THE UNIVERSITY OF READING

A Framework for Refining Functional Specifications into Parallel Reconfigurable Hardware Implementations

PhD Thesis in Computer Science

Department of Computer Science

John Hawkins

June 30, 2005

Declaration

I confirm that this thesis is my own work and the use of all material from other sources has been properly and fully acknowledged.

Abstract

Reconfigurable logic devices such as the FPGA have brought about a revolution in the field of hardware design. The reduction in development costs has had a huge impact on broadening the scope of applications for which a hardware implementation is a realistic possibility. Current FPGA devices run to many millions of gates, giving a huge potential for efficiency gains, benefiting from the inherently parallel nature of hardware circuits. These devices continue to grow in size, to the end that we can now seriously consider implementing even large scale systems purely in reconfigurable logic.

Despite these advances, we find ourselves somewhat lacking in the tools and methodologies required to fully exploit this potential. Issues of hardware implementation and parallelism introduce significant complexity into the design process. We argue that without the correct approach, not only will this potential be under used, but the inherent complexity will undermine people's confidence about the correctness of resulting implementations, limiting the scope in which they can be deployed.

This thesis presents a methodology in which an algorithm specified in a high level functional programming language can be transformed and refined into a parallel implementation suitable for execution on an FPGA device. All throughout the methodology, a clear path of proof is maintained, so that the resulting implementations are provably correct.

The work in this thesis concerns refining functional programs into parallel networks of concurrent processes. Also, a refinement step from concurrent processes in the CSP notation to the Handel-C hardware description language is introduced, which facilitates implementation on an FPGA. Additionally, an extended notion of data refinement is presented, providing the developer with choices for how to represent types from the specification in the implementation. Here we expand on the notion of the *stream*, a serial communication scheme, and also introduce the *vector*, a data parallel communication scheme. This gives new scope to tackle a number of algorithms which previously could not be addressed in a scalable manner, particularly those which involve quadratic sized data structures such as the Cartesian product. Furthermore this new scope is complemented with a rich library of provably correct, re-usable components, addressing both the stream and vector settings. These components refine commonly used higher order and list processing functions from the specification. Finally the usefulness of these contributions is illustrated with a number of case studies, covering not only academic style problems, but also real world applications.

Acknowledgements

First and foremost I owe a huge debt of gratitude to my supervisor, Ali Abdallah, who has been a tremendous source of inspiration, guidance and motivation over the last few years. Additionally I would like to thank my examiners for all their hard work in checking the thesis, and their insightful discussion which led to some significant improvements.

A number of organisations deserve acknowledgement for the part they have played in helping me to carry out this work. First, the EPSRC who provided me with a studentship for the first three years. Next, Reading University and London South Bank University for providing the environment and the support which has allowed this research to take place. Finally Softel Ltd, for being flexible and allowing me to continue my studies whilst working there.

On a personal level I am extremely grateful to a number of fellow PhD students, for spurring me on and creating many happy memories over the last few years. In particular from Reading - Rob Lang, Mark Gasson and Michelle Fountain, three truly great people who will no doubt remain life long friends. Additionally from the research group I was working at for the latter part of my PhD, I'd like to say a big thank you to Mark Green, Issam Damaj and Hamdan Dammag, all of whom showed me that there was a light at the end of the tunnel!

As anyone who has done a PhD will know, it can be a real feat of endurance, and often it is not just you, but the people close to you who have to endure it! As such may I extend my thanks - and apologies(!) to members of my family - particularly to Mum, Dad and Vera. Finally to Chie, who has suffered with it more than most, I can't ever say thank you enough for your patience and perserverance.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

As the field of computing has grown, so have demands on performance and reliability of systems. In many areas, such demands have forced us to consider moving away from more traditional computing paradigms and looking instead to alternative and novel architectures. These requirements incur a price, in that the design and implementation of such systems becomes ever more complex. Nowhere is this more true than in the fields of hardware design and parallel processing. The burden of complexity on developers and engineers becomes so great in these environments that it would be naive in the extreme to expect successful results from a purely ad-hoc design process. Arguably, rigorous design methodologies are not just desirable, but are in fact essential.

Hardware design introduces a number of development issues that are not so evident in the field of software engineering. To begin with, the lower level nature of a hardware implementation makes it far more difficult for an engineer to conceptualise, and therefore significantly more difficult to develop. A second, and perhaps larger concern, is that the more physical nature of the hardware development process traditionally makes it extremely costly. In software engineering, compilation is typically a quick and inexpensive task, which tends to encourage (for better or worse) something of a 'trial and error' approach to system development. In *traditional* hardware design (e.g. ASIC development), an equivalent iteration requires the physical construction/modification of a circuit or device which can be extremely time consuming and costly. Moreover, in many cases any errors present can be far more difficult to detect than in an equivalent software based solution. There is therefore a much stronger incentive to produce designs that are error-free before any actual physical implementation occurs.

The nature of parallel processing will also add significant complexity into the design of a system. An entire extra dimension is added to system development which not only brings with it a whole new area of potential problems, but also introduces new challenges in terms of maximising efficiency. The selection of a model for parallel processing is an extremely important part of designing

any parallel algorithm. Broadly the models fall into two categories; those using shared memory and those relying on message passing. Intuitively many developers creating 'ad-hoc' parallel implementations tend to opt initially for shared memory architectures as they are typically simpler to develop and seemingly incur less overhead. However, shared memory architectures are subject to the obvious bottlenecks caused by contention over simultaneous resource access. As a result these architectures do not scale as well as those relying on message passing. On the down side, the often complex interactions that occur in a message passing environment can make the reliable and efficient design of such systems extremely difficult. A methodology that helped to abstract away much of this complexity would be a very useful commodity.

Reconfigurable logic devices, such as the FPGA, have brought about something of a revolution in hardware design. Complex circuitry can be mapped onto these devices without the requirement for any expensive machinery or manual effort. This, coupled with the fact that a single device can be reconfigured over and over again, has potential to utterly transform the cost of the hardware design process. To begin with, devices like the FPGA only had a relatively small number of gates, so were typically only used for 'glue logic', as a bridge between other parts of a hardware design. Now, however, we are seeing FPGA chips with many millions of gates, and we are beginning to realise the potential for implementing entire complex systems using reconfigurable logic alone. As ever though, the burden of complexity on the developer presents a significant bottleneck in the progress of this technology. Without the correct tools to help manage this complexity it is unlikely the potential of reconfigurable logic devices could ever be exploited to its full.

Some headway has been made in this direction, with the introduction of hardware description languages such as VHDL [88], Verilog [89] and more recently Handel-C [34]. However, one might still consider these languages as relatively low level, when compared to the kinds of languages available for software design. Additionally, such languages do not enforce a complete design methodology, and as such the behaviour of the system is still left very much in the hands of the individual developer. This means that there is still a significant bulk of effort required by the developer, who in the absence of any other guidance is likely to make design decisions on an ad-hoc basis, which may be prone to errors and not necessarily optimal in terms of efficiency.

Moreover, whilst reconfigurable logic has made significant improvements to the timescales required in hardware design, the process is still notably slower than conventional software design. Generating FPGA configurations - from compilation of hardware description languages, through to the place and route - is by no means a trivial task for the tools involved, particularly where complex designs are concerned. Compilation cycles are likely to be somewhat more lengthy than in a traditional software development setting - in the extremes this can be hours or even days. Whilst we may consider trial-and-error more of a feasible approach in reconfigurable logic than in the case of ASIC design, it is still likely to be unacceptably time consuming. Again, an approach which could produce designs which are provably error-free would be highly desirable.

Functional programming languages [20] provide a very high level of abstraction. They allow algorithms to be specified in terms of a relationship between their input and output, rather than by a sequence of language specific processing instructions, as is the case in the imperative setting. An algorithm expressed in the functional style is typically much closer to a specification than it is to an implementation in the imperative sense. However, a functional program still remains a machine-executable piece of code. This high level nature can greatly speed up the development process - one can often write in just a few lines of functional code a program that might take hundreds in an imperative context. Functional languages tend to capture the 'essence' of an algorithm, without bogging the developer down so heavily with the intricacies of syntax and behaviour introduced by imperative languages. The quality of referential transparency held by functional languages is also something of great usefulness to us. This allows any part of a functional program to be replaced by an alternative expression that we know to be functionally equivalent, safe in the knowledge that the overall functionality of the algorithm will remain the same. This makes functional languages an ideal framework for program transformation, and for proving properties about our algorithms. We therefore have scope to construct programs which are provably correct and guaranteed to satisfy conditions we impose upon them.

Such properties make functional programming languages ideal candidates for the high level specification of parallel hardware implementations.

1.2 Objectives

1.2.1 Overall Objectives

The overall objective of this research is to develop a methodology which reduces the complexity of engineering efficient parallel hardware implementations. This we can break down into two principal goals.

First, we wish to reduce the complexity of design involved in the development of hardware implementations. At the lowest level of a hardware implementation, the interaction between parts of a circuit are extremely difficult for a human engineer to comprehend, particularly given the speed and the presence of concurrency. The larger reconfigurable devices become, the larger the designs that can be implemented on them, and consequently this complexity increases dramatically. We might reasonably expect an engineer to be able to explicitly design a circuit for a few hundred or perhaps a few thousand gates, but when we begin to look towards designs involving millions of gates then this task becomes simply insurmountable. Abstraction must be employed to let the engineer manage the scope and size of such designs. Arguably the greater the level of abstraction, the larger the design that can be intellectually managed.

Secondly, we wish also to reduce the complexity inherent in quality assurance for hardware implementations. Traditional approaches to quality assurance for hardware implementations, such

as model checking and exhaustive testing, may well have suited designs involving a few thousand gates. However, these approaches will simply not scale when we come to consider designs involving millions of gates. Without sufficient methods to assert the correctness of systems implemented in large scale reconfigurable logic the scope for application of this technology will be severely limited. Reconfigurable devices could prove to be of great benefit in a number of application areas, and arguably at present their full potential is far from being realised. However, without appropriate techniques to ensure they will operate correctly they are not likely to be adopted in many of those areas, particularly those with a safety critical aspect, such as aviation.

So, we wish to develop a methodology which gives as high level of abstraction as possible, whilst also providing us assurance of correct implementation. Both of these goals should be realised with scalability in mind; with a view to accommodating ever increasing sizes of implementations. As in the field of conventional software design, we can naturally expect that increased sizes of implementations will result in increased build costs. However, when implementing in hardware this issue can be even more acute. Development tools for reconfigurable logic can take a significant amount of time - hours, even days - to compile large hardware designs. This, coupled with the added difficulties of debugging in hardware, makes the traditional "build and test" cycle of software design very poorly suited indeed to the development of hardware implementations.

Both the issues of engineering complexity and the requirement for quality assurance would be greatly assisted by a framework in which implementations could be derived which are correct by construction. That is to say, given a specification which we know to satisfy our requirements, we require a methodology with which we can formally derive a correct implementation from it. In other words we require a framework in which clearly understandable high level specifications can be taken and refined into hardware implementations by means of a process in which proof of correctness is maintained throughout.

1.2.2 Requirements of the Methodology

Such a methodology brings with it its own set of objectives or requirements, which can be seen as sub-objectives of this thesis. We shall examine these here.

- The methodology should facilitate the refinement from a high level functional language, for which we have chosen Haskell [20], to a language which allows implementation on an FPGA, for which we have chosen Handel-C.
- The methodology should allow for parallelism to be exposed and exploited during the refinement process.
- The methodology should provide alternatives for supporting different kinds of parallelism, of both the data and functional (pipe) varieties.

- The methodology will be required to provide a mechanism for relating between the functional style used in the specification and the behavioural style required by an implementation.
- The methodology should express behavioural definitions in a framework which allows formal reasoning about processes and communication, and for this we choose CSP.
- The methodology should allow us to move from CSP definitions to Handel-C implementations.
- The methodology should promote a library of powerful higher order components to promote code reusability.
- The methodology should be intellectually manageable.
- The methodology should facilitate the development of efficient implementations. We shall concern ourselves particularly with scalable efficiency - for example, quadratic time algorithms which, through parallelism, can be implemented in linear time with linear resources.
- The methodology should maintain correctness of refinement throughout, such that we can reassure ourselves that the final resulting implementations is correct with respect to our starting specification.

1.3 Contributions of the Thesis

The work in this thesis is based on that of Abdallah [1, 2, 3] in deriving the core of the approach for refining functional specifications into parallel networks of CSP [42] processes. The work presented in this thesis adds to and extends this approach in a number of ways, which we shall outline here.

Extension to Hardware Design. With the introduction of an additional refinement step to Handel-C, we have added scope for algorithms developed using this approach to be implemented on an FPGA. This provides us with a real platform in which parallelism can be exploited and efficient implementations can be realised. This also results in a framework which allows us to create provably correct implementations in reconfigurable logic.

Data Refinement. In the earlier work a list type in a functional specification had just one interpretation in the process setting - as a sequential series of messages on a single channel. In this work we identify this as a *stream*, which is just one of two possible strategies for refining a list into communication between processes. Here we also introduce the *vector*, a data parallel communication scheme. This allows us to tackle many algorithms which could not previously be addressed in a scalable manner. A good example of this is Cartesian product - this results in a quadratic sized output. With only sequential communication available to us we cannot possibly derive a truly scalable implementation. This scope to deal with data parallelism and manage quadratic sized data structures within the framework opens up many interesting patterns of computation, and these are explored in this work.

Library of Components. We expand on those higher order functions which had already been investigated in the earlier work (chiefly *map*, $fold$ and $filter$) by exploring how they may be interpreted in the vector setting, as well as in stream terms. Furthermore, we explore a number of additional higher order and list processing functions which had not previously been examined, often because their implementations were not practical prior to the introduction of vectors. In addition we explore how all of these building blocks may be implemented in Handel-C, to provide a library of provably correct components for hardware design.

Case Studies. Finally in this work we present a number of case studies which illustrate new algorithms and applications which may be tackled thanks to the extensions made to the approach. These include both academic style problems such as sorting, and also some real world applications - a JPEG decoder and an algorithm for DNA processing. We typically take quadratic time algorithms, and use our methodology to derive linear time implementations in reconfigurable logic. Our implementations not only achieve the goal of truly scalable efficiency, but are also provably correct.

1.4 Notation

Several different types of notation will be employed throughout this thesis, and these are introduced here.

1.4.1 Haskell

We shall use Haskell [37, 18] as the functional programming language in which our original specifications will be provided. We shall write Haskell code in italics, with identifiers typically in lower case, with the exception of types and type constructors which, following the Haskell convention, begin with a capital letter, and proceed in lower case. An example Haskell function is given below:

$$\begin{array}{lcl} inc & :: & Int \rightarrow Int \\ inc\ x & = & x + 1 \end{array}$$

1.4.2 BMF

At times we shall employ the Bird-Meertens Formalism [19, 16], which can be considered as an abstract functional language with a very close correspondence to Haskell. BMF is particularly well suited to reasoning about functional programs as it enjoys a rich set of laws for program transformation. In fact, BMF can be used more or less interchangeably with Haskell, the only notable exception being that typically in BMF the commonly used higher order functions are written as binary infix operators.

1.4.3 CSP

We shall use CSP [42] for the formal specification of processes derived from our functional specifications. We shall use italics again for CSP, however we will write process names all in upper case to distinguish them from the names of functions. An example CSP process is given below:

$$INC \quad = \quad in \; ? \; x \; \rightarrow \; out \; ! \; (x + 1) \; \rightarrow \; SKIP$$

1.4.4 Handel-C

Our final implementations shall be provided in Handel-C [34], which is then compilable to produce actual configurations for FPGA devices. Handel-C closely follows many of the underlying semantics of CSP, including the communication model, allowing us to implement many CSP specifications directly with only small syntactic changes required. We shall use a fixed size font for Handel-C code, again with process names given all in upper case, as illustrated below:

```
macro proc INC (in,out)
{
   Int x;
   in ? x;
   out ! x + 1;
}
```

1.5 Related Work

The research reported in this thesis has been profoundly influenced by the work of numerous computer scientists on functional programming [16, 20, 18], transformational programming [53, 19, 15], concurrency [42, 26], declarative hardware description languages [24, 58], and hardware compilation techniques [34, 76].

In particular, the work of Abdallah [1, 2, 3] on devising the core of a transformational programming methodology for the synthesis of message-passing parallel algorithms in CSP from high-level functional specifications.

Other relevant related work includes formal frameworks and tools for hardware description, verification [32], testing and model checking FDR [30], as illustrated by Peleska [77], Gordon, Bowen, Josephs [50] and Jifeng [46].

In the remainder of this section we shall look at an overview of other frameworks for developing hardware designs. Principally we shall look at frameworks which are based on functional languages such as Haskell and MAL. These frameworks include μFP, Lava and SAFL. We shall also examine other frameworks which, although not based on these conventional functional languages, are still functional in nature, and share similar goals of abstraction and componentisation. As part of this

we look at techniques inspired by the work done on systolic arrays and regular architectures. We shall also take a look at the broader field of program transformation and refinement.

1.5.1 Functional Frameworks for Developing Hardware Designs

mu FP

In her early work Sheeran [86] proposed μFP (a variant of Backus's functional programming language FP [10]) for specifying the behaviour of *VLSI* circuits and their geometric layout. In this framework, a proof that two circuit descriptions have the same semantics may be obtained by transforming one of the description to the other using μFP algebraic laws. The approach has been applied by Jones and Sheeran [48] in order to derive parameterized representation of regular synchronous circuits from their specifications.

Lava

The Lava language [24], developed at Chalmers by Sheeran et al, is a hardware description language based on Haskell. Although Lava may have been intended originally as an experimental tool to aid research into hardware verification [23], a version of the language is also used at Xilinx in an industrial capacity for the implementation of real world applications on FPGA devices [87]. Lava has built on earlier work, in particular the languages μFP [86], and Ruby [84, 47, 49]. Guo and Luk developed a method for compiling Ruby programs onto an FPGA [33]. Ruby however differs from Lava in one significant aspect, that the abstract specifications it uses are relations rather than functions.

Functional programs in Lava are used to describe circuits directly, and as such specifications are at a much lower level than in this work. For example, the following Lava program describes a half adder circuit:

```
import Lava

halfadd (a, b) = (sum, carry)
  where
    sum   = xor2 (a, b)
    carry = and2 (a, b)
```

Lava provides a suite of attractive features to the prospective circuit developer. First we have a Haskell library of commonly used hardware circuit components. Examples of these include the `xor2` and `and2` components used above.

Also provided is the ability to perform simulations of a Lava circuit description from within the Haskell environment. Given a circuit description and a set of input values, an appropriate simulation function can then be employed to determine the output of that circuit. This coupled with the rich expressiveness of Haskell gives scope for very powerful automated testing. This rapid

and immediate testing of a circuit design makes Lava an environment which is particularly well suited to prototyping.

As an extension of the above, Lava allows properties to be defined and attributed to circuits. These properties allow the developer to specify logical constraints for the circuit. Lava then supplies a verification function which can hook into external theorem provers to determine whether or not a given property is observed by the circuit definition.

However, circuits designed in Lava are by no means limited to prototyping, verification and simulation. As already noted, Lava has been used to design and implement real world applications [87]. Therefore an important feature of the Lava environment is the ability to generate VHDL [72] from Lava circuit descriptions. These VHDL hardware descriptions are then implementable on FPGA devices.

One interesting area of analogy between this work and Lava is in Lava's notion of *connection patterns*. These are effectively "higher order circuits" and share some conceptual similarities with the notions of higher order functions/processes used here. For example, the `serial` connection pattern in Lava is defined as follows:

```
serial circ1 circ2 a = c
 where
   b = circ1 a
   c = circ2 b
```

Here we have a higher order circuit which takes in two other circuits as parameters and composes them together sequentially. As such the output of one circuit forms the input to another. This bears analogy to the concept of process feeding (see Section 4.2), which models function application; or process piping (see Section 4.4) which models function composition.

Similarly other connection patterns in Lava bear resemblance to the higher order functions/processes used in this work. Lava's `map` connection pattern has an obvious correspondence to our vector interpretation of *map*. Furthermore, Lava's `tree` connection pattern corresponds to the familiar funnel network presented here as a potential refinement of $fold$ in the vector setting. Finally, Lava's `row` connection pattern, has functionality similar to certain interpretations of either $fold$ or $unfold$.

However, although these connection patterns do bear analogy to some of our higher order processes, their treatment is somewhat different in Lava. The approach in Lava is somewhat more bottom-up than that of this work. In Lava, these patterns have been identified as useful constructs in circuit design, but not necessarily linked to the higher order functions that they represent. In this work we have chosen instead to start with the higher order functions and explore from there the different ways in which their functionality can be represented in hardware, with an emphasis on ensuring correct refinement throughout.

In summary, the principle difference between the approach of Lava and that of this work is in the level of abstraction. Lava advocates defining the behaviour of circuits directly in a functional framework. In this work we suggest instead to synthesise the behaviour of the eventual hardware

implementation from a high level functional specification.

SAFL

SAFL [71] (and its successor SAFL+ [83, 81, 82]) is a first order functional language designed for hardware description. It was developed principally by Sharp and Mycroft at the University of Cambridge. SAFL takes its functional aspects from ML [65] rather than Haskell and also shares some similarities with the HardwareC language [52]. SAFL+ implements synchronous channels as found in Handel-C, however they are treated with more generality. A SAFL+ channel is not restricted to a single transmitter/receiver pair as in Handel-C. An excerpt of SAFL+ demonstrating how channels are dealt with is given below:

```
fun Accumulate(state) [c] =
  let val read_value = c?
    in if read_value=0 then state
                       else Accumulate(state+read_value) end

fun GenNumbers(state) [c] =
  c!state; if c=0 then () else GenNumbers(state-1)

fun sum(x) =
  static channel connect
  in GenNumbers(x) [connect] || Accumulate(0)[connect] end
```

Here channels can be passed to `fun` constructs by way of a list of parameters. These are kept intentionally separate from parameters which represent normal data (note the use of round and square brackets). The above excerpt demonstrates how functions in SAFL+ represent a mixture of functional and imperative/behavioural definitions. In this work we strive to keep the two separate, given that, arguably, functions are best reasoned about in a purely functional environment (such as Haskell), and processes are best reasoned about in a purely process oriented environment (such as CSP).

