map\ head\ \ [es_i]_{i=c+1}{}^{d})])\ (ys:(map\ tail\ \ [es_i]_{i=c+1}{}^{d}))$

34. $\{g:=1,h:=f,m:=1,n:=1,y_1 = y,ys_1 = ys\}$

35. $transp'\ ([([e_i]_{i=a}{}^{b})_j]_{j=e}{}^{f}{+}{+}[[y_i]_{i=g}{}^{h}{+}{+}(map\ head\ \ [es_i]_{i=c+1}{}^{d})])\ ([ys_i]_{i=m}{}^{n}{+}{+}(map\ tail\ \ [es_i]_{i=c+1}{}^{d}))$

36. $\{if\ c+1>d\ then\ 37\ else\ 41\}$

37. $transp'\ ([([e_i]_{i=a}{}^{b})_j]_{j=e}{}^{f}{+}{+}[[y_i]_{i=g}{}^{h}])\ [es_i]_{i=m}{}^{n}$

38. $\{([e_i]_{i=a}{}^{b})_{f+1} := [[y_i]_{i=g}{}^{h}]$ **ON THE CONDITION THAT** $a-b = h-g\}$

39. $transp'\ ([([e_i]_{i=a}{}^{b})_j]_{j=e}{}^{f+1})\ [es_i]_{i=m}{}^{n}$

40. $\{f:=f+1,goto\ 27\}$

41. $\{es_{c+1} = z:zs\}$

42. $transp'\ ([([e_i]_{i=a}{}^{b})_j]_{j=e}{}^{f}{+}{+}[[y_i]_{i=g}{}^{h}{+}{+}(z:(map\ head\ \ [es_i]_{i=c+2}{}^{d}))])\ ([ys_i]_{i=m}{}^{n}{+}{+}(zs:(map\ tail\ \ [es_i]_{i=c+2}{}^{d})))$

43. $\{y_{h+1} := z,ys_{n+1}:=zs\}$

44. $transp'\ ([([e_i]_{i=a}{}^{b})_j]_{j=e}{}^{f}{+}{+}[[y_i]_{i=g}{}^{h+1}{+}{+}(map\ head\ \ [es_i]_{i=c+2}{}^{d})])\ ([ys_i]_{i=m}{}^{n+1}{+}{+}(map\ tail\ \ [es_i]_{i=c+2}{}^{d}))$

45. $\{h:=h+1,n:=n+1,c:=c+1,goto\ 35\}$

Since all nodes now are distinct end-nodes or contains a loop back to a previous node, the transformation is complete. Observe that in state 39 we have to know that the condition of the transformational rule is fulfilled (for example by means of the protocols) in order to ascertain that the transformation is valid.

11.3.6 Some observations about the transformation

Aided by some simple rules, **transp** is informally deforested. Key observations:

- We need to have a channel specification that informs the system how in and out-data is delivered to the processor. Otherwise it is unknown what data-quantities should be read and written by the deforested code.
- There is no way of guaranteeing the validity of the following transformation rule unless the datasize is specified:

$$[([e_i]^{b}{}_{i=a})_j]^{d}{}_{j=c} \mathbin{++} [([e_i]^{f}{}_{i=e})_j]^{h}{}_{j=g} = [([e_i]^{b}{}_{i=a})_j]^{d+(h-g)}{}_{j=c}$$

if

$$([e_i]^{b}{}_{i=a})_{j=d+1} \mathbin{..} ([e_i]^{b}{}_{i=a})_{j=d+(h-g)} = ([e_i]^{f}{}_{i=e})_{j=g} \mathbin{..} ([e_i]^{f}{}_{i=e})_{j=h}$$

11.4. Automatic transformations

Further investigations into how deforestation could be automated by formalizing the rules and incorporating them into a transformation system have been undertaken, specifically in relation to a simple framework written in Haskell by Thomas Johnsson. Unfortunately, time

did not allow these experiments to be taken to their conclusions, but no doubt exists in my mind regarding the feasibility an automatic transformation program.

12. Compilation to distributed architectures

12.1. The basic idea for a first attempt

If the optimized functional language signal-processing programs can be compiled to a *single* signal-processor, the next step is multiprocessor architectures.

Since no satisfactory way of automatically partitioning functional programs to more than one processor exists, annotations must be provided by the system designer that aids the transformation system. These annotation should tell the transformation system how the architecture is connected, with what protocols, and give hints for target locations of program segments.

Based on these annotations, that preferably should be given in the same functional language as the signal-processing programs, the transformation goal is to map the code to the architectural specification. This approach to parallel architecture code-generation is explored in [Sha90]. The program specification together with protocol and architecture descriptions should be enough information for individual processor code generation, given a good one-processor target compiler.

12.1.1 A sketchy methodology

As stated above, a verified and tested functional language specification of a signal-processing program, specification of an architecture, and some way of compiling a functional program to a single signal-processor should be sufficient basis for a multi-processor compiler. This compiler's responsibilities includes generation of code-segments that corresponds to all parts of the original program, and insertion of correct communication primitives needed for synchronization of processors and data exchange.

How should this be accomplished? Ideally the user should be able to mark which parts of the code to put on which processor(s). The compiler should then transform the program into a number of sub-programs, each assigned to one processor. These smaller programs, specifications of the architecture interconnections and the one-processor compiler is utilized to yield distributed code containing the functionality of the original program together with communication and synchronization calls.

The implementation of an compiler that automatically maps code to any architecture optimally would be a breakthrough in signal-processing development.

13. High-level development support

13.1. Why is it important?

High-level support for development of aids the programmer to write signal-processing software rapidly and correctly, thus reducing the costly implementation phase. This is one of the big advantages of approaching a problem previously viewed as a fundamentally low-level development problem from the angle adopted by this thesis. Once the paradigm of construction has changed from writing low-level code directly, the available benefits of high-level languages should be exploited immediately. To waste the knowledge painfully discovered by others in the field of program development is as dumb as refusing to use an electric drill instead of an ordinary hand-drill based on unfamiliarity with electric tools. Why not do a few changes when the rewards are so high?

In conventional program development, the need for high-level languages and associated concepts has been recognized for a long time. Efficiency issues has unfortunately hindered these paradigms from influencing high-performance signal-processing application development. If a bridge was successfully built between high-level specifications and efficient low-level code, it would be shameful not to utilize the possible benefits fully.

13.2. An example: Symbolic Evaluation

13.2.1 Why?

A datatype that supports both numerical values (Float, Complex, Int etc.) and symbolic expressions was introduced as the base type of the library and simulation to illustrate additional benefits of using a functional language for specification purposes. To be able to use non-standard interpretation, a number of standard prelude functions has been overloaded for this datatype. The goal is to automatically produce symbolic expressions results if variables is used as an input rather than numeric values. Thus, the correctness of routines can be ascertained by comparing the expressions generated by using symbolic inputs to the desired mathematical expressions. Symbolic evaluation has previously been utilized to generate hardware descriptions from functional specifications in the Ruby system [JS90].

13.2.2 An example

To elaborate a little on the use of non-standard interpretation as a mean for verifying the correctness of routines, the following illustrates how the FFT routine can be verified for vectors of size 4 and 8:

```
? sex4

[C[0], C[1], C[2], C[3]]

? vfft 4 sex4

[((C[0]+W(0,4)*C[2])+W(0,4)*(C[1]+W(0,4)*C[3])), ((C[0]-
W(0,4)*C[2])+W(1,4)*(C[1]-W(0,4)*C[3])), ((C[0]+W(0,4)*C[2])-
W(0,4)*(C[1]+W(0,4)*C[3])), ((C[0]-W(0,4)*C[2])-W(1,4)*(C[1]-
W(0,4)*C[3]))]
```

The vector **sex4** contains symbolic variables, instead of numeric values. When the vector FFT operation **vfft** of size 4 is applied, the result is a vector containing symbolic expressions. These expressions are built from the variables C[0]..C[3] and the phase-factors W(0,4)..W(3,4) where $W(x,y) = W_y^x$.

```
? sex8

[C[0], C[1], C[2], C[3], C[4], C[5], C[6], C[7]]

? vfft 8 sex8

[(((C[0]+W(0,8)*C[4])+W(0,8)*(C[2]+W(0,8)*C[6]))+W(0,8)*((C[1]+W(0,8
)*C[5])+W(0,8)*(C[3]+W(0,8)*C[7]))), (((C[0]-
W(0,8)*C[4])+W(2,8)*(C[2]-W(0,8)*C[6]))+W(1,8)*((C[1]-
W(0,8)*C[5])+W(2,8)*(C[3]-W(0,8)*C[7]))), (((C[0]+W(0,8)*C[4])-
W(0,8)*(C[2]+W(0,8)*C[6]))+W(2,8)*((C[1]+W(0,8)*C[5])-
W(0,8)*(C[3]+W(0,8)*C[7]))), (((C[0]-W(0,8)*C[4])-W(2,8)*(C[2]-
W(0,8)*C[6]))+W(3,8)*((C[1]-W(0,8)*C[5])-W(2,8)*(C[3]-
W(0,8)*C[7]))), (((C[0]+W(0,8)*C[4])+W(0,8)*(C[2]+W(0,8)*C[6]))-
W(0,8)*((C[1]+W(0,8)*C[5])+W(0,8)*(C[3]+W(0,8)*C[7]))), (((C[0]-
W(0,8)*C[4])+W(2,8)*(C[2]-W(0,8)*C[6]))-W(1,8)*((C[1]-
W(0,8)*C[5])+W(2,8)*(C[3]-W(0,8)*C[7]))), (((C[0]+W(0,8)*C[4])-
W(0,8)*(C[2]+W(0,8)*C[6]))-W(2,8)*((C[1]+W(0,8)*C[5])-
W(0,8)*(C[3]+W(0,8)*C[7]))), (((C[0]-W(0,8)*C[4])-W(2,8)*(C[2]-
W(0,8)*C[6]))-W(3,8)*((C[1]-W(0,8)*C[5])-W(2,8)*(C[3]-
W(0,8)*C[7])))]
```

The generated expressions is equivalent to the mathematical definition of the FFT, so our routine is correct (at least for the size 4 and 8).

For numerical values, the result is of course numerical:

```
? ex8

[1.0+2.0j, 3.0+1.0j, 7.0+6.0j, -9.0+7.0j, -4.0-1.0j, 8.0+3.0j,
9.0+0.0j, 7.0+0.0j]

? vfft 8 ex8

[22.0+18.0j, 22.3137+13.4853j, -22.0-18.0j, -5.24264+22.2132j, 4.0-
4.0j, -0.313709-3.48528j, -16.0+8.0j, 3.24264-20.2132j]
```

As you can see, the function **vfft** returns different result formats effortlessly. Granted, the symbolical expressions quickly becomes impossible to decipher by hand, but further simplification and structuring could be done before presentation. Important uses for the generated expressions include raw-data input to functions that check the validity of assumptions about code-properties. A similar technique could be employed to generate in-data to a commercial theorem prover such as the Logikkonsult NP AB's Prover [Log96].

13.2.3 How is it implemented?

For a discussion of the implementation of the symbolic evaluation, see section 8.6. at page 17.

13.2.4 Further uses of Symbolic Evaluation

Many interesting things can be done with the symbolic evaluation to aid the developer in verifying the library and dependent routines. The implementation of the **Expr** datatype could be used as a basis for partial evaluation in order to specialize generated expressions and algorithms. This could be a useful way of automatically deriving special instances of

general signal-processing programs. Another interesting experiment would be to generate data-flow code from the specifications and examine how compilated versions of this code compares to compiled deforested functional programs.

13.3. Proposed uses of an enriched type-system

13.3.1 Why use an enriched type system at all?

In the previous section covering compilation to distributed architectures, the protocol specifications was proposed to be implemented as an extension of the normal type-system. The protocols could easily be specified in other ways, so why this particular approach? My belief is that this kind of protective shell around vulnerable objects is important for several reasons, and should be implemented in the type system as it is the classical place for safeguards. I consider it of utmost importance to make the potential user of the signal-processing library aware of the power of a strong type system.

13.3.2 New possibilities

In the proposal for a notation for architectures, the use of an enriched type-system was advocated where types was used to capture the protocols used by processors to communicate with its neighbours. This concept could be used to address a number of other issues. These include:

- Use of some variant of **sized** or **shapely** types to tag matrices and vectors with sizes in order to make illegal manipulations (such as adding two matrices of different size) impossible. Matrix dimension errors are extremely strong indicators of a potentially faulty algorithm. As a consequence of implementing the dimensional information in the type-system, all mismatches are discovered at the earliest possible time, saving the developer expensive debugging. Since the type-checker would flag the code as faulty and hence uncompilable, the user could not compile the erroneous code even if he wanted to.

```
+::Matrix m n->Matrix m n->Matrix m n
m1::Matrix 3 2
m2::Matrix 3 1
m3=m1+m2 (ILLEGAL)
```

- Use of the same kind of types to make connection of data-streams of different rates illegal (here the combinator **>c>** is utilized to connect two streams).

```
>c>::Stream r1->Stream r1->Stream r1

s1::Stream 2000

s2::Stream (2000/2)

s3 = s1>c>s2 (ILLEGAL)
```

- Ensurance of observation of processor capacity limits. The complexity of a block should be part of its type and could be decided by applying a metric to expressions generated by the symbolic evaluation.

14. Conclusions

Concrete results of the thesis include the signal-processing library and the airborne radar simulation, both important as real-life input for transformation programs. This input is needed if further investigations into possible uses of functional languages to specify signal-processing is to be conducted. In fact, one use of the thesis is as a preliminary investigation into the recommended angle of solution, and as a technical survey of research applicable to this problem.

The generated code and the time put into writing it clearly shows that functional languages provide an interesting alternative to the present low-level development strategy. Furthermore, the deforestation work with the key problem **transp** seems extremely promising. It is very possible that an efficient way of compiling the specifications to signal-processing code can be devised, and that other transformations can be applied to generate distributed implementations. If this is the case, a problem hitherto deemed impossible to avoid can be elegantly solved, removing the source of a large portion of the costs associated with development of signal-processing systems.

The thesis has illustrated that additional advantages of using a functional language such as Haskell as a specification tool, are the support for non-standard interpretation and the powerful built-in type-system which simplifies development of new algorithms. The bridge that the deforestation transformation could build between a high-level language with this kind of features and efficient implementations is very valuable. An implementation of the proposed enriched type-system would make this statement even more true.

Personally, the thesis work and the collaboration with a number of skilled researchers has been a source of great pleasure, and a valuable learning experience. In six months, I have learned more than in the previous two years. It has been hard work, but fun.

15. Future work

The thesis work could be extended and carried forward in a number of ways. Further investigations that I strongly advocate are:

- Continued experiments with transformation software, directed at producing efficient executable code from functional programs. A number of experiments were conducted within the framework of this thesis, but as stated above, time did not permit a full implementation of all the rules proposed in connection to the transformation of **transp**.
- More work directed toward the invention of an good notation usable for the formulation of distributed architectures.
- Experiments with the intent of developing an fully automatic methodology for producing efficient distributed imperative programs containing synchronization and communication primitives inserted by the compiler. This compiler should work with the functional specification and the annotated architecture as an input.
- Inquiries into the how the ideas about extended type-systems really could aid the developer of signal processing software.
- Extension of the symbolic evaluation and research about how the generated expressions could be used.

- Inquiries into how formal methods could be utilized to verify specifications and generated software.
- Addition of further facilities for simulation and information visualization in order to develop a stand-alone tool for signal-processing development.

16. References

[AJ89] Augustsson, L. and Johnsson, T.,*Parallel Graph Reduction with the <nu, G>-machine*,Proceedings of the 1989 Conference on Functional Languages, 202-213,London, England,1989

[BG92] Berry,G. and Gonthier, G.,*The Esterel Synchronous Programming Language: Design, Semantics,Implementation*,Science of Computer Programming vol. 19, n˚2, 87-152, 1992

[CFB92] Cann, D.C and Feo, J.T. and Bohoem, A.D.W et. al.,*SISAL Reference Manual : language version 2.0* ,1992

[CH93] Carlsson, M. and Hallgren, T.,*FUDGETS - A Graphical User Interface in a Lazy Functional Language*,In Functional Programming & Computer Architecture,March 1993

[CHK+92] Cox, S. and Huang, S.-Y. and Kelly, P. et. al.,*An implementation of static functional process networks*,Parallel Architectures and Languages Europe,497-512,Springer Verlag,1992.

[Dar81] Darlington, J.,*Alice: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages*,Proceedings 1981 Conference on Functional Languages and Computer Architecture,Wenthworth-by-the-sea,Portsmouth,New Hampshire,1981

[Den94] Dennis, J.B.,*Stream data types for signal processing in Advances in Dataflow Architecture and Multithreading*,IEEE Computer Society Press,1994

[Den95] Dennis, J.B.,*Static mapping of functional programs: An example in signal processing*,High Performance Functional Computing,149-163,1995

[GHR+90] Genin, D. and Hilfinger, P. and Rabaey, J. et al.,*DSP Specification Using the Silage Language*,Proc. of Int. Conf. on Acoustics, Speech, and Signal Processing vol 2,1056-60,April 1990

[Gill96] Gill, A.,*“Cheap Deforestation for Non-strict Functional Languages”* ,Glasgow University, 1996

[HCAA94] Hicks, J. and Chiou, D. and Ang, B.-S. and Arvind,*Performance studies of id on the monsoon dataflow system*,CSG Memo 345-3,Laboratory for Computer Science, Computation Structures Group,MIT,1994

[HCR+91] Halbwachs, N. and Caspi, P. and Raymond, P. et. al.,*The synchronous dataflow programming language Lustre*,Proceedings of the IEEE vol. 79 nr. 9, September 1991

[HRR91] Halbwachs, N. and Raymond, P. and Ratel, C.,*Generating Efficient Code From Data-Flow Programs*,Third International Symposium on Programming Language Implementation and Logic Programming,Passau (Germany),August 1991

[Hud92] Hudak, P. et al.,*Report on the Programming Language Haskell: A Non-Strict,Purely Functional Language*,March 1992. Version 1.2. Also in Sigplan notices, May 1992

[HV92] Hartel, P. and Vree, W.,*Arrays in a lazy functional language - a case study: the fast Fourier transform*, Proceedings of the 2nd Workhop on Arrays, functional languages, and parallel systems (ATABLE),52-66,1992

[IG95] Isenstein, B.S and Greene J.,*Execution of parallel algorithms on a heterogenous multicomputer*,Mercury computer Systems, Inc.,1995

[JGS93] Jones, N. D. and Gomard, C.K. and Sestoft, P.,*Partial Evaluation and Automatic Program Generation*,Prentice-Hall,1993

[JS90] Jones, G. and Sheeran, M.,*Circuit design in Ruby*.In Staunstrup, J., editor,Formal Methods for VLSI Design.,North-Holland, 1990

[KGG+94] Kuner, V. and Grama, A. and Gupta, A. et. al.,*Introduction to parallel computing- design and analysis of parallel algorithms*,The Benjamin/Cummings Publishing Company,Redwood City,California,USA,1994

[Log96] Logikkonsult NP AB Formal methods Sweden, *NP-Tools 2.2 Reference Manual*, Logikkonsult NP AB Formal methods Sweden technical report NPT-01-07-02

[Lee95] Lee, E.A.,*Modeling Radar Systems Using Hierarchical Dataflow*,Proc. of IEEE Int. Conf. on Acoustics, Speech, and Signal Processing,1995

[PHEJ95] Pino, J.L. and Ha, S. and Lee, E.A. and Buck, J.T.,*Software Synthesis for DSP using Ptolemy*,Journal of VLSI Signal Processing, 7-21, 1995

[PJ87] Peyton~Jones, S.L., *GRIP: A Parallel Graph Reduction Machine*,In Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture, Portland, Oregon, U.S.A, September 1987

[PM92] Proakis, J. and Manolakis, D.,*Digital signal processing: principles, algorithms, and applications -2nd ed.*,Macmillan publishing company,New York,New York,USA,1992

[RdMV+88] Rabbaey, J. and deMan, H. and Vanhoof, J. and Goosens, J. and Cathorr, F.,*CATHEDRAL II: A Synthesis System for Multiprocessor DSP Systems*, 311-360,Addison Wesley, 1988.

[Ree95] Reekie, H.J.,*Realtime Signal Processing: Dataflow, Visual, and Functional Programming*,University of Technology at Sydney,1995

[Sha90] Sharp, D.W.N,*Functional Language Transformation For Parallel Computer Architecures*,Imperial College of Science,Technology and Medicine,1990

[Sha93] Sharp, D.W.N. and Cripps, M.D.,*Synthesis of the Fast Fourier Transform Algorithm by Functional Language Program Transformation*,Proc. Euromicro Workshop on Parallel and Distributed Processing,136-143,1993

[Tur86] Turchin, V.F,*The concept of a supercompiler*,ACM Transactions on Programming Languages and Systems,8(3):292-325, July 1986

[Wad84] Wadler, P., *Listlessness is better than laziness: lazy evaluation and garbage collection at compile time*,In Proc. ACM Conf. on Lisp and Functional Programming, 45-52, Austin, Texas, 1984

[Wad88] Wadler, P.,Deforestation: *Transforming programs to eliminate trees*,In European Symposium on Programming, 344-358, Nancy,March 1988.

[WFC+93] Wray, J. P. and Fitzpatrick, Stephen and Clint, M. and Kilpatrick, P. L.",*Deriving Distributed MIMD Implementations from Functional Specifications*,Advances in Parallel Computing 9,353-358,1993

[WW87] Watson, P. and Watson, I.,*Evaluating Functional Programs on the FLAGSHIP Machine*,In Proceedings 1987 Functional Programming and Computer Architecture, 80-97, 1987

17. Appendices

Appendix A. Introductions to related areas

Since the thesis brings together concepts from two different fields, five short introductions are included to introduce some basic terms and relations to readers less familiar with either Computer Science or Electrical Engineering:

A.1. Radar

A.1.1 The basics

Radars transmit short duration pulses as electromagnetic waves, and receive echoes from any electromagnetic reflector in the path from the radar. The (pulse delay) radar measures the time between the transmitted pulse and the received pulse in order to determine the distance between the reflector and the radar.

The distance R is given by

$$R = \frac{c\Delta t}{2}$$

where c is the speed of light. From this can be seen that the range resolution of a radar can be expressed as

$$\delta R = c\frac{\tau_r}{2}$$

where δR is the distance beyond which two reflectors must be separated so that their echoes can be seen as two separate pulses (i.e. one pulse ends before the second starts). The range resolution is determined by the time duration of the pulse which equals the reciprocal of the transmitted signal's bandwidth. The radar pulse repetition frequency τ_r determines the maximum unambiguous range of the radar:

$$R_{max} = c\frac{\tau_r}{2}$$

Targets with ranges greater than R_{max} produce echoes that return to the radar after one or more pulse repetition periods and appear to have ranges between 0 and R_{max}.

A.1.2 The returned signal

What information can then be obtained from the pulses returning from the target? The distance certainly, but also the velocity of the target since the Doppler effect forces the frequency of the reflected wave to be altered.

A general form of a narrowband received signal s(t) from a radar can be expressed as

$$s(t) = A(t)\cos(\omega_0 t + \phi(t))$$

where ω_0 represents the carrier frequency, A(t) and ϕ(t) represent amplitude and phase modulation.The value ϕ(t) includes any phase modulations of the transmitted signal, Doppler effects and constant phaseshift. It is hence possible to distinguish echoes reflected by moving objects from stationary ground reflections (“clutter”). The received signal is passed through a demodulator to yield the signals I and Q:

$$I = A(t)\cos(\phi(t))$$

$$Q = A(t)\sin(\phi(t))$$

The signals can be thought of as projections of a phasor representing the intensity of the echo rotating at the doppler frequency. Both the components are needed in order to know in which direction the phasor is rotating (corresponding to positive or negative velocity). Returned signals are passed through a number of parallel filters (the Doppler filter bank) to separate the signal into a number of frequency channels.

A.1.3 Integration

The dynamic range of the returned signal in each channel is very wide, and most targets are obscured by noise (caused by imperfect electronic components etc.) In order to pull the target signal out of the surrounding noise, the signal is integrated over more than one pulse. This means that subsequent returned signals in a channel are added to each other. Since the noise has random phase from pulse to pulse, the noise-amplitude converges to a medium value. The phase of the echo of the target is constant and the integration therefore has the effect of raising the signal-level compared to the noise.

A.1.4 Detection

At the end of each integration period, the output of each filter is applied to a separate detector. If the integrated signal plus the accompanying noise exceeds a certain threshold, the detector concludes that a target is present. Since completely random noise sometimes will exceed the detector threshold, the sensitivity must be set so that to keep the false alarms to a reasonable level. On the other hand, if the threshold is set too high, weak targets may be missed. Clearly optimal values for this parameter is a crucial factor for successful operation. Since the mean noise-level and system gain may vary over a wide range, the outputs of the radar's doppler filters must continually be monitored to adjust the thresholds for each filter. As nearly as possible the levels are set as to maintain false-alarm rate to the optimum value for each filter. For this reason automatic detectors are called constant false-alarm-rate detectors (CFAR).

In order to further increase the sensitivity of the detector, a technique called the "m out of n criteria" may be used. If the time-on-target spans *n* integration periods, *m* crossings of the threshold level is required to flag a detected target, as opposed to only one.

A.1.5 Pulse compression

Ideally we would want both long detection range and fine range resolution. To achieve this we would like to transmit extremely narrow pulses of exceptionally high peak power. There are however practical limits to the amount of peak power a signal may contain, and to obtain long detection ranges at pulse repetition frequencies low enough for pulse delay ranging fairly wide pulses must be transmitted. The solution to this problem is to transmit modulated pulses of sufficient width to provide the necessary average power at a reasonable level of peak power, and then "compress" the received echoes by decoding their modulation. All methods of pulse compression are essentially matched filtering schemes in which the transmitted pulses are coded, and the received pulses are passed through a filter whose time-frequency characteristic is the conjugate of the coding's. Pulses that are to be sent are "decompressed" to make them longer and of less amplitude, and the returned echoes are passed through a filter that "compress" them into shorter signals of higher amplitude.

A.1.6 Amplitude weighting

In both the Doppler filters and the pulse compression filters, short pulses are passed through a filtering system. This leads to sidelobes in the frequency domain of the transfer function. In order to reduce these (to avoid getting energy originating from the same target in unnec-

essary many doppler-channels) the amplitudes of the samples in each pulse are weighted to reduce fringing effects. This is operation is often referred to as “windowing”.

A.1.7 Doppler radar

The main reason for sensing doppler frequencies is the need for computation of parameters used for separation of returns received simultaneously from different objects in addition to range rate measurement. In the absence of Doppler ambiguities (covered below), a target’s Doppler frequency may be determined simply by noting in which Doppler filter bank the target appears. The range rate is given by:

$$R' = \frac{-f_d \lambda}{2}$$

where R’ = dR/dt (range rate), f_d = doppler frequency and λ = transmitted wavelength. Since clutter arises from stationary objects, clutter echoes drowns information in the lower filter banks (and in the uppermost filters due to the periodical nature of the spectrum of pulsed signals). When the radar itself has a velocity relative to the ground, all frequencies must be translated in order to offset the clutter to the lowest filter-banks so that no information is pushed outside the pass-band of the filter-banks.

A.1.8 Range ambiguities

Pulse delay ranging works without a hitch as long as the round-trip transit time for the most distant target the radar may detect is shorter than the interpulse period. But if the radar detects a target whose transit time exceeds the interpulse period (as is the case when a target is beyond the maximum unambiguous range), the echo of one pulse will be received after the next pulse has been transmitted, and the target will appear to be at a much shorter range than it actually is. As stated above, the maximum unambiguous range is dependent on the pulse repetition frequency.

How is then the presence of ambiguous range return detected? One simple technique is called “Pulse Repetition Frequency (PRF) jittering”. Since the return-time from targets within the unambiguous range is independent of PRF of the transmitted wave, a change of PRF will only affect targets outside the unambiguous range. In this manner, targets may easily be divided into “near” and “far”.

To resolve ambiguities, an extension of PRF jittering called “PRF switching” is used. If we measure how much the apparent distance of the targets change when the PRF is changed, the unambiguous range may be computed using the expression:

$$R_{true} = \frac{R_u \Delta R_{apparent}}{\Delta R_u} + R_{apparent}$$

where R_u = maximum unambiguous range and $R_{apparent}$ = the apparent range. Unfortunately this does not solve the problem completely since we cannot tell which target is which after a PRF change if multiple targets are present. The measurement of $\Delta R_{apparent}$ thus presents some difficulties. More PRF:s must be used to address this shortcoming. In general it can be shown that the number of different frequencies must be (N-1) to allow the radar to uniquely measure the range of N simultaneously detected targets. In addition to this number of frequencies that must be used for *deghosting*, an additional number of PRF:s might be needed depending on the desired maximum unambiguous range and the individual PRF:s.

A.1.9 Doppler ambiguities

As in the case of range measuring, the measured doppler frequencies is not by necessity unambiguous. This stems from the fact that the spectrum of a pulsed signal is repetitive with the first sidebands separated from the carrier by the Pulse Repetition Frequency (PRF). If some sideband carrier offset by a multiple of the doppler frequency is within the frequency range covered in the filterbanks, an erroneous velocity will be detected.

To resolve doppler ambiguities, we must have some way of telling what whole multiple of the PRF, if any, separates the observed frequency from the carrier frequency. If not too great, this multiple may readily be determined by a PRF switching technique similar to the one used to resolve range ambiguities, yielding the expression:

$$f_d = \frac{f_r \Delta f_{obs}}{\Delta f_r} + f_{obs}$$

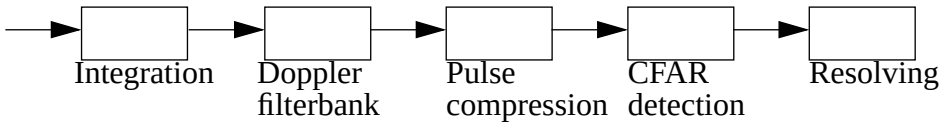
where f_d = true doppler range, f_r = amount PRF is changed and f_{obs} = observed frequency. As in the range resolving case, multiple PRF:s must be used to deghost multiple target returns.

A.1.10 The competing claims for PRF

Since a low PRF is needed to lessen the effects of range ambiguities, and a high PRF is called for to minimize the doppler ambiguities, the choice of PRF is crucial to the performance of the system. Low PRF radars need not worry about range ambiguities, but a lot of computing is needed to extract any usable doppler information. High PRF radars can accurately tell the velocities of targets, but need to resolve range ambiguities. Medium PRF radars need a little computing for both, but a lot less than high and low PRF radars.

A.1.11 An example system

To illustrate the concepts mentioned above, the layout of an example system is presented below. First the pulse-data from a number of pulses is filtered through the doppler filter-banks, to separate the return energy into doppler-channels, where each channel corresponds to a velocity interval. This separates the clutter from the moving targets (since the clutter has an zero absolute velocity). Successive pulses are then added to integrate the signal. Next, CFAR detection is performed on each doppler channel in order to separate targets from the noise with a certain probability of no errors. The target velocities and ranges are ambiguous, since the true range and velocity is “folded” into the interval given by the PRF. The final step is accordingly range resolving.



A.1.12 A quick explanation of some terms

- Range-velocity matrix, data matrix - The data from the receiver is a complex vector, with the amplitude of each range bin corresponding to the return in a given time interval, and the phase corresponding to the target’s velocity. A number of these vectors are grouped into a matrix, and then filtered to yield a matrix indexed by range and velocity.
- Range bin - The time for reflections of the sent pulse to return is measured and discretized, thus dividing the unambiguous distance into a number of bins. Each bin corresponds to a row in the data matrix.
- Doppler channel - After filtering, each frequency interval of the returned pulse is mapped into a separate column in the data matrix. This column corresponds to a velocity interval.
- Clutter - A very small percent of the return from receiver is actual reflections of the sought for moving targets. The ground and stationary objects return a large amount of energy that can mask important target information. This unwanted energy is called clutter. The counter for a cluttered return is to separate “targets” with an zero absolute velocity from targets with real movement.
- Noise - Due to imperfect analog electronics in the sender and receiver, a certain amount of noise is injected into the signal from the receiver. This is a potential source of error since the noise could drown out useful return energy. To alleviate this problem, the returned signal is integrated (meaning that mean value of a number of successive pulses is formed, hopefully lowering the noise in proportion to the target-return).

A.1.13 Some radar abbreviations

Tabell 2: Abbreviations of radar terms

Abbreviation	Meaning
PRI	Pulse range interval
INTI	Integration interval
PRF	Pulse repetition frequency
CFAR	Constant false alarm rate
RS	Range samples
RB	Range bins
DCH	Doppler channel

A.2. Digital signal-processing

For an in-depth treatment of digital signal-processing, see [PM92].

A.2.1 Discrete fourier transform

Informally the Discrete Fourier Transform of a sequence in the time-domain can be viewed as N values, each value X(k) corresponding to the amplitude of a sinewave with frequency f(k)=(k/2-1)/NT (where 1/T is F_s, the sampling frequency). The sum of these sinewaves will be a function deviating as little as possible from the original sequence.

The Discrete Fourier Transform is defined as a set of N samples of the Fourier transform X(ω) for a finite-duration sequence {x(n)} of length L less or equal to N, where the sampling of X(ω) occurs at the N equally spaced frequencies $\omega_k = 2*\pi*k/N$, k= 0,1,2..N-1. {X(k)} uniquely represents the sequence {x(n)} in the frequency domain, and can be transferred back to the time domain by means of the Inverse Fourier Transform:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_k^{(kn)}$$

$$x(k) = \frac{1}{N} \sum_{n=0}^{N-1} X(n) W_k^{(-k)n}$$

Notice the similarities between the transforms, only a factor N and the sign of k are different. Hence it is easy to retrieve the IDFT with an algorithm that computes the DFT for a given sequence.

The factor W is defined as

$$W = e^{\frac{-j2\pi}{N}}$$

Why is then the DFT important? Suppose that we have a finite-duration sequence x(n) of length L which excites a filter with an impulse response of finite duration M (a FIR-filter). Then the output sequence of y(n) of the FIR-filter in the time-domain may be expressed as the convolution of the input x(n) and the impulse response h(n), that is

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k)$$

If some simple conditions is met, this is equivalent to

$$Y(k) = H(k)X(k)$$

Thus the output sequence of a FIR filter can be computed by first applying the DFT to the input sequence, then multiplying the resulting transformed sequence on a per element basis with the DFT of the filter's impulse response and finally applying the IDFT to the final sequence. The reason for the importance of this approach to linear filtering is the multitude of effective algorithms (called FFT algorithms) for computing the DFT.

These algorithms exploit the symmetry and periodical properties of the phase factors W_N. In particular, two important properties are:

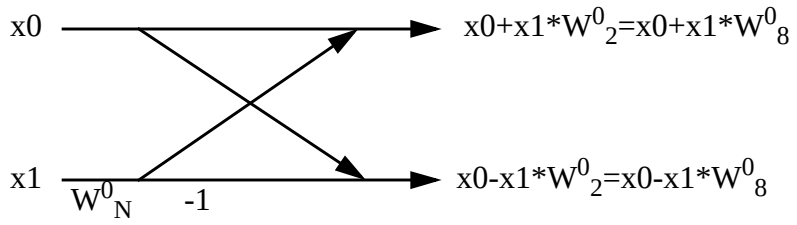
$$W_N^{k+N/2} = -W_N^k$$

$$W_N^{k+N} = W_N^k$$

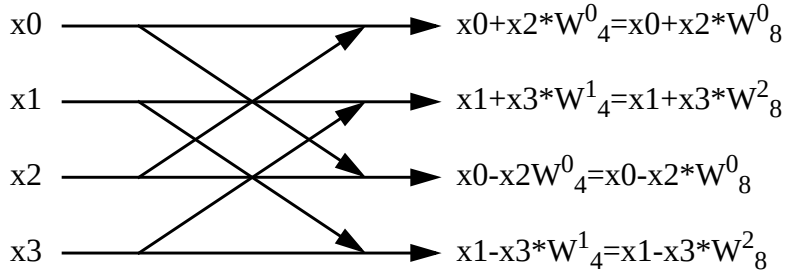
A simple algorithm named “the radix-2 FFT algorithm” exploits these properties by introducing the constraint that the number of elements in the complex array N must be 2^v where v is some integer number, in order to compute the DFT as follows:

The first step is to permute the elements of the complex array in bitreversed order (the new index into the array given as the binary representation of the old **read backwards**). The DFT of a one point sequence is then the sequence itself. Longer sequences are treated recursively as follows:

The arrays is grouped into adjacent pairs of two and the two-point-DFT is formed by a “butterfly computation”, here showed for the first two indexes:



The resulting output sequence is merged, and then split up into groups of four for which the four-point-DFT is computed using basic butterfly calculations on each element in the upper half of the group of four and a corresponding element in the lower half (here the first group of four):



Observe that at this stage x0 is the output of the prior stage, not the original x0. Next, groups of 8 is formed and the same procedure repeated. This continues until one block of N numbers have been computed, the N-point DFT.

A Haskell function that computes the FFT looks like this:

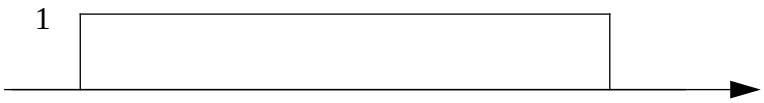
```
fft_twid::Int->[Complex]->[Complex]->[Complex]
fft_twid n tw cxl =
 let
       one_butterfly globn n twidvect cl = butterfly (div n 2) cl
           where
           butterfly num cl = upperpart ++ lowerpart
              where
              upperpart = apply3 upperfunc
              lowerpart = apply3 lowerfunc
              apply3 f =
                     vmap3 f (vtake num cl) (vdrop num cl) twidvect
              upperfunc = \a b w->a+w*b
              lowerfunc = \a b w->a-w*b
       stage_butterfly n pts tw = concat.(map (one_butterfly n pts stagetw)).(vsplit_l pts)
           where
           stagetw = map (\k->tw!!(k*level)) [0..((div pts 2)-1)]
           level = div n pts
```

```
        iter1 f [] val = val
        iter1 f (x:xs) val = iter1 f xs (f x val)
    in
        iter1 (\pts->stage_butterfly n pts tw) (takeWhile (<=n) (iterate (*2) 2))  (vbitreverse cxl)
```

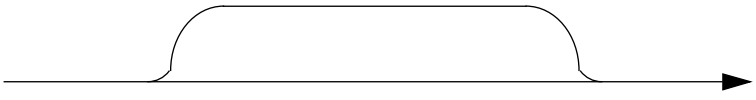
cxl is the datavector**, tw** is the corresponding vector of phasefactors and **n** is the length of the FFT (multiple of 2). This function operates by applying butterfly calculations of size 2,4,8..n to the result of the previous calculation. Applying size 0 butterflies is defined to be the bitreversed **cxl** vector, and the result of size **n** butterfly application is defined to be the FFT. This application is done by the *stage_butterfly* function that applies *one_butterfly* butterflies to the whole **pts** vector.

A.2.2 Windowing

The sidelobes present in the frequency domain of outputs from filters fed short sequences are due to the discontinuancies in the signal (abrupt start and stop of each signal sequence). To envision why this is the case, consider that the short input sequences can be thought of as a continuous time signal multiplied with an rectangular window function:



This corresponds to convolution in the frequency domain with the Fourier transform of the rectangular pulse function. The transform has a Sin(x)/x shape, and as a consequence of this, the resulting frequency response of the filter has sidelobes. To smooth these sidelobes, the sequence is multiplied with a function that has a frequency domain characteristic suitable for removing sidelobes. The desired frequency characteristic could be something like:



A suitable characteristic will smooth the filter frequency response, preserving important information. This technique, called “windowing”, is used to treat short segments of a longer signal.

A.3. Parallel architectures

A.3.1 Why parallel solutions?

Many applications involve such an abundance of data, or such a high data-rate (as in the case of signal-processing for radar) that conventional serial computers simply can't process the data in reasonable time or keep up the pace. Many computational problems are too hard to be solved with a conventional one-processor architecture even for moderate sizes of input data.

A.3.2 A taxonomy of parallel architectures

Two control styles are possible, *centralized* where a single control unit dispatches instructions to each processing unit and *localized* where each unit works independently. The former is also referred to as *single instruction multiple data stream* (SIMD) and the latter *multiple instruction multiple data stream* (MIMD). In a SIMD parallel computer, the same instruction is executed synchronously at all processing units. If some operations need to be performed only when a criteria in the data is met, the processing units that tests false are turned off until the rest of the processors are done.

The need for communication between processors is solved in one of two ways. Either all processors share the same address-space (*shared-address-space computers)* or they pass messages to each other (*message-passing computers)*. Shared-address-space computers are divided into two categories, *uniform-memory-access computers* (UMA), where the time to access global data and data local to the processor is equal, and *nonuniform-memory-access computers* (NUMA).

Interconnection networks between memory units and processors can either be *static* or *dynamic*. Static networks consists of point-to-point links between the units, and dynamic networks switch the messages to simulate links.

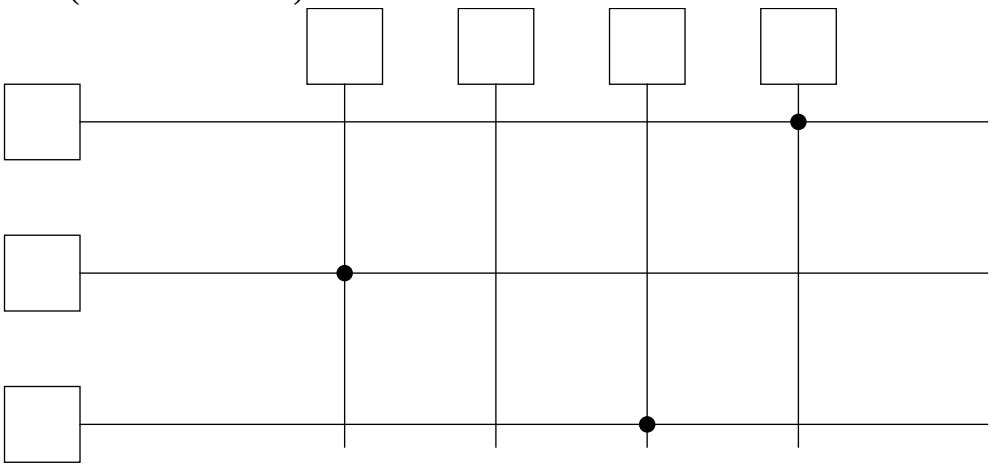
Processor granularity divides parallel architectures into two classes. If the number of processors is small and each processor is very powerful the resulting architecture is termed *coarse-grain*. On the opposite end of the spectrum lies the *fine-grain* architecture which consists of many small processors.

A.3.3 Interconnection networks

All data processing units need to communicate, both with each other and with possibly separate memory units. To do this, some facility for connecting units must be provided. Two main strategies exists: to have a *dynamic*ally reconfigurable network that adjusts the connections when one unit requests to communicate with another, and to have a *static* network consisting of fixed connections that can't be changed (possibly combined with routing). A routed static network architecture is often used for distributed signal-processing applications.

A.3.4 Dynamic interconnection networks

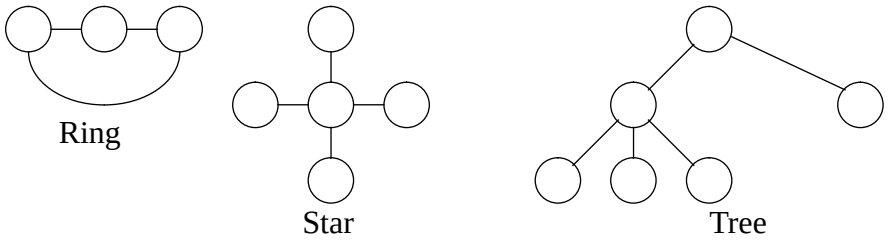
Simple types of dynamic interconnection networks include variations of the *crossbar* switching network, consisting of a matrix of switching elements used to connect units to each other (illustrated below).



Other common solutions are built around the idea of *busbased networks* where each processor sends messages on a shared bus, and *multistage interconnection networks.*

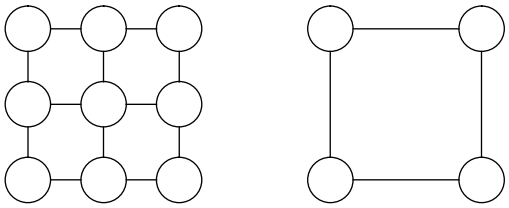
A.3.5 Static interconnection network concepts

The most basic types of static interconnection networks are the *completely-connected-network (*where every processor has a direct communication link with all others*), Ring and Star-connected Networks, Tree-connected-networks* and the more advanced *Mesh-connected* and *Hypercube-connected-networks*. Ring, Star and Tree-connected networks are simply processors connected in the way the names implies:



The two-dimensional mesh connection network is an extension of an linear network to two dimensions. Each (central) processor has a direct communication link to four other processors. This idea can be generalized to any number of dimensions. The hypercube is a multidimensional mesh of processors with exactly two processors in each dimension. Higher orders of hypercubes can be formed by connecting the corresponding processors of two lower order hypercubes (a zero-dimensional hypercube is defined to be a single processor).

A two-dimensional mesh and two-dimensional hypercube (composed of two one-dimensional hypercubes) looks like this:



A.3.6 Routing

Since the static networks restricts the number of processors able to communicate with a certain other processor, different means of communication than direct point-to-point message passing must be devised. When routing is used, messages are passed through possible intermediary processors before arriving to the sought destination. Since the delay the message suffers depends on the route taken, algorithms must be used to choose which way to send a message.

Routing mechanisms can be classified as either *minimal* or *non-minimal* and either *deterministic* or *adaptive*. A minimal routing mechanism always chooses the shortest way in contrast to the non-minimal that may choose a longer route to avoid congestion. Deterministic routing chooses a way based only on information on start and destination node, while adaptive routing takes into consideration information regarding the current state of the network.

A commercial architecture directed towards applications such as signal-processing, the RACEways architecture from Mercury Computer Systems, utilizes a tree-connected network of processors with internal message routing [IG95]. Parallel

A.4. Parallel computations

A number of properties of parallel computers needs to be understood by the designer when an algorithm must run efficiently on a parallel architecture, since a naive splitting into parts for distribution often generates a result that is computed less efficiently than a direct sequential implementation. For an more in-depth discussion of parallel computations, see [KGG+94].

A.4.1 Communication overhead

In order for a re-distribution of workload from one processor to two to be worthwhile, the communication overhead induced by the migration must be less than half the original execution time. If not, the overall execution time *increases* instead of decreases. This is one of the most important problems associated with parallel execution.

A.4.2 Location of data

If the access time for messages from local-store is different from the time needed to get an element of data from a remote storage, some processors in a network may spend a lot of

time just waiting for data to arrive. A smart parallel implementation keeps the most used data in a local store.

Deciding where to store data is often non-trivial.

A.4.3 Load balancing

In general, not all processors work at their full capacity. Some may be finished with their tasks early, some may have a lot of data to process. Balancing of the workloads may be a must if the processors have unequal burdens. If no balancing at all is done, many processors may soon be waiting for a few processors overflowed with work.