SAFL abstracts away certain hardware details which must be made explicit in a Lava design. As such it may be suggested that SAFL is a higher level description language than Lava. However, reasoning about circuits is still done directly in the functional framework, and this is the one main way in which the approach of SAFL differs from that of this work.

Pebble

Pebble [58], short for *Parameterised Block Language*, is a hardware design language with an emphasis on reusability and efficiency. It was developed by Luk, McKeever et al, at Imperial College London. Pebble was born in part of earlier work to develop a parameterised, re-usable library for FPGA development [59]. Attempts to develop this kind of library in existing versions of VHDL were presumably one of the main areas highlighting the need for a new hardware description lan-

guage. Pebble can be considered as a much simplified variant of VHDL, but with greater support for parameterisation in an attempt to promote better reusability.

Pebble programs are defined as blocks. An example block defining a half adder circuit is given below.

```
BLOCK hadd (x,y:GENERIC) [a,b:WIRE] [cout,sum:WIRE] BEGIN
  xor2 [a,b] [sum ] MAP rloc IS "X,x,Y,y,";
  and2 [a,b] [cout] MAP rloc IS "X,(x+1),Y,y,"
END;
```

As with Lava, Pebble programs can be compiled into VHDL allowing implementation on an FPGA. The Pebble compiler also supports direct compilations to netlists. One of the major benefits of Pebble is the ability to target different variants of VHDL, allowing greater portability of library code when differing flavours of hardware description are required for the implementation. VHDL is a surprisingly non-standard language, there are many different variants, and porting code from one type of VHDL to another can be extremely laborious. By implementing library code in the more general Pebble, this porting effort is greatly reduced.

Other Functional Frameworks

SMALL [74] is a programming language originally intended for state machine design, which is also capable of generating FPGA configurations [85]. The approach in this work shares some similarities with Handel-C and its predecessors [76]. Although an imperative language, as an interesting aside the SMALL compiler is implemented in Haskell.

Jazz [45] is a language and framework designed for hardware description/synthesis, which has some similarities with Lava. Although it is not built on pure functional foundations as Lava is, it does support a polymorphic type system through an object oriented approach to hardware design. Hawk [64], is a framework for the description and verification of microprocessors. Like Lava, it is effectively a set of libraries based in Haskell. The emphasis here is more towards simulation rather than actual development, however. HML [75] is a hardware description language based on ML.

Frameworks Based on Regular Arrays

Manjunathaiah and Megson described a technique for designing hardware components, as regular array architectures, for high-throughput embedded systems applications [62]. The method is based on deriving component designs using classical regular (or systolic) array synthesis techniques and composing these separately evolved component design into a unified global design. A tool for regularizing systems of affine recurrence equations (SARE) into uniform recurrence format has also been developed [61]. The tool supports the designer's task of specifying algorithms and for synthesizing regular arrays.

1.5.2 Program Transformation and Refinement

Bird and Meertens have developed a calculus for functional algorithmic manipulations [19, 16]. de Moor [15] has generalized this calculus to the relational level and Gibbons [21] has, among others, shown its applicability in the derivation of several challenging problems.

Although this calculus has been mainly used for deriving efficient functional programs from their specifications, the final form can also be used as a starting point for deriving efficient imperative programs or parallel programs.

The formal concept of refinement can be traced to Dijkstra [29]. Hoare, Morgan, Sanders, Woodcock and Jifeng have formalized the concept of data refinement [43] whilst Morgan provided a full framework for specification refinement. Both the specification and the program are expressed in the same (pre/post condition) notation. Davies and woodcock showed how data refinement can be done on small Z specifications. Barros showed, in her PhD thesis [12], how large Z specifications be systematically refined into functional programs using careful composition of data refinement and functional refinement steps. Abdallah [1] showed using algebraic methods how to formally refine functional programs into pipelines of CSP processes.

Page has shown how to compile high level circuit descriptions, expressed in the programming language Handel-C [34], into hardware.

1.6 Thesis Overview

The remainder of this thesis is divided as follows.

Chapter 2 gives a general overview to the style of the methodology presented in this work. Here we begin to explore the relationship between functions and processes, between data and communication, and give a general feel for the methodology.

Chapter 3 presents the first step of the methodology, data refinement. This looks at how data types in the specification can be refined into different models of transmission in the implementation, thus beginning to define the behaviour and scope for parallelism.

Chapter 4 looks at process refinement; how we can move from a functional specification to a behavioural implementation. Here we give criteria for the correctness of process refinement and look at how different forms of functional definition can be refined to processes.

Chapter 5 introduces two key higher order functions - map and $fold$ which will constitute the 'building blocks' of our implementations, and considers each of them in turn with respect to our data refinement strategies presented in the previous chapter.

Chapter 6 expands this library with a number of other powerful components such as $filter$ and $unfold$.

Chapter 7 looks at a particular subset of components within out library of components - those that refine list processing functions.

Chapter 8 details the refinement procedure - the steps that a developer would take in order to derive an implementation using this methodology.

Chapter 9 presents some case studies demonstrating the application of this methodology.

Chapter 10 concludes the thesis. We first discuss the potential for future work. Following this we summarise the findings of this research and evaluate the success of the methodology created.

Chapter 2

Methodology Overview

2.1 Introduction

The field of computing has, quite naturally, resulted in the development of a diverse array of different paradigms and frameworks. The number of programming languages in existence today runs to many thousands, and the difference between these is of course not purely syntactic. These languages are generally designed with a particular problem area or implementation environment in mind. As such a language promotes not just a means of communication with a processor, but also an underlying ethos and design methodology. With so much diversity in these methodologies, they understandably all have their relative strengths and weaknesses. It is therefore quite common to employ more than one language in a design process - this of course happens implicitly every time a compiler is involved. This transformation process can also be a human one though - for example the interpretation of a formal specification to produce a concrete implementation.

The two classic goals of language design are abstraction and efficiency. Unfortunately these tend to pull in opposite directions, and often the provision of one will be at the detriment of the other. As a result these characteristics will feature heavily in the comparison of any two languages or methodologies. Listen to any two programmers arguing about the relative merits of their favourite language and almost certainly the battle ground will be drawn over these two areas.

2.2 Functions and Processes

Functional languages such as Haskell provide a very high level of abstraction. They allow us to think more in terms of relationships between input and output, rather than defining a strict series of actions to facilitate the computation, as is the way in imperative languages. In many ways functional programs can be considered as closer to specifications than implementations. They are, however, implementable, given an appropriate compiler or interpreter. The level of abstraction does of course come at a price - functional implementations are typical less efficient than their imperative

counterparts. In addition, the overheads brought about by this high level of abstraction mean that they are currently not generally considered good candidates for implementation directly on novel architectures such as the FPGA.

The world of processes, as inhabited by CSP, allows us to reason about behaviour and interaction in a manner that functional languages are, quite deliberately, not equipped to address. In the world of processes we can talk explicitly about parallelism. Thanks to the language Handel-C, such process definitions are directly implementable onto an FPGA.

The requirement therefore seems clear - ideally we would like to exploit the abstraction, clarity, correctness and ease of design inherent in the world of functions, whilst still being able to take advantage of the efficiency of implementation in hardware via the world of processes. We would like a framework that would give us the best of both worlds - the ability to specify an algorithm functionally and from that derive a behavioural implementation in processes.

Given that they exist in different worlds, we cannot of course provide direct equivalences between functions and processes. However, we can still move from one space to the other via refinement. So, we cannot prove that a process implementation is *equal* to a functional specification, but we may be able to supply criteria to prove that it correctly *refines* it.

In our functional space there are two types of entities that we shall need to consider, as demonstrated in Figure 2.1.

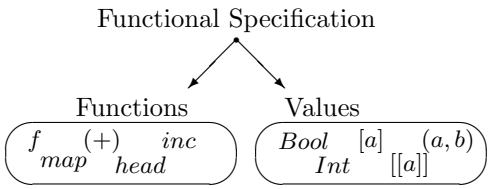


Figure 2.1: Functions and Values.

So, we have both functions and data types to deal with. How will these manifest themselves in our implementation? In effect this brings about two parts to deriving our implementation.

For values or data types in our specification, we have the task of *data refinement* to carry out. This is concerned with determining how values passed around in our functional specification will be dealt with in our implementation. Different architectures naturally result in different implementation issues, and as such certain data structures are better suited to some architectures than others. For example, whilst linked lists may be a convenient structure in a traditional CPU based architecture, they are not nearly as straightforward to implement on an FPGA. However, it is often not required to have a representation in our implementation that exactly mimics the behaviour of the type in the specification. What we are actually trying to achieve is just to preserve those abstract characteristics of our data structures which are important to us. So, for example, a list is a structure that contains a group of items. Typically we assume this should have an ordering,

but even this most basic characteristic may not be essential for our implementation to function correctly. We may or may not require that items can be inserted into (or removed from) the list in constant time; the list may or may not be required to change in size over time, and so on.

In a message passing environment it seems likely that at least some values will be refined to communication between processes. How will this communication occur? For simple types, such as booleans or integers, this may be a fairly straightforward decision. For more complicated structures, such as lists and lists of lists, we may need to think more carefully about the alternatives available to us. We may want to consider how these alternatives will effect the efficiency of our implementation, and its scope for parallelism. Correctness is also of great importance. It will be crucial to provide criteria to prove that any data refinement is valid, such that we can re-assure ourselves that our final implementation correctly implements our specification. Data refinement is investigated further in Chapter 3.

For functions in our specification, we have the task of *process refinement* to carry out. This is concerned with mapping from the input/output relationships given by functions to the more concrete behaviour of processes in the implementation. Again here, correctness is of the upmost importance, and so we will require criteria to prove that a given function is correctly refined by the process we chose to implement it with. Process refinement is investigated further in Chapter 4.

2.3 Example

Let us look at a general example of this refinement process. Consider that we have a function f that we wish to refine to a process. The function f has the following type:

$$f \;::\; A \to A$$

So, f takes in some value of type A and returns a value of the same type. Let us provide a specific definition for f, which will imply that A is in fact some form of integer type:

$$f\ x \;=\; x + 1$$

Let us consider the data refinement aspect of this procedure. In the specification, we have a value x, of type A, which is passed as a parameter to the function f. In our implementation we would like this value to be refined to some form of communication. For this, we shall require that the specification type A is refined to some type which can be communicated. Depending on the type we are refining, we may have more than one option for a method of communication. Let us assume, for the purposes of this example, that values of the type A can be communicated "as is".

So, in order to refine our data type to communication, we will need to appeal to Prd, short for 'produce'. This will be discussed in more detail in Chapters 3 and 4. Although Prd may at first appear to be a process, it is in fact a function *which returns a process.* Once given its parameter

(the value to produce) we then have a process in a particular state. In CSP terms we have the following:

$$Prd(x) \ = out\ !x \ \rightarrow \ SKIP$$

In effect, this represents our data refinement. The input value x in our specification is refined to the communication of a value x on the channel *out*.

We now require a process refinement for the function f. An obvious candidate is a simple process which has two channels. As such our candidate process P has the following alphabet:

$$\alpha P \ = \ [in, out]$$

The behaviour of the process P is very simple. It reads a value on its input channel, and then outputs that value plus one on its output channel. In CSP we have:

$$P \ = \ in?x \ \rightarrow \ out!(x+1) \ \rightarrow \ SKIP$$

Informally, we may be content to visually inspect this definition and reassure ourselves it implements the required functionality. However, for the sake of correctness it is necessary to introduce some formal criteria for asserting that the process P is indeed a valid refinement for the function f.

The criteria for refinement pivots around the special function/process Prd. Additionally for our process refinement criteria we will require a CSP operator which can link the output channel of one process to the input channel of another. This can be achieved with the 'feed' operator, represented by the $\triangleright$ symbol. This can be defined as follows, using CSP's channel renaming and hiding operators:

$$P \triangleright Q = (P[mid/out] \ || \ Q[mid/in])\backslash\{mid\}$$

In effect the feed operator models function application in the process world. Given this we can supply a criteria for valid refinement as follows:

$$Prd(x) \triangleright P = Prd(f\ x)$$

That is to say that the process P refines the function f if the act of producing x and feeding it to P is equivalent to producing $(f\ x)$ directly.

Through substitution into the left hand side of the above rule, we have:

$$(out\ !x \ \rightarrow \ SKIP) \triangleright (in?x \ \rightarrow \ out!(x+1) \ \rightarrow \ SKIP)$$

Expanding the definition for the feed operator gives us:

$$((mid\ !x \ \rightarrow \ SKIP) \ || \ (mid?x \ \rightarrow \ out!(x+1) \ \rightarrow \ SKIP))\backslash\{mid\}$$

As both sides of channel *mid* are willing to communicate, and the channel is hidden, we can simplify to the following:

$$out!(x+1) \ \rightarrow \ SKIP$$

This, of course, is equivalent to $Prd\ (f\ x)$

2.4 Summary

We have seen here that with the aid of some basic constructs like the feed operator and Prd, we can move from the world of functions to the world of processes. These constructs have only been introduced in very general terms here to give an overall feel for the methodology advocated in this thesis. They will be dealt with in more depth and detail later. The important thing to note, though, is that we can be assured to be maintaining correctness throughout development. At each refinement step we can provide proofs to reassure ourselves that our implementations correctly refine our specifications. In this way we are able to benefit from the best of both worlds - the high level abstract nature of functional languages to aid in the specification and design, and the lower level nature of languages like Handel-C which facilitate implementation in hardware and therefore greater performance and reliability.

Chapter 3

Data Refinement

3.1 Introduction

One significant difference between the functional and process environments will be the manner in which data is passed between components. In functional languages the flow of data is in parameters to functions. When implementing in a message passing environment, this passing of parameters may correspond to communication between processes. There may be more than one option available to us when it comes to determining how this communication should take place. The choice of communication method may have a significant impact on the efficiency of the implementation. In this Chapter we shall take a detailed look at data refinement, how values and types in the specification are represented in the implementation.

3.2 Refinement in General

It may be useful to introduce the notion of refinement in general terms as used in this work. In this section we shall not talk specifically about refining to types suitable for use in reconfigurable logic - we shall leave this for Section 3.3. We shall instead just discuss some illustrative examples.

Generally refinement will be employed as a means to move from an abstract specification towards a concrete implementation. Let us consider an example. Let us assume that the set is an abstract type. A set containing elements of type A can be written simply:

$$\{A\}$$

Given that the set is an abstract type, our implementation environment will not allow us to implement it directly. We shall assume our environment does, however, have lists as a built-in type. A list containing elements of type A can be written:

$$[A]$$

Let us consider employing the list as a refinement for the set. First, it is important to introduce an abstraction function. This allows us to move from a value of our concrete type to a value of our abstract type. In many cases, the inverse of this function will be a relation. That is to say, a single value in the abstract type may have several corresponding values in the concrete type. Our abstraction function, named *abs*, will be of the following type. Note often it is wise to give some more detail in the name of this function to describe the types it will abstract to and from. Here, however, for simplicity, we shall keep to merely *abs*:

$$abs \ :: \ [A] \rightarrow \{A\}$$

Informally, we shall give it the following definition:

$$abs \ [x_1, x_2, ..., x_n] \ = \ \{x_1, x_2, ..., x_n\}$$

3.2.1 Example - Set Maximum

Now for an example refinement of a function within this setting. The function *setmax* takes in a set of values and returns the highest value in the set:

$$setmax \ :: \ \{A\} \rightarrow A$$

Given the binary maximum operator ($\uparrow$), we can define it informally as follows:

$$setmax \ \{x_1, x_2, ..., x_n\} \ = \ x_1 \uparrow x_2 \uparrow ... \uparrow x_n$$

In the list setting, we have a potential refinement of *setmax* we shall call *listmax*.

$$listmax \ :: \ [A] \rightarrow A$$

An informal definition is similar to that for *setmax*.

$$listmax \ [x_1, x_2, ..., x_n] \ = \ x_1 \uparrow x_2 \uparrow ... \uparrow x_n$$

For *listmax* to be deemed a valid refinement of *setmax* we shall require that the act of evaluating *listmax* in list terms is equivalent to evaluating *setmax* in set terms. In other words we require that the result of *listmax* applied to a given list is the same as the result of first abstracting the list to a set, and then applying *setmax* to that. We can capture the required equivalence in the following diagram:

$$\begin{array}{ccc} \{A\} & \xrightarrow{setmax} & A \\ \uparrow{\scriptstyle abs} & & \uparrow{\scriptstyle id} \\ [A] & \xrightarrow{listmax} & A \end{array}$$

We shall use this kind of diagram frequently in this chapter, and so it may serve here to illustrate how exactly it is to be interpreted. The starting point is the bottom left hand corner, where in this case we have the type $[A]$. The end point is the top right hand corner, where we have the type A. The arrows then describe two possible routes. The first (up, then across) corresponds to the expression $(setmax \circ abs)$, and the second (across, then up) corresponds to the expression $(id \circ listmax)$. The intermediate types for each of the two routes are also given - in the top left and bottom right hand corners respectively. The question that the diagram poses is whether or not taking either of the two routes will arrive at the same result. That is to say, for any list of type $[A]$, will we get the same result (of type A) by applying $(setmax \circ abs)$ as we would by applying $(id \circ listmax)$?

In other words our task is to prove that this diagram commutes, that is, we can move from the bottom left hand corner, to the top right, via either route, and arrive at the same value. Note no refinement is made of the result type (a single value of type A), so only the identity function is used to move between this in the specification and implementation. In this case, the proof will follow simply from a series of substitutions, based on the definitions previously presented:

$$\begin{array}{lll} & (setmax \circ abs)\ [x_1, x_2, ..., x_n] & \{def.\} \\ = & setmax\ \{x_1, x_2, ..., x_n\} & \{def.\ abs\} \\ = & x_1 \uparrow x_2 \uparrow ... \uparrow x_n & \{def.\ setmax\} \\ = & listmax\ [x_1, x_2, ..., x_n] & \{def.\ listmax\} \\ = & (id \circ listmax)\ [x_1, x_2, ..., x_n] & \{def.\ id\} \end{array}$$

Thus we can state formally that $listmax$ is a valid refinement of $setmax$.

3.2.2 Example - Adding to a Set

As another example, let us consider the function $addset$. This takes in a value of type A, and a set, and returns the set with the value added. This has the following type:

$$addset :: A\ \rightarrow\ \{A\} \rightarrow \{A\}$$

Another way to look at this, by way of currying, is as the function $(addset\ a)$. This takes in a set, and returns that set with the value a added to it. This has the following type:

$$(addset\ a) ::\ \{A\} \rightarrow \{A\}$$

We can define it informally as follows:

$$addset\ a\ \{x_1, x_2, ..., x_n\}\ =\ \{a\} \cup \{x_1, x_2, ..., x_n\}$$

In a list setting we may wish to consider more than one possible implementation, depending on our exact interpretation of the set implemented in list terms. Where our list implementation of

the set is unordered and we are not concerned about duplicates, we can implement using the cons operator (:)

$$addlist\ a\ [x_1, x_2, ..., x_n]\ =\ a\ :\ [x_1, x_2, ..., x_n]$$

Should we wish to guard against duplicates in our list based representation of the set we may wish to offer a different version:

$$addlist'\ a\ [x_1, x_2, ..., x_n]\ =\ a\ :\ filter\ (\neq a)\ [x_1, x_2, ..., x_n]$$

Alternatively, where our list based representation is ordered, we may want to instead choose the following:

$$addlist''\ a\ [x_1, x_2, ..., x_n]\ =\ insert\ a\ [x_1, x_2, ..., x_n]$$

3.3 Transmission Values

Efficiency in an implementation, and in particular the scope for parallelism, will be heavily influenced by the way in which the types in the specification are refined. When refining to a message passing environment, we need to consider how values passed as parameters to functions in the specification correspond to communications between processes in the implementation.

In functional programming terms, the concept of a *value* is very broad. So, an integer is a value, and so is a list of integers, and indeed a list of lists of integers. It is our job here to consider the counterparts of these values in the implementation environment.

We shall employ the term *transmission values* to encapsulate all values that are communicable in our target environment. We may also occasionally term such values simply *transmissions* for brevity. The different kinds of transmission values - items, streams, vectors and so on, will be introduced in the following sections.

We shall be chiefly concerned with alternatives for refining the list, a type intrinsic to most functional specifications. The list forms a good abstraction for most linear types, including the array, commonly found in imperative programs.

3.3.1 Nothing

A function with a given return type must return a value of that type. A process, however, is not necessarily obliged to output anything. In some cases we will find it useful to encapsulate a special transmission value which results in no output, the value *nothing*. This value is valid in place of a transmission value of any other type, be it an item, stream, vector or other transmission. We shall see examples of its use later.

3.3.2 Items

Items are the simplest form of transmission value that correspond to some actual output. Items encapsulate single, fixed size values such as integers, characters and booleans. We shall generally assume items to be atomic - they require only a single action to communicate.

We may occasionally need to make a distinction between items, as transmission values, and the functional values they represent in the specification. Given a type A in the specification, the corresponding item type can be written:

$$\dot{A}$$

Alternatively, using a Haskell style syntax, we have the type:

$$Item\ a$$

We shall use these two forms interchangeably.

3.3.3 Streams

The stream forms one of our two principal strategies for refining a list. The stream is a purely sequential method of communicating a group of values. It comprises a sequence of messages on a channel, with each message representing a value. Values are communicated one after the other, so communication of the entire structure will require linear time with respect to the size of the list. It is assumed that the receiving process does not know the size of a given stream in advance, so the sending process will have to notify it somehow when transmission is complete.