This distribution can be done once and for all or dynamically while the parallel algorithm is executed. High performance signal-processing programs can not rely on dynamic load-balancing, since run-time task assignment is very time consuming.

A.4.4 Granularity

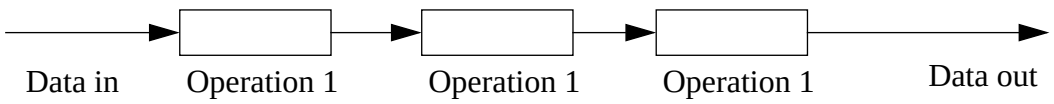
The issue of how to distribute work is dependent of the granularity of the problem. Would it be advantageous to split it into a few big chunks, or could computational power be saved by dividing it into many small? If the resulting granularity is high, the risk of clogging the system with messages between the tasks is high. A low granularity forces each processor to handle a heavier load, thus forcing the system architect to use more expensive processors.

A.4.5 Parallel algorithms

A.4.5.1 Algorithmic paradigms

A.4.5.1.1 Pipelined parallelism

When a sequence of operations need to be performed on a stream of data, each operation (possibly consisting of other, more fine-grained operations) may be applied by a different processor. Each time a new piece of data is accepted in one end of the pipe, the other emits a processed result. Pipelined systems are often fairly efficient, with a comparatively low development time. Unfortunately, few real-life signal-processing problems can be solved by a straight-forward pipeline architecture only.



A.4.5.1.2 Algorithmic parallelism

Since a program often is comprised by a number of independent functions that are applied to data, in each processing step some may be applied in parallel since they do not need the result of others. This is called algorithmic parallelism. Whether to exploit a possibility to perform computations in parallel or not depends on the factors mentioned above.

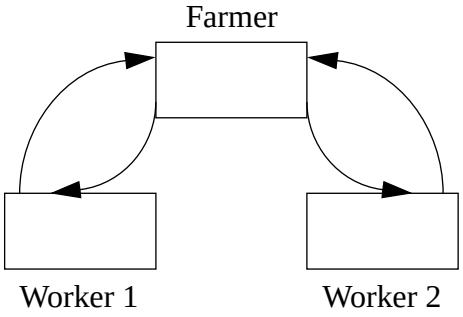
A.4.5.1.3 Geometric parallelism

Some problems easily map unto the architecture of the system. This is known as *geometric parallelism*. An example of this is operations performed on a two-dimensional image. A rectangular piece of data can be assigned to a each processor and all processors run the same program. If border information is needed, each unit receives a certain data-overlap.

If present, geometric parallelism is often easier to exploit efficiently than algorithmic parallelism, since no splitting of code is required.

A.4.5.1.4 Processor farms

Many problems can be statically divided into subtasks of an known complexity. This is not always the case, though. In a general case, no information on how to divide the problem may be available at compile time. The solution to this problem is to map the problem dynamically to a number of subprocessors, controlled by a master processor. This algorithmic paradigm is known as *processor farming*. Processors in control are referred to as farmers, and the subordinate processors as workers. The farmers distribute work among the workers, achieving load-balancing by only sending tasks to the least utilized workers. Processed tasks are sent back to the farmer, who combine the returned data to form some element of output. Processor farms needs a lot of message passing, so a good static decomposition of the computational problem is always preferable. Signal-processing programs are often statically dividable, but the decomposition requires a lot of work.



A.4.5.2 Static and dynamic algorithms

Algorithms can be divided into two categories depending on how they process data: static and dynamic. A static algorithm performs the same operations on the indata irrespective of their value. Thus the complexity of the algorithm is the same no matter what the indata may be. An example of this is a *static feed-forward process network*, where each unit does some processing and feeds the results forward. No loops or conditional datapaths are allowed. Almost all signal-processing problems are of the static feed-forward process network type

Dynamic algorithms uses the run-time data to determine what operations should be performed. This implies that scheduling might be needed, since the workload is unknown at

compile-time. If scheduling is employed, overhead is incurred as the price to pay for the run-time flexibility.

A.4.5.3 Converting sequential to parallel algorithms

Given a sequential algorithm, how do you convert it to a a form suitable for parallel execution? One possible way is to analyse the algorithm looking for steps that are taken sequentially, even if they could be taken in parallel. This could either be done automatically (as parallelizing FORTRAN compilers try to do), or by hand. To this date, no good automatic tool for performing this analysis has been devised, many opportunities for parallel computations induce more overhead than the distribution of workload makes worthwhile. In some cases, the algorithms may also be sequential by definition. A new algorithm must then be chosen or synthesized from the original if parallelism is to be exploited in a substantial way.

A.4.5.4 Annotations

Since (as stated above) no good way exists for dividing an algorithm automatically, the algorithm architect is forced to resort to some sort of annotations to aid tools in mapping to a form suitable for a multiprocessor machine. Many forms of notation have been devised, and more are in the making as this area is the scope of a lot of research.

Currently, the trend is going toward the use of *algorithmic skeletons*, which are abstractions of patterns of parallel computation (such as pipeline parallelism and processor farms). Ideally a programmer should be able to “plug” the logic of a certain algorithm into a pre-programmed skeleton, handling all processor communication and synchronization transparently. This methodology is unfortunately not mature.

A.5. Deforestation

For a more in-depth treatment of deforestation, see [Wad88] and [Gill96].

A.5.1 What is deforestation?

Deforestation is the somewhat witty term for removal of intermediary data-structures by transformations. Historically, first efforts was put into eliminating lists used to compute results but non-existent in the answer itself (“listlessness techniques”). This way of writing programs is a hallmark of functional programming and greatly improves understandability of code, at the cost of space-inefficient programs. Later extensions of the listlessness concept included removal of the more general tree-structures, which explains the term “deforestation”.

A.5.2 Why deforestation?

As stated above, one of the main reasons for the relative space-inefficiency of compiled functional programs is the many data-structures introduced by the programmer to store intermediary results. Simple functions can use up a lot of unnecessary heap space when written in the “listfull” style endorsed by functional languages. Simple changes in the definitions of functions could produce code that made less lavish use of the limited heap. To elaborate, consider the function:

```
sumOfDoubles::Int->Int->Int
sumOfDoubles start end =
                sum (map (*2) [start..end])
```

Clearly the list containing numbers in the interval [start..end] is unnecessary overhead in view of the fact that heap will be used for storage even though the list is non-existent in the final result. Another way of writing an equivalent function could be:

```
sumOfDoubles::Int->Int->Int
sumOfDoubles start end = sumOfDoubles' 0 start end
 where
 sumOfDoubles' sum start end
    | start == end = sum
    | otherwise = sumOfDoubles' (sum+2*start) (start+1) end
```

This equivalent function is tailrecursive, and thus space-efficient on the stack. Furthermore, this piece of code avoids building an intermediary list on the heap just for later consumption with the intent of collapsing the list again to form an element of output. Accordingly, this function would be preferable if minimum requirement of heap-space was the judgement criteria even at the penalty of decreased readability.

A good solution to these competing claims would be to encourage the programmer to write readable high-level-style definitions, and then produce intermediate code suitable for production of efficient executables through transformations. Otherwise a a great deal of expressive power would be lost by restricting the allowed coding style. This is the simple remedy for the seemingly contradictory demands of an expressive language and efficent code.

To summarize: *we want to write the functions in the listful style of the first example and then use some transformation to produce output in the listless style of the second example*. Compilation of the resulting code could the be done in the ordinary fashion. A classic example of eating the cake and having it left at the same time!

Why is then space-efficiency so especially important when one is concerned with compiling functional programs to signal-processors? Consider that the typical (treefull) functional program of a medium size requires a heap-depth in the order of some megabytes in order to evaluate. This should be compared to the size of primary memory on a fully expanded ordinary signal processor (something like 128K). Consequently the maximum depth of data required on the heap is a critical factor if we intend to write anything more than trivial programs.

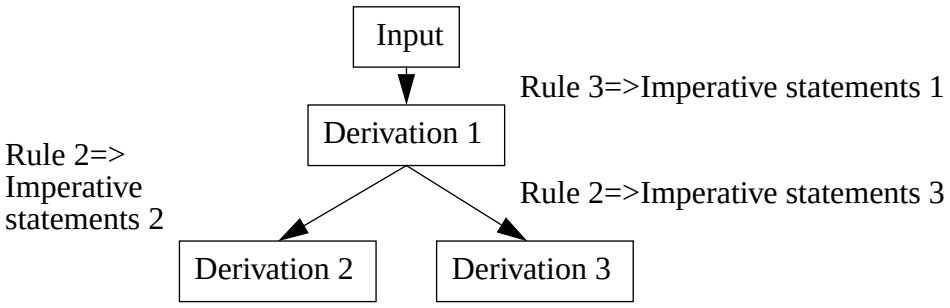
Operations that remove unnecessary datastructures are thus of great interest if we hope to compile functional code to signal-processors, especially if we want an relatively unrestricted set of programming constructs allowed in top-level specifications. If we took the alternative route of directly writing listless programs, we would have to spend a large amount of time reintroducing and removing intermediary data structures every time we made any modification of the definitions. This would be the sour grapes we would be required to consume in order to arrive at an optimal treeless output after each modification of specifications.

A.5.3 A general idea for an deforestational transformation

An automatic system for deforestation of functional programs could be a function of type **Functional_program-> ([Functional_program],[Imperative_statements])**. The function should take the functional program provided as input, and then produce a tuple of two

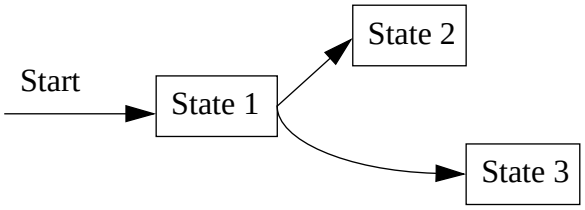
lists. Each element in the first list [D_1..D_n] is a **derivation** produced by the application of an appropriate transformation rule T_k to some previous derivation of the original program. The second list [S_1..S_n] contains groups of imperative statements that corresponds to the application of the transformation rules. Each group of statements is a **state** S.

Here is an example of how an deforestation could proceed:



The nodes in this graph contains derivations of previous programs. The arrows indicate the origin of each derivation. New derivations are produced by the application of transformation rules T. A rule-application can produce more than one derivation. The aim of the deforestation function is to generate all possible unique and fully simplified derivations D.

Each rule-application corresponds to a group of imperative statements, forming states. The states forms an finite automata, reading and writing elements and choosing outgoing arrows according to the imperative code in each state:



Arrows between the states in the state-graph are goto-statements corresponding to the arrows in the derivation graph.

The example starts by the application of transformational rule 3 to the input program, producing the derivation 1. This application corresponds to some imperative statements, who constitute state 1. Rule 2 is then applied to derivation 1, yielding derivation 2 and 3. Neither derivation 2, nor 3 proves to be transformable, so the deforestation is complete.

Some transformation sequences result in previous derivations ($T(D_k) = D_{k+1}$,$D_{k+1} == D_p$, $p<k$). It is important to make a link between the state corresponding to the source-derivation D_k and the state corresponding to D_p instead of generating the sequence again. This strategy of avoiding repeated sequences is termed **knot tying**, or **folding with respect to new functions**.

A.5.4 Possible requirements of a deforestational transformation

It goes without saying that the deforestation system must map functional programs equivalent imperative code, but this is not the only important demand one can place on such a transformation. As explained below, efficiency of output and the number of steps taken to arrive at an output can be other constraints on the problem specifications.

As in the case of all transformations, termination of the resulting programs must be verified even if the individual transformations themselves are legal if full correctness is an important

prerequisite. This might be very important in some cases, but considered a minor problem in others.

Deforestation transformations intended for incorporation in a compiler must be proved to terminate given any expression legal as input in a **finite** number of steps, since it would be inappropriate for a compiler to require an infinite amount of time to compile a finite program.

A last possible requirement of a deforestation transformation, is that the treeless programs should be at least as time-efficient as the trefull. The application of a transformation may, if applied without consideration, result in an inability by the compiler to optimize the resulting output. Steps must be carefully taken so as not to render standard optimizations impossible, if we are obliged to produce equally lean output in a time-complexity sense as the original program.

A.5.5 A simple informal example

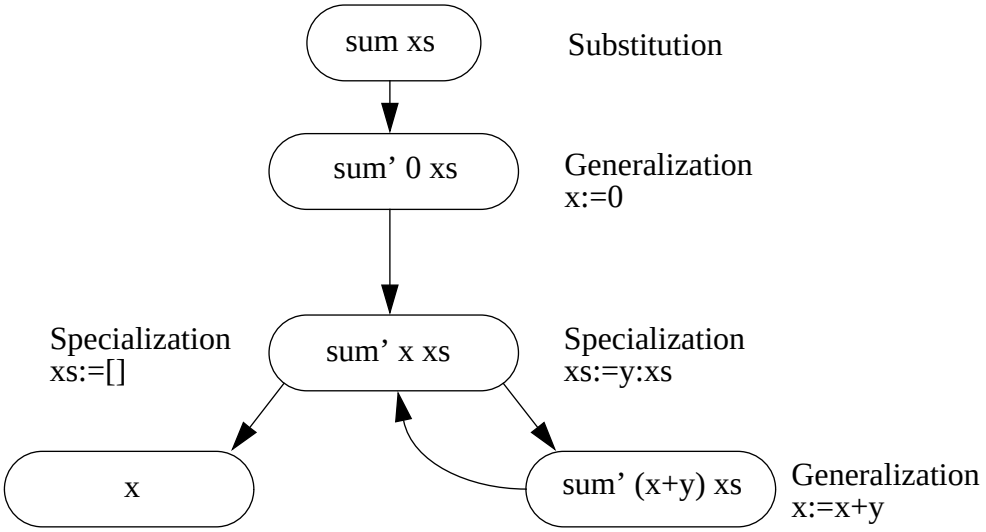
Consider the example of deforesting the expression "**sum xs**" where "**sum**" is given by the definitions in the figure. Our strategy can be expressed as follows:

- **Generalize** all constants or expressions involving linear combinations of constants and variables to new variables.
- **Substitute** functions with their definitions where necessary.
- **Specialize** expressions if it is necessary in order for the transformation to go further by generating different derivations for different cases.
- Whenever **knot-tying** is possible, do so.

Each state has a label. Generalization generates a new variable in the resulting output, and a subsequent assignment. Specialization corresponds to if-then-else clauses, and knot-tying to a goto-statement directed to another label. Whenever the input list is divided into a cons-cell and a rest, one "in?" statement in the output reads an element of input. Every time a derivation only contains a variable, an "out!" statement is generated. The transformation continues until no further derivations can be generated. Treeless input form is here defined to be full Haskell minus application of the **cons** operator. Legal output form is expressions formed from algebraic and equality operators, the control structure "if-then-else", constants, variables, labels and the four operators "**in?**", "**out!**", "**goto**" and "**:=**".

```
DEFINITIONS
sum xs = sum' 0 xs
sum' x [] = x
sum' x (y:ys) = sum' (x+y) ys
```

The following figure shows informally how the transformation proceeds, and the resulting states. Note the loop introduced by knot-tying.




```
L0:
goto L1
L1:
x:=0
goto L2
L2:
if eof() then goto L3
 else goto L4
L3:
out! x
stop
L4:
in? y
x:=x+y
goto L2
```

Appendix B. Program documentation

B.1. Program documentation of the signal-processing library

B.1.1 Functions that generate data

B.1.1.1 gen_lfm_pulse

Typical call:

gen_lfm_pulse::Int->Expr->Expr->Vector Expr

gen_lfm_pulse n_rs_pulse fs sweep_bw

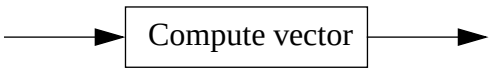
Arguments:

- n_rs_pulse: Number of samples in pulse
- fs : sampling frequency
- sweep_bw: sweep bandwidth

Returns:

A complex linearly frequency modulated pulse.

Functionality:



The vector is computed in a straight forward manner.

B.1.1.2 gen_noise

Typical call:

gen_noise::Int->Int->Expr->Int->Matrix Expr

gen_noise n_rs n_pri noise_level_db seed

Arguments:

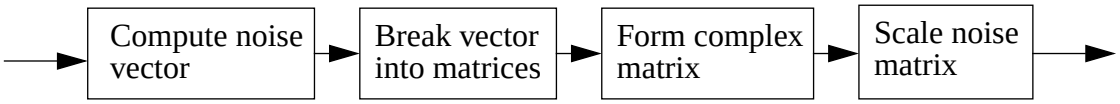
- n_rs: Number of range samples (number of rows in generated matrix)

- n_pri: Number of pulse repitition intervals in each integration interval (number of columns in generated matrix)
- noise_level_db: Noise level in decibel
- seed: Seed to feed the random number generator

Returns:

A complex matrix containing normal-distributed noise.

Functionality:



The complex noise matrix is computed by generating a noise vector from the seed provided, and then breaking this vector into two matrices (one for the real part, and one for the complex). These are then merged into one complex noise matrix, which is scaled to the proper maximum noise level.

B.1.1.3 gen_clutter

Typical call:

gen_clutter::Int->Int->Expr->Int->Matrix Expr

gen_clutter n_rs n_pri level_db seed

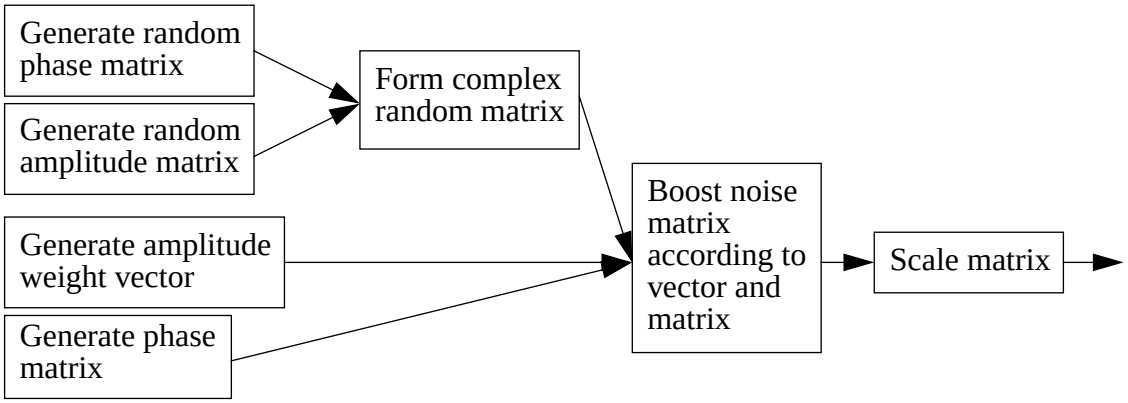
Arguments:

- n_rs: Number of range samples (number of rows in generated matrix)
- n_pri: Number of pulse repetition intervals in each integration interval (number of columns in generated matrix)
- level_db: Maximal clutter level in decibel
- seed: Seed to feed the random number generator

Returns:

A complex matrix containing random clutter with a certain maximal level.

Functionality:



First a random phase and amplitude matrix is formed from the provided seed. An amplitude vector and phase matrix is computed that will be used to make sure that the random matrix rows have large clutterlike amplitudes at the highest and lowest phase-shifted elements that falls off and rises linearly and has close to no amplitude for the rest of the elements. The resulting clutter matrix is then scaled to the desired maximum amplitude.

B.1.1.4 gen_target

Typical call:

gen_target::Vector Expr->Int->Expr->Expr->Matrix Expr

gen_target pulse n_pri amp shift

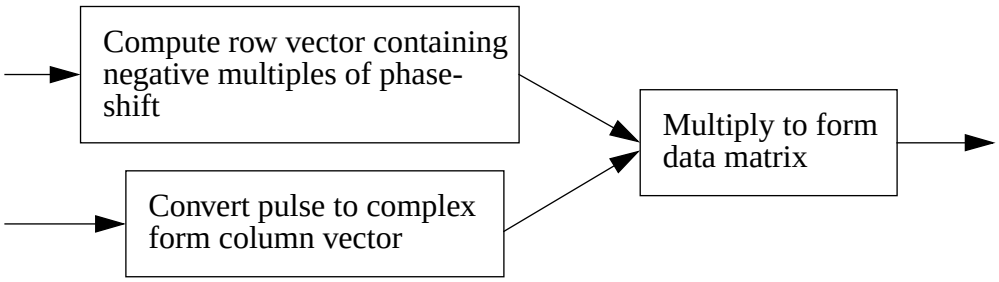
Arguments:

- pulse: a complex pulse from the transmitter
- n_pri: Number of pulse repitition intervals in each integration interval (number of columns in generated matrix)
- amp: Amplitude of target return
- shift: Frequency shift of return

Returns:

A complex matrix containing a target return with a certain amplitude and frequency shift generated by a specified transmitter pulse.

Functionality:



In order to produce a target return matrix, the function constructs a row vector with phase-shifts of [0,(-shift),-(2*shift)..] and multiplies this result with the column vector containing the sent pulse.

B.1.1.5 gen_targets

Typical call:

gen_targets::Int->Int->Vector Expr->Expr->Expr->Int->Int->[Expr]->[Expr]->

[Expr]->Matrix Expr

gen_targets n_rs n_pri send_pulse r_prf v_prf n_sample n_rs_min target_range

target_vel target_ampl

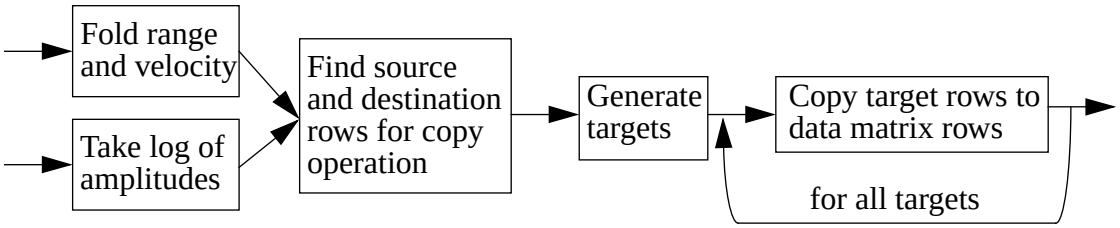
Arguments:

- n_rs: Number of range samples (number of rows in generated matrix)
- n_pri: Number of pulse repitition intervals in each integration interval (number of columns in generated matrix)
- send_pulse: A vector representing the pulse to be reflected
- r_prf: Unambigous range
- v_prf: Unambigous velocity
- n_sample: Number of samples between send-gen_targets::Vector Expr->Int->Expr->Expr->Matrix Exprpulses
- n_rs_min: Number of range samples before output starts
- target_range: List of target ranges
- target_vel: List of target velocities
- target_ampl: List of target amplitudes

Returns:

A complex matrix containing a target-returns from specified ranges with velocities and amplitudes specified in lists.

Functionality:



The function imposes a number of computed target returns on a blank matrix. In order to find out which rows of the generated targets to impose on which rows in the data-matrix, the ranges and the velocities are folded into the unambigous intervals. Based on this data together with the target amplitude and frequency shift, targets are generated and imposed on the data-matrix one at a time.

B.1.2 Filtering and related functions

B.1.2.1 hanning

Typical call:

hanning::Int->Vector Expr

hanning n

Arguments:

- n: Number of weightcoefficents to calculate

Returns:

A vector containing weight-coefficents specified by the hanning function.

Functionality:

This function simply computes the hanning function values for a vector of specified length.