Let us consider the notation we shall use for representation of the stream type. Given some type A, a stream containing values of type A is denoted simply:

$$\lfloor A \rfloor$$

Alternatively, using a Haskell style syntax, we have the type:

$$Stream\ a$$

Again, we shall use these two forms interchangeably.

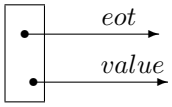


Figure 3.1: The Stream

Depending on the implementation environment, there may be several different strategies for signalling the end of transmission (EOT). Given knowledge of the type of values being communicated, one might intuitively suggest to set a side a rogue value to represent EOT. This approach

has its disadvantages, however. Not only does it lack genericity, we may often encounter situations where all possible values in the given type are used - for example, booleans. Another alternative might be to pair every value communicated with an additional boolean value to determine whether or not it indicates EOT. Although this should work with values of any type, it does introduce a communication overhead which may be unacceptable in many settings. Perhaps the most generic approach, with the least overhead, is to introduce a second channel on which the end of transmission can be signalled. After all of the values in the stream have been communicated along the value channel, a single bit is communicated along the *eot* channel to signal the end of transmission. This mechanism is depicted in Figure 3.1.

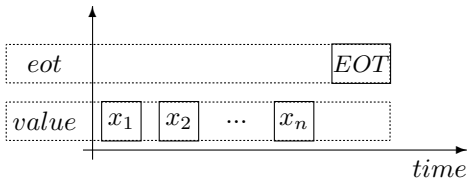


Figure 3.2: Messages required over time to communicate a stream.

Figure 3.2 illustrates the sequence of messages which occur in order to communicate a stream using this model. It is important to make clear some assumptions about this communication strategy, and we shall discuss these here. The values in the stream (here $x_1, x_2, ..., x_n$) are transmitted in order, one after the other, on the *value* channel. As is illustrated on the diagram there may be a gap in time between consecutive values being transmitted. Furthermore this gap may not necessarily be constant. After the last value (here x_n) has been transmitted the producer has the responsibility of then signalling the end of transmission on the *eot* channel. Again, there may be a gap in time between the last value being transmitted and the end of transmission being signalled. The responsibilities of the producer and consumer with respect to a stream can therefore be clearly mapped out. The producer must only be willing to engage in output on the *value* channel, until the last value has been transmitted. After this it must only be willing to engage in output on the *eot* channel, and after a successful transmission of the *EOT* signal it may then terminate. The consumer must be willing to engage in input on either the *eot* or *value* channels throughout transmission. Upon an input occurring on the *eot* channel it then terminates. We can express these roles formally in CSP, defining two behavioural specifications. All producers and consumers of streams must adhere to the appropriate one of these specifications if they are to correctly implement their part of the stream transmission procedure. The alphabets of these processes must contain the same two channels, as illustrated below:

$$\alpha SPRODUCE,\ \alpha SCONSUME \quad = \quad \{value, eot\}$$

First, the definition for the producer:

$$
\begin{array}{lcl}
SPRODUCE\ (Stream\ (x : xs)) & = & value\ !\ x\ \rightarrow SPRODUCE\ (Stream\ xs) \\
SPRODUCE\ (Stream\ [\,]) & = & eot\ !\ EOT\ \rightarrow SKIP
\end{array}
$$

For the consumer we have the following:

$$
\begin{array}{lcl}
SCONSUME & = & value\ ?\ x\ \rightarrow\ SCONSUME \\
 & & | \\
 & & eot\ ?\ EOT\ \rightarrow SKIP
\end{array}
$$

3.3.4 Vectors

The vector is an alternative refinement for a list. Whereas the stream implements a totally sequential mechanism, the vector is a totally parallel one. Each item to be communicated by the vector will be dealt with independently in parallel. A vector refinement of a simple list of items will communicate the entire structure in a single step, in constant time. The vector is depicted in Figure 3.3.

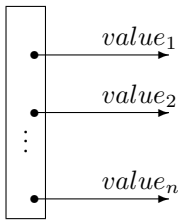


Figure 3.3: The Vector

Given some type A, a vector of length n, containing values of type A, is denoted:

$$\langle A \rangle_n$$

Alternatively, using a Haskell style syntax, we have the type:

$$Vector\ a$$

We may consider the value n, the size of the vector, to be part of the type of the vector. Vectors must be of the same size to be deemed 'compatible'. Given two differing values n and m, the type $\langle A \rangle_n$ is not the same as the type $\langle A \rangle_m$.

Figure 3.4 illustrates the messages required to communicate a vector of items. Whilst all the items must be communicated independently in parallel, we do not however assert that they must all be transmitted at exactly the same point in time. See Section 3.4.2 for an example of where this may be a desirable feature. The communication of the structure as a whole is complete when the last (i.e. slowest) component item has been transmitted. As with the stream case, the producer

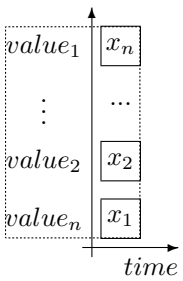


Figure 3.4: Messages required over time to communicate a vector.

and consumer at either side of the vector communication must adhere to a particular behaviour. This can be defined by the following two CSP processes. The alphabets of these processes must contain the same set of channels, as illustrated below:

$$\alpha VPRODUCE_n,\ \alpha VCONSUME_n \quad = \quad \{value_1, value_2, ..., value_n\}$$

First, the producer:

$$VPRODUCE_n\ (Vector\ v) \quad = \quad \underset{i=1}{\overset{n}{||}} \quad value_i\ !\ v_i \rightarrow SKIP$$

Secondly, for the consumer:

$$VCONSUME_n \quad = \quad \underset{i=1}{\overset{n}{||}} \quad value_i\ ?\ x_i \rightarrow SKIP$$

3.3.5 Distributed Lists

One additional strategy to consider when refining a list is to partition it into sub-segments, which may then be viable for independent communication. In essence, by employing distributed lists, we are introducing an additional refinement step which may add scope for parallelism. A distributed list refinement will refine from a list into a list of lists. This will not be directly usable in our implementation - we will then have to perform a further refinement step into some combined structure, described below.

3.4 Combined Transmission Structures

Whenever dealing with multi-dimensional data structures, for example, lists of lists, we need to think carefully about the alternatives for implementation and the consequences that may arise.

Effectively implementation options arise from differing compositions of our "primitive" data refinements - streams and vectors. We shall consider just two dimensions here, however there is no theoretical barrier to prohibit the refinement of three or more dimensional structures in a similar manner.

3.4.1 Stream of Streams

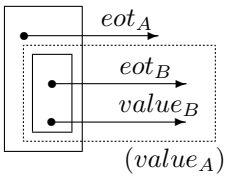


Figure 3.5: The Stream of Streams

The stream of streams is a totally sequential means of communicating a list of lists, and as such will offer us the least scope for data parallelism. This is depicted in Figure 3.5, and as the diagram hopefully illustrates, this is in effect a stream embedded in another stream. Here a single channel is used to communicate every single value, one at a time. Additionally, two separate means of signalling EOT are required. One to denote that each sub-segment of the list has been fully communicated, and another to denote that the communication of the structure as a whole has completed. To illustrate this, consider the following stream of streams.

$$\lfloor \lfloor x_1, x_2, ..., x_k \rfloor, \lfloor y_1, y_2, ..., y_m \rfloor, \lfloor z_1, z_2, ..., z_n \rfloor \rfloor$$

This would correspond to the sequence of messages depicted in Figure 3.6:

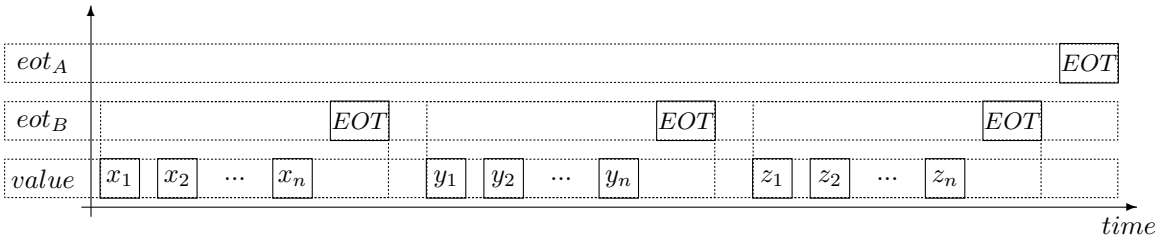


Figure 3.6: Messages required over time to communicate a stream of streams.

Using this method, for a quadratic sized structure, communication will require quadratic time. However, only a single processing element will be required.

3.4.2 Vector of Streams

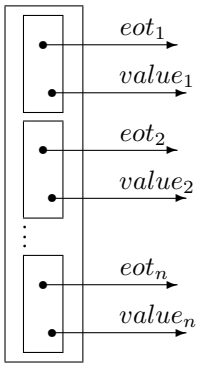


Figure 3.7: The Vector of Streams

The vector of streams, depicted in Figure 3.7, communicates a number of streams independently in parallel. Each stream will have its own value channel and an independent means of signalling EOT. If we consider this mechanism as a refinement for a lists of lists, we may observe that it provides us the ability to cope with sub-lists of differing lengths To illustrate this method of communication, consider the following vector of streams.

$$\langle \lfloor x_1, x_2, ..., x_k \rfloor, \lfloor y_1, y_2, ..., y_m \rfloor, \lfloor z_1, z_2, ..., z_n \rfloor \rangle_3$$

Communication of this structure can be considered as three processes composed together in parallel. Each process has a separate value channel, and a separate means of signalling EOT. The first process communicates the values $x_1, x_2, ..., x_k$ in order and then signals EOT. The second communicates the values $y_1, y_2, ..., y_m$ in order and then signals EOT. The third communicates the values $z_1, z_2, ..., z_n$ in order and then signals EOT. Communication of such a quadratic sized structure in this way will take linear time and require linear processing elements.

The messages required to communicate a vector of streams are illustrated in Figure 3.8. It is important to note here that the streams work quite independently of each other. One component stream may well take longer than another to transmit - it may contain more items, for example. Communication of the structure as a whole completes when the last component stream has been fully transmitted.

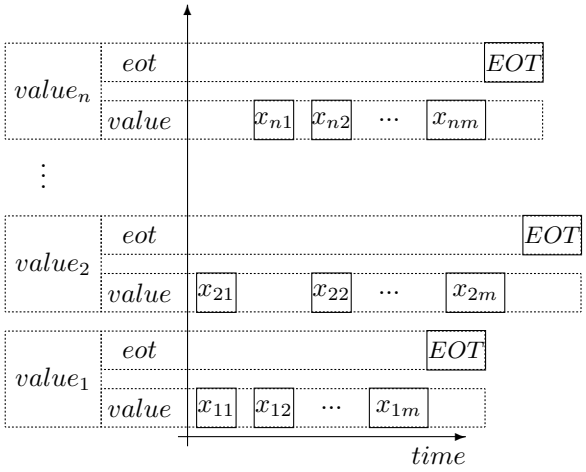


Figure 3.8: Messages required over time to communicate a vector of streams.

3.4.3 Stream of Vectors

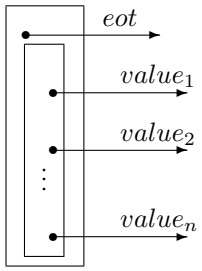


Figure 3.9: The Stream of Vectors

The stream of vectors is a sequence of communication stages, where at each stage a number of values are communicated at the same time in parallel. As a refinement for a list of lists, it is only applicable where the sub-lists are of equal length. To illustrate this method of communication, consider the following stream of vectors.

$$\lfloor \langle x_1, x_2, ..., x_n \rangle_n, \langle y_1, y_2, ..., y_n \rangle_n, \langle z_1, z_2, ..., z_n \rangle_n \rfloor$$

At the first stage, the entire sequence $x_1, x_2, ..., x_n$ is communicated in parallel as a vector, at the same time. Following this, the values $y_1, y_2, ..., y_n$ will be communicated as a vector. Thirdly, $z_1, z_2, ..., z_n$ will be communicated as a vector, and then EOT will be signalled for the structure as a whole. As with the vector of streams, communication of such a quadratic sized structure in this way will take linear time and require linear processing elements. The messages required to communicate this structure are illustrated in figure 3.10.

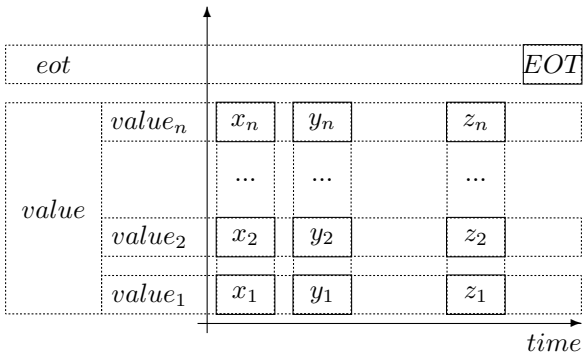


Figure 3.10: Messages required over time to communicate a stream of vectors.

3.4.4 Vector of Vectors

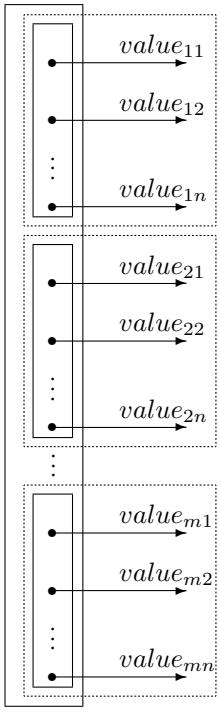


Figure 3.11: The Vector of Vectors

The vector of vectors communicates every item in every sub-list in the structure completely independently in parallel. Let us consider the following vector of vectors.

$$\langle\langle x_1, x_2, ..., x_n\rangle_n, \langle y_1, y_2, ..., y_n\rangle_n, \langle z_1, z_2, ..., z_n\rangle_n\rangle_3$$

All values in this structure will be communicated at the same time, in parallel. Despite the quadratic size of this structure, communication can complete in constant time. However, the requirement on processing elements will be quadratic. The messages required to communicate a vector of vectors are illustrated in Figure 3.12.

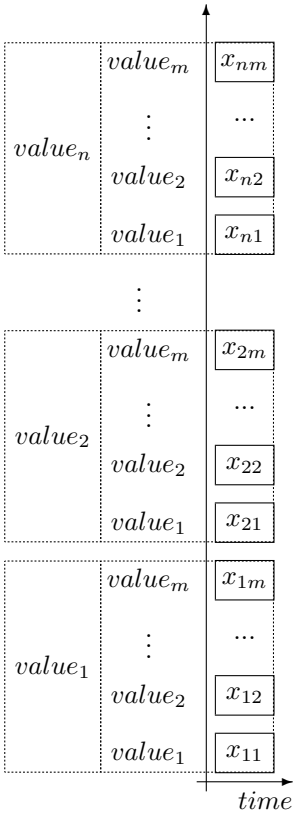


Figure 3.12: Messages required over time to communicate a vector of vectors.

3.5 Transmission Values in Haskell

It may be useful to consider how transmission values can be modeled in a real functional language, for example, in Haskell. One possibility is through usage of a single recursive datatype, $Transmission\ a$:

$$\begin{aligned} data\ Transmission\ a \quad = \quad & Item\ a\ | \\ & Stream\ [Transmission\ a]\ | \\ & Vector\ [Transmission\ a]\ | \\ & Nothing \end{aligned}$$

This has several drawbacks. First of all, the lack of clear type information. Regardless of the complexity of a transmission value, whether an item, a stream, a vector of streams, or even a stream of vectors of streams, the type will remain simply $Transmission\ a$. Secondly, the above definition allows for values such as:

$$Stream[Vector\ a, Stream\ a, Item\ a]$$

As with lists, the components of streams and vectors must all be of the same type. Although technically the above definition defines that a stream *is* the same type as a vector, we require a model where the two are treated as different types, but with some shared properties. A better solution is to consider $Transmission\ a$ as a Haskell class with three instances:

$$
\begin{aligned}
&class\ Transmission\ a\\
&instance\ Transmission\ (Item\ a)\\
&instance\ Transmission\ (Stream\ a)\\
&instance\ Transmission\ (Vector\ a)
\end{aligned}
$$

Basic definitions of $Item$, $Stream$ and $Vector$ may proceed as follows:

$$
\begin{aligned}
data\ Item\ a &= Item\ a\\
data\ Transmission\ a\ =>\ Stream\ a &= Stream\ [a]\\
data\ Transmission\ a\ =>\ Vector\ a &= Vector\ [a]
\end{aligned}
$$

Strictly speaking, we should allow for the value *nothing* in each of these types. Haskell type constructors must be unique to a particular type, so we shall have to introduce a different extra constructor in each case:

$$
\begin{aligned}
data\ Item\ a &= Item\ a\ |\\
&\quad INothing\\
data\ Transmission\ a\ =>\ Stream\ a &= Stream\ [a]\ |\\
&\quad SNothing\\
data\ Transmission\ a\ =>\ Vector\ a &= Vector\ [a]\ |\\
&\quad VNothing
\end{aligned}
$$

We could then introduce a member to the class called *nothing*, which could return the corresponding value for each instance. This form of definition allows us to combine these types together easily and clearly to form more complex transmission values. For example, a simple integer transmission would be defined as follows:

$$Item\ Int$$

A stream of integers would be defined:

$$Stream\ (Item\ Int)$$

A vector of streams of integers would be defined:

$$Vector\ (Stream\ (Item\ Int))$$

This nesting can continue up to any arbitrary level in theory, although in practice we will seldom go much further than the three levels seen in the above example.

3.6 Refinement to Transmission Values

3.6.1 Items

Let us assume for now that refinement of items is trivial - an integer in the specification can be directly refined to an integer in the implementation, a character to a character, a boolean to a boolean and so on.

3.6.2 Streams

Let us consider the act of refining a list into a stream. The abstraction function for abstracting from streams to lists has the following type:

$$abs_S :: \lfloor A \rfloor \rightarrow [A]$$

This is effectively an adaptation of the identity function on lists:

$$abs_S \ \lfloor x_1, x_2, ..., x_n \rfloor = [x_1, x_2, ..., x_n]$$

In general, to prove that a function f_a which operates on lists is correctly refined by a function f_c which operates on streams, we require that the following diagram commutes:

$$\begin{array}{ccc} [A] & \xrightarrow{f_a} & [B] \\ \uparrow{\scriptstyle abs_S} & & \uparrow{\scriptstyle abs_S} \\ \lfloor A \rfloor & \xrightarrow{f_c} & \lfloor B \rfloor \end{array}$$

Let us look at an example: the function *reverse*. This takes a list and returns its reverse. As such it has type:

$$reverse \ :: \ [A] \rightarrow [A]$$

Informally, *reverse* may be defined as follows:

$$reverse \ [x_1, x_2, ..., x_n] \ = \ [x_n, x_{n-1}, ..., x_1]$$

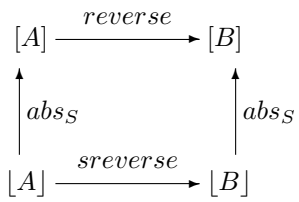
In stream terms, let us consider a refinement called *sreverse*.

$$sreverse \ :: \ \lfloor A \rfloor \rightarrow \lfloor A \rfloor$$

The functionality of *sreverse* mimics that of *reverse*.

$$sreverse \ \lfloor x_1, x_2, ..., x_n \rfloor \ = \ \lfloor x_n, x_{n-1}, ..., x_1 \rfloor$$

For this to be a valid refinement of *reverse*, we will require the following diagram to commute:



We can prove this as follows:

$$\begin{array}{lll} & (reverse \circ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\ = & reverse\ [x_1, x_2, ..., x_n] & \{def.\ abs_S\} \\ = & [x_n, x_{n-1}, ..., x_1] & \{def.\ reverse\} \\ = & abs_S\ \lfloor x_n, x_{n-1}, ..., x_1 \rfloor & \{def.\ abs_S\} \\ = & abs_S\ (sreverse\ \lfloor x_1, x_2, ..., x_n \rfloor) & \{def.\ sreverse\} \\ = & (abs_S \circ sreverse)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ \circ\} \end{array}$$

We may also find useful functions for dealing with tuples of streams. Most commonly we shall encounter tuples of size two (pairs), and an abstraction function here will assist us in the refinement of binary operators. For a pair of streams, to abstract to a corresponding pair of lists, we shall require an abstraction function with the following type:

$$abs2_S :: (\lfloor A \rfloor, \lfloor B \rfloor) \rightarrow ([A], [B])$$

This will have the following informal definition:

$$abs2_S\ (\lfloor x_1, x_2, ..., x_n \rfloor, \lfloor y_1, y_2, ..., y_m \rfloor) = ([x_1, x_2, ..., x_n], [y_1, y_2, ..., y_m])$$

As one might expect, we can define it more formally in terms of abs_S:

$$abs2_S\ (xs, ys)\ =\ (abs_S\ xs, abs_S\ ys)$$

3.6.3 Vectors

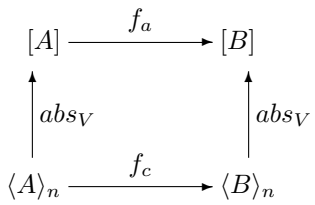
Let us consider the act of refining a list into a vector. The abstraction function for abstracting from vectors to lists has the following type:

$$abs_V :: \langle A \rangle_n \rightarrow [A]$$

This is, again, an adaptation of the identity function on lists:

$$abs_V\ \langle x_1, x_2, ..., x_n \rangle_n = [x_1, x_2, ..., x_n]$$

As before, we can prove that a function f_a which operates on lists is correctly refined by a function f_c which operates on vectors if and only if the following diagram commutes:



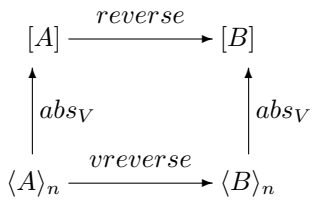
As an example, let us again consider refinement the function *reverse*. In vector terms, let us consider a refinement called *vreverse*.

$$vreverse \;::\; \langle A\rangle_n \to \langle A\rangle_n$$

The functionality of *vreverse* mimics that of *reverse*.

$$vreverse\ \langle x_1, x_2, ..., x_n\rangle_n \;=\; \langle x_n, x_{n-1}, ..., x_1\rangle_n$$

For this to be a valid refinement of *reverse*, we will require the following diagram to commute:



We can prove this as follows:

$$\begin{array}{lll}
 & (reverse \circ abs_V)\ \lfloor x_1, x_2, ..., x_n\rfloor & \{id\} \\
= & reverse\ [x_1, x_2, ..., x_n] & \{def.\ abs_V\} \\
= & [x_n, x_{n-1}, ..., x_1] & \{def.\ reverse\} \\
= & abs_V\ \langle x_n, x_{n-1}, ..., x_1\rangle_n & \{def.\ abs_V\} \\
= & abs_V\ (vreverse\ \langle x_1, x_2, ..., x_n\rangle_n) & \{def.\ vreverse\} \\
= & (abs_V \circ vreverse)\ \langle x_1, x_2, ..., x_n\rangle_n & \{def. \circ\}
\end{array}$$

As with the stream case, we may also find useful functions for dealing with tuples of vectors. For a pair of vectors, to abstract to a corresponding pair of lists, we shall require an abstraction function with the following type:

$$abs2_V :: (\langle A\rangle_n, \langle B\rangle_m) \to ([A], [B])$$

This will have the following informal definition:

$$abs2_V\ (\langle x_1, x_2, ..., x_n\rangle_n, \langle y_1, y_2, ..., y_m\rangle_m) = ([x_1, x_2, ..., x_n], [y_1, y_2, ..., y_m])$$

As one might expect, we can define it more formally in terms of abs_V:

$$abs2_V\ (xs, ys) \;=\; (abs_V\ xs, abs_V\ ys)$$

3.6.4 Combined Structures

Distributed Lists

Functionally, we shall model our distributed list as a list of lists. As such, given some type A, a distributed list refinement of a list of A is denoted simply $[[A]]$.