B.1.2.2 do_weight

Typical call:

do_weight::Vector Expr->Matrix Expr->Matrix Expr

do_weight weight m

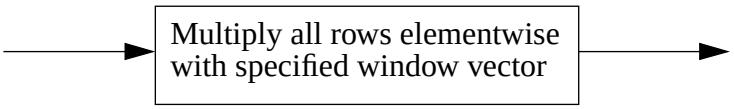
Arguments:

- weight: A weightvector to apply to all rows in m
- m: Complex matrix with the same number of columns as weight has elements

Returns:

A complex matrix with the rows weighted by the weigh-tvector.

Functionality:



All rows of the matrix is weighted by elementwise multiplication with the weigh-vector. The purpose of the weighting is to reduce the filtering sidelobes induced by using a rectangular window to cut out a short segment of a longer signal. A weight-vector such as the one generated by the hanning function reduces these side-lobes considerably (but reduces the amplitude of the main-lobe conversely).

B.1.2.3 vfft

Typical call:

vfft::Int->Vector Expr->Vector Expr

vfft num v

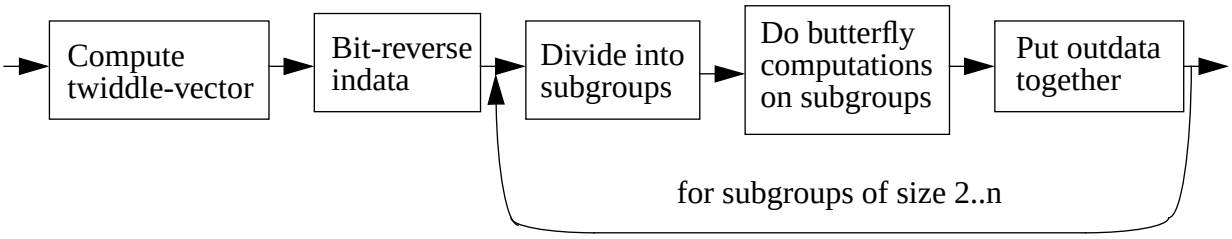
Arguments:

- num: Number of columns in m
- v: Complex vector

Returns:

A complex vector containing the FFT of *v*.

Functionality:



The vfft function computes the fft by progressing in steps. As the datavectormust be produced in the correct order, it is permuted by bitreversing. Dividing the data into groups of two and applying a butterfly computation produces output groups that are fused together to form a new vector. This is the input for another pass, which divides the data into groups of four. Progressing like this, the computation following the above pattern operates on larger and larger groups. When the calculation has been performed for a group containing the whole vector, the resulting fused vector is the FFT vector.

B.1.2.4 mfft_r

Typical call:

mfft_r::Int->Matrix Expr->Matrix Expr

mfft_r num m

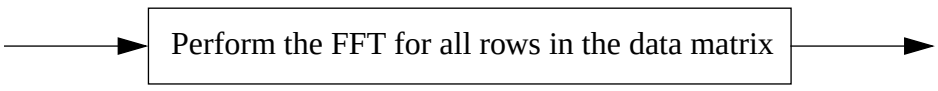
Arguments:

- num: Number of columns in m
- m: Complex matrix

Returns:

A complex matrix with the rows containing the FFT of the corresponding rows in *m*.

Functionality:



This function simply maps the FFT operation for vectors on all rows in a data matrix.

B.1.2.5 mfft_c

Typical call:

mfft_c::Int->Matrix Expr->Matrix Expr

mfft_c num m

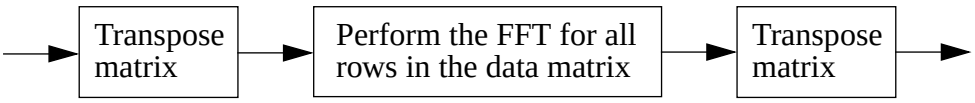
Arguments:

- num: Number of rows in m
- m: Complex matrix

Returns:

A complex matrix with the columns containing the FFT of the corresponding columns in *m*.

Functionality:



This function simply maps the FFT operation for vectors on all columns in a data matrix by transponating the matrix before and after mapping the FFT on the rows.

B.1.2.6 vifft

Typical call:

vifft::Int->Vector Expr->Vector Expr

vifft num m

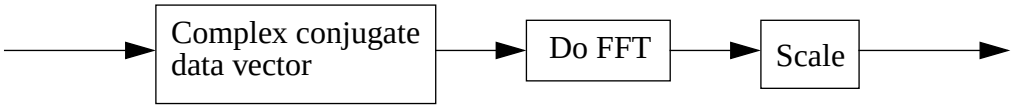
Arguments:

- num: Number of columns in m
- m: Complex vector

Returns:

A complex vector containing the IFFT of *v.*

Functionality:



As known from the theory of signal-processing, the inverse FFT of a vector is simply the scaled FFT of the complex conjugated data vector.

B.1.2.7 mifft_r

Typical call:

mifft_r::Int->Matrix Expr->Matrix Expr

mifft_r num m

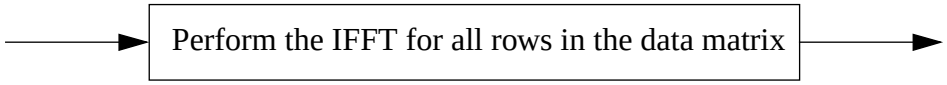
Arguments:

- num: Number of columns in m
- m: Complex matrix

Returns:

A complex matrix with the rows containing the IFFT of the corresponding rows in *m*.

Functionality:



This function simply maps the IFFT operation for vectors on all rows in a data matrix.

B.1.2.8 mifft_c

Typical call:

mifft_c::Int->Matrix Expr->Matrix Expr

mifft_c num m

Arguments:

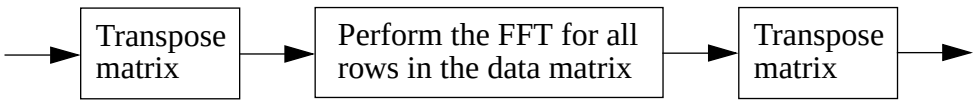
- num: Number of rows in m

- m: Complex matrix

Returns:

A complex matrix with the columns containing the IFFT of the corresponding columns in *m*.

Functionality:



This function simply maps the FFT operation for vectors on all columns in a data matrix by transponating the matrix before and after mapping the FFT on the rows.

B.1.2.9 firfilt

Typical call:

firfilt::Vector Expr->Vector Expr->Vector Expr

firfilt filter indata

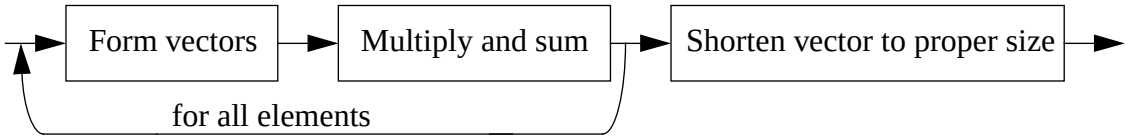
Arguments:

- filter: Vector specifying a FIR-filter
- indata: Indata

Returns:

A vector containing the filtered indata.

Functionality:



Convolution of the two vectors progresses as follows: for each element in the target vector, two new vectors is formed. These vectors are elementwise multiplied and the resulting vector is summed, forming the output element. Since the resulting output vector is too long, it is trimmed to the correct length.

B.1.2.10 matrix_fir_c

Typical call:

matrix_fir_c::Matrix Expr->Vector Expr->Matrix Expr

matrix_fir_c m v

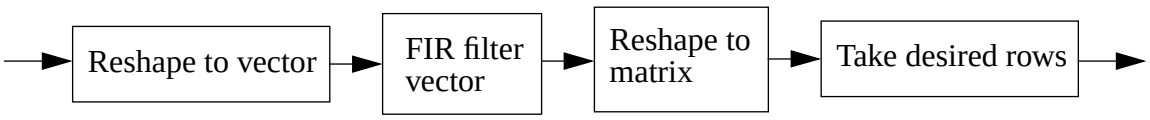
Arguments:

- m: Complex matrix
- v: Impulse response of an FIR-filter

Returns:

A complex matrix with the columns containing the corresponding columns of *m* filtered in the time-domain with the specified filter.

Functionality:



Conversion of the data matrix to a vector is the first step in computing the filtered columns of *m*. FIR filtering of this long vector by convolution with the coefficient vector *v* produces an output vector, of which only a part is interesting. Reshaping and trimming is therefore applied in order to produce the output matrix.

B.1.3 Pulsecompression functions

B.1.3.1 pcomp_time

Typical call:

pcomp_time::Matrix Expr->Vector Expr->Int->Matrix Expr

pcomp_time m pulse n_zero

Arguments:

- m: Complex matrix to be pulsecompressed
- pulse: Filter coefficients in time domain
- n_zero: How many zeros to pad each row before filtering

Returns:

A complex matrix pulsecompressed in the time-domain by linear filtering. This is advantagous for short filters.

Functionality:

This function pulse-compresses the filter by first padding the matrix to the correct dimensions, and then filtering the columns of the data matrix with the conjugated and reversed coefficients. The filtering is done with the **matrix_fir_c** function.

B.1.3.2 pcomp_freq

Typical call:

pcomp_freq::Matrix Expr->Vector Expr->Int->Matrix Expr

pcomp_freq m pulse n_zero

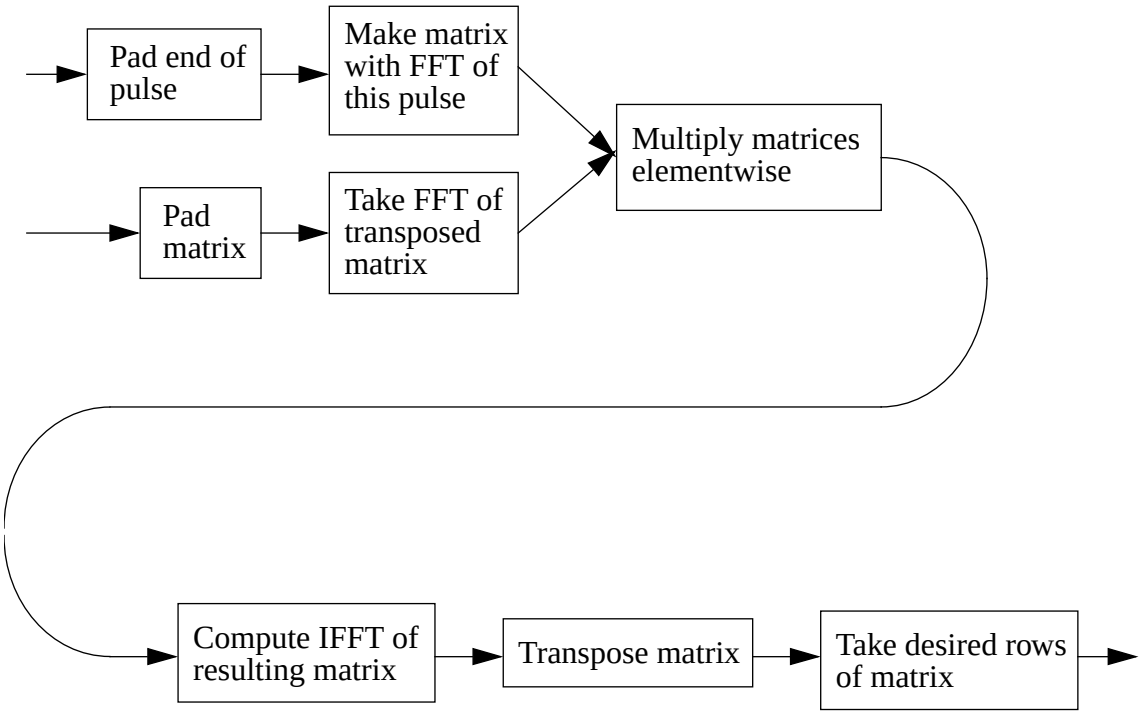
Arguments:

- m: Complex matrix to be pulsecompressed
- pulse: Filter coefficients in time domain
- n_zero: How many zeros to pad each row before filtering

Returns:

A complex matrix pulsecompressed in the frequency-domain by FFT/IFFT.

Functionality:



Pulsecompression in the frequency domain is accomplished by the use of the FFT/IFFT approach to linear filtering. Converting the matrix and pulse to suitable sizes by padding prepares for the subsequent filtering. The necessity of this operation stems from the fact that the Fast Fourier Transform is easier to compute for lengths of vectors that are powers of two. The filtering is performed by computing the Inverse Fast Fourier Transform of the elementwise multiplication of the FFT:s of the computed matrices. To convert the result back to the expected size, a certain trimming of the resulting matrix is a must.

B.1.4 CFAR detection functions

B.1.4.1 cfar_lmax

Typical call:

cfar_lmax::Matrix Expr->Expr->Matrix Bool

cfar_lmax m minimum

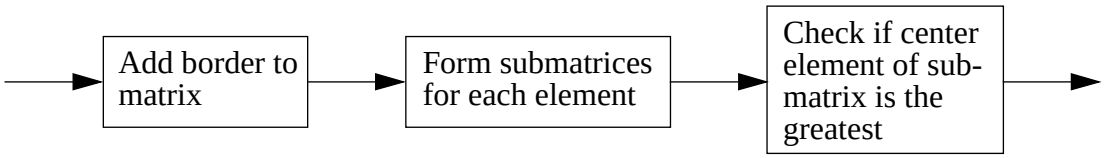
Arguments:

- m: Matrix indata
- minimum: value used when comparing with elements outside the matrix

Returns:

A matrix where each element containing **True** means that the corresponding element in the indata matrix is larger than all the surrounding values. When the comparisons are performed for the borders, the nonexistent elements are presumed to have a value of *minimum*.

Functionality:



First a “border” is added to the matrix by appending two new columns first and last, and two new rows first and last. These rows and columns simply consists of the *minimum* value provided to the function. New elements are formed by computing the complete 3*3 submatrices (as tuples of tuples). For all these submatrices, a boolean value of **True** or **False** is computed, based on whether the center element in the submatrix is the greatest of all the adjoining elements.

B.1.4.2 mean_columns

Typical call:

mean_columns::Int->Matrix Expr->Matrix Expr

mean_columns nr m

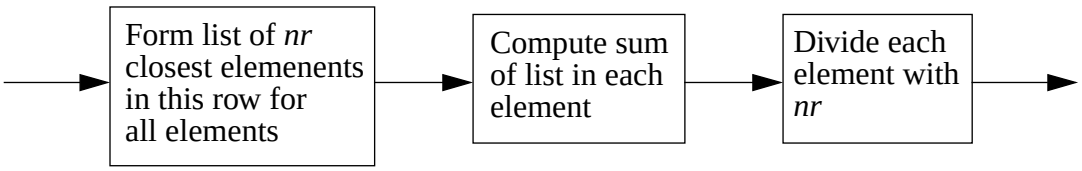
Arguments:

- nr: Number of doppler-channels to average over
- m: Matrix indata

Returns:

A matrix containing mean values of a number of adjacent doppler-channels (columns) for all range-bins. The mean value for each element is calculated on the following *nr* elements in the same row.

Functionality:



For each element in the matrix, a list of the nearest *nr* elements in the same row is produced. The sum of this list is then calculated, and divided by *nr* in order to produce a mean value for each element based on the adjacent elements in the column.

B.1.4.3 cfar_gof

Typical call:

cfar_gof::Int->Int->Int->Matrix Expr->Matrix Expr

cfar_gof n gf m

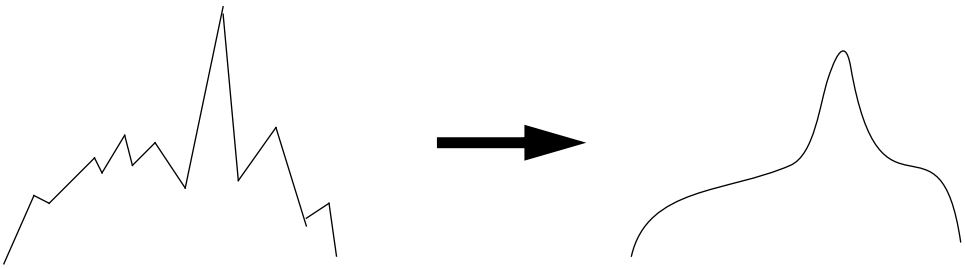
Arguments:

- n: Number of range bins to average over
- gf: Number of guard bins
- m: Matrix indata

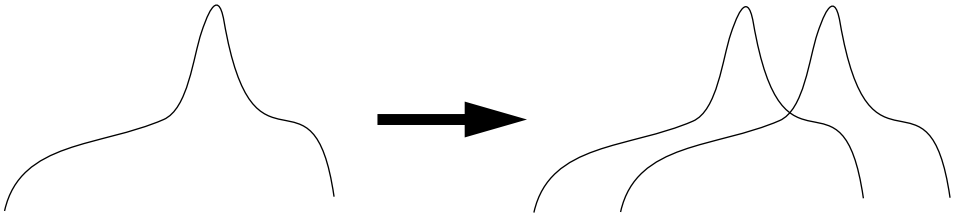
Returns:

A matrix with the greatest of an early and an late average of the values of the adjoining amplitudes in all frequency channels. The mean value is calculated on the next *nr* elements in an column. In order to be able to provide every element with an early and late average, the matrix is padded both in the beginning and the end with rows taken from itself. As a guiding line for which rows to pad with, the early and late averages of both the first and last row should be computed on the basis of the same adjoining rows in the same order.

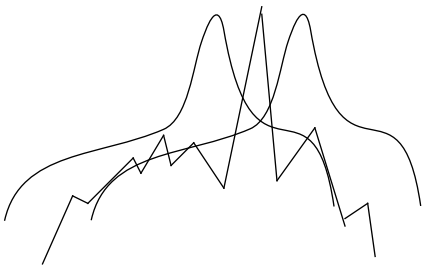
The objective of this function is to first smooth out each curve (where a curve is taken to mean the plot of the intensities in a doppler channel)



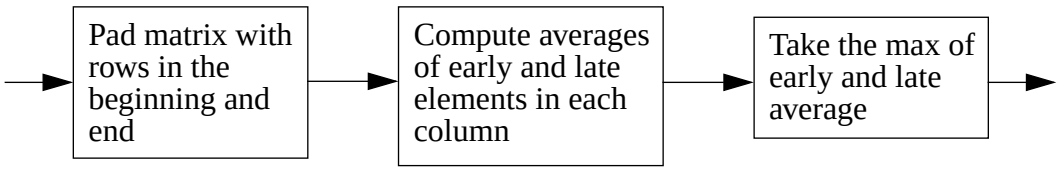
The next step is to lessen the sensitivity in the area near the peaks, by taking the envelope of a left-displaced smoothed curve and a right displaced smoothed curve. This is done to heighten the mean curve in the vicinity of peaks so that the sidelobes is suppressed from detection.



Now the peaks could be detected by comparing the original jagged curve to this GOF curve offset upward by a suitable amplitude. A target is detected at those places where the jagged original curve is greater than the envelope of the smoothed curves.



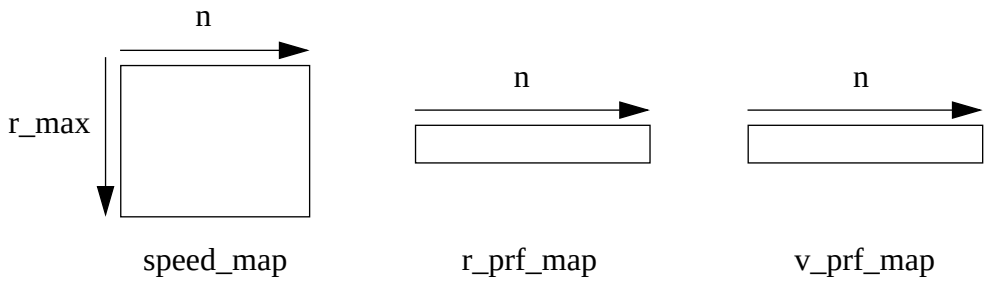
Functionality:



First the matrix is padded so that early and late averages can be computed for *all* rows. Each element in a new matrix is formed by making a tuple of the mean value of the early elements in the same column, and the late. The maximum element of this resulting tuple is then taken for all tuples in the matrix, producing the GOF-matrix.

B.1.5 Functions used for resolving ranges and velocities

State-monad internal datastructures

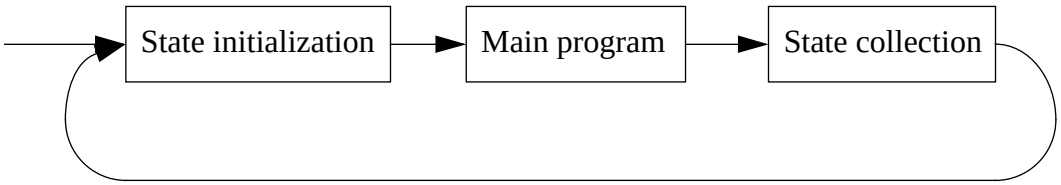


All the resolving routines use a state-monad that remembers prior target reflections. This is accomplished by storing the latest *n* r_prf and v_prf values in addition to the latest *n* out-folded target vectors in three internal datastructures (where a target vector is simply taken to mean a vector containing reflections in the proper range-bins).

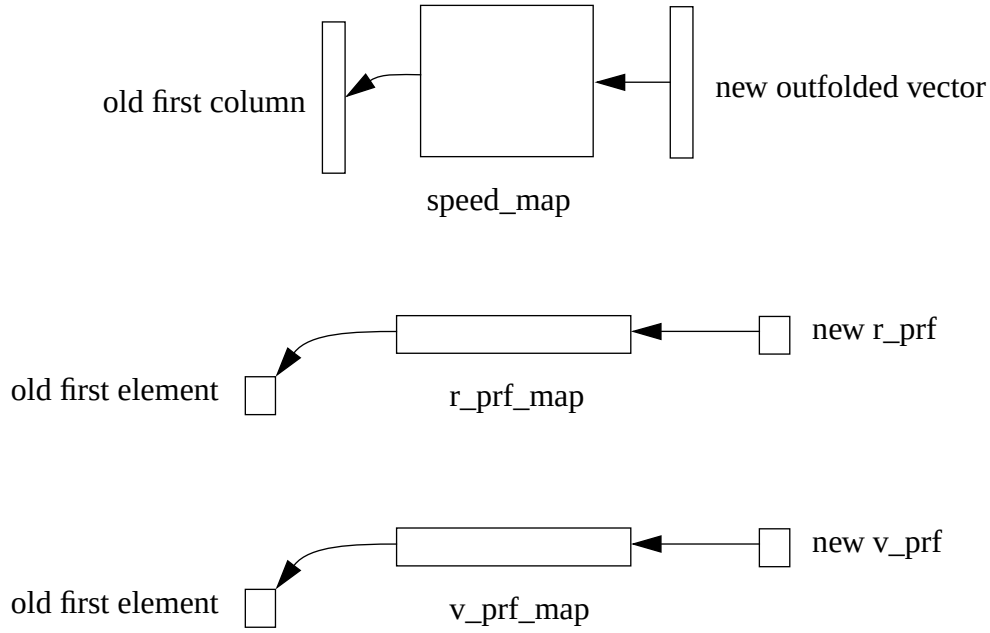
The dimension variable *n* corresponds to the *n* in the “m out of n” criteria and *r_max* to the maximum number of range-bins to fold out to. In addition to determining the longest range

and the maximum number of targets resolvable at any one time, the values of these parameters decide the computation time of a resolving pass.

When plumbed together, the resolve-routines will automatically update the internal data-structures in the necessary manner. Preservation of the internal state is vital for correct functioning of the routines should be guaranteed. This can be accomplished by collecting the new state after each resolve pass in the main program and subsequently initializing the monads prior to the next pass.



Updating of internal datastuctures



The resolving functions updates the internal data structures of the state-monads by removing the first element or column and adding this iterations values at the end of the data-structures. In this manner, the last *n* critical parameters is preserved internally and are at hand when the resolving are to be undertaken. Observe that prior to any resolving, the structures must be initialized to null if the resolving is to function properly.

B.1.5.1 resolv_init

Typical call:

resolv_init::Int->Int->S ResolvState ()

resolv_init n rmax

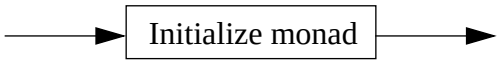
Arguments:

- n: N in the “m out of n” detection criteria
- r_max: Maximum range to fold out to

Returns:

A state-monad that remembers unresolved targets and returns nothing.

Functionality:



This function simply initializes the state by creating null-filled maps of the indicated dimensions.

B.1.5.2 resolv_range

Typical call:

resolv_range::Int->Vector Expr->S ResolvState [Int]

resolv_range m vect

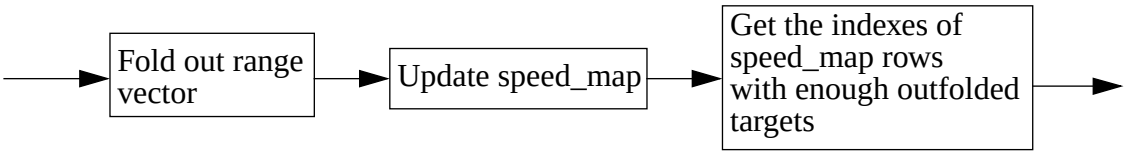
Arguments:

- m: m in the “m out of n” detection criteria
- vect: Row vector containing the target returns in the correct range-bins.

Returns:

A state-monad that remembers old, not resolved targets and returns resolved range-bins.

Functionality:



In order to update the speed_map (which contains the speed of the outfolded reflections in each rangebin), the target vector is outfolded by padding it with itself over and over again until the resulting vector is of equal length to the columns in the speed_map. This padded vector is then added to the end of the speed_map, and the first column is discarded.

To get the resolved hits, a list is formed containing the indexes of rows in the updated speed-map containing more than *m* non-zero elements. The range-bin numbers corresponds to the true range to the targets.

B.1.5.3 resolv_vel

Typical call:

resolv_vel::[Int]->Int->Expr->Expr->Expr->Expr->S ResolvState ([Int],[Expr])

resolv_vel r_det_index m v_prf v_min v_max v_win

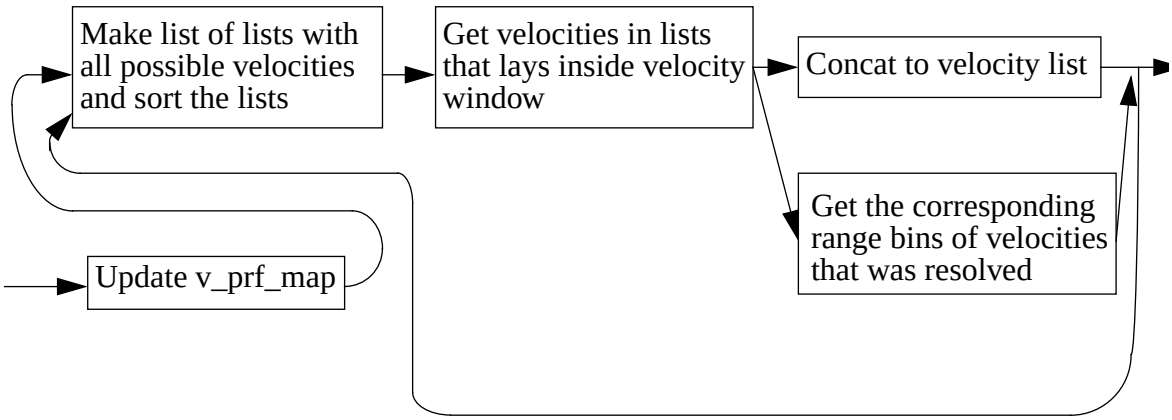
Arguments:

- r_det_index: List of range bins where targets has been discovered
- m: M in the “m out of n” detection criteria
- v_prf: Unambigous velocity limit
- v_min: Minimum detectable velocity
- v_max: Maximum detectable velocity
- v_win: The difference between two velocities necessary for them to be seen as different

Returns:

A state-monad that remembers old, unresolved targets, and returns a tuple of list of range bins and list of corresponding velocities.

Functionality:



First the v_prf-map is updated by adding the new v_prf at the end, and dropping the first element of the resulting vector. The new map, together with the range-bin numbers of range-resolved targets is the basis of the velocity-resolving algorithm.

For each non-zero element in a vector contained in a range-bin indicated by the target-list, a list of all the possible real velocities is produced and sorted. The true velocity is taken to be the mean of any *m* successive velocities that spans *v_win* or less. This velocity (if any) in conjunction with the range-bin number is stored in lists for each target range-bin and constitutes the output of the function.

B.1.5.4 resolv_cancel

Typical call:

resolv_cancel::[Int]->Expr->Expr->S ResolvState ()

resolv_cancel rv_det_index r_prf rs_res

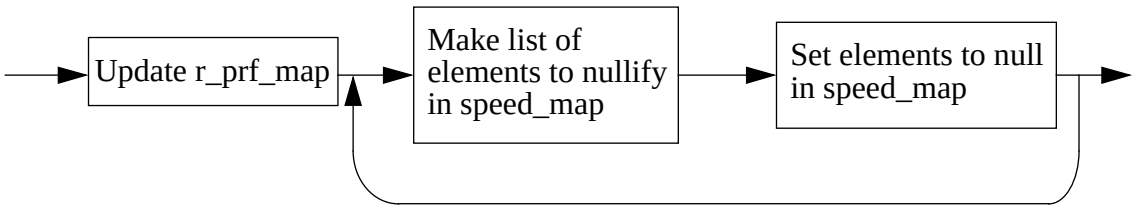
Arguments:

- rv_det_index: List of range bins where targets has been discovered
- r_prf: Unambigous range limit
- rs_res: Range resolution (one bin corresponds to rs_res meters)

Returns:

A state-monad that remembers old, unresolved targets with the indicated hits removed.

Functionality:



This function first updates the r_prf-map by adding the new r_prf last, and then removing the first element (the oldest r_prf). For each range-bin indicated by the list of fully resolved targets, a list is formed for each non-zero element containing all other target positions that would be folded to the same range-bin inside the unambigous range. The speed-map is then modified one row at a time by setting the corresponding elements to null if the current range-bin is in the list produced earlier. Iterating this procedure for each real target range-bin produces a speed-map where the only outfolded targets present are unresolved.

B.2. Program documentation of the radar simulation

B.2.1 Main program

Typical call:

go::IO ()

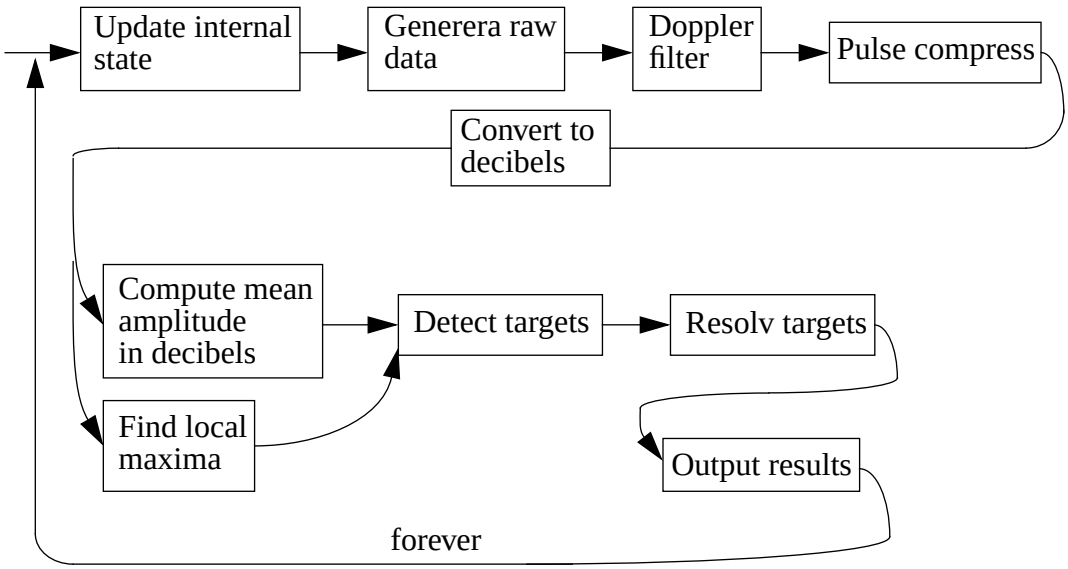
go

Arguments:

Returns:

An IO monad that outputs the result of the computation.

Functionality:



The main function proceeds in stages, where each stage ends with output of result of the current iteration and a recursive call to the function itself. All internal parameters are used to compute the new state of each iteration is stored in a **MainPrgState** structure. These are used to compute the variables in the **RadarVariables** structure. The next step is to generate simulated data from the receiver antenna. This data is passed through the doppler filter banks to separate the energy into frequency channels. To collect the energy with the aim of maximizing the returned signal energy, a matched filter pulse-compression scheme is used. The logartithm of the amplitudes in the resulting matrix is then taken, and mean values are computed. Local maxima computations is done to check where the locally greatest elements are, and this data together with the mean levels are used for detection. In order to determine the true range and velocity of the targets, resolving is performed. True range and speed is then written to the output, and the function concludes the iteration by calling itself.

B.2.2 Supporting functions

B.2.2.1 compute_variables

Typical call:

compute_variables::MainPrgState->Int->Int->RadarVariables

compute_variables state new_n_pri_select new_n_sample_select

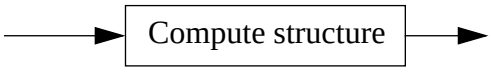
Arguments:

- state: state of the main program
- new_n_pri_select: integer number used to compute new n_pri
- new_n_sample_select: integer number used to compute new n_sample

Returns:

A RadarVariables structure containing variables to use for this iteration.

Functionality:



The structure is computed in a straight-forward manner.

B.2.2.2 gen_data

Typical call:

gen_data::[Int]->RadarVariables->Int->RadarVariables->(Matrix Expr,Matrix Expr,Vector Expr)

gen_data seedlist v

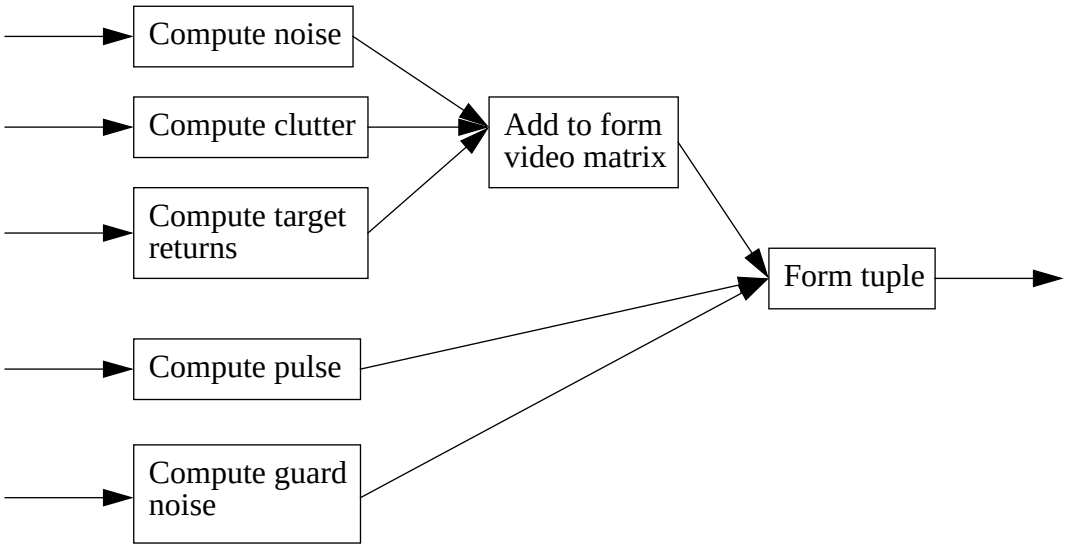
Arguments:

- seedlist: list of three seeds used in noise-generation
- v: RadarVariables structure containing assorted variables used in computations

Returns:

A 3-tuple containing video matrix, guard matrix and the sent pulse vector.

Functionality:



The video matrix is formed by adding noise, clutter and target returns together. A tuple is formed from this matrix,the computed pulse vector and guard noise matrix. Seedvalues used for the generation of noise is taken from the seedlist provided.

B.2.2.3 pulse_compress

Typical call:

pulse_compress send_pulse sum_video_d guard_video_d v

pulse_compress::Vector Expr->Matrix Expr->Matrix Expr->RadarVariables->(Matrix Expr,Matrix Expr)

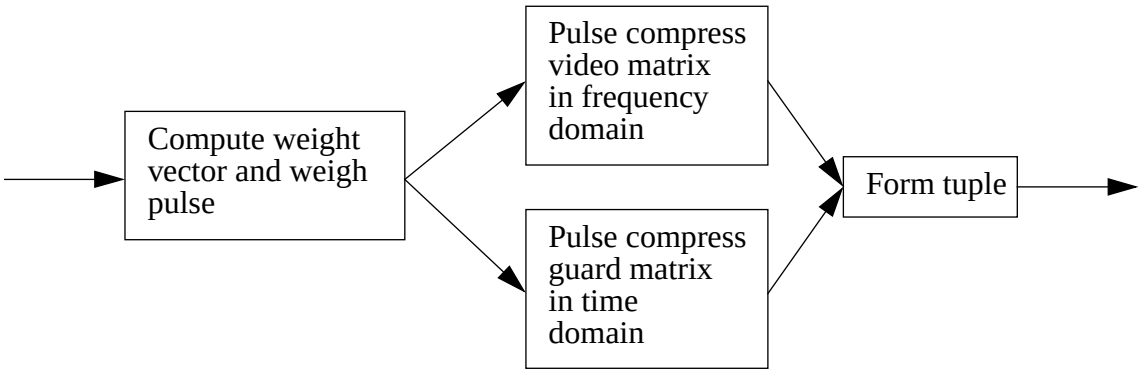
Arguments:

- send_pulse: pulse sent
- sum_video_d: doppler-filtered video matrix
- guard_video_d: doppler-filtered guard matrix
- v: RadarVariables structure containing assorted variables used in computations

Returns:

A tuple of the pulse-compressed video matrix and the pulse-compressed guard matrix.

Functionality:



The pulse that should be used for the pulse-compression is windowed by weighting with the Hanning function. This weighted pulse is used to pulse-compress the video matrix in the frequency domain (by the application of the FFT/IFFT). Traditional computation of the convolution sum is utilized for the filtering of the guard matrix. The resulting matrices are merged into a tuple and returned.

B.2.2.4 convert_db

Typical call:

convert_db sum_video_p guard_video_p v

convert_db::Matrix Expr->Matrix Expr->RadarVariables->(Matrix Expr,Matrix Expr)

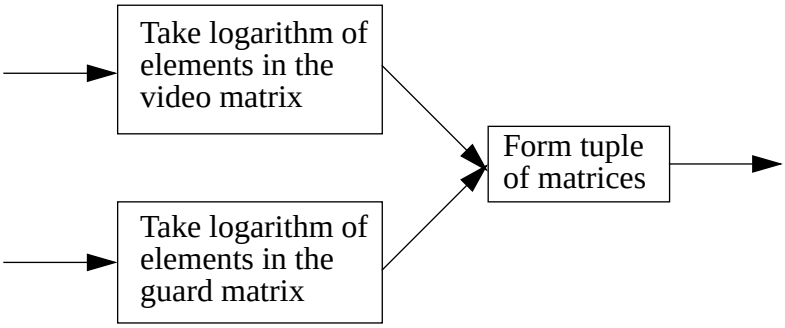
Arguments:

- sum_video_p: doppler-filtered and pulse-compressed video matrix
- guard_video_p: doppler-filtered and pulse-compressed guard matrix
- v: RadarVariables structure containing assorted variables used in computations

Returns:

A tuple of the video matrix and the guard matrix containing the logarithm of the elements .

Functionality:



The logarithm of all elements in both matrices is taken and a tuple is formed. This tuple is then returned.

B.2.2.5 mean_ampl_db

Typical call:

mean_ampl_db sum_db guard_db v

mean_ampl_db::Matrix Expr->Matrix Expr->RadarVariables->(Expr,Expr)

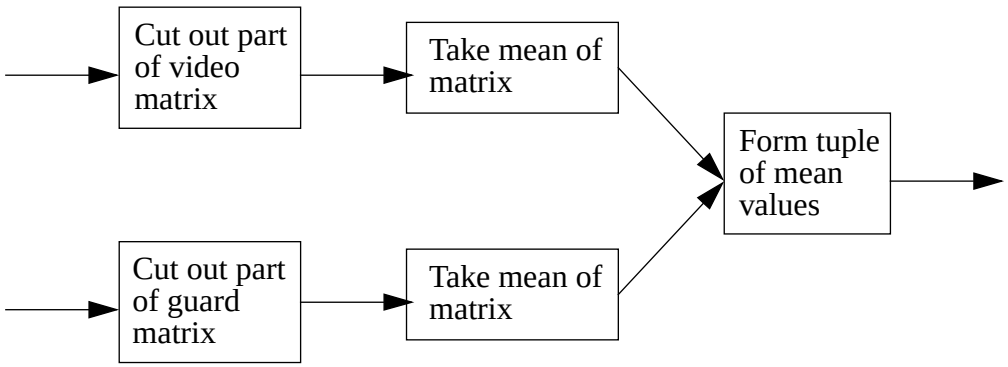
Arguments:

- sum_db: doppler-filtered, pulse-compressed and logarithmed video matrix
- guard_db: doppler-filtered,pulse-compressed and logarithmed guard matrix
- v: RadarVariables structure containing assorted variables used in computations

Returns:

A tuple containing the result of computing the mean values of a selected region of the two matrices, where the first is the result derived from the video matrix and the second the result from the guard matrix.

Functionality:



Parts of the matrices are discarded, and mean values are calculated for the remaining parts. A tuple is then formed from the resulting values.

B.2.2.6 get_local_max_mean

Typical call:

get_local_max_mean sum_db sum_noise_db

get_local_max_mean::Matrix Expr->Expr->(Matrix Bool,Matrix Expr,Matrix Expr)

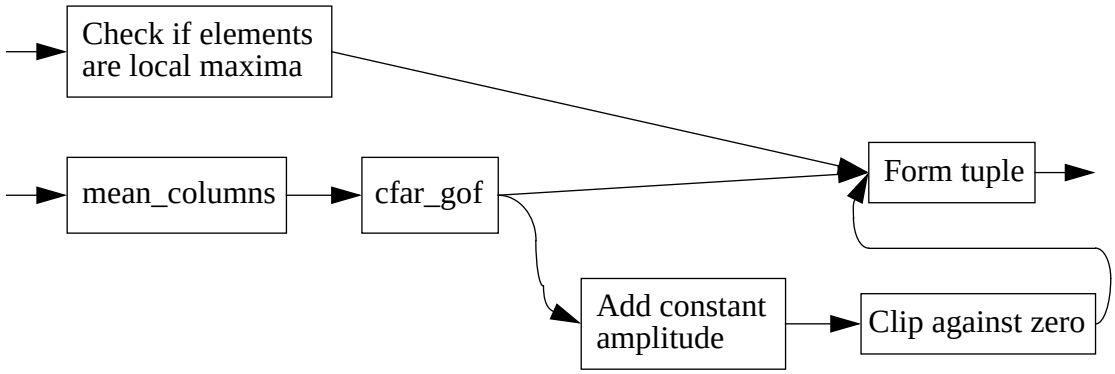
Arguments:

- sum_db: doppler-filtered, pulse-compressed and logarithmed video matrix
- sum_noise_db: mean value of a selected part of the matrix

Returns:

A three-tuple containing (*sum_lmax_det*,*sum_gof_db*,*sum_thr_db*) where *sum_lmax_det* is a boolean matrix containing **True** if the corresponding element is a local maxima, *sum_gof_db* is the result of applying the *mean_columns* and *cfar_gof* calculations to the in-data matrix and *sum_thr_db* is the result of adding a constant amplitude to all elements and clipping this value against zero.

Functionality:



The mean of the indata matrix is taken in the time and frequency direction, and the result of this is paired into a tuple together with the logical matrix containing **True** in the positions corresponding to local maxima and theclipped offset mean matrix.

B.2.2.7 detect

Typical call:

detect sum_db guard_noise_db guard_db sum_lmax_det sum_thr_db v

detect::Matrix Expr->Expr->Matrix Expr->Matrix Bool->Matrix Expr->RadarVariables->(([Int],[Int]),[Int],[Int],Int])

Arguments:

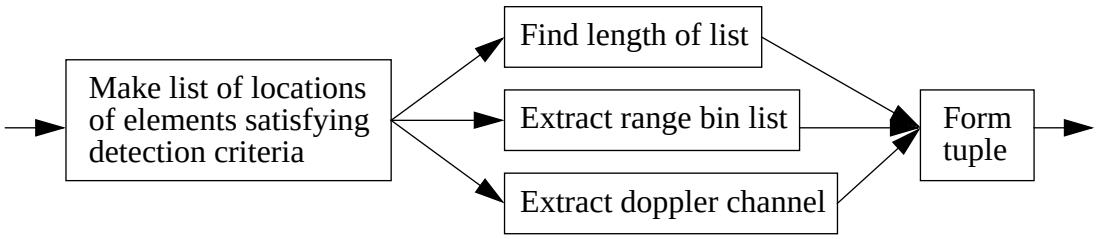
- sum_db: doppler-filtered, pulse-compressed and logarithmed video matrix
- guard_noise_db: mean value of a selected part of the guard matrix
- guard_db: doppler-filtered, pulse-compressed and logarithmed guard matrix
- sum_lmax_det: boolean matrix indicating if corresponding video element is local maxima
- sum_thr_db: local threshold matrix
- v: RadarVariables structure containing assorted variables used in computations

Returns:

A four-tuple containing (*det_list*,*det_dch*,*det_rb*,*n_det*) where

— *det_list* is a list of tuples containing a doppler channel and range bin where a target has been detected
— *det_dch* is a list of doppler-channels where targets reside
— *det_rb* is a list of range bins where targets reside
— *n_det* is the number of detected targets

Functionality:



A list consisting of tuples of doppler-channel number and range-bins of elements that satisfies the following criteria is computed:

- The corresponding element in *sum_db* should be larger than *sum_thr_db*
- The corresponding element in *guard_db* should not be larger than *(guard_noise_db+g_t0_db)*
- The corresponding element in *sum_lmax_det* should be **True** and located within *first_dch* to *last_dch*

The length of this list, a list of the doppler-channels, a list of the range-bins and the list itself is merged into a four-tuple.