An abstraction function for distributed lists should have the following type:

$$abs_D :: [[A]] \rightarrow [A]$$

There may be several different interpretations of the implementation of abs_D, depending on the partitioning scheme used. Any partition and abstraction function pair must satisfy the following equation:

$$abs_{Dn} \circ parts_n = id_{[]}$$

Let us consider two here. The first corresponds to a partitioning scheme in which contiguous subsections of the list are used in order. An example partitioning function could take the following form:

$$\begin{array}{lcl} parts_1\ n\ [] & = & [] \\ parts_1\ n\ xs & = & take\ n\ xs\ :\ parts_1\ n\ (drop\ n\ xs) \end{array}$$

In this scheme, the corresponding abstraction is simply a case of concatenating the segments together:

$$abs_{D1} = fold\ (+\!\!+)$$

In an alternative scheme, the partitions may have been constructed in a transposed manner. Here an abstraction function might proceed as follows (see Section 7.15 for a discussion of the function *transpose*):

$$abs_{D2}\ =\ abs_{D1} \circ transpose$$

A reverse transposition composed with our previous partitioning scheme provides us with our new partitioning scheme.

$$parts_2\ n\ =\ transpose \circ parts_1\ n$$

Again, we can prove that a function f_a which operates on lists is correctly refined by a function f_c which operates on distributed lists if and only if the following diagram commutes:

$$\begin{array}{ccc} [A] & \xrightarrow{f_a} & [B] \\ \uparrow{\scriptstyle abs_D} & & \uparrow{\scriptstyle abs_D} \\ [[A]] & \xrightarrow{f_c} & [[B]] \end{array}$$

Let us again consider our example *reverse*. Given that any function operating on lists can be said to be homomorphic so long as it can be expressed as a fold composed with a map, we can highlight the homomorphic nature of *reverse* by defining it as follows:

$$reverse = fold\ (flip\ (+\!\!+))\ \circ\ map\ (\lambda\, x\ \bullet\ [x])$$

Here $flip$ is a function which takes in a binary operator, and reverses the order of its operands:

$$flip\ (\oplus)\ a\ b\ =\ b\ \oplus\ a$$

Interestingly we can generalise this definition of $flip$ when using in conjunction with $fold$:

$$(flip\ (\oplus))\ /\ \ =\ (\oplus)\ /\ \circ\ reverse$$

In a two dimensional setting, we have the function *tdreverse*. This will reverse the overall structure as well as each individual sub-list.

$$tdreverse\ ::\ [[A]] \rightarrow [[A]]$$

It can be defined as follows:

$$tdreverse\ =\ reverse\ \circ\ map\ reverse$$

This should form a valid refinement for *reverse* in a distributed list setting. To prove this, we require the following diagram to commute:

$$\begin{array}{ccc} [A] & \xrightarrow{\ reverse\ } & [B] \\ \uparrow{\scriptstyle abs_D} & & \uparrow{\scriptstyle abs_D} \\ [[A]] & \xrightarrow{\ tdreverse\ } & [[B]] \end{array}$$

We can prove the diagram commutes for this definition of *tdreverse*, along with our first definition of abs_D as follows. Here $[l_1, l_2, ..., l_n]$ represents a distributed list (a list of lists) where l_1 is the first sub-segment, l_2 is the second segment and so on.

$$\begin{array}{lll} & (reverse\ \circ\ abs_{D1})\ [l_1, l_2, ..., l_n] & \{id\} \\ = & (reverse\ \circ\ (+\!\!+)/\ )\ [l_1, l_2, ..., l_n] & \{def.\ abs_{D1}\} \\ = & ((flip\ (+\!\!+))/\ \circ\ map\ reverse)\ [l_1, l_2, ..., l_n] & \{reverse\ promotion\} \\ = & ((+\!\!+)/\ \circ\ reverse\ \circ\ map\ reverse)\ [l_1, l_2, ..., l_n] & \{def.\ flip\} \\ = & ((+\!\!+)/\ \circ\ tdreverse)\ [l_1, l_2, ..., l_n] & \{def.\ tdreverse\} \\ = & (abs_{D1} \circ\ tdreverse)\ [l_1, l_2, ..., l_n] & \{def.\ abs_{D1}\} \end{array}$$

Stream of Streams

To abstract from a stream of streams to a list of lists, we can use the function abs_{SS}. This has type $\lfloor\lfloor A\rfloor\rfloor \rightarrow [[A]]$. One possible definition might proceed as follows:

$$abs_{SS} \;=\; map\ abs_S \;\circ\; abs_S$$

To prove that some function, f_a which operates on lists of lists is correctly refined by a function f_c which operates on streams of streams, we will require the following diagram to commute:

$$\begin{array}{ccc} [[A]] & \xrightarrow{f_a} & [[B]] \\ \uparrow{\scriptstyle abs_{SS}} & & \uparrow{\scriptstyle abs_{SS}} \\ \lfloor\lfloor A\rfloor\rfloor & \xrightarrow{f_c} & \lfloor\lfloor B\rfloor\rfloor \end{array}$$

As an example of this, let us consider again our function $reverse$. Specifically, we are interested in the two dimensional version $tdreverse$. Intuitively, we might construct a stream of stream refinement of $tdreverse$ as follows:

$$ssreverse \;=\; sreverse \;\circ\; smap\ sreverse$$

Let us assume here we already have some function $smap$ (which will be introduced later) which is a valid refinement for map in stream terms.

$$\begin{array}{ccc} [[A]] & \xrightarrow{tdreverse} & [[B]] \\ \uparrow{\scriptstyle abs_{SS}} & & \uparrow{\scriptstyle abs_{SS}} \\ \lfloor\lfloor A\rfloor\rfloor & \xrightarrow{ssreverse} & \lfloor\lfloor B\rfloor\rfloor \end{array}$$

The proof that this diagram commutes, that is, $ssreverse$ is a valid refinement of $tdreverse$, is as follows:

$$
\begin{array}{lll}
 & (tdreverse \circ abs_{SS})\ \lfloor s_1, s_2, ..., s_n \rfloor & \{id\} \\
= & tdreverse\ [l_1, l_2, ..., l_n] & \{def.\ abs_{SS}\} \\
= & [reverse\ l_n, reverse\ l_{n-1}, ..., reverse\ l_1] & \{def.\ tdreverse\} \\
= & [reverse \circ abs_S\ s_n, ..., reverse \circ abs_S\ s_1] & \{def.\ abs_S\} \\
= & [abs_S \circ sreverse\ s_n, ..., abs_S \circ sreverse\ s_1] & \{sreverse\} \\
= & abs_S\ \lfloor abs_S \circ sreverse\ s_n, ..., abs_S \circ sreverse\ s_1 \rfloor & \{sreverse\} \\
= & (abs_S \circ smap\ (abs_S \circ sreverse))\ \lfloor s_n, ..., s_1 \rfloor & \{def.\ smap\} \\
= & (abs_S \circ smap\ (abs_S \circ sreverse) \circ sreverse)\ \lfloor s_1, s_2, ..., s_n \rfloor & \{def.\ sreverse\} \\
= & (abs_S \circ smap\ abs_S \circ smap\ sreverse \circ sreverse)\ \lfloor s_1, s_2, ..., s_n \rfloor & \{map\ dist\} \\
= & (abs_S \circ smap\ abs_S \circ sreverse \circ smap\ sreverse)\ \lfloor s_1, s_2, ..., s_n \rfloor & \{reverse\} \\
= & (abs_S \circ smap\ abs_S \circ ssreverse)\ \lfloor s_1, s_2, ..., s_n \rfloor & \{sreverse\} \\
= & (abs_{SS} \circ ssreverse)\ \lfloor s_1, s_2, ..., s_n \rfloor & \{abs_{SS}\}
\end{array}
$$

Vector of Streams

To abstract from a vector of streams to a list of lists, we can use the function abs_{VS}. This has type $\langle \lfloor A \rfloor \rangle_n \rightarrow [[A]]$. One possible definition might proceed as follows:

$$abs_{VS} \ = \ map\ abs_S \circ abs_V$$

To prove that some function, f_a which operates on lists of lists is correctly refined by a function f_c which operates on vectors of streams, we will require the following diagram to commute:

$$
\begin{array}{ccc}
[[A]] & \xrightarrow{\ f_a\ } & [[B]] \\
\uparrow{\scriptstyle abs_{VS}} & & \uparrow{\scriptstyle abs_{VS}} \\
\langle \lfloor A \rfloor \rangle_n & \xrightarrow{\ f_c\ } & \langle \lfloor B \rfloor \rangle_n
\end{array}
$$

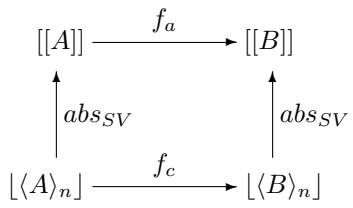
A refinement of *tdreverse* in vector of stream terms, *vsreverse*, could proceed in a fashion analogous to that for the stream of streams, above.

Stream of Vectors

To abstract from a stream of vectors to a list of lists, we can use the function abs_{SV}. This has type $\lfloor \langle A \rangle_n \rfloor \rightarrow [[A]]$. One possible definition might proceed as follows:

$$abs_{SV} \ = \ map\ abs_V \circ abs_S$$

To prove that some function, f_a which operates on lists of lists is correctly refined by a function f_c which operates on streams of vectors, we will require the following diagram to commute:



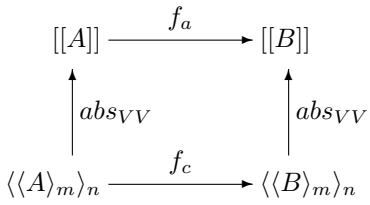
A refinement of *tdreverse* in stream of vector terms, *svreverse*, could proceed in a fashion analogous to that for the stream of streams, above.

Vector of Vectors

To abstract from a vector of vectors to a list of lists, we can use the function abs_{VV}. This has type $\langle\langle A\rangle_n\rangle_n \to [[A]]$. One possible definition might proceed as follows:

$$abs_{VV} \;=\; map\ abs_V \;\circ\; abs_V$$

To prove that some function, f_a which operates on lists of lists is correctly refined by a function f_c which operates on streams of vectors, we will require the following diagram to commute:



A refinement of *tdreverse* in vector of vector terms, *vvreverse*, could proceed in a fashion analogous to that for the stream of streams, above.

3.7 Conduits

A conduit is the mechanism by which a transmission is carried. It is important here to note the distinction between transmission values and conduits. Conduits can be considered as the physical 'wires' which allow transmission of a particular structure of values. Transmission values are the actual messages which flow along those wires. A conduit can be viewed as a lifting of the CSP concept of a channel. In fact, it will often be formed as a grouping of several CSP channels. Here we assume all of these channels to be uni-directional, and to communicate between exactly two processes. Each channel or conduit has exactly one producer on one side, which may only output to that channel; and exactly one consumer on the other side which may only input from that channel. We have three basic types of conduits: item conduits, stream conduits and vector conduits. Conduits are recursive structures. Item conduits are the most primitive form, and may

not contain other conduits. Stream and vector conduits, however, may contain other conduits. A Haskell style syntax may be helpful in outlining the potential for nesting of conduits:

$$\begin{aligned} Conduit\ a \quad = \quad & ItemConduit\ a \mid \\ & StreamConduit\ (Conduit\ a) \mid \\ & VectorConduit\ (Conduit\ a) \end{aligned}$$

When defining alphabets of CSP processes, it may be helpful for clarity to state the type of conduits used. We shall use the functional style operator (::). Additionally, we shall distinguish between conduit types and transmission value types. So, given a process P, that includes a conduit c of some conduit type $\underline{T}$ in its alphabet, we may write:

$$\alpha P = \{c :: \underline{T}\}$$

It is important to make a clear distinction between transmission values and conduits here. If T expresses a type of transmission value, then $\underline{T}$ is a conduit type suitable for carrying such values.

3.7.1 Item Conduits

An *item conduit* is just a single CSP channel. If i is an item conduit we can input to and from it directly. Thus the following are valid, providing x and y are of the correct type:

$$i\ ?\ x$$
$$i\ !\ y$$

To define i as an item conduit to convey some type A we shall write:

$$i :: \underline{\dot{A}}$$

So, given a process P, which has i in its alphabet, we may write:

$$\alpha P = \{i :: \underline{\dot{A}}\}$$

As a more concrete example, let us consider a conduit c which is suitable for the conveyance of boolean transmission values. First of all, a transmission value b which is a boolean is described as follows:

$$b :: \dot{Bool}$$

Our conduit type is therefore:

$$c :: \underline{\dot{Bool}}$$

As such, a process P which has c in its alphabet may be specified as follows:

$$\alpha P = \{c :: \underline{\dot{B}ool}\}$$

In Handel-C terms, our item conduits should be directly implementable as Handel-C channels, given that our definition of items here is limited to simple fixed size types. In Handel-C, integers can be of any arbitrary size in bits, not necessarily a multiple of bytes (i.e. 8 bits). This size must be explicitly stated when declaring a variable. similarly, channels to carry those integers must have their size explicitly stated. So, to set up an item conduit to transmit 32 bit integer values, we have the following:

```
chan int 32 ci;
```

An item conduit to deal with 7 bit ASCII values could be given as follows:

```
chan unsigned int 7 cc;
```

Similarly an item conduit to deal with Boolean values:

```
chan unsigned int 1 cb;
```

Of course, we may wish to add a little more meaning to these by providing some type synonyms:

```
typedef int 32 Int;
typedef unsigned int 7 Char;
typedef unsigned int 1 Bool;
```

We could then express our channels / conduits as follows:

```
chan Int ci;
chan Char cc;
chan Bool cb;
```

We may also find the following preprocessor definition useful:

```
#define Item(x) chan x
```

This would allow us to re-write the above definitions as follows:

```
Item (Int) ci;
Item (Char) cc;
Item (Bool) cb;
```

3.7.2 Stream Conduits

A *stream conduit* is a pair of conduits. The first conduit is a simple channel, used for the signalling of the end of transmission (EOT). The second conduit will carry the values of this stream. In the case of a simple stream of items, the value conduit will be an item conduit (i.e. a single CSP channel). However, for different types of streams (streams of streams, streams of vectors and so on) this conduit may have some more complex structure in itself. In terms of the syntax for dealing with stream conduits, we shall consider a stream conduit as a structure with two members. Thus, if s is a stream conduit, then the following are also conduits:

$$s.eot$$
$$s.value$$

Were s stream of items, then $s.value$ would be an item conduit, and as such we could apply CSP's input and output operators to it directly. Were s a stream of a more complex nature, say for example a stream of streams, $s.value$ would comprise another stream conduit. Similarly, were s a stream of vectors, then $s.value$ would be a vector conduit.

To define s as a stream conduit to convey some type A we shall write:

$$s :: \underline{\lfloor A \rfloor}$$

So, given a process P, which has s in its alphabet, we may write:

$$\alpha P = \{s :: \underline{\lfloor A \rfloor}\}$$

In Handel-C terms we can represent this pairing of conduits using a `struct`. To create a 'one off' instance of a conduit named `s`, to carry a stream of integers, we could use the following:

```
struct
{
   Item(Int) value;
   Item(Bool) eot;
} s;
```

Alternatively, given the following preprocessor definition:

```
#define Stream(x) struct { x value; Item(Bool) eot; }
```

We could then define the above stream conduit `s` as follows:

```
Stream (Item(Int)) s;
```

3.7.3 Vector Conduits

A *vector conduit* is simply an array of other conduits. As with the stream conduit, the exact structure of these conduits will depend on the type of the vector. For a simple vector of items,

each conduit will be a single CSP channel. We shall use subscripts to identify the conduits that make up the vector. If v is a vector conduit of size n, then the following are also conduits:

$$v_1, v_2, \ldots, v_n$$

As before, were v a vector of items, v_i would be a simple item conduit. However, in the case of a vector of streams or a vector of vectors, we shall find v_i shall be either a stream conduit or a vector conduit correspondingly.

To define v as a vector conduit of size n, to convey some type A we shall write simply:

$$v :: \underline{\langle A \rangle_n}$$

So, given a process P, which has v in its alphabet, we may write:

$$\alpha P = \{v :: \underline{\langle A \rangle_n}\}$$

In Handel-C terms vectors are represented as arrays of other conduit types. So, for example, a vector of integers of size 5:

```
Item(Int) value[5];
```

3.7.4 Conduit Renaming

The CSP channel renaming operator shall be lifted to apply to conduits as well. Given a process P with an alphabet containing a conduit c of some conduit type T.

$$\alpha P \ = \ \{c :: \underline{T}\}$$

We may produce a new process, P', which behaves in the same way as P, but operates on conduit c' instead of c. This can be specified using the renaming operator.

$$P' \ = \ P[c'/c]$$

The alphabet of P' will contain c' in place of c. The type of the conduit will, however, be the same.

$$\alpha P' \ = \ \{c' :: \underline{T}\}$$

In this manner, we can rename whole stream, vector or combined conduits with a single substitution.

Conduit renaming in CSP terms will correspond to parameter passing in Handel-C. All conduits on which a process wishes to communicate externally should be included in the parameter list for that process. Consider the following process `P`:

```
macro proc P (in,out)
{
   ...
}
```

Here `P` is a process which communicates on two conduits which, from its point of view, are named `in` and `out`. Should we wish to make use of this process, and have it communicate on channels `xin` and `xout` we simply pass these as parameters:

```
P (xin, xout);
```

The above is the equivalent of the following in CSP:

$$P[xin/in,\ xout/out]$$

3.7.5 Conduit Hiding

Similarly, the CSP channel hiding operator shall be lifted to apply to conduits. Given a process P with a conduit c in its alphabet:

$$\alpha P \ = \ \{c :: \underline{T}\}$$

The expression:

$$P \backslash \{c\}$$

Describes a process where the conduit c only has scope within P. As with renaming, this applies to the entire conduit, allowing us to rename whole stream, vector or combined conduits in a single expression.

In Handel-C terms, channels, like variables, have scope limited to the context block within which they are declared. Context blocks are the areas between curly brackets: { ... }. As such conduit hiding is simply an issue of scope. Consider the following example:

```
macro proc P ()
{
   Item(Int) c;
   ...
}
macro proc Q ()
{
   Item(Int) c;
   ...
}

void main()
{
   par
```

```
   {
      P();
      Q();
   }
}
```

Here we have two process, `P` and `Q`, which are composed together in parallel. Within each of the processes a simple item conduit is declared, both of which happen to have the same name. They are, however, two entirely separate and independent conduits because of their local scope. In CSP terms this is analogous to composing processes P and Q in parallel, each of which have a hidden conduit c. We can express this as follows:

$$P\backslash\{c\} \parallel Q\backslash\{c\}$$

3.8 Produce

The produce process is fundamental to data refinement. It effectively describes the mapping between transmission values, which may exist in the functional world, and the world of processes. One way to model produce is as a function which returns processes. Specifically, we have a function which inputs transmission values (items, streams, vectors and so on) and returns processes. Let us consider such a function Prd. The function has the following type:

$$Prd \ :: \ Transmission\ a \ => \ a \ -> \ Process$$

We shall assume in all instances that Prd employs a single output conduit, *out*, however, the structure of this conduit will vary depending on the particular flavour we are considering.

3.8.1 Nothing

The meaning of the special transmission value *nothing* will be largely defined by its relationship with Prd.

$$Prd\ (Nothing) = SKIP$$

Strictly speaking, the value *nothing* does not exist in its own right, as we have a separate constructor for each transmission value type. So, in effect, we shall require three definitions for *Prd nothing*. However, they will all be fundamentally the same as the above.

3.8.2 Items

For simple, single item types (integers, characters, booleans and so on), the produce process is very simple. This is depicted in Figure 3.13. Here the output conduit is just a single channel.

The definition is very straightforward:

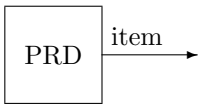


Figure 3.13: The produce process for items.

$$Prd\ (Item\ a) = out\ !\ a \rightarrow SKIP$$

For example:

$$\begin{aligned} Prd\ (Item\ True) &= out\ !\ True \rightarrow SKIP \\ Prd\ (Item\ 10) &= out\ !\ 10 \rightarrow SKIP \end{aligned}$$

The alphabet of Prd in item terms, given a value x of type A is as follows:

$$\alpha\ (Prd\ (Item\ x)) = \{out :: \underline{\dot{A}}\}$$

In Handel-C terms item conduits should be implementable directly as individual channels. As such we should be able to communicate using the standard CSP style communication operators supplied in Handel-C. Thus, to produce a value x on item conduit *out*, we have simply:

```
out ! x;
```

3.8.3 Streams

The produce process for streams is depicted in Figure 3.14. As already noted, the output conduit in this case is a pair of two other conduits. One conduit will produce the values of the stream, and the other will be a simple channel used to signal EOT.

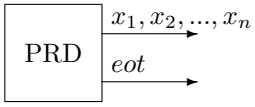


Figure 3.14: The produce process for streams.

Let us first consider the simple case. For streams where the elements are simple items, each item in the stream must be output in turn on the value channel, after which EOT must be signalled on the eot channel.

$$\begin{aligned} Prd\ (Stream\ []) &= out.eot\ !\ any \rightarrow SKIP \\ Prd\ (Stream\ (a : s)) &= out.value\ !\ a \rightarrow (Prd\ s) \end{aligned}$$

In a more general case, we do not necessarily know the structure of the values which the stream is carrying. These may be simple items, but may also be streams or vectors. We can appeal to Prd

for each item, to ensure it is produced correctly. The member of the output conduit designated for signalling eot (*out.eot*) will always be a simple channel no matter what, so we can signal EOT with this directly through the event $out.eot\ !\ any$. The other member of the output conduit (*out.value*) will be a conduit in its own right, and each instance of Prd will output using this conduit. This is achieved using channel renaming.

$$\begin{aligned} Prd\ (Stream\ []) &= out.eot\ !\ any \rightarrow SKIP \\ Prd\ (Stream\ (a : s)) &= (Prd\ a)[out.value/out];\ Prd\ (Stream\ s) \end{aligned}$$

We can also give an alternative definition in a less functional style, i.e. without employing tail recursion. This might proceed as follows:

$$Prd\ (Stream\ s) = \left(\begin{array}{l} \#s \\ \ ;\quad (Prd\ s_i)[out.value/out] \\ i = 1 \end{array} \right) ; out.eot\ !\ any \rightarrow SKIP$$

The alphabet of Prd in stream terms, given stream containing values of type A is as follows:

$$\alpha\ (Prd\ (Stream\ xs)) = \{out :: \underline{\lfloor A \rfloor}\}$$

As an example, let us consider the act of producing a stream xs, with values $\lfloor x_1, x_2, ..., x_n \rfloor$:

$$\begin{aligned} Prd\ \lfloor x_1, x_2, ..., x_n \rfloor = {} & (Prd\ x_1)[out.value/out]; \\ & (Prd\ x_2)[out.value/out]; \\ & ... \\ & (Prd\ x_n)[out.value/out]; \\ & out.eot\ !\ any \rightarrow SKIP \end{aligned}$$

Assuming the elements of xs are items (not streams or vectors), we can apply our definition of Prd to produce each of the elements, and expand the above as follows:

$$\begin{aligned} Prd\ \lfloor x_1, x_2, ..., x_n \rfloor = {} & (out.value\ !\ x_1 \rightarrow SKIP); \\ & (out.value\ !\ x_2 \rightarrow SKIP); \\ & ... \\ & (out.value\ !\ x_n \rightarrow SKIP); \\ & out.eot\ !\ any \rightarrow SKIP \end{aligned}$$

Finally, with a little simplification, we can arrive at the following:

$$\begin{aligned} Prd\ \lfloor x_1, x_2, ..., x_n \rfloor = {} & out.value\ !\ x_1 \rightarrow \\ & out.value\ !\ x_2 \rightarrow \\ & ... \\ & out.value\ !\ x_n \rightarrow \\ & out.eot\ !\ any \rightarrow SKIP \end{aligned}$$

In Handel-C we have something like the following, for a simple stream of items:

```
macro proc SPRODUCE (out,n,values)
{
    typeof (n) i;
    for (i=0;i<n;i++)
    {
        out.value ! values[i];
    }
    out.eot ! True;
}
```

Alternatively, using `seq` replication, we have the following:

```
macro proc SPRODUCE (out,n,values)
{
    seq (i=0;i<n;i++)
    {
        out.value ! values[i];
    }
    out.eot ! True;
}
```

3.8.4 Vectors

For vectors, we compose together n instances of the produce process in parallel, one for each item in the vector. The output conduit here is an array of conduits. This is depicted in Figure 3.15.

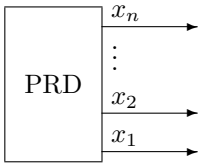


Figure 3.15: The produce process for items.

As with the stream definition, let us first consider the simple case. Here the members of the vector are simple items and can be output directly:

$$Prd\ (Vector\ v) \quad = \quad \|_{i=1}^{\#v} \quad out_i\ !\ v_i \rightarrow SKIP$$

A more general definition is given below. As with the stream version of Prd, the function Prd is used recursively to return a process which will correctly produce each member of the vector.

$$Prd\ (Vector\ v) \quad = \quad \|_{i=1}^{\#v} \quad (Prd\ v_i)[out_i/out]$$

The alphabet of Prd in vector terms, given a vector containing values of type A is as follows:

$$\alpha\ (Prd\ (Vector\ xs))\ =\ \{out :: \underline{\langle A \rangle_n}\}$$

Let us again consider an example, a vector xs, with values $\langle x_1, x_2, ..., x_n \rangle_n$:

$$\begin{aligned} Prd\ \langle x_1, x_2, ..., x_n \rangle_n \quad = \quad & (Prd\ x_1)[out_1/out]\ || \\ & (Prd\ x_2)[out_2/out]\ || \\ & ...\ || \\ & (Prd\ x_n)[out_n/out] \end{aligned}$$

As before, if we assume the elements of xs are simple items, we can expand the definitions of Prd as follows:

$$\begin{aligned} Prd\ \langle x_1, x_2, ..., x_n \rangle_n \quad = \quad & (out_1\ !\ x_1 \rightarrow SKIP)\ || \\ & (out_2\ !\ x_2 \rightarrow SKIP)\ || \\ & ...\ || \\ & (out_n\ !\ x_n \rightarrow SKIP) \end{aligned}$$

In Handel-C we have something like the following, for a simple vector of items:

```
macro proc VPRODUCE (out,n,values)
{
    par (i=0;i<n;i++)
    {
        out[i] ! values[i];
    }
}
```

3.9 Summary

In this chapter we have shown how data types in our specification can be refined to communication mechanisms in our implementation. We have looked at two fundamental alternatives - the stream and the vector, which correspond to sequential and parallel computation respectively.

Chapter 4

Process Refinement

4.1 Introduction

In this chapter we look at how functions in our specification are refined into processes in our implementation. Not only do we need to consider how behaviour can be derived from individual functions, but also we need to determine how application and composition of those functions is to be translated into the process environment.

4.2 Feed

Consider some function f and some value x. In a functional language, we express the application of function f to value x simply with the following:

$$f\ x$$

In effect the space here is our function application operator. Let us consider how we would refine the above specification into processes. As we have seen in the chapter on data refinement, the value x can be refined into a process $Prd(x)$ which transmits that value on a channel (or more generally a conduit). Let us assume we have some process F which refines function f. We require some means of specifying that the output of $Prd(x)$ should form the input to process F. This can be achieved with the feed operator (written $\triangleright$), defined in [1]. We can therefore refine the above specification using the feed operator as follows:

$$Prd(x)\ \triangleright\ F$$

This is depicted in Figure 4.1, wherein both the conduit names, as well as the values carried on those conduits are given.

The feed operator takes two processes, composes them together in parallel, and renames both the output conduit of the first and the input conduit of the second to a new name, which is then

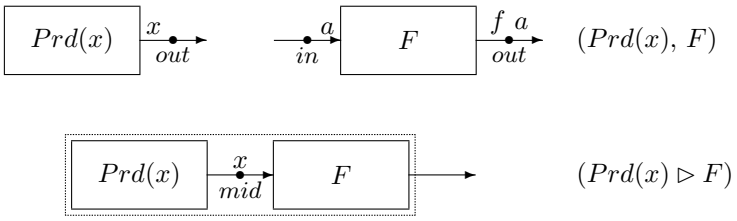


Figure 4.1: Refinement of the function application to process feeding.

hidden. Given the lifted concepts of CSP channel renaming and hiding, the definition can remain the same regardless of the type of conduit (item, stream, vector or any combination thereof).

$$P \rhd Q = (P[mid/out] \parallel Q[mid/in]) \backslash \{mid\}$$

Here the process P must be a producer only, and have no input. The composite process is a producer.

This requires that *out* is the only member of the alphabet of P, *in* is a member of the alphabet of Q, and that these conduits are of the same type. Or, more formally we have:

$$(\alpha P = \ \{out :: \underline{A}\}) \wedge (\alpha Q = \ \{in :: \underline{A}, out :: \underline{B}\})$$

4.2.1 Vectors

Let us take a while to explore the use of the feed operator in the vector setting. Let us consider two processes, PS and QS, each formed from a parallel composition of processes P and Q respectively. The first process, PS, outputs a vector:

$$PS \ = \ \left(\begin{array}{cc} n & \\ \parallel & P[out_i/out] \\ i = 1 & \end{array} \right)$$

We can define the alphabet of PS as follows:

$$\alpha PS \ = \ \{out :: \underline{\langle A \rangle_n}\}$$

The second process, QS, inputs a vector.

$$QS \ = \ \left(\begin{array}{cc} n & \\ \parallel & Q[in_i/in] \\ i = 1 & \end{array} \right)$$

We can define the alphabet of QS as follows:

$$\alpha QS \ = \ \{in :: \underline{\langle A \rangle_n}\}$$

These processes satisfy the above conditions for use of the feeding operator. So, if the output vector of PS is fed to QS, .i.e.

$$PS \rhd QS$$

We have the following process:

$$\left(\begin{array}{c} n \\ || \\ i=1 \end{array} P[out_i/out] \right) \rhd \left(\begin{array}{c} n \\ || \\ i=1 \end{array} Q[in_i/in] \right)$$

To illustrate the use of vector feeding, the above definition is equivalent to the following:

$$(PS\ [mid/out] \mid\mid QS\ [mid/in]) \setminus \{mid\}$$

To write this out in full we have the (somewhat unwieldy!) definition:

$$\left(\left(\begin{array}{c} n \\ || \\ i=1 \end{array} P[out_i/out] \right) [mid/out] \mid\mid \left(\begin{array}{c} n \\ || \\ i=1 \end{array} Q[in_i/in] \right) [mid/in] \right) \setminus \{mid\}$$

Note carefully the lifted use of CSP's renaming operator here. In the definition above, we see the expressions $[mid/out]$ and $[mid/in]$. Here whole vector conduits are being renamed. By applying these renaming operators, the above expression can be simplified to arrive at the following:

$$\left(\left(\begin{array}{c} n \\ || \\ i=1 \end{array} P[mid_i/out] \right) \mid\mid \left(\begin{array}{c} n \\ || \\ i=1 \end{array} Q[mid_i/in] \right) \right) \setminus \{mid\}$$

Note in the above the lifted use of CSP's hiding operator. The hiding operator is used above to hide an entire vector conduit, mid. All the component conduits that make up this vector (mid_1 up to mid_n) will be hidden.

When we consider vector feeding between processes of the likes of PS and QS, we may observe that feeding between the processes as whole is equivalent to composing together in parallel n instances of a process which feeds between the component processes. In other words:

$$\left(\begin{array}{c} n \\ || \\ i=1 \end{array} P[out_i/out] \right) \rhd \left(\begin{array}{c} n \\ || \\ i=1 \end{array} Q[in_i/in] \right) = \begin{array}{c} n \\ || \\ i=1 \end{array} P \rhd Q$$

4.3 Process Refinement

Given that we now have a definition of a feed operator that operates on processes, we are now able to give a formal definition of process refinement. Let us consider a function f, which takes

in values of type A and returns values of type B. We shall assume the data refinement step has already been performed, such that A and B are both types of some transmission value:

$$f :: A \rightarrow B$$

Let us now consider a potential refinement for f, a process F. This should have an input conduit suitable for receiving transmission values of type A, and an output conduit suitable for sending transmission values of type B.

$$\alpha F \ = \ \{in :: \underline{A}, out :: \underline{B}\}$$

The operator $\prec$ denotes a process refinement, where the left hand side is a function, and the right hand side a process. To state that f is refined by F, or in other words, the process F is a valid refinement of the function f, we may write:

$$f \prec F$$

To prove this is a valid refinement, we require the following condition to hold:

$$\forall\, x :: A \bullet (Prd\ x) \rhd F \ = \ Prd\ (f\ x)$$

That is to say, for all values x of type A, the process formed by producing x and feeding this to F is equal to the process formed by producing the result of f applied to x directly. This equality here is strictly speaking an algebraic equivalence, although in simple terms we can consider it as engaging in the same events, or, in other words, producing the same output.

4.4 Pipe

The pipe operator is written $\gg$, and is introduced in [42]. The pipe operator, like the feed operator, takes two processes, composes them together in parallel, and renames both the output conduit of the first and the input conduit of the second to a new name, which is then hidden. Given the lifted concepts of CSP channel renaming and hiding, the definition can remain the same regardless of the type of conduit (item, stream, vector or any combination thereof).

$$P \gg Q = (P[mid/out] \ || \ Q[mid/in]) \backslash \{mid\}$$

This requires that *out* is a member of the alphabet of P, *in* is a member of the alphabet of Q, and that these conduits are of the same type. Or, more formally we have:

$$(out \ \epsilon \ \alpha P) \wedge (in \ \epsilon \ \alpha Q) \wedge \exists \underline{T} \bullet (in :: \underline{T} \wedge out :: \underline{T})$$

The pipe operator in CSP refines function composition ($\circ$) in our specifications. Consider the following functional specification:

$$h \;=\; g \circ f$$

Here we have the following types:

$$\begin{aligned} f &:: A \to B \\ g &:: B \to C \\ h &:: A \to C \end{aligned}$$

Let us assume we have some process P which refines f and some process Q which refines g. These processes should therefore have the following alphabets:

$$\begin{aligned} \alpha P &:: \{in :: \underline{A},\ out :: \underline{B}\} \\ \alpha Q &:: \{in :: \underline{B},\ out :: \underline{C}\} \end{aligned}$$

In our functional specification, the output of function f becomes the input to function g. Similarly in our process refinement we wish to state that the output of process P forms the input to process Q. We can use the pipe operator for this, and as such we can then refine our compositional specification h to the following:

$$P \gg Q$$

So we can say:

$$(g \circ f) \;\prec\; (P \gg Q),\ \text{if}\ (f \prec P \;\wedge\; q \prec Q)$$

This is illustrated in Figure 4.2, wherein both the conduit names, as well as the values carried on those conduits are given.

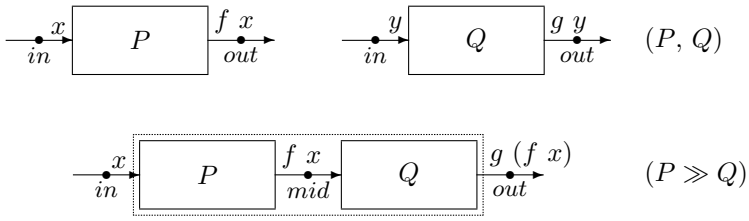


Figure 4.2: Refinement of the composition operator to process piping.

The CSP definition of the pipe operator - in terms of the parallelism operator, with some channel renaming/hiding - carries through to our Handel-C implementation. As noted in Sections 3.7.4 and 3.7.5, conduit renaming and hiding correspond to parameter passing and locally scoped conduits in Handel-C. So, given Handel-C definitions for processes `P` and `Q`, we could specify the piping together of these two processes as follows:

```
macro proc PIPE (in,out)
{
    // ( definition for mid )
    par
    {
        P (in,mid);
        Q (mid,out);
    }
}
```

4.5 Pipelining

In Section 4.4 we saw how functional composition can be refined into process piping. This scheme for two functions/processes can also be generalised to n functions/processeses as shown in [2]. The functional composition operator is a binary operator like any other. We have the following type:

$$(\circ) \ :: \ (B \to C) \to (A \to B) \to (A \to C)$$

As a special case of the above, quite general, type definition, we can of course deal with functions which both input and output values of the same type. So given some function f:

$$f \ :: \ A \to A$$

So the expression $(f \circ f)$ is a function which is also of type $(A \to A)$. We can, in fact, compose f with itself *ad nauseum*:

$$f \ \circ \ f \ \circ \ f \ \circ \ ... \ \circ \ f$$

Expressions of this form can instead be given as a fold (see Section 5.3) with the composition operator. We have:

$$fold \ (\circ) \ [f, \ f, \ ..., \ f] \quad = \quad f \ \circ \ f \ \circ \ ... \ \circ \ f$$

Or alternatively, to use the infix form of fold. we have:

$$(\circ) \ / \ [f, \ f, \ ..., \ f] \quad = \quad f \ \circ \ f \ \circ \ ... \ \circ \ f$$

Evidently, we may not always be simply re-applying the same function over and over. So long as all the functions in our composition have the same type:

$$f_1, f_2, ..., f_n \ :: \ A \to A$$

We can then give the following:

$$(\circ)/ \ [f_1, f_2, ..., f_n] \quad = \quad f_1 \ \circ \ f_2 \ \circ \ ... \ \circ \ f_n$$

One common way to create a series of n functions which have the same type but perform a differing action is to 'specialise' each with an additional parameter. Consider the following function f:

$$f \ :: \ B \rightarrow A \rightarrow A$$

Here we have a function which, once supplied with a value of type B will then take in a value of type A and return a value of type A. We can then map f to a list of values of type B to give us our list of composable functions. This results in the following pattern:

$$(\circ) \ / \ (map \ f \ [x_1, x_2, ..., x_n]) \quad = \quad (f \ x_1) \ \circ \ (f \ x_2) \ \circ \ ... \ \circ (f \ x_n)$$

In process terms, each occurrence of the composition operator $(\circ)$ will be refined to an application of the process piping operator $(\gg)$. In the simple case, where we are composing n instances of the same function f, given some process P which refines f we have the following:

$$(\circ) \ / \ [f, \ f, \ ..., \ f] \ \prec \ \left(\begin{array}{c} n \\ \gg \quad P \\ i = 1 \end{array} \right), \text{ if } f \prec P$$

In the more general case, we have to take into the account that the pipe operator and composition operators work in opposite directions.

$$(\circ) \ / \ [f_n, \ f_{n-1}, \ ..., \ f_2, \ f_1] \ \prec \ \left(\begin{array}{c} n \\ \gg \quad P_i \\ i = 1 \end{array} \right), \text{ if } f_{1..n} \prec P_{1..n}$$

Alternatively we may wish to introduce the function $reverse$ here:

$$(\circ) \ / \ (reverse \ [f_1, f_2, ..., f_n]) \ \prec \ \left(\begin{array}{c} n \\ \gg \quad P_i \\ i = 1 \end{array} \right), \text{ if } f_{1..n} \prec P_{1..n}$$

Similarly for the parameterised case we have:

$$(\circ) \ / \ (map \ f \ (reverse \ [x_1, x_2, ..., x_n])) \ \prec \ \left(\begin{array}{c} n \\ \gg \quad P(x_i) \\ i = 1 \end{array} \right), \text{ if } (f \ x) \prec P(x)$$

To illustrate this, the construction of a pipeline of processes is depicted in Figure 4.3.

In Handel-C terms, a basic pattern for pipelining can be represented by the following:

```
macro proc PIPELINE (n, mids, P)
{
    par (i=0;i<n;i++)
    {
        P (mids[n],mids[n+1]);
    }
}
```

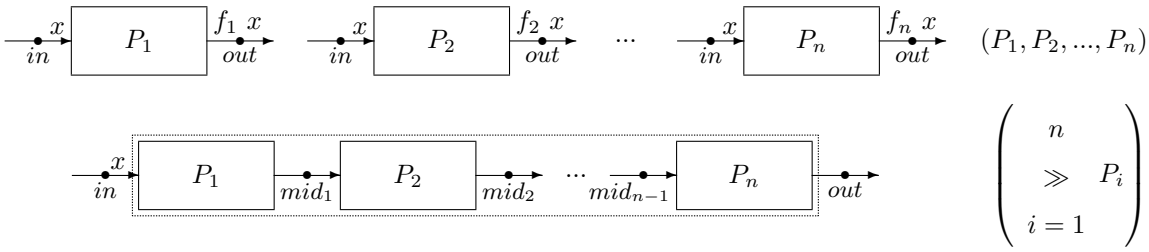


Figure 4.3: Refinement of composed functions to pipelined process.

Alternatively, where we wish to parameterise each process with a different value, we have:

```
macro proc PIPELINE1 (n, channels, values, P)
{
    par (i=0;i<n;i++)
    {
        P (mids[n],mids[n+1],values[n]);
    }
}
```

4.6 Recursion Unrolling

Functional definitions are often given recursively. Although this is a useful construct in a specification, we are not typically able to implement recursive algorithms directly in our target environment (the FPGA). In terms of list processing functions, there are, broadly speaking, two forms of recursion.