B.2.2.8 resolv_targets

Typical call:

resolv_targets det_rb det_dch this_reslv_state v

resolv_targets::[Int]->[Int]->ResolvState->RadarVariables->([Expr],[Expr],[Int],ResolvState)

Arguments:

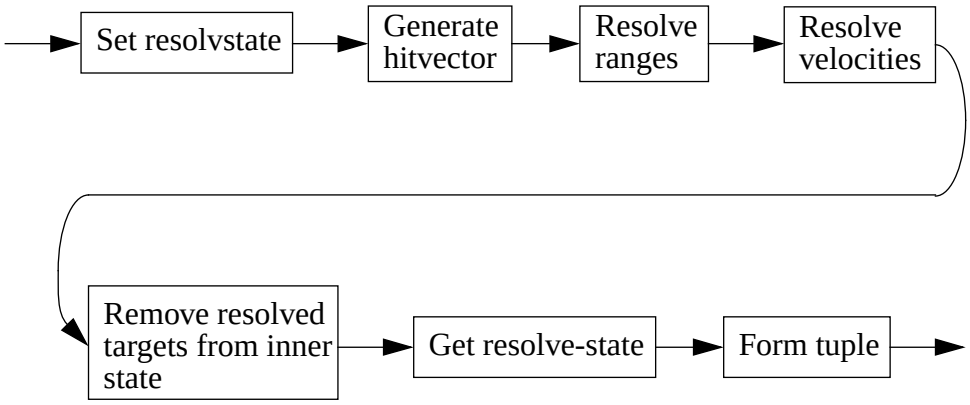
- det_rb: list of a number of range-bins where targets has been detected
- det_dch: list of the corresponing doppler-channels where targets has been detected
- this_resolv_state: the state of the resolving routines for this iteration
- v: RadarVariables structure containing assorted variables used in computations

Returns:

A four-tuple containing (*target_range,target_velocity,rv_ind_l,new_rslv_state*) where *target_range* is a list containing the resolved ranges, *target_velocity* is the corresponding re-

solved velocitys, *rv_ind_l* is the range indexes of the resolved targets and *new_rslv_state* is the resulting inner state of the resolving routines.

Functionality:



The inner state of the resolving routines are set as to rember all the previous unresolved targets. A “hitvector” is generated, which is a vector containing targets in the proper range-bins. This is the input to the range-resolving routine, which returns the resolved target ranges. Final steps is the velocity-resolving and the formation of the return tuple.

B.2.2.9 build_result

Typical call:

build_result n_det det_list target_range target_velocity rv_ind_l v

build_result::Int->[[Expr],[Expr]]->[Expr]->[Expr]->[Int]->RadarVariables->String

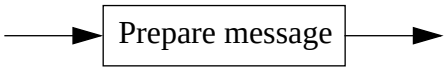
Arguments:

- n_det: number of detected targets
- det_list: tuple of target_range,target_velocity
- target_range: list of ranges to targets
- target_velocity: list of velocitys of targets
- rv_ind_l: range index numbers of targets
- v: RadarVariables structure containing assorted variables used in computations

Returns:

A prepared message containing iteration information and a list of target detections.

Functionality:



The message is simply prepared in an thoroughly uninteresting manner.

Appendix C. Program code

C.1. The library

```
--(C) Per Bjesse (master thesis in computer science)
--The Haskell signal processing library

module Signbeh where
import Signdata
import RandomHBC


--Some goodies that is good to have
infixl 9 >.>
infixl 9 >$>


(>.>) f g = g . f
(>$>) x f = f x

iter1::(a->b->b)->[a]->b->b
iter1 f [] val = val
iter1 f (x:xs) val = iter1 f xs (f x val)

iter2::(a->b->c->c)->[a]->[b]->c->c
iter2 f [] _ val = val
iter2 f (x:xs) (y:ys) val = iter2 f xs ys (f x y val)

iter3::(a->b->c->d->d)->[a]->[b]->[c]->d->d
iter3 f [] _ _ val = val
iter3 f (x:xs) (y:ys) (z:zs) val = iter3 f xs ys zs (f x y z val)
--End of assorted goodies


--     Start of fourier auxillary functions
nextLarger2Pot::Int->Int
nextLarger2Pot n = head $ dropWhile (<n) (iterate (*2) 2)

--Computes the phasefactors (Wk)expN
w::Expr->Expr->Expr
w (INum n) (INum k)  = form rew imw
       where
       rew = toFNum (cos (2*pi*fk/fn))
       imw = toFNum (- sin (2*pi*fk/fn))
       fk = fromInt k
       fn = fromInt n
w (Var v1) (Var v2) = CVar (“W” ++ v1 ++ v2)


--Computes twiddlevector
twidvect::Int->Vector Expr
twidvect n = vmap ((\k->w (INum n) k) . toINum) (vgen_f_t 0 (n-1))

symtwidvect::Int->Vector Expr
symtwidvect n = [CVar (“W(“++ (show x) ++ “,” ++  show n ++ “)”)
        | x<- [0..(n-1)] ]


--Computes FFT from complex vector and twiddle vector by radix-2 method:
--First reordering the 1-point FFT (the list as it is), then forming
--2-point FFT..2expM (=N)-point FFT
```

```
--Can take symbolic as well as numerical arguments
fft_twid::Int->Vector Expr->Vector Expr->Vector Expr
fft_twid n tw cxl =
 let
    combine2_FFT::Int->Int->Vector Expr->Vector Expr->Vector Expr
    combine2_FFT globn n twidvect cl = butterfly (div n 2) cl
           where
           butterfly num cl = upperpart ++ lowerpart
                  where
                  upperpart = apply3 upperfunc
                  lowerpart = apply3 lowerfunc
                  apply3 f = vmap3 f (vtake num cl) (vdrop num cl) twidvect
                  upperfunc = \a b w->a+w*b
                  lowerfunc = \a b w->a-w*b


    --Combines a stage of lower order fft:s to form higher order
    combineN_FFT::Int->Int->Vector Expr->Vector Expr->Vector Expr
    combineN_FFT n pts tw=concat.(map (combine2_FFT n pts stagetw)).(vsplit_l pts)
            where
           stagetw = map (\k->tw!!(k*level)) [0..((div pts 2)-1)]
           level = div n pts

 in
    iter1 (\pts->combineN_FFT n pts tw) (takeWhile (<=n) (iterate (*2) 2))
              (vbitreverse cxl)

--Computes IFFT from complex vector and twiddle vectorby radix-2 method:
ifft_twid::Int->Vector Expr->Vector Expr->Vector Expr
ifft_twid n tw cxl@((CNum _):_) = vmap (*scalef) (fft_twid n conjtw cxl)
       where
       conjtw = vmap (\k->conj k) tw
        scalef = CNum (1/(fromInt n),0)


--Frontends used to separate symbolic datatype

vfft::Int->Vector Expr->Vector Expr
vfft n cxl@((CNum _):_) = fft_twid n (twidvect n) cxl
vfft n cxl@((CVar _):_) = fft_twid n (symtwidvect n) cxl

vifft::Int->Vector Expr->Vector Expr
vifft n cxl@((CNum _):_) = ifft_twid n (twidvect n) cxl
vifft n cxl@((CVar _):_) = ifft_twid n (symtwidvect n) cxl

mfft_c,mfft_r::Int->Matrix Expr->Matrix Expr --On columns,row
mfft_c n = mtransp . (mfft_r n) . mtransp
mfft_r n = mmap_r (vfft n)

mifft_c,mifft_r::Int->Matrix Expr->Matrix Expr --On columns,row
mifft_c n = mtransp . (mifft_r n) . mtransp
mifft_r n = vmap (vifft n)
--END of fourier functions




--The generate functions are written naively, since they aren't meant to be
--compiled anyway
```

```
--***********************gen_lfm_pulse************************************
--Generates a lfm pulse of desired length specified bandwidth coverage

--n_rs_pulse = number of samples in desired pulse
--fs = sampling frequency of system
--sweep_bw = bandwidth to sweep over
--result: A linearly frequency modulated pulse in the frequency domain

gen_lfm_pulse::Int->Expr->Expr->Vector Expr
gen_lfm_pulse n_rs_pulse fs sweep_bw = [gen_f k| k<-[1..n_rs_pulse]]
   where

   -- br = sweep bandwidth relative to sampling frequency)
   br = fromFNum (sweep_bw/fs)

   -- generate frequency lfm chirp
   gen_f k = exp (form (toFNum 0) (im k))
   im k= toFNum (-2*pi*br*((fromInt k)-
               (fromInt n_rs_pulse)/2-0.5)^2/(2*(fromInt n_rs_pulse)))

--************************End of gen_lfm_pulse*****************************


--************************gen_noise****************************************
--Generates a matrix filled with normal distributed noise

--n_rs = number of rangesamples to generate (number of rows in matrix)
--n_pri = number of pulse repetition intervals (number of columns in matrix)
--result: A matrix of pulses filled with noise

gen_noise::Int->Int->Expr->Int->Matrix Expr
gen_noise n_rs n_pri noise_level_db seed= noisemat
   where

   --Scale factor in db:s to get noise with the specified level
   scale = toCNum $ 10**((fromFNum noise_level_db)/20)/(sqrt 2)

   --Form a complex random matrix by breaking up normaldistributed random lists
   noisemat =
          map (toFNum . fromDouble)
                 (
                   take (2*n_rs*n_pri)
                        (normalRandomDoubles seed seed)
                 ) >$>
           \norml->mgen_break_list n_rs n_pri norml >$>
            \remat->mgen_break_list n_rs n_pri
                        (
                          drop (n_rs*n_pri) norml
                        ) >$>
                 \immat->mmap2 (\r i -> form r i) remat immat >$>
                  \mat->mmap (*scale) mat
--************************End of gen_noise*********************************


--************************gen_clutter**************************************
--Generates a matrix filled with clutter (noise with low frequencies)

--n_rs = number of rangesamples to generate (number of rows in matrix)
--n_pri = number of pulse repetition intervals (number of columns in matrix)
```

```
--level_db = maximum clutter level
--seed = seed for the pseudo-random number generator
--result: A matrix filled with clutter

gen_clutter::Int->Int->Expr->Int->Matrix Expr
gen_clutter n_rs n_pri level_db seed = clutter_matrix
 where
 clutter_matrix =
    vmap (\k->toCNum $ 10**(-1.25/81*(k*2048)^2)) where_vect >$>
     \ampl->mmap (\k->exp (k * (form (toFNum 0) (toFNum 1))))
                    (
                      vrcmult (ltovect [(toCNum.fromInt) e |e<-[1..n_pri]])
                              (vmap (\e-> toCNum $ 2*pi*e) where_vect)
                    ) >$>
        \freq-> mmap_r (\v->mult_rv_m v freq)
                         (
                            mmap_r (vmult_elem ampl) randmat
                         ) >$>
           \unscaled_matrix-> let
                                scale = toCNum $  10**((fromFNum level_db)/20)
                              in
                                mmap (*scale) unscaled_matrix

 --Vector specifying where clutter is distributed
 where_vect = ltovect $ map (*0.005) [-1,-0.8..1]
 n = vsize where_vect

 --Generate complex random matrix with random amplitude and phase
 randmat =
    map (toFNum.fromDouble)
            (
              take (2*n_rs*n)  (randomDoubles seed seed)
            ) >$>
     \randl->mgen_break_list n_rs n randl >$>
      \mat1->mgen_break_list n_rs n (drop (n_rs*n) randl) >$>
       \mat2->let
                makec = \f1 f2 ->
                            (
                              (toCNum.fromFNum) (sqrt (-(toFNum 2)*(log f1)))
                            )
                               * exp (
                                      form (toFNum 0) ((toFNum (2*pi))*f2)
                                     )

              in
                mmap2 makec mat1 mat2


--***********************End of gen_clutter*******************************


--***********************gen_target*************************************
--Generates a (sub)matrix containing a reflection against a target

--pulse = the pulse sent by the transmitter
--n_pri = number of pulse repetition intervals (number of columns in matrix)
--amp = the amplitude of the reflected signal
--shift = doppler frequency shift induced by moving target
--returns: a  (sub)matrix containing the reflection of the pulse against the
--         moving target
```

```
gen_target::Vector Expr->Int->Expr->Expr->Matrix Expr
gen_target pulse n_pri amp shift = the_target
 where
 the_target =
   let
      phasedata = vmap (*(exp (form (toFNum 0) (-shift))))
                       (
                         vgen_n_val (n_pri-1) (toCNum 1)
                       )
   in
      vcumprod (vadd_e_first (toCNum 1) phasedata) >$>
       \phases->vrcmult phases
                       (
                         vmap (*(form amp (toFNum 0))) pulse
                       )


--***********************End of gen_target*********************************


--***********************gen_targets***************************************
--Generates a whole matrix with the reflections from the targets

--n_rs = number of rangesamples to generate (number of rows in matrix)
--n_pri = number of pulse repetition intervals (number of columns in matrix)
--send_pulse = the transmitted pulse
--r_prf = unamibigous range
--v_prf = unambigous velocity
--n_sample = number of samples between sendpulses
--n_rs_min = distance in range samples that the first row of output corresponds to
--target_range = list of distances to the targets
--target_vel = list of target velocities
--target_ampl = list of amplitudes of reflections from the targets
--returns: a matrix containing target reflections

gen_targets::Int->Int->Vector Expr->Expr->Expr->Int->Int->[Expr]->[Expr]->
  [Expr]->Matrix Expr
gen_targets n_rs n_pri send_pulse r_prf v_prf n_sample n_rs_min target_range
                 target_vel target_ampl = the_targets
  where

   the_targets =
     mgen_val n_rs n_pri (toCNum 0) >$>
      \zerom->vmap (\k->
                       (frem k v_prf)*(toFNum (2*pi))/v_prf
                )
                   target_vel >$>
       \shift->vmap (\k->
                        (toFNum 10)**(k/(toFNum 20))
                    )
                    target_ampl >$>
         \amp->vmap (\k->
                         round ((frem k r_prf)/rs_res) - n_rs_min
                    )
                     target_range >$>
            \rs->map (\k->[k..(k+n_rs_pulse-1)]) rs >$>
             \rs_list->map (\k->[x | x<-k, x>=0, x<n_rs]) rs_list >$>
              \dest_parts->zipWith (\rsi rsprt->
                                    map (\k->k-rsi) rsprt
```

```
                                  )
                                    rs dest_parts >$>
                  \source_parts->
                        zipWith (\ampl frshift->
                                   gen_target send_pulse n_pri ampl frshift
                                 )
                                   amp shift >$>

                       \targets-> iter3 (\k l m->imposeTarget k l m)
                                          dest_parts source_parts targets zerom


  --Compute sizes and conversion factors
   rs_res = r_prf/(toFNum $ fromInt n_sample)
   n_target = vsize target_range
   n_rs_pulse = length send_pulse


--Imposes a target unto s matrix by changing specified rows
imposeTarget::[Int]->[Int]->Matrix Expr->Matrix Expr->Matrix Expr
imposeTarget drws srws tdata m =
        iter2 (\drw srw -> mmod_f_row (vadd (mrow srw tdata)) drw) drws srws m
--**********************End of gen_targets*******************************




--**********************clip*********************************************
--Clips the input matrix against specified high and low limits

--input = input matrix
--low = lower limit
--high = higher limit
--returns: a clipped matrix

clip::Matrix Expr->Expr->Expr->Matrix Expr
clip input low high =
  mmap
    (\k->
      if k<low then
        low
      else
        (
         if k>high then
            high
         else
            k
        )
    ) input
--**********************End of clip**************************************



--**********************Random number testfkts from **********************
mean v = sum v/(fromInt $ length v)
variance v = (1/(fromInt (n-1)))*(sum (map (\x->(x-m)^2) v))
  where
  n = vsize v
  m = mean v
```

```
--***********************End of random number generator*******************



--***********************Hanning routine*********************************
--Computes a weight vector with hanning characteristic

--n = length of weight vector
--returns: the wieght coefficients of the hanning window

hanning::Int->Vector Expr
hanning n = vmap (toCSpec . (*0.5).(1-).cos.(/divf).(*(2*pi)).fromInt)
  (vgen_f_t 1 n)
      where
      divf = fromInt n + 1.0
      toCSpec = \x->CNum (x,0)


make_window n = vmap (\k->k/vnrm) vect
 where
 vect = hanning n
 vnrm = form (vnorm vect) (toFNum 0)

--***********************End of Hanning routine****************************


--***********************Do_weight routine*******************************
--Weight the rows of the data matrix with the specified window coefficients

--weight = weight vector containing the window coefficients
--m = a matrix on which to perform weighting on rows
--returns: a weighted matrix
--REMARK: Weightcoefficients is a row vector instead of columnvector


do_weight::Vector Expr->Matrix Expr->Matrix Expr
do_weight weight m = mmap_r (vmult_elem weight) m
--***********************End of Do_weight routine**************************


--***********************Pulse_compress routine time***********************
--Pulse compresses columns by filtering in the time domain (advantageous for short
--inverse filters)

--m = a matrix to pulse compress
--pulse = filtercoefficients to use for the inverse filter
--n_zero = number of zeros to add in front of each column before filtering
--returns: a pulsecompressed matrix

pcomp_time::Matrix Expr->Vector Expr->Int->Matrix Expr
pcomp_time m pulse n_zero = compressed_matrix
   where
   compressed_matrix =
     (mpad_b_c n_zero . mpad_e_c (filtsize-(n_zero+n_rs))) m >$>
       \padded_m->(vmap conj . vreverse) pulse >$>
        \filter_coeff->matrix_fir_c padded_m filter_coeff

   (n_rs,n_pri) = msize m
   n_pulse = vsize pulse
   filtsize = n_rs+n_pulse-1
```

```
--***********************End of pulse compress routine time*****************


--***********************Matrix fir for columns routine*********************
--FIR-filter the columns of a matrix, with the coefficients provided

--m= matrix to filter the columns of
--v= vector containing filter coefficients
--returns: A filtered matrix

matrix_fir_c::Matrix Expr->Vector Expr->Matrix Expr
matrix_fir_c m v = filtered_matrix
      where

      filtered_matrix =
         firfilt v (mtovect $ mreshape 1 m) >$>
          \filtered_seq->mreshape n_rs (vtomatr filtered_seq) >$>
           \mprim->mtake_first_r (n_rs-n_pulse+1)
                              (
                                mdrop_first_r (n_pulse-1) mprim
                              )
      (n_rs,n_pri) = msize m
      n_pulse = vsize v
--***********************Matrix fir for rows routine************************


--***********************firfilt*******************************************
--FIR-filters a vector with the specified coefficients

--b= coefficients of FIR-filter (b0..bn)
--x= invector to be filtered
--returns: a filtered vector
firfilt::Vector Expr->Vector Expr->Vector Expr
firfilt b x = shortened_convolved_seq
  where
  shortened_convolved_seq =
     map (\k->dosum $ vmult_elem (take k b) (indata k x)) [1..num] >$>
       \filtseq->vtake (vsize x) filtseq
   where
   indata k x = reverse $ take k (vpad_e (numb-1) x)
   dosum x = foldr (+) (toCNum 0) x
  num = numb + numx - 1
  numb = vsize b
  numx = vsize x
--***********************End of firfilt************************************


--***********************Pulse_compress routine freq***********************
--Pulse compresses columns by filtering in the frequency domain

--m = a matrix to pulse compress
--pulse = filtercoefficients to use for the inverse filter
--n_zero = number of zeros to add in front of each column before filtering
--returns: a pulsecompressed matrix

pcomp_freq::Matrix Expr->Vector Expr->Int->Matrix Expr
pcomp_freq m pulse n_zero = compressed_matrix
      where
      compressed_matrix =
         vpad_e (n_fft-n_pulse) pulse >$>
```

```
          \pad_p->mpad_e_r (n_fft-(n_rs + n_zero))
                                (
                                  mpad_b_r n_zero (mtransp m)
                                ) >$>
            \pad_m->mgen_n_v n_pri
                                 (
                                  vmap conj (vfft n_fft pad_p)
                                 ) >$>
               \fft_pulse_matrix->
                   let
                     elemult m1 m2 = mmap2 (\x y->x*y) m1 m2
                   in
                     mifft_r n_fft (
                                    elemult (mfft_r n_fft pad_m) fft_pulse_matrix
                                   ) >$>
                    \big_filtered_m->mtake_first_r n_rs (mtransp big_filtered_m)
      --Compute the length of matrix,pulse and size of FFT
      (n_rs,n_pri) = msize m
      n_pulse = length pulse
      n_fft = nextLarger2Pot (n_rs + n_zero + n_pulse)

--************************End of pulse compress routine freq*****************


--************************CFAR local maximum routine*************************
--Indicates if matrix elements is the local max in the adjoining region

--mat = matrix to search for local maxima in
--minimum = value of elements in “virtual border” outside the matrix dimensions
--returns: Matrix containning booleans indicating whether the corresponding
--         elements is local maxima

cfar_lmax::Matrix Expr->Expr->Matrix Bool
cfar_lmax mat minimum=
    (mmap m_sm_nbs_c_gtst . m_sm_nbs . m_add_border minimum) mat
--************************End of cfar_lmax***********************************


--************************Mean columns***************************************
--Forms a mean for the closest nr of elements (columnwise)

--nr = the number of elements to form a mean value for
--step = the number of elements to step when performing the mean value computation
--m = the matrix to perform the meanvalue computation on
--returns: A matrix where each element contains the mean value of the
--         corresponding element


mean_columns::Int->Matrix Expr->Matrix Expr
mean_columns nr = mmap ((/((toFNum.fromInt) nr)) . foldr1 (+)).
                     mnum_nbs_r nr

--************************End of mean columns********************************


--************************CFAR gof*******************************************
--Performs CFAR calculations on the rows of a matrix to give a CFAR value for
--each element

--n: the number of elements to form a mean over
```

```
--gf: number of guard bins
--step: the number of elements to step with when selecting elements for mean
--returns: A matrix containing the greatest of the “early” and “late” average

cfar_gof::Int->Int->Matrix Expr->Matrix Expr
cfar_gof n gf= mmap (uncurry max) .
               m_early_late_avg_c gf n .
               m_gof_padends n gf 1

--**********************End of CFAR  gof************************************


--**********************Local copy*****************************************
--Converts a datareduced matrix to original size by repeating elements in
--both directions (needed if step>1 in CFAR_GOF or MEAN_COLUMNS)

--r_copy: Number of rows to generate by repeating original rows
--c_copy: Number of columns to generate by repeating columns of resulting matrix
--m: matrix to enlarge (by factor one or greater in each direction)
--result: a matrix of the same dimensions as original data matrix

local_copy::Int->Int->Matrix Expr->Matrix Expr
local_copy r_copy c_copy = menlarge r_copy c_copy
--**********************End of local copy***********************************


--*************Here comes some monad stuff needed for resolv functions*********
data S s a = S (s -> (a,s))
unS (S a) = a

instance Monad (S s) where
  return a = S $ \s -> (a,s)
  m >>= k = S $ \s -> let (a,s') = (unS m) s in (unS (k a)) s'
  m >> k = S $ \s -> let (_,s') = (unS m) s in (unS k) s'

showS::S Int Int->String
showS m = case (unS m) 0 of (a,s)->"State: " ++ show s ++ " Count: " ++ show a

--*************End of the monadic stuff****************************************




--**********************Resolv_init*****************************************
--The resolv internal state datatype
data ResolvState=Maps {r_prf_map,v_prf_map::Vector Expr,speed_map::Matrix Expr}
 deriving Show

showres::S ResolvState a->a
showres m = case (unS m) zeroResState of (res,st) -> res

showstate::S ResolvState a->String
showstate m = case (unS m) zeroResState of (res,st) -> shw st
 where
 shw st= show (speed_map st) ++ show (v_prf_map st) ++ show (r_prf_map st)

getstate::S ResolvState a->ResolvState
getstate m = case (unS m) zeroResState of (res,st) -> st

setstate::ResolvState->S ResolvState ()
```

```
setstate st = S $ \_-> ((),st)

getStRes::S ResolvState a->(ResolvState,a)
getStRes m = case (unS m) zeroResState of (res,st) -> (st,res)

zeroResState::ResolvState
zeroResState = Maps {speed_map = [[]],v_prf_map = [],r_prf_map = []}


--Initializes the inner state of the resolv routines

--n= the n in the “m out of n” criteria
--rmax = the maximum length to fold out to
--returns: A monad which contains previous echoes and associated parameters
--         and returns nothing

resolv_init::Int->Int->S ResolvState ()
resolv_init n rmax = S $ \_-> ((),newstate)
 where
 --Install blank datastructures
 newstate = Maps {v_prf_map = m1,r_prf_map = m2,speed_map = m3}
 m1 = vgen_n_val n (toFNum 0.0)
 m2 = vgen_n_val n (toFNum 0.0)
 m3 = mgen_val rmax n (toFNum 0.0)
--***********************End of resolv_init***********************************



--***********************Resolv_range*****************************************
--Resolvs ranges, applies “m out of n”

--m = m in “m out of n” criteria
--vect = vector containing reflections folded into range bins of unambigous range
--returns: A monad which contains previous echoes and associated parameters
--         and returns list of indexes of the true range bins where the targets
--         are found. Updates the speed_map

resolv_range::Int->Vector Expr->S ResolvState [Int]
resolv_range m vect = S $ \s -> returntupl s vect m
 where
 returntupl s vect m = (list,newstate)
   where

   (list,newstate) =
      floor ( (fromInt (b_rows+in_rows-1)) / (fromInt in_rows) ) >$>
       \n_fold-> vtake b_rows (
                                vgen_rep_n_v n_fold vect
                               ) >$>
          \lcol->madd_v_last_c lcol (
                                        mdrop_first_c 1 (speed_map s)
                                    ) >$>
             \newmap->s {speed_map = newmap} >$>
              \newstate->mmap (\x-> if not (x==(toFNum 0)) then 1 else 0)
                              newmap >$>
                \bit_map->vmap2 (&&)
                                (
                                  vmap (>=m) ( msum_r bit_map )
                                )
                                (
                                  vmap (>(toFNum 0)) lcol
```

```
                                ) >$>
                        \det->vfind_pred_ind (==True) det >$>
                         \list->(list,newstate)


   (b_rows,_) = msize (speed_map s)
   in_rows = vsize vect
--***********************End of resolv_range*********************************


--***********************Resolv_vel and related friends******************
--Resolves the detected velocity to true velocity by folding out and checking
--for specified number of velocities inside velocity window. Updates the
--v_prf map

--r_det_index = list of true rangebins where targets have been detected
--m = m in “m out of n” criteria
--v_prf = unambigous velocity limit
--v_min = minimum velocity to fold out to
--v_max = maximum velocity to fold out to
--v_win = maximum velocity difference allowed for resolving data
--result: monad which returns list of tuples of (true range bins, true velocities
--        and keeps track of previous echoes and associated parameters

resolv_vel::[Int]->Int->Expr->Expr->Expr->Expr->S ResolvState ([Int],[Expr])
resolv_vel r_det_index m v_prf v_min v_max v_win =
             S $ \s->returntupl s r_det_index m v_prf v_min v_max v_win

  where
  returntupl s r_det_index m v_prf v_min v_max v_win = (tpl,newstate)
   where
   tpl = (rv_det_index,vel)
   newstate = s {v_prf_map = newvmap}

   newvmap =
     let
       vmap = v_prf_map s
     in
       vadd_e_last v_prf $ tail vmap

   (rv_det_index,vel) =
     let
        smap = speed_map s
        vellist =
              map (\r-> (
                          make_v_list r v_min v_max newvmap smap >$>
                           \vel_list->get_vel v_win m vel_list
                        )
                  ) r_det_index
     in
        (\(a,b)->(a,concat b)) $ unzip $ filter (\(a,b)->not (null b)) $
         zip r_det_index vellist


make_v_list::Int->Expr->Expr->Vector Expr->Matrix Expr->[Expr]
make_v_list rnum v_min v_max vprf_row smap =
   let
     genlist min max step start =
       if start == (toFNum 0) then
         []
```

```
       else
         [start,(start-step)..min]++[(start+step),(start+((toFNum 2)*step))..max]
   in
     vqsort $ concat $ zipWith (genlist v_min v_max) vprf_row (mrow rnum smap)



get_vel::Expr->Int->[Expr]->[Expr]
get_vel window m list =
   let
     mean_n n list = (/((toFNum.fromInt) n)) $ foldr1 (+) list
     span_ord_list list = last list - head list
     group_n n vlist =
         let
          part = take n vlist
         in
          if (length part < n) then
            []
          else
            part:(group_n n (tail vlist))
   in
      map (mean_n m) $ filter (\l->span_ord_list l < window)  $ group_n m list


--***********************End of resolv_vel**********************************


--***********************Resolv_cancel**************************************
--Removes the outfolded targets that has been succesfully resolved, and updates
--the r_prf map. Updates r_prf_map and speed_map

--rv_det_index = list of range bins where targets has been detected
--r_prf = unambigous range
--rs_res = range resolution

resolv_cancel::[Int]->Expr->Expr->S ResolvState ()
resolv_cancel rv_det_index r_prf rs_res= S $ \s ->
                                          returntupl s rv_det_index r_prf rs_res
 where
 returntupl s rv_det_index r_prf rs_res = ((),newstate)
  where
   newstate = s {speed_map = newsmap,r_prf_map = newrmap}
   newrmap =
     let
       rmap = r_prf_map s
     in
       vadd_e_last r_prf $ tail rmap
   newsmap =
     let
       smap = speed_map s
     in
       iter1 (\row map-> (
                           make_mod_list row map newrmap rs_res >$>
                             \modlist->setnull modlist map
                         )
             ) rv_det_index smap
```

```
make_mod_list::Int->Matrix Expr->Vector Expr->Expr->[[Int]]
make_mod_list rnum smap r_prf_row rs_res=
          zipWith constrLists (vtolist r_prf_row) (vtolist $ mrow rnum smap)
  where
  constrLists rprf e =
      let
        null_index_list rnum n_row
               |e == (toFNum 0) = []
               |otherwise = [strt,(strt+stp)..end]
        stp = (floor $ fromFNum (rprf/rs_res))
        strt = mod rnum stp
        end = n_row-1
      in
         null_index_list rnum n_row
  (n_row,n_col) = msize smap



setnull::[[Int]]->Matrix Expr->Matrix Expr
setnull rlist smap = mfrom_lol $ setnull' 0 rlist (mto_lol smap)
setnull' n rlist (srow:srows) =
  let
    nullf n selem [] = selem
    nullf n selem list = if (head list) == n then (toFNum 0) else selem
    newlist = map listmod rlist
     where
      listmod [] = []
      listmod l =  if ((head l) < n) then (tail l) else l
  in
    (zipWith (nullf n) srow rlist) : (setnull' (n+1) newlist srows)
setnull' _ _ _ = []


--***********************End of resolv_cancel *******************************
```