4.6.1 Tail Recursion

The first, tail recursion, is illustrated by the example below:

$$\begin{aligned} insert\ a\ [] &= [a] \\ insert\ a\ (x : xs) &= \text{if } a < x \\ &\quad \text{then } a : x : xs \\ &\quad \text{else } x\ :\ insert\ a\ xs \end{aligned}$$

This kind of recursion is easier to deal with as computation proceeds progressively as the input list is consumed. In CSP we are quite at liberty to specify processes recursively, so we can provide the following analogous definition:

$$
\begin{aligned}
INSERT(a) \quad = \quad & in.eot\ ?\ any \rightarrow (out.value\ !\ a \rightarrow out.eot\ !\ True \rightarrow SKIP) \\
& | \\
& in.value\ ?\ x \rightarrow \left(\begin{array}{c} out.value\ !\ a \rightarrow out.value\ !\ x \rightarrow COPY \\ \nleftarrow a < x \nrightarrow \\ out.value\ !\ x \rightarrow INSERT(a) \end{array} \right)
\end{aligned}
$$

This definition is, however, not directly implementable in hardware. Thankfully tail recursion is easily refined to iteration, so we can replace the above definition with a simple iteration, as demonstrated in Figure 4.4. This iterative version can be implemented almost directly in Handel-C. The Handel-C implementation for this process is also given in Figure 4.4.

```
macro proc INSERT (in,out,a)
{
    typeof(a) x;
    Bool eot;
    eot = False;
    while (!eot)
    {
        prialt
        {
        case in.eot ? eot:
            out.value ! a;
            out.eot ! True;
            break;
        case in.value ? x:
            if (a<x)
            {
                out.value ! a;
                out.value ! x;
                COPY (in,out,eot);
            }
            else
            {
                out.value ! x;
            }
            break;
        }
    }
}
```

$$
\begin{aligned}
& INSERT(a) = \\
& \mu X \bullet \\
& in.eot\ ?\ any \rightarrow (out.value\ !\ a \rightarrow out.eot\ !\ True \rightarrow SKIP) \\
& | \\
& in.value\ ?\ x \rightarrow \left(\begin{array}{c} out.value\ !\ a \rightarrow out.value\ !\ x \rightarrow COPY \\ \nleftarrow a < x \nrightarrow \\ out.value\ !\ x \rightarrow X \end{array} \right)
\end{aligned}
$$

Figure 4.4: The Handel-C and CSP definitions of the process INSERT.

4.6.2 Head Recursion

The other form of recursion in list terms is head recursion. This usually presents more of a problem when it comes to refinement as it generally requires the entire input list to have been consumed before any actual computation takes place. A good example of this is the function $foldr$ - see Section 5.3.

$$\begin{aligned} foldr\ (\oplus)\ e\ [] &= e \\ foldr\ (\oplus)\ e\ (x : xs) &= x\ \oplus\ (foldr\ (\oplus)\ e\ xs) \end{aligned}$$

Although the head recursive nature of $foldr$ here many not be immediately apparent, consider the application of the $(\oplus)$ operator. We cannot fully apply $(\oplus)$ until we have reached the end of the list, so in effect the computation begins at the end of the list and proceeds backwards. The informal definition for $foldr$ may help to clarify this:

$$foldr\ (\oplus)\ e\ [x_1, x_2, ..., x_n] = x_1 \oplus (x_2 \oplus (... \oplus (x_n \oplus e)))$$

We can provide an alternative definition for $foldr$ which may also help to highlight its head recursive style. Here we gradually accumulate the result in the e parameter by progressing through the list backwards, applying $(\oplus)$ at each step.

$$\begin{aligned} foldr\ (\oplus)\ e\ [] &= e \\ foldr\ (\oplus)\ e\ (xs +\!\!+ [x]) &= foldr\ (\oplus)\ (x\ \oplus\ e)\ xs \end{aligned}$$

Sometimes it may be acceptable to implement a buffering scheme to deal with this head recursion. Another approach, perhaps more efficient, may be to attempt to remove the head recursion from the specification via some program transformation. For example, in many cases $foldr$, which is head recursive, can be replaced with $foldl$, its tail recursive counterpart, following a little manipulation of the specification. The relationship between these two functions is explored further in Section 5.3.

4.6.3 Pattern Matching

Pattern matching is a convenient way to specify possible input cases in a functional specification. Where pattern matching is applied to list which are inputs to a function, and these lists are refined to streams in the implementation, we shall need to think carefully about how they are to be represented. Some examples of the sorts of pattern matching expressions for lists we may encounter are given in Figure 4.5.

$[]$	An empty list
$[x]$	A list containing exactly one item, x.
$[x, y]$	A list containing exactly two items, x and y.
$(x : xs)$	A list containing at least one item x.
$(x : y : xs)$	A list containing at least two items x and y.

Figure 4.5: Typical pattern matching expressions on lists.

Let us consider how these constructs might be refined to a process implementation in stream terms.

Delayed Signalling of EOT

Consider the refinement of some part of a function which uses a pattern match involving the empty list. In a stream refined process, broadly speaking, this condition will correspond to checking the EOT channel of the appropriate stream. The signalling of EOT will denote that no further values are to be transmitted on the corresponding stream. So, once the receiving process has been given notification of EOT, it knows it will not be required to consume anything more from that stream. The reverse statement is, however, not necessarily true. That is to say, when EOT is *not* yet signalled, the consuming process should not assume that there *are* more values to be received from that stream. There may sometimes be a gap between a sending process transmitting the last value in a stream, and the signalling of EOT.

This issue can be resolved as long as the consumer adheres to the CSP specification given in Section 3.3.3. That is to say, the consumer must at any point (before the stream has ended) be willing to accept either a value or the EOT message. In effect it should wait indefinitely for either of these to occur. This can be achieved with CSP's choice operator (or Handel-C's `prialt` construct). The consumer must never assume that EOT not being signalled definitely means another value can be received.

Peeking without Consuming

Another important issue to consider is that in CSP style communication, as implemented in Handel-C, we cannot 'peek' at an input without actually consuming from it. This kind of 'peeking' behaviour is, however, commonplace in functional specifications. This is well illustrated by the function *merge*, as used in the merge sort algorithm:

$$
\begin{array}{lcl}
merge\ xs\ [] & = & xs \\
merge\ []\ ys & = & ys \\
merge\ (x : xs)\ (y : ys) & = & \text{if } x < y \\
 & & \quad \text{then } x : merge\ xs\ (y : ys) \\
 & & \quad \text{else } y : merge\ (x : xs)\ ys
\end{array}
$$

In the third statement of this definition the pattern matching on the left hand side asserts that there is at least one item available from each of the two input lists. These items are then compared, and the lower of the two is output. The higher item, however, is effectively put back into the list it came from before the function recurses. Whilst this does not present any particular problem in the world of functions, in process terms this kind of behaviour certainly is an issue that requires some thought. In our model of CSP communication, where we assume all channels to be uni-directional, we cannot simply 'put back' a value that has been received on a channel in this way.

One possibility would be to add extra behaviour to sending processes to deal with the possibility of values being sent back by the receiver via some kind of return channel. However, this is

probably not the best way to handle this issue. For one thing, adding channels will have a cost in hardware terms and it is likely many of these return channels will go unused - only certain receiving processes are likely to employ this peeking behaviour. Additionally, as the function *merge* above demonstrates, generally values which are peeked at but not consumed in this way are refused on a "not yet" rather than a "not at all" basis. In other words, they are often not rejected altogether, but instead are consumed at some later stage. The introduction of a return channel mechanism could create a kind of ping pong behaviour between the sender and receiver, which is likely to be highly inefficient.

A simpler, and arguably more efficient solution would be to add some kind of buffering to the receiving process. In this way we need not add any more communication overhead, and we need not complicate the sending process in any way. Conceptually, this buffer would act as an interface between the receiving process and the stream from which it is consuming. Rather than simply reading from the stream, the receiving process would be furnished with two possible operations. A *peek* operation would attempt to fill the buffer with the required number of items, which may result in one or more values being received from the stream. Where this number of items is already present in the buffer, then no actual communication with the stream would be required. A *consume* operation would effectively remove items from the buffer.

The size of the buffer required for a given stream can be determined from the depth of the pattern matching for the corresponding list in the functional specification. By depth here we mean the maximum number of items extracted from that list in any pattern match for that list in the definition. As an example, for the *merge* function above, the required buffer size is just one item for each of the two input lists. The 'deepest' pattern match is the $(x : xs)$ for the first list and $(y : ys)$ for the second list. In each of these expressions we are only attempting to match one item.

Let us also consider the function *bpass*. This performs a single pass of the bubble sort algorithm. We can define this as follows:

$$
\begin{array}{lcl}
bpass\ [] & = & [] \\
bpass\ [x] & = & [x] \\
bpass\ (x : y : ys) & = & \text{if } x < y \\
 & & \text{then } x : bpass\ (y : ys) \\
 & & \text{else } y : bpass\ (x : xs)
\end{array}
$$

Here the pattern matching depth is two - in the third statement we attempt to match two items from the input list, with the expression $(x : y : ys)$. Thus in a stream process refinement of this function we would a buffer size of two.

4.7 Lazy Evaluation

In the functional setting we are not particularly concerned with the fate of any part of an input list we do not choose to consume. Parts of intermediate data structures which are not required to compute the eventual output can be entirely ignored without any cause for concern in a functional program. This is, of course, a direct result of lazy evaluation - we only calculate that which we absolutely need to in order to produce the end result. A good example of this is given by the *take* function. This inputs a list and returns just the first n elements. Consider the following functional expression:

$$take\ 20\ (map\ (*2)\ [1..1000000])$$

Here the expression [1..1000000] generates a list of all numbers from one to a million. We are then applying *map* (see Section 5.2) with the function $(*2)$ to this list - so we are in effect doubling every item. Finally, we are requesting that we take just the first 20 results. The question here is, of course, how many calculations are made - that is to say how many times is the multiplication operator applied. Is it twenty or one million? Whereas an imperative equivalent of this algorithm might make a million calculations, lazy evaluation means that the functional version makes only twenty - the minimum required to produce the desired output.

Moreover, functional programs allow us to deal with infinite lists. The enumeration given above does not have to have an upper limit - by removing this we are effectively requesting an infinite list of ascending integers. Thus the expression below actually results in the same number of calculations and the same result, whereas arguably an imperative equivalent may *never* produce a result:

$$take\ 20\ (map\ (*2)\ [1..])$$

We may find it important in certain cases to be reassured that this model of lazy evaluation can be preserved in our process implementations. Certainly for the above example, were we to discover that the process implementation performed one million calculations we would definitely have cause for concern. We could refine the above definition to a simple network of processes:

$$\begin{array}{lcl}
ENUM(from, to) & = & \text{if } (from > to) \\
 & & \text{then } out.eot\ !\ True \rightarrow SKIP \\
 & & \text{else } out.value\ !\ from \rightarrow ENUM(from + 1, to) \\
DOUBLE & = & in.eot\ ?\ any \rightarrow out.eot\ !\ True \rightarrow SKIP \\
 & & | \\
 & & in.value\ ?\ x \rightarrow out.eot\ !\ (x * 2) \rightarrow DOUBLE \\
TAKE(n) & = & \text{if } (n == 0) \\
 & & \text{then } out.eot\ !\ True \rightarrow SKIP \\
 & & \text{else } in.value\ ?\ x \rightarrow out.value\ !\ x \rightarrow TAKE(n-1) \\
NETWORK & = & ENUM(1, 1000000) \triangleright DOUBLE \gg TAKE(20)
\end{array}$$

This is depicted below:

It should hopefully be clear that the characteristics of lazy evaluation are, to an extent, preserved by this refinement. After twenty steps the output of the network as a whole will signal EOT. We could construe from this that the computation as a whole has completed. However - although the last process in the pipeline, $TAKE$, will then terminate, the other two processes will continue to run, waiting indefinitely to perform communications which will never actually occur[1]. Of course, this issue may be entirely inconsequential, it really depends on the setting in which the implementation is placed.

It is important to bear in mind the potential side effects of these kinds of refinements in the world of processes. Consider as an example the following two functions.

$$\begin{aligned} dup\ a &= (a, a) \\ fork\ f\ (a, b) &= (f\ a, f\ b) \end{aligned}$$

Then consider the following expression which takes advantage of these functions:

$$fork\ (take\ 5)\ (take\ 10)\ (dup\ [1..20])$$

The implication of this is that we wish to generate the integers between one and twenty in a list, and then make two copies of that list. In the first copy of the list we wish to discard all but the first five items, and in the second list all but the first ten. Thus our result should be equivalent to:

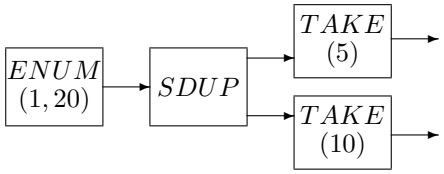
$$([1..5], [1..10])$$

As *dup* in our specification above is operating on lists, we may choose to provide a stream refinement for it in our implementation. In this context it will have a single stream as input, and will, from this, be required to produce two identical streams as output. A possible definition is given below:

$$\begin{aligned} SDUP = \ & in.eot\ ?\ any \rightarrow out1.eot\ !\ True \rightarrow out2.eot\ !\ True \rightarrow SKIP \\ & | \\ & in.value\ ?\ x \rightarrow out1.value\ !\ x \rightarrow out2.value\ !\ x \rightarrow SDUP \end{aligned}$$

Let us consider this process composed together into a network refining our functional specification.

[1] You almost feel sorry for them, don't you?



So, we would expect to have two streams as output - one containing the values (1..5) and another containing the values (1..10). Actually, given the definition of $SDUP$ as above, this is not, in fact, what will happen. After consuming five values, the upper $TAKE$ process will stop consuming on its input conduit, which corresponds to the conduit $out1$ in the process $SDUP$. The definition of $SDUP$ is such that, upon receiving each value from its input, it will first attempt to output it on $out1$, then on $out2$, and then repeat. Whilst the receiving end of $out1$ is unwilling to communicate, the process will wait there indefinitely. So in effect, once one $TAKE$ process stops consuming, the other will no longer be fed. So actually our network as a whole will output five values on each output stream before becoming deadlocked. The additional five values required from the second $TAKE$ process will never appear.

This issue raises a few questions. Should $TAKE$ (and similar processes) be responsible for consuming the entirety of its input stream, including the segment it merely discards? A definition which satisfies this additional criteria is given below:

$$
\begin{aligned}
TAKE(n) \quad = \quad & in.eot \; ? \; any \rightarrow \; out.eot \; ! \; True \rightarrow SKIP \\
& | \\
& in.value \; ? \; x \rightarrow \left(\begin{array}{ll} \text{if} & (n == 0) \\ \text{then} & TAKE(n) \\ \text{else} & out.value \; ! \; x \rightarrow TAKE(n-1) \end{array} \right)
\end{aligned}
$$

In this version we continue to consume from the input stream until EOT is signalled for that conduit, at which point we also then signal EOT on the output conduit. This definition arguably does not satisfy the criteria of lazy evaluation - EOT will not be signalled on the output conduit until the entirety of the input stream has been consumed. We could, of course, supply a different definition that signals EOT at the point at which n reaches zero. Again though, whether or not lazy evaluation is implemented by this behaviour is arguable. Although the output will be produced in a minimal number of steps as we would hope, the network as a whole will continue to operate after this, potentially indefinitely. Is this desirable?

Another alternative strategy for addressing this issue is the possibility of EOT signalling being bi-directional. Currently the sender has a mechanism for informing the receiver that it will not produce any more values, but there is no mechanism for the receiver to notify the sender that it is not prepared to consume any more. So, in addition to the EOT (end of transmission) channel, we might also consider an EOR (end of reception) channel. Whereas we may want to bear this option in mind for certain specific scenarios, we certainly do not which to add this EOR mechanism to

every single stream - it will generally go unused, which is potentially very wasteful.

So, for general purposes, in the absence of the previous two solutions, we pass the responsibility for resolving this issue to the sending process. That is to say, any sender which outputs to more than one receiver in this fashion should not allow the cessation of one receiver to affect the transmission to the other(s).

Considering our process $SDUP$, in effect we shall require that the two output streams are asynchronous of each other and therefore asynchronous with respect to the input stream. As always, we can translate from a synchronous communication to an asynchronous one via buffering. We effectively then require one buffering process per output stream. This process should always be willing to transmit, assuming there is data in the buffer, and always be willing to receive, assuming there is unused capacity in the buffer. The input stream is then managed by a process which reads a value at a time, and then attempts to pass this value onto each of the buffer processes, accepting that it may fail.

4.8 Conversions

We may often find it useful to have processes which "convert" between our different data refinements. For example, a process which inputs a stream and outputs a vector, as well as one which does the reverse, inputs a vector and outputs a stream.

In fact, these conversions are specific cases of more general schemes; specialised refinements of *map*. We simply use a refinement of the function id for the characteristic function f. See Section 5.2.3 and Section 5.2.4 for more details.

4.9 Summary

We have seen how functions in our specification can be refined to processes in our implementation. A formal criteria for valid refinement to a process has been given which allows us to reassure ourselves that our implementation correctly satisfies the requirements of our specification. We have shown how function application can be refined to the feed operator in the process world. We have also shown how we can translate from the combination of components in the functional setting (composition) to the equivalent in the process setting (pipelining). With these basic building blocks in our toolbox we are now prepared to tackle a wide range of refinement tasks.

Chapter 5

Refinement of Key Higher Order Functions

5.1 Introduction

Two particular higher order functions, *map* and $fold$, are key to many functional algorithms and as such it is necessary to thoroughly explore their refinement to processes.

5.2 Map

Let us consider the function *map*. Given a function f of type $A \rightarrow B$, $map\ f$ has type:

$$[A] \rightarrow [B]$$

Informally, it has the following definition:

$$map\ f\ [x_1, x_2, ..., x_n]\ =\ [f\ x_1, f\ x_2, ..., f\ x_n]$$

We may occasionally find it useful to employ *map* as a binary infix operator. To accomplish this, we are presented with two possibilities. First, the Haskell convention of surrounding a function name with single quotes:

$$f\ `map`\ xs\ =\ map\ f\ xs$$

Secondly, the BMF *map* operator:

$$f\ *\ xs\ =\ f\ map\ xs$$

5.2.1 Streams

The corresponding stream refinement of $map\ f$, $smap\ f$, has type $\lfloor A \rfloor \rightarrow \lfloor B \rfloor$. An informal definition of *smap* is a simple analogy of *map*:

$$smap\ f\ \lfloor x_1, x_2, ..., x_n \rfloor\ =\ \lfloor f\ x_1, f\ x_2, ..., f\ x_n \rfloor$$

Thus to state that *smap* is a valid refinement of *map* we require the following diagram to commute.

$$\begin{array}{ccc} [A] & \xrightarrow{map\ f} & [B] \\ \uparrow{\scriptstyle abs_S} & & \uparrow{\scriptstyle abs_S} \\ \lfloor A \rfloor & \xrightarrow{smap\ f} & \lfloor B \rfloor \end{array}$$

Given a stream of length n containing items x_1 up to x_n we can construct a proof that the diagram will commute as follows:

$$\begin{array}{lll} & (map\ f\ \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\ = & map\ f\ [x_1, x_2, ..., x_n] & \{def.\ abs_S\} \\ = & [f\ x_1, f\ x_2, ..., f\ x_n] & \{def.\ map\} \\ = & abs_S\ \lfloor f\ x_1, f\ x_2, ..., f\ x_n \rfloor & \{def.\ abs_S\} \\ = & abs_S\ (smap\ f\ \lfloor x_1, x_2, ..., x_n \rfloor) & \{def.\ smap\} \\ = & (abs_S\ \circ\ smap\ f)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ \circ\} \end{array}$$

This highlights an important equivalence which will be useful in later proofs:

$$map\ f\ \circ\ abs_S\ =\ abs_S\ \circ\ smap\ f$$

We shall term this $smap - r1$. This property can be of particular benefit to us, as it allows us to translate commonly used map laws in terms of *smap*. Take for example map distributivity.

$$\begin{array}{ll} (abs_S\ \circ\ smap\ (f\ \circ\ g))\ \lfloor x_1, x_2, ..., x_n \rfloor & \\ abs_S\ \lfloor (f\ \circ\ g)\ x_1, (f\ \circ\ g)\ x_2, ..., (f\ \circ\ g)\ x_n \rfloor & \{def.\ smap\} \\ [(f\ \circ\ g)\ x_1, (f\ \circ\ g)\ x_2, ..., (f\ \circ\ g)\ x_n] & \{def.\ abs_S\} \\ map\ (f\ \circ\ g)\ [x_1, x_2, ..., x_n] & \{def.\ map\} \\ (map\ f\ \circ\ map\ g)\ [x_1, x_2, ..., x_n] & \{map\ distributivity\} \\ (map\ f\ \circ\ map\ g\ \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ abs_S\} \\ (map\ f\ \circ\ abs_S\ \circ\ smap\ g)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{smap - r1\} \\ (abs_S\ \circ\ smap\ f\ \circ\ smap\ g)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{smap - r1\} \end{array}$$

Form this we can assert that:

$$smap\ (f\ \circ\ g)\ =\ smap\ f\ \circ\ smap\ g$$

Thus map distributivity holds for *smap*.

Process Refinement

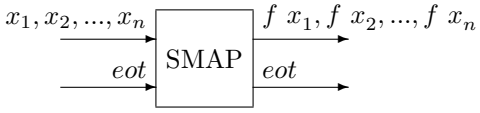


Figure 5.1: The map process for streams.

A process implementing the functionality of *map* f in stream terms should input a stream of values, and output a stream of values with the function f applied. This is depicted in Figure 5.1.

Simple Case Definition

To illustrate this behaviour, let us consider a process candidate for $SMAP$ in the simple case that the members of the stream are items. Here, at each stage, if the input *eot* channel is willing to communicate, we receive a message from it, echo the message to the output *eot* channel, and then skip. If the input value channel is willing to communicate, we receive a single value from it, and output the result of f applied to that value on the output value channel, then repeat. In this version, the parameter f can be passed as a function parameter to $SMAP$.

$$\begin{aligned} SMAP(f) \quad = \mu\, X \bullet \quad & in.eot\ ?\ any \rightarrow out.eot\ !\ any \rightarrow SKIP \\ & \mid \\ & in.value\ ?\ x \rightarrow out.value\ !\ (f\ x) \rightarrow X \end{aligned}$$

General Case Definition

In the more general case the handling of the *eot* channels will be the same. However, the handling of the value conduit will vary depending on the type of the elements of the input and output stream. In the simple case above, the process inputs an item and outputs an item at each stage. In practice, the process might be required to input and output whole streams or vectors at each stage, and the input and output types may differ. Given *map* is polymorphic, it has no knowledge about the structure of individual items in the list - only the function f passed as a parameter to *map* will have the required know how to be able to input and output such values. As such, it is necessary to pass a refinement of the original function f *as a process*, rather than as a function. Thus the process F will be responsible for reading from the input value conduit of the input stream (which may itself be a stream or vector), and outputting to the output value conduit, with the function f applied. Note the use of process choice ($\Box$) here in place of prefix choice ($\mid$) earlier. This substitution was necessary given the level of abstraction placed around the input value conduit - at this level, we no longer have two visible events to choose between.

$$\begin{aligned} SMAP(F) \quad = \mu\, X \bullet \quad & in.eot\ ?\ any \rightarrow out.eot\ !\ any \rightarrow SKIP \\ & \Box \\ & F[in.value/in, out.value/out]; X \end{aligned}$$

Given a process F, with alphabet:

$$\alpha F \ = \ \{in :: \underline{A}, out :: \underline{B}\}$$

The alphabet of $SMAP(F)$ can be defined as follows:

$$\alpha SMAP(F) \ = \ \{in :: \underline{\lfloor A \rfloor}, out :: \underline{\lfloor B \rfloor}\}$$

Proof

To prove this definition of the process $SMAP$ is a valid refinement of the function *smap*, we shall require that the following diagram commutes:

$$\begin{array}{ccc} \lfloor A \rfloor & \xrightarrow{smap\ f} & \lfloor A \rfloor \\ \downarrow Prd & & \downarrow Prd \\ Process & \xrightarrow{\triangleright\ SMAP(F)} & Process \end{array}$$

In other words, we require that the following equivalence holds for any stream s, given a process F which is a valid refinement of the function f:

$$(Prd\ s) \ \triangleright \ SMAP(F) \ = \ Prd\ (smap\ f\ s)$$

The proof of this proceeds as follows. Firstly let us consider the case of the empty stream. We start by substituting the empty stream for s in the above expression.

$$(Prd\ (Stream\ [])) \ \triangleright \ SMAP(F)$$

Now let us expand our definition of $SMAP$ into the above.

$$\begin{aligned} & (out.eot\ !\ any \rightarrow SKIP) \ \triangleright \\ & (\mu\, X \ \bullet \ (in.eot\ ?\ any \rightarrow out.eot\ !\ any \rightarrow SKIP) \\ & \Box \\ & F[in.value/in, out.value/out]; X) \end{aligned}$$

Following this we may find it useful to apply the definition of the feed operator ($\triangleright$):

$$
\begin{aligned}
&((mid.eot\ !\ any \rightarrow SKIP)\ ||\\
&(\mu X\ \bullet\ (mid.eot\ ?\ any \rightarrow out.eot\ !\ any \rightarrow SKIP)\\
&\Box\\
&F[mid.value/in, out.value/in]; X))\backslash\{mid\}
\end{aligned}
$$

Given the willingness of the two processes to communicate on *mid.eot*, it is clear which branch of the choice will be followed. This leads to:

$$(mid.eot\ !\ any \rightarrow out.eot\ !\ any \rightarrow SKIP)\backslash\{mid\}$$

Given that *mid* is hidden, from the outside world, this is equivalent to the following:

$$out.eot\ !\ any \rightarrow SKIP$$

Which is the same as the definition of Prd when applied to an empty stream:

$$Prd\ (Stream\ [])$$

Finally, given that *smap* applied to an empty stream will result in an empty stream, we have the following:

$$Prd\ (smap\ f\ (Stream\ []))$$

In the case of the non-empty stream, we will need to prove the equivalence:

$$(Prd\ x)\ \triangleright\ F\ =\ Prd\ (f\ x)$$

Where x is any member of the input stream. This equates to F being a valid refinement of f, which we have already stated as a requirement for $SMAP(F)$ to be a valid refinement of *smap* f. Thus we can claim $SMAP(F)$ to be a valid refinement of *smap* f in all cases.

Handel-C Implementation

Given that we now have a valid refinement in CSP terms for *map*, let us consider how we may implement it in Handel-C.

First, let us consider the simple case where we are dealing with a stream of items. Here we have a Handel-C `macro proc`, parameterised with the input stream, the output stream, and the expression `f`. The expression `f` here could be defined as a Handel-C `macro expr`.

The process hinges around a loop which terminates when the variable `eot` is set to true. At each step of the loop, we wait until either the *eot* or *value* channel of the input stream is willing to communicate. If the *eot* channel is willing to communicate, we consume the input from it and store it in the variable `eot`, remembering also to output an EOT message for the output stream.

```
macro proc SMAP_SIMPLE (streamin,streamout,f)
{
   Bool eot;
   messagetype (streamin) x;
   eot  = False;
   do
   {
      prialt
      {
      case streamin.eot ? eot:
         streamout.eot ! True;
         break;
      case streamin.value ? x:
         streamout.value ! f(x);
         break;
      }
   } while (!eot)
}
```

Figure 5.2: The simple case definition of the process SMAP.

If the value channel of the input stream is willing to communicate, we consume a value from the input stream, and output that value with f applied to it on the output stream. The definition is given in Figure 5.2.

More generally we have the following process. Here, the function f is refined to a process `F`, rather than an expression. When the *eot* channel of the input stream is not willing to communicate, we engage instead `F`, passing it as parameters the value conduit of the input and out streams. This is given in Figure 5.3.

```
macro proc SMAP (streamin,streamout,F)
{
   Bool eot;
   eot  = False;
   do
   {
      prialt
      {
      case streamin.eot ? eot:
         streamout.eot ! True;
         break;
      default:
         F (streamin.value,streamout.value);
         break;
      }
   } while (!eot)
}
```

Figure 5.3: The general case definition of the process SMAP.

This behaviour may incur a potential problem if there is some delay between the feeding process

outputting the last component of the stream and EOT being signalled. It may then occur that at some point in time, the *eot* conduit is not yet willing to communicate, but all the values in the stream have already been transmitted. To cope with this the process `F` here will require some kind of pass-through mechanism. That is to say, if the value channel (or conduit) of the input stream is not willing to communicate, the process `F` must not wait indefinitely for it.

Handel-C's `prialt` statement has provision for this. This has two behaviours depending on the presence or lack of a default case. Where no default case is present, execution will wait until one of the specified channels is willing to communicate. Where a default case is present, the choice is made without waiting. If none of the specified channels are willing to communicate immediately, execution will proceed instead to the default clause.

As an example, if we wish to use our generic version of `SMAP` to implement a map operation on a simple stream of items, applying some function f, our definition for the process `F` should be as shown in Figure 5.4.

```
macro proc F (conduitin,conduitout)
{
   prialt
   {
   case conduitin ? x:
      conduitout ! f (x);
      break;
   default:
      break;
   }
}
```

Figure 5.4: The process F.

5.2.2 Vectors

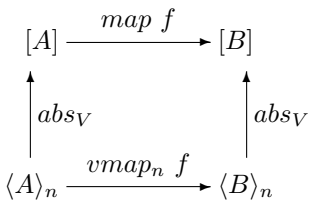
Taking again the example of *map* f, we have a refinement in vector terms called *vmap* f. This has type:

$$\langle A \rangle_n \rightarrow \langle B \rangle_n$$

In a similar fashion to *smap*, an informal definition is analogous to that of *map*.

$$vmap\ f\ \langle x_1, x_2, ..., x_n \rangle_n\ =\ \langle f\ x_1, f\ x_2, ..., f\ x_n \rangle_n$$

Strictly speaking we should write *vmap* as $vmap_n$, as we are required to parameterise the function with the size of the vector used, however, we will generally omit the subscript where the value is clear from context. Proof that this is a valid refinement of *map* requires, as before, the following diagram to commute.



We shall see that the proof is analogous to that of *smap*.

$$
\begin{array}{lll}
 & (map\ f\ \circ\ abs_V)\ \langle x_1, x_2, ..., x_n\rangle_n & \{id\} \\
= & map\ f\ [x_1, x_2, ..., x_n] & \{def.\ abs_V\} \\
= & [f\ x_1, f\ x_2, ..., f\ x_n] & \{def.\ map\} \\
= & abs_V\ \langle f\ x_1, f\ x_2, ..., f\ x_n\rangle_n & \{def.\ abs_V\} \\
= & abs_V\ (vmap\ f\ \langle x_1, x_2, ..., x_n\rangle_n) & \{def.\ vmap\} \\
= & (abs_V\ \circ\ vmap\ f)\ \langle x_1, x_2, ..., x_n\rangle_n & \{def.\ \circ\}
\end{array}
$$

Process Refinement

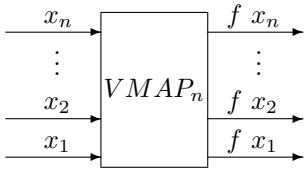


Figure 5.5: The map process for vectors.

So we have established that the functionality of *map f* in a list setting is modeled by *vmap f* in the vector setting. We should now consider a process refinement of *vmap f*. We shall require a process F, which is a valid refinement of our function f. The implementation of $VMAP$ can then proceed by composing together n instances of F together in parallel, and directing an item from the input vector to each instance for processing. We should also ensure that the output of each instance of F can proceed along a separate conduit in the output vector. In CSP we have:

$$VMAP_n(F) \quad = \quad \underset{i=1}{\overset{n}{||}} \quad F[in_i/in, out_i/out]$$

Given a process F, with alphabet:

$$\alpha F \ = \ \{in :: \underline{A}, out :: \underline{B}\}$$

The alphabet of $VMAP_n(F)$ can be defined as follows:

$$\alpha VMAP_n(F) \ = \ \{in :: \underline{\langle A\rangle_n}, out :: \underline{\langle B\rangle_n}\}$$

Proof

To prove this is a valid refinement, we shall require that the following diagram commutes:

$$\begin{array}{ccc} \langle A \rangle_n & \xrightarrow{vmap\ f} & \langle A \rangle_n \\ \Big\downarrow Prd & & \Big\downarrow Prd \\ Process & \xrightarrow{\triangleright\ VMAP_n(F)} & Process \end{array}$$

In other words, we require that the following equivalence holds for any vector v of length n, given a process F which is a valid refinement of the function f:

$$(Prd\ v)\ \triangleright\ VMAP_n(F)\ =\ Prd\ (vmap\ f\ v)$$

Expanding the definitions of Prd and $VMAP$, the left hand side of the above expression is equivalent to:

$$\left(\begin{array}{ll} \#v & \\ || & (Prd\ v_i)[out_i/out] \\ i = 1 & \end{array} \right) \triangleright \left(\begin{array}{ll} n & \\ || & F[in_i/in, out_i/out] \\ i = 1 & \end{array} \right)$$

Given as we have already stated that the length of the vector v is equal to n, we may slightly modify this definition to create the following:

$$\left(\begin{array}{ll} n & \\ || & (Prd\ v_i)[out_i/out] \\ i = 1 & \end{array} \right) \triangleright \left(\begin{array}{ll} n & \\ || & F[in_i/in, out_i/out] \\ i = 1 & \end{array} \right)$$

Given the law previously defined for feeding between vector processes, we can simplify to the following:

$$\begin{array}{ll} \#v & \\ || & (Prd\ v_i)\ \triangleright\ F[out_i/out] \\ i = 1 & \end{array}$$

Given the requirement that F is a valid refinement of f, then the following equivalence must hold for any item x of the appropriate type.

$$(Prd\ x)\ \triangleright\ F\ =\ (Prd\ (f\ x))$$

So the previous definition becomes:

$$\begin{array}{ll} \#v & \\ || & (Prd\ (f\ v_i))[out_i/out] \\ i = 1 & \end{array}$$

This is clearly an instance of the vector version of Prd, and as f is applied to every item in the vector, it is evident the above is equivalent to:

$$Prd\ (vmap\ f\ v)$$

This is the right hand side of the original expression, thus proving that the previous diagram commutes.

Handel-C Implementation

We may now turn our attention to providing a definition in Handel-C for the behaviour of this process. Here we can employ Handel-C's enumerated `par` construct to place n instances of the process F together in parallel. Each instance is passed the corresponding conduits from both the input and output conduits. This definition is given in Figure 5.6.

```
macro proc VMAP (size,vectorin, vectorout, F)
{
   typeof (size) c;
   par (c=0;c<size;c++)
   {
      F(vectorin[c],vectorout[c]);
   }
}
```

Figure 5.6: The Handel-C implementation of the process VMAP.

5.2.3 Streams to Vectors

In certain situations we may require a specialised refinement of $map\ f$ which inputs a stream and outputs a vector. This has type:

$$s2vmap_n\ f :: \lfloor A \rfloor \rightarrow \langle B \rangle_n$$

An informal definition of $s2vmap_n$ is a simple analogy of map:

$$s2vmap_n\ f\ \lfloor x_1, x_2, ..., x_n \rfloor\ =\ \langle f\ x_1, f\ x_2, ..., f\ x_n \rangle_n$$

Proof this is a valid refinement requires the following diagram to commute:

$$\begin{array}{ccc} [A] & \xrightarrow{map\ f} & [B] \\ \uparrow{\scriptstyle abs_S} & & \uparrow{\scriptstyle abs_V} \\ \lfloor A \rfloor & \xrightarrow{s2vmap_n\ f} & \langle B \rangle_n \end{array}$$

We can provide a proof for this as follows:

$$\begin{array}{lll} & (map\ f\ \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\ = & map\ f\ [x_1, x_2, ..., x_n] & \{def.\ abs_S\} \\ = & [f\ x_1, f\ x_2, ..., f\ x_n] & \{def.\ map\} \\ = & abs_V\ \langle f\ x_1, f\ x_2, ..., f\ x_n \rangle_n & \{def.\ abs_V\} \\ = & abs_V\ (s2vmap_n\ f\ \lfloor x_1, x_2, ..., x_n \rfloor) & \{def.\ s2vmap_n\} \\ = & (abs_V\ \circ\ s2vmap_n\ f)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ \circ\} \end{array}$$

Process Refinement

A process refinement for this specialised refinement of *map*, $s2vmap_n$, can be supplied in the form of the process $S2VMAP_n$, depicted in Figure 5.7.

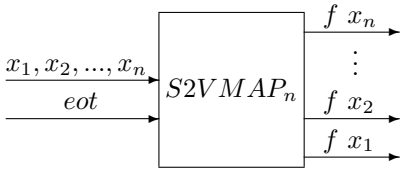


Figure 5.7: The map process with stream input and vector output.

5.2.4 Vectors to Streams

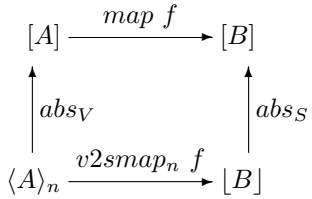
Similarly we may on occasion require a specialised refinement of $map\ f$ which inputs a vector and outputs a stream. This has type:

$$v2smap_n\ f :: \langle A \rangle_n \rightarrow \lfloor B \rfloor$$

An informal definition of $v2smap_n$ is a simple analogy of *map*:

$$v2smap_n\ f\ \langle x_1, x_2, ..., x_n \rangle_n\ =\ \lfloor f\ x_1, f\ x_2, ..., f\ x_n \rfloor$$

Proof this is a valid refinement requires the following diagram to commute:



We can provide a proof for this as follows:

$$
\begin{array}{lll}
 & (map\ f\ \circ\ abs_V)\ \langle x_1, x_2, ..., x_n\rangle_n & \{id\} \\
= & map\ f\ [x_1, x_2, ..., x_n] & \{def.\ abs_V\} \\
= & [f\ x_1, f\ x_2, ..., f\ x_n] & \{def.\ map\} \\
= & abs_S\ \lfloor f\ x_1, f\ x_2, ..., f\ x_n \rfloor & \{def.\ abs_S\} \\
= & abs_S\ (v2smap_n\ f\ \langle x_1, x_2, ..., x_n\rangle_n) & \{def.\ v2smap_n\} \\
= & (abs_S\ \circ\ v2smap_n\ f)\ \langle x_1, x_2, ..., x_n\rangle_n & \{def.\ \circ\}
\end{array}
$$

Process Refinement

The specialised refinement of *map*, $v2smap_n$, was introduced in Section 5.2.4. A process refinement for this function will be supplied in the form of the process $V2SMAP_n$, depicted in Figure 5.8.

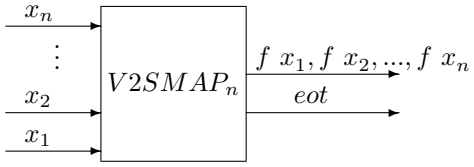


Figure 5.8: The map process with vector input and stream output.

5.2.5 Combined Structures

Let us consider the act of mapping some function f (of type $A \rightarrow B$) to every item of every sub segment of a two dimensional structure (for example, a list of lists). As such, in list terms we have an expression we shall term *tdmap* (two dimensional map):

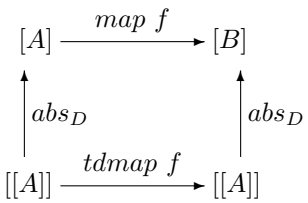
$$tdmap\ f\ =\ map\ (map\ f)$$

The expression $tdmap\ f$ has type:

$$[[A]] \rightarrow [[A]]$$

5.2.6 Distributed Lists

The function *tdmap* should provide a valid refinement for *map* in a distributed list setting. Proof that this is a valid refinement of *map* requires, as always, the following diagram to commute.



We can prove the diagram commutes for this definition of *tdmap*, along with our first definition of abs_D as follows. Here $[s_1, s_2, ..., s_n]$ represents a distributed list (a list of lists) where s_1 is the first sub-segment, s_2 is the second segment and so on.

$$\begin{array}{lll}
 & (map\ f\ \circ\ abs_{D1})\ [s_1, s_2, ..., s_n] & \{id\} \\
= & (map\ f\ \circ\ (+\!\!+)/\ )\ [s_1, s_2, ..., s_n] & \{def.\ abs_{D1}\} \\
= & ((+\!\!+)/\ \circ\ map\ (map\ f))\ [s_1, s_2, ..., s_n] & \{map\ promotion\} \\
= & ((+\!\!+)/\ \circ\ tdmap\ f)\ [s_1, s_2, ..., s_n] & \{def.\ tdmap\} \\
= & (abs_{D1} \circ\ tdmap\ f)\ [s_1, s_2, ..., s_n] & \{def.\ abs_{D1}\}
\end{array}$$

5.2.7 Stream of Streams

In stream terms, we could refine $tdmap\ f$ to a function $ssmap\ f$. This has type:

$$\lfloor\lfloor A\rfloor\rfloor \rightarrow \lfloor\lfloor B\rfloor\rfloor$$

and can be defined as follows:

$$ssmap\ f\ = smap\ (smap\ f)$$

To prove this is a valid refinement, the following diagram must commute:

$$\begin{array}{ccc}
[[A]] & \xrightarrow{map\ (map\ f)} & [[B]] \\
\uparrow{\scriptstyle abs_{SS}} & & \uparrow{\scriptstyle abs_{SS}} \\
\lfloor\lfloor A\rfloor\rfloor & \xrightarrow{ssmap\ f} & \lfloor\lfloor B\rfloor\rfloor
\end{array}$$

Given a stream of streams denoted $\lfloor s_1, ..., s_n\rfloor$, the proof may proceed as follows:

$$\begin{array}{lll}
 & (map\ (map\ f)\ \circ\ abs_{SS})\ \lfloor s_1, ..., s_n\rfloor & \\
= & map\ (map\ f)\ [l_1, ..., l_n] & \{def.\ abs_{SS}\} \\
= & [map\ f\ l_1, ..., map\ f\ l_n] & \{def.\ map\} \\
= & abs_S\ \lfloor map\ f\ l_1, ..., map\ f\ l_n\rfloor & \{def.\ abs_S\} \\
= & abs_S\ \lfloor (map\ f\ \circ\ abs_S)\ s_1, ..., (map\ f\ \circ\ abs_S)\ s_n\rfloor & \{def.\ abs_S\} \\
= & abs_S\ \lfloor (abs_S\ \circ\ smap\ f)\ s_1, ..., (abs_S\ \circ\ smap\ f)\ s_n\rfloor & \{smap - r1\} \\
= & abs_S\ \circ\ smap\ (abs_S\ \circ\ smap\ f)\ \lfloor s_1, ..., s_n\rfloor & \{def.\ smap\} \\
= & abs_S\ \circ\ smap\ abs_S\ \circ\ smap\ (smap\ f)\ \lfloor s_1, ..., s_n\rfloor & \{smap\ distributivity\} \\
= & map\ abs_S\ \circ\ abs_S\ \circ\ smap\ (smap\ f)\ \lfloor s_1, ..., s_n\rfloor & \{smap - r1\} \\
= & abs_{SS}\ \circ\ smap\ (smap\ f)\ \lfloor s_1, ..., s_n\rfloor & \{def.\ abs_{SS}\} \\
= & abs_{SS}\ \circ\ ssmap\ f\ \lfloor s_1, ..., s_n\rfloor & \{def.\ ssmap\}
\end{array}$$

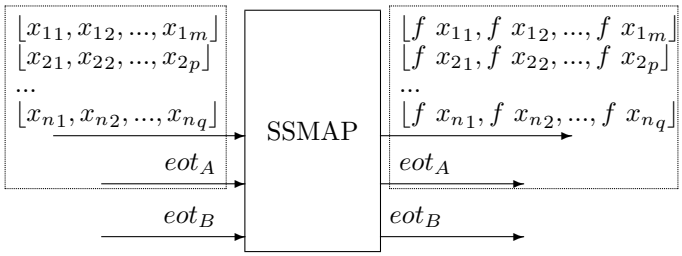


Figure 5.9: The map process for streams of streams.