C.2. The datatype definitions

```
module Signdata.hs where

--(C) Per Bjesse (master thesis in computer science)
--The Haskell signal processing library (data module)



--     Start of stream datatype
infixr 5 :-
data Stream a = NullS | a :- (Stream a)

stream::[a]->Stream a
stream [] = NullS
stream (x:xs) = x:-stream xs

unitS::a->Stream a
unitS x = x :- NullS

nullS::Stream a->Bool
nullS NullS = True
nullS (_:-_) = False
```

```
headS::Stream a->a
headS (x:-_) = x

tailS::Stream a->Stream a
tailS (_:-xs) = xs

mapS::(a->b)->Stream a->Stream b
mapS f NullS = NullS
mapS f (x:-xs) = f x :- mapS f xs

dropS::Int->Stream a->Stream a
dropS k NullS  = NullS
dropS 0 s = s
dropS k (x:-xs) = dropS (k-1) xs

--     End of stream datatype

--     Start of symbolic datatype
data Expr = Var String | CVar String | OneArg Op1 Expr | TwoArg Op2 Expr Expr |
       FNum Float | INum Int | CNum (Float,Float) deriving (Eq,Ord)
data Op1 = Neg | Conj | Phase | Magn | Cos | Sin | Sqrt | Exp |
        Log | Round  deriving (Eq,Ord)
data Op2 =
       Add | Sub | Div | Mul | Rem | Mod | CForm | Raised
             deriving (Eq,Ord)

instance Num Expr where
        (+) (FNum f1) (FNum f2) = FNum (f1+f2)
       (+) (CNum (c1r,c1i)) (CNum (c2r,c2i)) = CNum (c1r+c2r,c1i+c2i)
       (+) s1 s2 = TwoArg Add s1 s2
       (-) (FNum f1) (FNum f2) = FNum (f1-f2)
       (-) (CNum (c1r,c1i)) (CNum (c2r,c2i)) = CNum (c1r-c2r,c1i-c2i)
       (-) s1 s2 = TwoArg Sub s1 s2
       (*) (FNum f1) (FNum f2) = FNum (f1*f2)
       (*) (CNum (a,b)) (CNum (c,d)) = CNum (a*c-b*d,c*b+d*a)
       (*) s1 s2 = TwoArg Mul s1 s2
       negate (FNum f1) = FNum (-f1)
       negate (CNum (cr,ci)) = CNum (-cr,-ci)
       negate s = OneArg Neg s


instance Fractional Expr where
        (/) (FNum f1) (FNum f2) = FNum (f1/f2)
       (/) (CNum (a,b)) (CNum (c,d)) = ((CNum (a/abscd,b/abscd)) *
             (CNum (c/abscd,-d/abscd)))
                    where abscd = sqrt (c*c + d*d)
       (/) s1 s2 = TwoArg Div s1 s2


instance Floating Expr where
        cos (FNum f1) = FNum (cos f1)
       cos e1 = OneArg Cos e1
       sin (FNum f1) = FNum (sin f1)
       sin e1 = OneArg Sin e1
       sqrt (FNum f1) = FNum (sqrt f1)
       sqrt e1 = OneArg Sqrt e1
        (**) (FNum f1) (FNum f2) = FNum (f1**f2)
       (**) e1 e2 = TwoArg Raised e1 e2
        exp (CNum (x,y)) = CNum ((exp (x))*(cos y),(exp (x))*(sin y))
```

```
       exp (FNum f) = FNum (exp f)
        exp e1 = OneArg Exp e1
        log (FNum f) = FNum (log f)
       log e1 = OneArg Log e1

log10 x = (log x) / (toFNum $ log 10)

instance Enum Expr where
       enumFromTo (INum s) (INum e) = map toINum [s..e]
       enumFromTo (FNum s) (FNum e) = map toFNum [s..e]
        enumFromThenTo (FNum s) (FNum n) (FNum e) = map toFNum [s,n..e]


fix::Expr->Int
fix (FNum f) = floor f

instance Integral Expr
instance Ix Expr
instance Real Expr
instance RealFrac Expr where
       round (FNum f) = round f

--A workaround due to the definition of round
eround::Expr->Expr
eround (FNum f) = INum (round f)
eround e = OneArg Round e

frem,fmod::Expr->Expr->Expr
frem x@(FNum x') y@(FNum y') = x-n*y
 where
 n = FNum (fromInt $ floor (x'/y'))
frem a b = TwoArg Rem a b

fmod x@(FNum x') y@(FNum y')  = FNum (fromInt $ floor (x'/y'))
fmod a b = TwoArg Mod a b

toINum::Int->Expr
toINum i = INum i
toFNum::Float->Expr
toFNum f = FNum f
toCNum c = CNum (c,0)

fromFNum::Expr->Float
fromFNum (FNum f) = f

fromFNumToInt::Expr->Int
fromFNumToInt (FNum f) = round f

conj,magn,phase::Expr->Expr
form::Expr->Expr->Expr
conj (CNum (f1,f2)) = CNum (f1,-f2)
conj s1 = OneArg Conj s1
magn (CNum (f1,f2)) = FNum (sqrt (f1^2+f2^2))
magn s1 = OneArg Magn s1
phase (CNum (f1,f2)) = FNum (atan (f2/f1))
phase s1 = OneArg Phase s1
form (FNum f1) (FNum f2) = CNum (f1,f2)
form e1 e2 = TwoArg CForm e1 e2

symbCVect::Int->String->Vector Expr
```

```
symbCVect n name = [CVar (name++"["++(show x)++"]") |x<-[0..(n-1)]]

instance Show Expr where
       showsPrec n e = showString (showExpr e)
showExpr::Expr->String
showExpr (CNum (re,im)) | im >= 0= show re ++ "+" ++ show im ++ "j"
showExpr (CNum (re,im)) = show re ++ show im ++ "j"
showExpr (FNum num) = show num
showExpr (INum num) = show num
showExpr (CVar s) = s
showExpr (Var s) = s
showExpr (OneArg Neg e1) = "-" ++ brackets e1
showExpr (OneArg Conj e1) = "Conj" ++ brackets e1
showExpr (OneArg Phase e1) = "Phase" ++ brackets e1
showExpr (OneArg Magn e1) = "Magn" ++ brackets e1
showExpr (OneArg Sin e1) = "Sin" ++ brackets e1
showExpr (OneArg Cos e1) = "Cos" ++ brackets e1
showExpr (OneArg Sqrt e1) = "Sqrt" ++ brackets e1
showExpr (TwoArg Add e1 e2) =lbrck++showExpr e1++"+"++showExpr e2++rbrck
showExpr (TwoArg Sub e1 e2) = lbrck++showExpr e1++"-"++showExpr e2++rbrck
showExpr (TwoArg Mul e1 e2) =showExpr e1 ++"*"++showExpr e2
showExpr (TwoArg Div e1 e2) = showExpr e1++"/"++showExpr  e2
brackets::Expr->String
brackets e1 = "(" ++ (showExpr e1) ++ ")"



rbrck = ")"
lbrck = "("

--      End of symbolic datatype

--     Start of Matrix,Vector datatypes
type Vector t = [t]
type Matrix t = [Vector t]


vsize::Vector t->Int
vsize = length

ltovect::[a]->Vector a
ltovect = id


vrcmult::Num a=>Vector a->Vector a->Matrix a
--v2 is presumed to be representing a "columnvector"
vrcmult v1 v2 = [map (*e) v1 | e<-v2]

vsub,vadd::Num a=>Vector a->Vector a->Vector a
vsub v1 v2 = vmap2 (-) v1 v2
vadd v1 v2 = vmap2 (+) v1 v2

vsum::Num a=>Vector a->a
vsum = sum

vmult_elem::Num a=>Vector a->Vector a->Vector a
vmult_elem v1 v2 = zipWith (*) v1 v2

mult_rv_m::Num a=>Vector a->Matrix a->Vector a
mult_rv_m v m = map (foldr1 (+)) [vmult_elem v v2 | v2 <- (mtransp m)]
```

```
vmap::Functor c=>(a->b)->c a->c b
vmap = map

vmap2::(a -> b -> c) -> [a] -> [b] -> [c]
vmap2 f v1 v2 = zipWith f v1 v2

vmap3::(a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
vmap3 = zipWith3

velem::Int->Vector a->a
velem n v = v!!n

vchelem::Int->a->Vector a->Vector a
vchelem n x v = take n v ++ [x] ++ drop (n+1) v

vgen_rep_n_v::Int->Vector a->Vector a
vgen_rep_n_v n v = concat $  replicate n v

vgen_f_t::Enum a=>a->a->Vector a
vgen_f_t start end = [start..end]

vgen_n_val::Int->a->Vector a
vgen_n_val n x = replicate n x

vgen_choose_e::[Int]->Vector a->Vector a
vgen_choose_e chl v = [velem x v | x<-chl]

vgen_sub_e::Num a=>[Int]->[Int]->Vector a->Vector a
vgen_sub_e sl1 sl2 v =
 [(velem (sl1!!x) v)-(velem (sl2!!x) v) | x<-[0..length sl1-1]]

vadd_e_last,vadd_e_first::a->Vector a->Vector a
vadd_e_last elem v = v++[elem]
vadd_e_first elem v = elem:v

vfoldr::(a->b->b)->b->Vector a->b
vfoldr = foldr

vfoldr1::(a->a->a)->Vector a->a
vfoldr1 = foldr1

vmin::Ord a=>Vector a->a
vmin = minimum

vfind_elem_ind::Eq a=>a->Vector a->Int
vfind_elem_ind e v = maybe (-1) id $ lookup e $ zip v [0..length v-1]

vconcat2::Vector a->Vector a->Vector a
vconcat2 v1 v2 = v1++v2

vconcat_lov::[Vector a]->Vector a
vconcat_lov = concat

vfilter::MonadZero b => (a -> Bool) -> b a -> b a
vfilter = filter

vreverse::Vector a->Vector a
vreverse = reverse
```

```
vtake,vdrop::Int->Vector a->Vector a
vtake = take
vdrop = drop

vqsort::Ord a=>Vector a->Vector a
qsort [] = []
qsort (a:x) = vqsort [y|y<-x,y<=a] ++ [a] ++ vqsort [y|y<-x,y>a]
vqsort = qsort


vpad_b,vpad_e::Int->Vector Expr->Vector Expr
vpad_b n= ((vgen_n_val n (CNum (0,0))) ++ )
vpad_e n= (++ (vgen_n_val n (CNum (0,0))))

vnorm,vmean::Vector Expr->Expr
vnorm = sqrt . vfoldr (+) (toFNum 0) . vmap (\k->(magn k)^2)
vmean v = (vfoldr1 (+) v) / ((toFNum.fromInt) (length v))

vabs::Vector Expr->Vector Expr
vabs v = vmap magn v

vsplit_to_m::Int->Vector t->Matrix t
vsplit_to_m  n cl
       | length cl <= n = [cl]
       | otherwise = (take n cl) : (vsplit_to_m n (drop n cl))

vsplit_l::Int->Vector t->[Vector t]
vsplit_l  n cl
       | length cl <= n = [cl]
       | otherwise = (take n cl) : (vsplit_l n (drop n cl))

vfind_pred_ind::(a->Bool)->Vector a->[Int]
vfind_pred_ind f v = [x | x<-[0..(vsize v) -1],f (velem x v)]

vcumprod::Vector Expr->Vector Expr
vcumprod = scanl1 (*)

vtomatr::Vector a->Matrix a
vtomatr v = [v]
vtolist::Vector a->[a]
vtolist = id

vbitreverse::Vector t->Vector t
vbitreverse [x]  = [x]
vbitreverse c = vbitreverse (evenIndexes c) ++ vbitreverse (oddIndexes c)
 where
 evenIndexes [] = []
 evenIndexes (x:y:zs) = x:evenIndexes zs
 evenIndexes (x:_) = [x]
 oddIndexes::Vector t->Vector t
 oddIndexes [] = []
 oddIndexes (x:xs) = evenIndexes xs




mtransp::Matrix t->Matrix t
mtransp a@(x:xs)
```

```
       |null x = []
       |otherwise = (map head a):(mtransp (map tail a))

mconj::Matrix Expr->Matrix Expr
mconj = (map (map conj)).mtransp

mreshape::Int->Matrix t->Matrix t
mreshape rows a@(x:xs)= (mtransp . (makematrix rows) . concat . mtransp) a
       where
       makematrix n [] = []
       makematrix n a= (take n a):(makematrix n (drop n a))


melem::Int->Int->Matrix a->a
melem m n matr = (matr!!m)!!n

mcol::Int->Matrix t->Vector t
mcol n m = (mtransp m) !! n

mrow::Int->Matrix t->Vector t
mrow n m = m!!n

mfrom_lov::[Vector a]->Matrix a
mfrom_lov = id

mfrom_lol::[[a]]->Matrix a
mfrom_lol = id
mto_lol::Matrix a->[[a]]
mto_lol = id

msum_r,msum_c::Num a=>Matrix a->Vector a
msum_r = map (foldr1 (+))
msum_c = map (foldr1 (+)) . mtransp

emptymatrix = []
madd_v_first_r,madd_v_last_r::Vector a->Matrix a->Matrix a
madd_v_first_r v m = v:m
madd_v_last_r v m = m ++ [v]

madd_v_first_c,madd_v_last_c::Vector a->Matrix a->Matrix a
madd_v_first_c [] _ = []
madd_v_first_c (x:xs) (r:rs) = ((x:r):(madd_v_first_c xs rs))
madd_v_last_c [] _ = []
madd_v_last_c (x:xs) (r:rs) = ((r++[x]):(madd_v_last_c xs rs))

m_add_border::a->Matrix a->Matrix a
m_add_border min x = [padrow] ++ (map (\r->[min]++r++[min]) x) ++ [padrow]
 where
 padrow = replicate ((+2) $ length $ head x) min


mattach_row::Matrix a->Matrix a->Matrix a
mattach_row m1 m2 = m1++m2
mattach_col::Matrix a->Matrix a->Matrix a
mattach_col m1 m2 = mtransp (m1'++m2')
 where
 m1' = mtransp m1
 m2' = mtransp m2

mgen_val::Int->Int->t->Matrix t
```

```
mgen_val m n matval = genvect m (genvect n matval)
       where
       genvect num val = take num (repeat val)

mgen_n_v::Int->Vector a->Matrix a
mgen_n_v n v= replicate n v


mgen_f_m_n::Int->Int->(Int->Int->a)->Matrix a
mgen_f_m_n m n f =  (mmap (\x->f (fst x) (snd x)) . mgen_numb_tupl m) n

mgen_subtr::Num a=>[Int]->[Int]->Matrix a->Matrix a
mgen_subtr r1 r2 mat = mfrom_lov (map (uncurry (subcomb mat)) (zip r1 r2))
 where
 subcomb m x1 x2 = vsub (mrow x1 m) (mrow x2 m)

mgen_choose_r::[Int]->Matrix a->Matrix a
mgen_choose_r l m = mfrom_lov (map (\x-> mrow x m) l)

mgen_choose_c::[Int]->Matrix a->Matrix a
mgen_choose_c l = mtransp . mgen_choose_r l .mtransp

mgen_numb_tupl::Int->Int->Matrix (Int,Int)
mgen_numb_tupl rows cols= [genrow y cols | y<-[0..(rows-1)]]
       where
       genrow rnum cols= [(rnum,x) | x<-[0..(cols-1)]]


mgen_numb_row::Int->Int->Matrix Int
mgen_numb_row m n= take n (repeat [x | x<-[0..(m-1)]])

mgen_break_list::Int->Int->[a]->Matrix a
mgen_break_list rws cls list =
       mreshape rws ((mtransp.vtomatr.ltovect.take (rws*cls)) list)

mchrow::Int->Vector a->Matrix a->Matrix a
mchrow n v m = take n m ++ [v] ++ drop (n+1) m

mchcol::Int->Vector a->Matrix a->Matrix a
mchcol n v m = mtransp $ (\m->take n m ++ [v] ++ drop (n+1) m) $ mtransp m

msize::Matrix t->(Int,Int)
msize m1 = (length m1, length (m1!!0))

mtovect::Matrix a->Vector a
mtovect m = head m

mtake_first_r,mdrop_first_r::Int->Matrix a->Matrix a
mtake_first_c,mdrop_first_c::Int->Matrix a->Matrix a
mtake_first_r = take
mdrop_first_r = drop
mtake_first_c n m = map (take n) m
mdrop_first_c n m = map (drop n) m

mmap_r::Functor c=>(a->b)->c a->c b
mmap_r f m = map f m

mmap::(a->b)->Matrix a->Matrix b
mmap f m = map (map f) m
```

```
mmap2::(a->b->c)->Matrix a->Matrix b->Matrix c
mmap2 f m1 m2 = mmap (uncurry f) (zipWith zip m1 m2)

mmap3::(a->b->c->d)->Matrix a->Matrix b->Matrix c->Matrix d
mmap3 f m1 m2 m3 = mmap (\(a,b,c)->f a b c) (zipWith3 zip3 m1 m2 m3)

mmap4::(a->b->c->d->e)->Matrix a->Matrix b->Matrix c->Matrix d->Matrix e
mmap4 f m1 m2 m3 m4= mmap (\(a,b,c,d)->f a b c d) (zipWith4 zip4 m1 m2 m3 m4)
 where
 zip4 [] _ _ _ = []
 zip4 (x:xs) (y:ys) (z:zs) (w:ws) = (x,y,z,w):(zip4 xs ys zs ws)
 zipWith4 f [] _ _ _ = []
 zipWith4 f (x:xs) (y:ys) (z:zs) (w:ws) = (f x y z w):(zipWith4 f xs ys zs ws)

mconcat::Matrix a->Matrix b->Matrix (a,b)
mconcat m1 m2 = zipWith zip m1 m2

mconv_list::Matrix a->[a]
mconv_list = concat

mmod_f_row::(Vector a->Vector a)->Int->Matrix a->Matrix a
mmod_f_row func r m = beg ++ [func the_row] ++ end
 where
 beg = take r m
 the_row = head $ drop r m
 end = drop (r+1) m

mpad_b_r,mpad_e_r,mpad_b_c,mpad_e_c::Int->Matrix Expr->Matrix Expr
mpad_b_r n = mmap_r (vpad_b n)
mpad_e_r n = mmap_r (vpad_e n)
mpad_b_c n = mtransp . mpad_b_r n . mtransp
mpad_e_c n = mtransp . mpad_e_r n . mtransp

mmax::Ord t=>Matrix t->Matrix t->Matrix t
mmax m1 m2 = mmap2 max m1 m2

mfind::Matrix Bool->[(Int,Int)]
mfind matr =  (strp . mconv_list . rmv . mconcat matr . mgen_numb_tupl m) n
       where
       (m,n) = msize matr
       filtfunc (x1,x2) = x1
       rmv = mmap_r (vfilter filtfunc)
       strp = map (\(x1,x2)->x2)

mmean::Matrix Expr->Expr
mmean m = vmean (map vmean m)

mmean_c::Matrix Expr->Vector Expr
mmean_c m = mmap_r vmean (mtransp m)

mcumsum_c,mcumprod_c::Matrix Expr->Matrix Expr
mcumsum_c = mtransp . mmap_r (scanl1 (+)) . mtransp
mcumprod_c = mtransp . mmap_r (scanl1 (*)) . mtransp

msort_c::Matrix Expr->Matrix Expr
msort_c = mtransp . mmap_r vqsort . mtransp

mmin::Matrix Expr->[(Int,Expr)]
mmin matr=  (indmax . mconcat (mgen_numb_row m n) . mtransp) matr
       where
```

```
       (m,n) = msize matr
       indmax = mmap_r (vfoldr1 maxtuple)
       maxtuple t1@(_,x1) t2@(_,x2) = if x1>x2 then t1 else t2

mabs::Matrix Expr->Matrix Expr
mabs m = mmap magn m

enlarge _ [] = []
enlarge n (x:xs) = (replicate n x) ++ enlarge n xs
menlarge rcopy ccopy m= enlarge rcopy (map (enlarge ccopy) m)

every_nth n [] = []
every_nth n r@(x:_)= if (length r) < n then [x] else
                        (x:(every_nth n (drop n r)))

mevery_nth_c n = map (every_nth n)
mevery_nth_r n = every_nth n


nbs (x:y:z:ws) = (x,y,z) : (nbs (y:z:ws))
nbs _ = []
m_sm_nbs::Matrix a->Matrix ((a,a,a),(a,a,a),(a,a,a))
m_sm_nbs m = map (\(r1,r2,r3)->nbs (zip3 r1 r2 r3)) (nbs m)

m_sm_nbs_c_gtst::Ord a=>((a,a,a),(a,a,a),(a,a,a))->Bool
m_sm_nbs_c_gtst (r1,r2@(_,c,_),r3) =  foldr1 (&&) $ map (gt_tpl c) [r1,r2,r3]
 where
 gt_tpl c (e1,e2,e3) = foldr1 (&&) $ map (<=c) [e1,e2,e3]

rnbs _ [] = []
rnbs num y@(x:xs) = if (length y)<num then [] else (take num y):(rnbs num xs)
mnum_nbs_r num = map (rnbs num)
mnum_nbs_c num = mtransp . mnum_nbs_c num . mtransp



m_gof_padends n gf stp r=
   t1 ++ r ++ t2
    where
    t1 = take (n+gf) $ drop (stp+gf) r
    t2 = take (n+gf) $ drop (rws-stp-gf-n-gf) r
    rws = length r


avg_rows d n r@(r1:rs)=
  if (length r) > (2*n+2*d) then
     (take n r,take n $ drop (n+2*d+1) r)  : avg_rows d n rs
  else
     []

transp a@(x:xs)
       |null x = []
       |otherwise = (map head a):(transp (map tail a))


m_early_late_avg_c d n r@(r1:rs) =
  if (length r) > (2*n+2*d) then
     (zip (avg (transp $ take n r))
          (avg (transp (take n $ drop (n+2*d+1) r))))
      : m_early_late_avg_c d n rs
```

```
  else
     []
             where
             avg = map $ (/(toFNum $ fromInt n)) . (foldr1 (+))


--     End of Matrix,Vector datatypes



--     Matrix IO to matlab
compFile::String->String->(Matrix Expr->Matrix Expr)->IO ()
compFile path path2 comp = readFile path >>= \n-> writeFile path2 (transf n)
       where
       transf = convMatrString . comp . convStringMatr

convStringMatr::String->Matrix Expr
convStringMatr x = repcomb mdata
       where
       m =  (round . read) (words ((lines x) !! 0) !! 0)
       n =  (round . read) (words ((lines x) !! 0) !! 1)
       mdata = map ((\x->FNum x) . read) (words ((lines x) !! 1))
       rerow d= take n d
       imrow d = (take n . drop n) d
       rest d =  drop (2*n) d
       clxrow d= zipWith form (rerow d) (imrow d)
       repcomb [] = []
       repcomb d = (clxrow d):(repcomb (rest d))

convMatrString::Matrix Expr->String
convMatrString x = sym m ++ " "++ sym n ++ "\n" ++ (repcomb (concat x))
       where
       deform (CNum (a,b)) = (a,b)
       m = length x --Space-leak
       n = length (x!!0)
       sym = (pad8 . (++".0") . show . fromInt)
       rerow l= (concat . map ((++" ") .pad8 . show . fst . deform)) l
       imrow l= (concat . map ((++" ") .pad8 . show . snd . deform)) l
       pad8 x
              | length x < 8 = x ++ (take (8 - length x) (repeat '0'))
              | otherwise = x
       repcomb::Vector Expr->String
        repcomb [] = ""
       repcomb l = rerow (take n l) ++ imrow (take n l) ++ repcomb (drop n l)



writeMatr::String->Matrix Expr->IO ()
writeMatr path = writeFile path . convMatrString
readMatr::String->IO ()
readMatr path = readFile path >>= \n-> return (convStringMatr n) >>= print
rcomp path todo  = readFile path >>= \n-> return (action n) >>= print
      where
      action n = (todo . convStringMatr) n
--     End of IO
```

C.3. Noise generation module

```
{-
   This module implements a (good) random number generator.

   The June 1988 (v31 #6) issue of the Communications of the ACM has an
   article by Pierre L'Ecuyer called, "Efficient and Portable Combined
   Random Number Generators".  Here is the Portable Combined Generator of
   L'Ecuyer for 32-bit computers.  It has a period of roughly 2.30584e18.

   Transliterator: Lennart Augustsson
-}

module RandomHBC(randomInts, randomDoubles, normalRandomDoubles) where
-- Use seeds s1 in 1..2147483562 and s2 in 1..2147483398 to generate
-- an infinite list of random Ints.
randomInts :: Int -> Int -> [Int]
randomInts s1 s2 =
    if 1 <= s1 && s1 <= 2147483562 then
       if 1 <= s2 && s2 <= 2147483398 then
           rands s1 s2
       else
           error "randomInts: Bad second seed."
    else
       error "randomInts: Bad first seed."

rands :: Int -> Int -> [Int]
rands s1 s2 = z' : rands s1'' s2''
       where  z'   = if z < 1 then z + 2147483562 else z
              z    = s1'' - s2''

              k    = s1 `quot` 53668
              s1'  = 40014 * (s1 - k * 53668) - k * 12211
              s1'' = if s1' < 0 then s1' + 2147483563 else s1'

              k'   = s2 `quot` 52774
              s2'  = 40692 * (s2 - k' * 52774) - k' * 3791
              s2'' = if s2' < 0 then s2' + 2147483399 else s2'

-- Same values for s1 and s2 as above, generates an infinite
-- list of Doubles uniformly distibuted in (0,1).
randomDoubles :: Int -> Int -> [Double]
randomDoubles s1 s2 = map (\x -> fromIntegral x * 4.6566130638969828e-10) (randomInts s1 s2)

-- The normal distribution stuff is stolen from Tim Lambert's
-- M*****a version

-- normalRandomDoubles is given two seeds and returns an infinite list of random
-- normal variates with mean 0 and variance 1.  (Box Muller method see
-- "Art of Computer Programming Vol 2")
normalRandomDoubles :: Int -> Int -> [Double]
normalRandomDoubles s1 s2 = boxMuller (map (\x->2*x-1) (randomDoubles s1 s2))

-- boxMuller takes a stream of uniform random numbers on [-1,1] and
-- returns a stream of normally distributed random numbers.
boxMuller :: [Double] -> [Double]
boxMuller (x1:x2:xs) | r <= 1    = x1*m : x2*m : rest
                     | otherwise = rest
                              where r = x1*x1 + x2*x2
                                    m = sqrt(-2*log r/r)
```

```
                                    rest = boxMuller xs
```

C.4. Radar simulation code

```
--(C) Per Bjesse (master thesis in computer science)
--Simulation of the typical operations performed in an airborne radar

--Uses the Haskell signal processing library
--To start: type <go> and then wait (a substantial amount of time) for the
--results




import Signbeh
import Debug

--**********************Constants***********************************
c,hf,fs,sweep_bw,rs_res,noise_level_db,mainlob_level_db,duty_cycle::Expr
min_value_db::Expr
n_sample_list,n_pri_list::[Int]
gof_n_rs,gof_n_pri,gof_n_guard::Int
gof_delta_db,g_t0_db,s_t0_db::Expr
m,n,r_max,r_window,mode_change_interval::Int
v_min,v_max,v_window::Expr
t_range,t_vel,t_amp_db::[Expr]
lambda::Expr
first_dch::Int




c = toFNum 2.997e+8
hf = toFNum 3.0e+9              -- frequency of send pulse [Hz]
fs = toFNum 1.0e+6              -- sampling frequency [Hz]
sweep_bw = fs                   -- bandwidth of chirp pulse [Hz]
rs_res = (c/ ((toFNum 2)*fs))   -- range sample resolution [meter]
noise_level_db = toFNum (-75)   -- noise level relative to ADO-max   [dB]
mainlob_level_db = toFNum 40    -- level of mainlob above noise level [dB]
min_value_db = noise_level_db - (toFNum 10)
duty_cycle = toFNum 0.1         -- relative pulse length
n_sample_list = [25,32,35,37,39,43]
                                -- list of the number of samples between pulses
n_pri_list = [32,64,128]
                                -- list of the number of pulses to integrate over

-- local threshold parameters
gof_n_rs = 16
gof_n_pri = 1
gof_n_guard = 1
gof_delta_db = toFNum 0         -- decibel
g_t0_db = toFNum 15             -- decibel
s_t0_db = toFNum 15             -- decibel


-- resolv parameters
```

```
-- require m “hits” in the latest n inti:s to accept a new target
m = 3
n = length n_sample_list

-- Maximum number of range bins to fold out to.
r_max = fix ((toFNum 200000)/rs_res)


-- Range window size; detections must be inside
-- for a range resolve sucess (not changable here).
r_window = 1                    --range bins


-- Velocity limits to fold out to.
v_min = toFNum (-1000)          -- m/s
v_max = toFNum 1000             -- m/s


-- Velocity window size; detections must be inside
-- for a velocity resolve sucess.
v_window  = toFNum 30           -- m/s
mode_change_interval = 6


-- target parameters
t_range  = map (\k->toFNum $ k*1000) [50, 100]
t_vel    = map toFNum [300, -450]
t_amp_db = map (\k->(toFNum k)+noise_level_db) [10, 30]


--derived parameters
lambda = c/hf
first_dch = 7
--***********************End of constants*********************************




--***********************THE MAIN PROGRAM IN ALL ITS GLORY*****************
--Things to do:
--Fix the symbolics

data MainPrgState = MS {
  counter::Int,
  n_pri_select::Int,
  n_sample_select::Int,
  rslv_state::ResolvState,
  random_seed::Int
} deriving Show

startstate = MS {
  counter = 0,
  n_pri_select = 0,
  n_sample_select = 0,
  rslv_state = getstate (resolv_init n r_max),
  random_seed = 1499
}

data RadarVariables = RV {
  vn_pri::Int,
```

```
  vn_sample::Int,
  vn_rs_pulse::Int,
  vn_rs::Int,
  vn_rs_zero::Int,
  vn_rs_min::Int,
  vn_rb_min::Int,
  vf_prf::Expr,
  vr_prf::Expr,
  vv_prf::Expr,
  vn_fft::Int,
  vlast_dch::Int
}



--The function that does all the work:
go::IO ()
go = do_proc startstate

do_proc state = putStr showstr >> (do_proc) newstate
    where
     new_counter = ((counter state) +1)
     new_n_pri_select =
           if (mod new_counter mode_change_interval) == 1 then
            (mod ((n_pri_select state)) (length n_pri_list) + 1)
           else
            (n_pri_select state)
     this_reslv_state =
           if (mod new_counter mode_change_interval) == 1 then
              getstate (resolv_init n r_max)
           else
              (rslv_state state)
     new_n_sample_select = (mod (n_sample_select state) (length n_sample_list)+1)
     seedlist =  map randclip $ take 4 $ randomInts r r
           where
           r = random_seed state
           randclip x =
             let
              absx = abs x
             in
              if absx < 1 then
                1
              else
               if absx > 2147483562 then
                 2147483562
               else
                 absx
     new_seed = head seedlist
     (showstr,new_rslv_state) =
        --First step: computation of this iterations variables
        compute_variables state new_n_pri_select new_n_sample_select >$>

          --Now generate the simulated data
          \v-> gen_data seedlist v >$>

            --Filter data
            \(sum_video,guard_video,send_pulse)->
                do_doppler sum_video guard_video v >$>
             \(sum_video_d,guard_video_d)->
                  pulse_compress send_pulse sum_video_d guard_video_d v >$>
```

```
              --Find and resolv targets
              \(sum_video_p,guard_video_p)->
                  convert_db sum_video_p guard_video_p >$>
                \(sum_db,guard_db)->
                   mean_ampl_db sum_db guard_db v >$>
                  \(sum_noise_db,guard_noise_db)->
                     get_local_max_mean sum_db sum_noise_db >$>
                   \(sum_lmax_det,sum_gof_db,sum_thr_db)->
                      detect sum_db guard_noise_db guard_db
                      sum_lmax_det sum_thr_db v >$>
                     \(det_list,det_dch,det_rb,n_det)->
                        resolv_targets det_rb det_dch this_reslv_state v >$>

                        --Build presentation strings
                        \(target_range,target_velocity,rv_ind_l,new_rslv_state)->
                           build_result n_det det_list
                                target_range target_velocity rv_ind_l v >$>
                           \showstr-> (showstr,new_rslv_state)

     --Compute the new state of the program
     newstate = state {
       counter = new_counter,
       n_pri_select = new_n_pri_select,
       n_sample_select = new_n_sample_select,
       rslv_state = new_rslv_state,
       random_seed = new_seed
     }

--***********************END OF THE MAIN PROGRAM**************************




compute_variables state new_n_pri_select new_n_sample_select = variables
   where
   variables = RV {
    vn_pri=n_pri,
    vn_sample=n_sample,
    vn_rs_pulse=n_rs_pulse,
    vn_rs=n_rs,
    vn_rs_zero =n_rs_zero ,
    vn_rs_min =n_rs_min ,
    vn_rb_min =n_rb_min ,
    vf_prf =f_prf ,
    vr_prf =r_prf ,
    vv_prf =v_prf ,
    vn_fft =n_fft ,
    vlast_dch = last_dch
   }
   n_pri=n_pri_list!!(new_n_pri_select-1)
   n_sample=n_sample_list!!(new_n_sample_select-1)
   n_rs_pulse=floor ((fromInt n_sample)*(fromFNum duty_cycle))
   n_rs=n_sample - n_rs_pulse
   n_rs_zero = floor ((fromInt n_rs_pulse)/2)
   n_rs_min = n_rs_pulse
   n_rb_min = n_rs_pulse - n_rs_zero
   f_prf = fs / (toFNum $ fromInt n_sample)
```

```
   r_prf = (toFNum $ fromInt n_sample) * rs_res
   v_prf = lambda * f_prf /(toFNum 2)
   n_fft = n_pri
   last_dch = n_pri - 6


--Generate matrixdata to perform the processing on
gen_data seedlist v= (sum_video,guard_video,send_pulse)
   where
   n_rs = vn_rs v
   n_pri = vn_pri v
   r_prf = vr_prf v
   v_prf = vv_prf v
   n_sample = vn_sample v
   n_rs_min = vn_rs_min v
   n_rs_pulse = vn_rs_pulse v
   seed1 = head $ tail seedlist
   seed2 = head $ tail $ tail seedlist
   seed3 = head $ tail $ tail $ tail seedlist
   sum_video =
        gen_targets n_rs n_pri send_pulse r_prf v_prf n_sample
                n_rs_min t_range t_vel t_amp_db >$>
          \sum_targets->gen_clutter n_rs n_pri
                           (noise_level_db+mainlob_level_db) seed3 >$>
            \sum_mainlob_clutter->
              gen_noise n_rs n_pri noise_level_db seed2 >$>
              \sum_noise->
                mmap3 (\x1 x2 x3->x1+x2+x3) sum_noise sum_mainlob_clutter sum_targets
   send_pulse = gen_lfm_pulse n_rs_pulse fs sweep_bw
   guard_video = gen_noise n_rs n_pri noise_level_db seed1




--Pass the generated data through the doppler filterbank
do_doppler sum_video guard_video v= (sum_video_d,guard_video_d)
   where
   n_pri = vn_pri v
   n_fft = vn_fft v
   dwindow = make_window n_pri
   do_fft_r n m = mmap conj $ mfft_r n (mmap conj m)
   sum_video_d = do_fft_r n_fft (do_weight dwindow sum_video)
   guard_video_d = do_fft_r n_fft (do_weight dwindow guard_video)


--Pulsecompress each range bin in the filtered matrix
pulse_compress send_pulse sum_video_d guard_video_d v=
    (sum_video_p,guard_video_p)
   where
   n_rs_pulse = vn_rs_pulse v
   n_rs_zero = vn_rs_zero v
   pwindow = make_window n_rs_pulse
   pulse = vmult_elem pwindow send_pulse
   sum_video_p = pcomp_time sum_video_d pulse n_rs_zero
   guard_video_p = pcomp_freq guard_video_d pulse n_rs_zero


--Convert all amplitudes to decibel
convert_db sum_video_p guard_video_p = (sum_db,guard_db)
   where
```

```
   sum_db = mmap (\k->(toFNum 20) * (log10 (magn k))) sum_video_p
   guard_db =  mmap (\k->(toFNum 20) * (log10 (magn k))) guard_video_p


--Get a the mean value of the amplitudes of a selected part of the whole
--matrix (in decibels)
mean_ampl_db sum_db guard_db v= (sum_noise_db,guard_noise_db)
   where
   n_rs = vn_rs v
   last_dch = vlast_dch v
   to_db first_dch last_dch nodb = mmean db
      where
      db = mtake_first_r (last_dch-first_dch+1) $ mdrop_first_r first_dch $
            mtransp nodb
   sum_noise_db = to_db first_dch last_dch sum_db
   guard_noise_db = to_db first_dch last_dch guard_db


--Get the local maxima, local mean for the matrix
get_local_max_mean sum_db sum_noise_db = (sum_lmax_det,sum_gof_db,sum_thr_db)
   where
   sum_lmax_det = cfar_lmax sum_db min_value_db
   sum_gof_db = mean_columns gof_n_pri sum_db >$>
                \t1-> cfar_gof gof_n_rs gof_n_guard t1 >$>
                  \t2-> mmap (\k->k-(sum_noise_db+gof_delta_db)) t2
   sum_thr_db = mmap (\k-> k+sum_noise_db+s_t0_db)
                 (mmap (\k->
                         if k > (toFNum 0) then
                          k
                         else
                          (toFNum 0)
                       )
                       sum_gof_db
                 )



--Now for the detections of targets

mask n_rs n_pri first_dch last_dch = mgen_n_v n_rs $
 vgen_n_val (first_dch-1) False ++
 vgen_n_val (last_dch-first_dch+1) True ++
 vgen_n_val (n_pri-last_dch) False

detect sum_db guard_noise_db guard_db sum_lmax_det sum_thr_db v=
     (det_list,det_dch,det_rb,n_det)
   where
   n_rs = vn_rs v
   n_pri = vn_pri v
   last_dch = vlast_dch v
   det_list = mmap2 (\k l-> k>l) sum_db sum_thr_db >$>
              \sum_det->mmap (> (g_t0_db+guard_noise_db)) guard_db >$>
               \guard_det-> mgen_n_v n_rs (
                             vgen_n_val (first_dch-1) False ++
                             vgen_n_val (last_dch-first_dch+1) True ++
                             vgen_n_val (n_pri-last_dch) False) >$>
                  \mask-> mmap2 (\k l->k && l)
                            (mmap2 (\k l->k && l) mask sum_lmax_det)
                            (mmap2 (\k l->k && (not l)) sum_det guard_det) >$>
                    \detections->mfind detections
```

```
   det_amp_db = map (\(a,b)->melem a b sum_db) det_list
   det_dch = map (\(a,b)->b) det_list
   det_rb = map (\(a,b)->a) det_list
   n_det = length det_amp_db


--Resolv the detected targets
resolv_targets det_rb det_dch this_reslv_state v=
    (target_range,target_velocity,rv_ind_l,new_rslv_state)
   where
   n_sample = vn_sample v
   n_rb_min = vn_rb_min v
   n_fft = vn_fft v
   v_prf = vv_prf v
   r_prf = vr_prf v
   (new_rslv_state,(rv_ind_l,vel_l)) =
      getStRes
       (
        setstate this_reslv_state >>
           iter2 (\dch rb ->
                   vchelem (n_rb_min + rb)
                           ((toFNum ((fromInt (dch+1))/(fromInt n_fft)))*v_prf)
                 )
                   det_dch det_rb
                     (
                      vgen_n_val n_sample (toFNum 0)
                     ) >$>
           \inhits-> resolv_range m inhits >>=
             \r_ind_list-> resolv_vel r_ind_list m v_prf v_min v_max v_window >>=
              \(rv_ind_list,target_vel) ->
                                    resolv_cancel rv_ind_list r_prf rs_res >>
                return (rv_ind_list,target_vel)
       )
   target_range = map (\k->(toFNum $ ((fromInt k)/1000))*rs_res) rv_ind_l
   target_velocity = map (\k-> (-k)) vel_l


--Construct the messages to present to the user
build_result n_det det_list target_range target_velocity rv_ind_l v= showstr
   where
   n_rs = vn_rs v
   n_pri = vn_pri v
   showstr = “Dimensions:” ++ show (n_rs,n_pri) ++ “\n” ++
             “Num detections:” ++ show n_det ++ “\n” ++
             “List:”++show det_list++ “\n” ++ hitstr

   hitstr = if ((length rv_ind_l)>0) then
      constrHitMess (zip target_range target_velocity) ++ “\n”
    else
      “\n”

   constrHitMess [] = “\n”
   constrHitMess ((r,v):rvs) = “Range: “ ++ show r ++ “ Vel: “ ++ show v ++
                               “\n” ++ constrHitMess rvs
```