Process Refinement

We have already shown that wherever F is a valid refinement of f, then $SMAP(F)$ is a valid refinement of *smap* f. It naturally follows then that a refinement of *ssmap* f, the map function in stream of streams terms, can be defined quite intuitively:

$$SSMAP(F) \;=\; SMAP(SMAP(F))$$

Given a process F, with alphabet:

$$\alpha F \;=\; \{in :: \underline{A}, out :: \underline{B}\}$$

The alphabet of $SSMAP(F)$ can be defined as follows:

$$\alpha SSMAP(F) \;=\; \{in :: \underline{\lfloor\lfloor A \rfloor\rfloor}, out :: \underline{\lfloor\lfloor B \rfloor\rfloor}\}$$

It may be useful to explore the behaviour of $SSMAP$ a little. From the point of view of the outer $SMAP$, we have a process which repeatedly applies $SMAP(F)$ to the input, until the end of transmission is signalled. In doing so, at each step of the outer $SMAP$, an entire stream is input, processed and output.

Handel-C Implementation

Ideally, we'd be able to construct a definition for $SSMAP$ in Handel-C entirely analogous to that we used in CSP. Unfortunately, as is typical of imperative languages, Handel-C does not support currying. So we would like to be able to give a definition along the lines of the following:

```
macro proc SSMAP (streamofstreamsin,streamofstreamsout,F)
{
   SMAP (streamofstreamsin,streamofstreamsout,SMAP(F));
}
```

The only definition Handel-C will natively understand requires expansion of one level of $SMAP$, thus we the definition given in Figure 5.10.

```
macro proc SSMAP (streamofstreamsin,streamofstreamsout,F)
{
   Bool eot;
   eot  = False;
   do
   {
      prialt
      {
      case streamofstreamsin.eot ? eot:
         streamofstreamsout.eot ! True;
         break;
      default:
         SMAP (streamofstreamsin.value,
               streamofstreamsout.value,
               F);
         break;
      }
   } while (!eot)
}
```

Figure 5.10: The simple Handel-C definition of the process SMAP.

5.2.8 Vector of Streams

In vector of stream terms, we could refine *tdmap f* to a function *vsmap f*. This has type:

$$\langle \lfloor A \rfloor \rangle_n \rightarrow \langle \lfloor B \rfloor \rangle_n$$

and can be defined as follows:

$$vsmap\ f\ = vmap\ (smap\ f)$$

Proof of the validity of this refinement would be analogous to that for the stream of streams, above.

Process Refinement

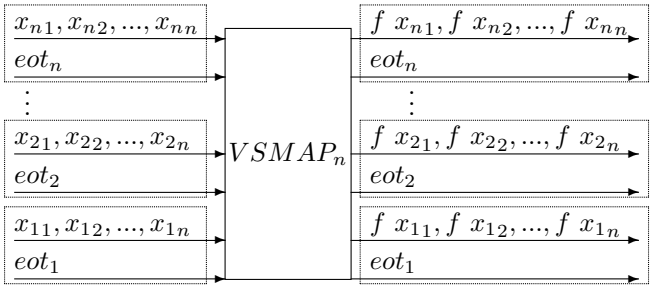


Figure 5.11: The map process for vectors of streams.

Yet again, given $VMAP(F)$ and $SMAP(F)$ forming valid refinements for *vmap f* and *smap f*, we have the following as a refinement of *vsmap f*:

$$VSMAP_n(F) \;=\; VMAP_n(SMAP(F))$$

Given a process F, with alphabet:

$$\alpha F \;=\; \{in :: \underline{A}, out :: \underline{B}\}$$

The alphabet of $VSMAP_n(F)$ can be defined as follows:

$$\alpha VSMAP_n(F) \;=\; \{in :: \underline{\langle \lfloor A \rfloor \rangle_n}, out :: \underline{\langle \lfloor B \rfloor \rangle_n}\}$$

Handel-C Implementation

Bearing the usual constraints in mind, our Handel-C definition can proceed as given in Figure 5.12.

```
macro proc VSMAP (size,vectorofstreamsin, vectorofstreamsout, F)
{
   par (c=0;c<size;c++)
   {
      SMAP (vectorofstreamsin[c],
            vectorofstreamsout[c],
            F);
   }
}
```

Figure 5.12: The Handel-C definition of the process VSMAP.

5.2.9 Stream of Vectors

In stream of vector terms, we could refine *tdmap f* to a function *svmap f*. This has type:

$$\lfloor \langle A \rangle_n \rfloor \rightarrow \lfloor \langle B \rangle_n \rfloor$$

and can be defined as follows:

$$svmap\ f\ = smap\ (vmap\ f)$$

Proof of the validity of this refinement would be analogous to that for the stream of streams, above.

Process Refinement

As before, having already proven $VMAP(F)$ is a valid refinement of *vmap f*, wherever F is a valid refinement of f, we can infer the following as a valid refinement of *svmap f*:

$$SVMAP_n(F) \;=\; SMAP(VMAP_n(F))$$

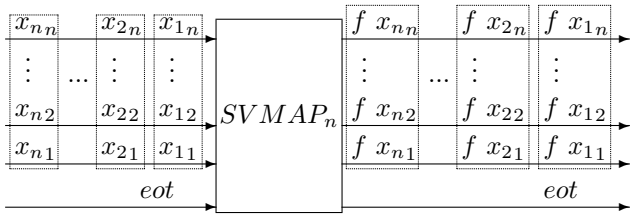


Figure 5.13: The map process for streams of vectors.

Given a process F, with alphabet:

$$\alpha F \ = \ \{in :: \underline{A}, out :: \underline{B}\}$$

The alphabet of $SVMAP_n(F)$ can be defined as follows:

$$\alpha SVMAP_n(F) \ = \ \{in :: \underline{\lfloor \langle A \rangle_n \rfloor}, out :: \underline{\lfloor \langle B \rangle_n \rfloor}\}$$

Handel-C Implementation

Again, the lack of currying in Handel-C will require us to provide an expanded definition, as demonstrated in Figure 5.14.

```
macro proc SVMAP (vectorsize,streamofvectorsin,streamofvectorsout,F)
{
   Bool eot;
   eot  = False;
   do
   {
      prialt
      {
      case streamofvectorsin.eot ? eot:
         streamofvectorsout.eot ! True;
         break;
      default:
         VMAP (vectorsize,
               streamofvectorsin.value,
               streamofvectorsout.value,
               F);
         break;
      }
   } while (!eot)
}
```

Figure 5.14: The Handel-C definition of the process SVMAP.

5.2.10 Vectors of Vectors

In vector of vector terms, we could refine *tdmap* f to a function *vvmap* f. This has type:

$$\langle\langle A\rangle_m\rangle_n \rightarrow \langle\langle B\rangle_m\rangle_n$$

and can be defined as follows:

$$vvmap\ f\ = vmap\ (vmap\ f)$$

Proof of the validity of this refinement would be analogous to that for the stream of streams, above.

Process Refinement

Finally, the vector of vector case, for a refinement of $vvmap\ f$, is no exception.

$$VVMAP_{(n,m)}(F)\ =\ VMAP_n(VMAP_m(F))$$

Given a process F, with alphabet:

$$\alpha F\ =\ \{in :: \underline{A}, out :: \underline{B}\}$$

The alphabet of $VVMAP_{(n,m)}(F)$ can be defined as follows:

$$\alpha VVMAP_{(n,m)}(F)\ =\ \{in :: \underline{\langle\langle A\rangle_m\rangle_n}, out :: \underline{\langle\langle B\rangle_m\rangle_n}\}$$

Handel-C Implementation

The Handel-C implementation is given in Figure 5.15.

```
macro proc VVMAP (sizen,sizem,vectorofvectorsin, vectorofvectorsout, F)
{
   typeof (sizen) c;
   par (c=0;c<sizen;c++)
   {
      VMAP (sizem,
            vectorofvectorsin[c],
            vectorofvectorsout[c],
            F);
   }
}
```

Figure 5.15: The Handel-C definition of the process VVMAP.

5.3 Fold

The fold family of functions are also known as catamorphisms. In list terms, the function $fold$ (also written /) has the following type:

$$fold \ :: \ (A \to A \to A) \to [A] \to A$$

Informally we can define it as follows:

$$fold \ (\oplus) \ [x_1, x_2, ..., x_n] = x_1 \oplus x_2 \oplus ... \oplus x_n$$

This function is usually implemented in terms of one of four common variants, $foldr$, $foldl$, $foldr1$ and $foldl1$. Firstly, let us examine $foldr$.

$$foldr :: (A \to B \to B) \to B \to [A] \to B$$

Here evaluation starts from the right, and a base value e is employed:

$$foldr \ (\oplus) \ e \ [x_1, x_2, ..., x_n] = x_1 \oplus (x_2 \oplus (... \oplus (x_n \oplus e)))$$

Next we have $foldl$.

$$foldl :: (A \to B \to A) \to A \to [B] \to A$$

Again a base value e is employed, and in this version, evaluation starts from the left:

$$foldl \ (\oplus) \ e \ [x_1, x_2, ..., x_n] = (((e \oplus x_1) \oplus x_2) \oplus ...) \oplus x_n$$

For non-empty lists, we can omit the base value e, to provide a further two variations. First we have $foldr1$.

$$foldr1 :: (A \to A \to A) \to [A] \to A$$

This has the following definition, evaluating from the right:

$$foldr1 \ (\oplus) \ [x_1, x_2, ..., x_n] = x_1 \oplus (x_2 \oplus (... \oplus (x_{n-1} \oplus x_n)))$$

Finally we have $foldl1$:

$$foldl1 :: (A \to A \to A) \to [A] \to A$$

This can be defined as follows, evaluating from the left:

$$foldl1 \ (\oplus) \ [x_1, x_2, ..., x_n] = (((x_1 \oplus x_2) \oplus x_3) \oplus ...) \oplus x_n$$

There exists certain relationships between the $fold$ variants which we may find useful. Each left or right variant may be expressed in terms of its opposite, by reversing the input list (using $reverse$) and switching the operand order of $\oplus$ (using the function $flip$):

$$
\begin{array}{lcl}
foldr\ (\oplus)\ e & = & foldl\ (flip\ (\oplus))\ e \circ reverse \\
foldl\ (\oplus)\ e & = & foldr\ (flip\ (\oplus))\ e \circ reverse \\
foldr1\ (\oplus) & = & foldl1\ (flip\ (\oplus)) \circ reverse \\
foldl1\ (\oplus) & = & foldr1\ (flip\ (\oplus)) \circ reverse
\end{array}
$$

To summarise, the types of the four $fold$ variants are as follows:

$$
\begin{array}{lcl}
foldr & :: & (A \to B \to B) \to B \to [A] \to B \\
foldl & :: & (A \to B \to A) \to A \to [B] \to A \\
foldr1 & :: & (A \to A \to A) \to [A] \to A \\
foldl1 & :: & (A \to A \to A) \to [A] \to A
\end{array}
$$

Their informal definitions are as follows:

$$
\begin{array}{lcl}
foldr\ (\oplus)\ e\ [x_1, x_2, ..., x_n] & = & x_1 \oplus (x_2 \oplus (... \oplus (x_n \oplus e))) \\
foldl\ (\oplus)\ e\ [x_1, x_2, ..., x_n] & = & (((e \oplus x_1) \oplus x_2) \oplus ...) \oplus x_n \\
foldr1\ (\oplus)\ [x_1, x_2, ..., x_n] & = & x_1 \oplus (x_2 \oplus (... \oplus (x_{n-1} \oplus x_n))) \\
foldl1\ (\oplus)\ [x_1, x_2, ..., x_n] & = & (((x_1 \oplus x_2) \oplus x_3) \oplus ...) \oplus x_n
\end{array}
$$

As with *map*, we may, from time to time, find a binary infix operator version of any of the $fold$ functions useful. As before, we may make use of the Haskell convention of surrounding a function name in single quotes, for example:

$$(\oplus)\ `fold`\ xs\ =\ fold\ (\oplus)\ xs$$

Or alternatively we may resort to BMF style operators:

$$
\begin{array}{lcl}
(\oplus)/\ xs & = & fold\ (\oplus)\ xs \\
((\oplus)\not\rightarrow e)\ xs & = & foldl\ (\oplus)\ e\ xs \\
((\oplus)\not\leftarrow e)\ xs & = & foldr\ (\oplus)\ e\ xs \\
(\oplus)\not\rightarrow xs & = & foldl1\ (\oplus)\ xs \\
(\oplus)\not\leftarrow xs & = & foldr1\ (\oplus)\ xs
\end{array}
$$

One useful feature of the fold family of functions is that they can also be implemented as a series of steps composed together. Take for example $foldr$:

$$foldr\ (\oplus)\ e\ [x_1, x_2, ..., x_n]\ =\ x_1 \oplus (x_2 \oplus (... \oplus (x_n \oplus e)))$$

Using sections, we can re-package this definition into a series of steps composed together, as follows:

$$foldr\ (\oplus)\ e\ [x_1, x_2, ..., x_n]\ =\ ((x_1\oplus) \circ (x_2\oplus) \circ ... \circ (x_n\oplus))\ e$$

This we can then simplify using a map and a fold to arrive at the following:

$$foldr\ (\oplus)\ e\ [x_1, x_2, ..., x_n]\ =\ ((\circ)/\ (map\ (\oplus)\ [x_1, x_2, ..., x_n]))\ e$$

Similarly, consider $foldl$:

$$\begin{aligned} foldl\ (\oplus)\ e\ [x_1, x_2, ..., x_n] &= (((e \oplus x_1) \oplus x_2) \oplus ...) \oplus x_n \\ &= ((\oplus\ x_n) \circ (\oplus\ x_{n-1}) \circ ... \circ (\oplus\ x_2) \circ (\oplus\ x_1))\ e \\ &= ((\circ)/\ (map\ (flip\ (\oplus))\ (reverse\ [x_1, x_2, ..., x_n])))\ e \end{aligned}$$

5.3.1 Streams

A stream refinement of $fold$, $sfold$, could have the following type:

$$sfold\ ::\ (A \to A \to A) \to \lfloor A \rfloor \to A$$

As before, the implementation could proceed informally as follows:

$$sfold\ (\oplus)\ \lfloor x_1, x_2, ..., x_n \rfloor = x_1 \oplus x_2 \oplus ... \oplus x_n$$

To prove this is a valid refinement of $fold$, we require that the following diagram commutes.

$$\begin{array}{ccc} [A] & \xrightarrow{fold\ (\oplus)} & A \\ \uparrow{\scriptstyle abs_S} & & \uparrow{\scriptstyle id} \\ \lfloor A \rfloor & \xrightarrow{sfold\ (\oplus)} & A \end{array}$$

We can prove this as follows:

$$\begin{aligned} & (fold\ (\oplus)\ \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor && \{id\} \\ = \ & fold\ (\oplus)\ [x_1, x_2, ..., x_n] && \{def.\ abs_S\} \\ = \ & x_1 \oplus x_2 \oplus ... \oplus x_n && \{def.\ fold\} \\ = \ & sfold\ (\oplus)\ \lfloor x_1, x_2, ..., x_n \rfloor && \{def.\ sfold\} \\ = \ & (id\ \circ\ sfold\ (\oplus))\ \lfloor x_1, x_2, ..., x_n \rfloor && \{id\} \end{aligned}$$

Similarly, we could prove refinements in stream terms of the four $fold$ variants. These would have the following types:

$$\begin{aligned} sfoldr &:: (A \to B \to B) \to B \to \lfloor A \rfloor \to B \\ sfoldl &:: (A \to B \to A) \to A \to \lfloor B \rfloor \to A \\ sfoldr1 &:: (A \to A \to A) \to \lfloor A \rfloor \to A \\ sfoldl1 &:: (A \to A \to A) \to \lfloor A \rfloor \to A \end{aligned}$$

Their informal definitions are as follows:

$$
\begin{aligned}
sfoldr\ (\oplus)\ e\ \lfloor x_1, x_2, ..., x_n \rfloor &= x_1 \oplus (x_2 \oplus (... \oplus (x_n \oplus e))) \\
sfoldl\ (\oplus)\ e\ \lfloor x_1, x_2, ..., x_n \rfloor &= (((e \oplus x_1) \oplus x_2) \oplus ...) \oplus x_n \\
sfoldr1\ (\oplus)\ \lfloor x_1, x_2, ..., x_n \rfloor &= x_1 \oplus (x_2 \oplus (... \oplus (x_{n-1} \oplus x_n))) \\
sfoldl1\ (\oplus)\ \lfloor x_1, x_2, ..., x_n \rfloor &= (((x_1 \oplus x_2) \oplus x_3) \oplus ...) \oplus x_n
\end{aligned}
$$

Process Refinement

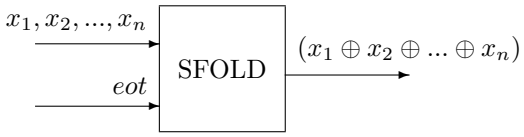


Figure 5.16: The fold process for streams.

A process implementing the functionality of $fold\ (\oplus)$ in stream terms should input a stream of values, and output the result of folding the operator $\oplus$ over the values input. This is depicted in Figure 5.16. As noted, $fold$ will not be generally be directly implemented. Instead we should consider the implementations of the four variants.

Simple Case Definition

To illustrate the behaviour of these functions, let us initially consider a process candidate for $SFOLDL(\oplus, e)$, which is a potential refinement for the function $sfoldl\ (\oplus)\ e$. We shall begin by considering the simple case, where the members of the input stream are items. In this case, we shall pass the first parameter $(\oplus)$ as a function, not a process. The implementation is therefore a simple instance of tail recursion.

$$
\begin{aligned}
SFOLDL(\oplus, e) \quad = \quad & in.eot\ ?\ any \rightarrow out\ !\ e \rightarrow SKIP \\
& | \\
& in.value\ ?\ x \rightarrow SFOLDL(\oplus, e \oplus x)
\end{aligned}
$$

Alternatively, with the introduction of a local variable, we can construct a non-recursive version.

$$
\begin{aligned}
SFOLDL(\oplus, e) \quad = \quad & a := e; \\
& \mu\, X \bullet \\
& in.eot\ ?\ any \rightarrow out\ !\ a \rightarrow SKIP \\
& | \\
& in.value\ ?\ x \rightarrow a := a \oplus x; X
\end{aligned}
$$

The process $SFOLDR$ is not quite so straightforward to implement. The right ordered computation it employs requires that the last value of the list must be known before computation may commence. Recall the relationship between $foldr$ and $foldl$:

$$foldr\ (\oplus)\ e\ =\ foldl\ (flip\ (\oplus))\ e\ \circ\ reverse$$

This gives us one possible implementation for $SFOLDR$ in which we reverse the incoming stream, and flip the order of the operands for the operator $(\oplus)$. So in functional terms, we have the following definition for $sfoldr$:

$$sfoldr\ (\oplus)\ e\ =\ sfoldl\ (flip\ (\oplus))\ e\ \circ\ sreverse$$

This could be refined in process terms by the following definition:

$$SFOLDR(\oplus, e)\ =\ SREVERSE\ \gg\ SFOLDL(flip\ (\oplus), e)$$

The process $SREVERSE$ will have to be implemented via some form of buffering. As a slight improvement on efficiency, we may wish to make that buffering functionality local to $SFOLDR$. Here, at each step, if we can receive a value from the input stream, we append that to the end of the buffer and continue. When EOT is signalled on the input stream, we then perform the fold on our local buffer, and output the result on the output conduit. In this scheme, we would have something akin to the following:

$$\begin{aligned} SFOLDR(\oplus, e)\ =\ & xs := [];\\ & \mu\, X \bullet\\ & in.eot\ ?\ any \rightarrow out\ !\ (sfoldr\ (\oplus)\ e\ xs) \rightarrow SKIP\\ & \mid\\ & in.value\ ?\ x \rightarrow xs := xs +\!\!+ [x]; X \end{aligned}$$

The remaining two variants, $SFOLDR1$ and $SFOLDL1$, can be expressed in terms of their already introduced counterparts. In the case of $SFOLDR1$ we can use the first value received from the input stream as the base value:

$$SFOLDL1(\oplus)\ =\ in.value\ ?\ x\ \rightarrow SFOLDL(\oplus, x)$$

Similarly, for $SFOLDR1$ we can use the *last* value received from the input stream as the base value:

$$\begin{aligned} SFOLDR1(\oplus)\ =\ & SREVERSE\ \gg\\ & (in.value\ ?\ x\ \rightarrow SFOLDL(flip\ (\oplus), x)) \end{aligned}$$

Note that, as with their functional equivalents, both $SFOLDR1$ and $SFOLDL1$ will not operate correctly when fed an empty stream. Deadlock will occur, given that both processes will wait indefinitely for a value from the stream which they will never receive.

General Case Definition

The above group of implementations work fine for simple streams of items. However, for two or more dimensional structures (for example, streams of vectors or streams of streams) they are not general enough. Let us consider some alternatives.

When attempting to generalise process definitions for *map* earlier in this chapter, it became clear that it would be necessary to shift the responsibility for reading from the input stream into the process passed in as a parameter. However, in terms of $fold$ this modification is complicated somewhat by the non-atomic nature of the individual operations. That is to say, we can not deal with any single item from the input stream in complete isolation. It is necessary to maintain some kind of state between successive reads from the input stream.

Perhaps the "purest" way to model this progression of state changes in CSP is through communication. Here each stage of the fold is a process which is responsible for reading in a value from the stream, as well as the current state value, applying the binary operator to these two values, and then outputting the result, which may in turn form the input to the next stage. This is perhaps best illustrated diagrammatically, as can be seen in Figure 5.17. We shall see that this bears a close resemblance to vector implementations for fold introduced later in this section.

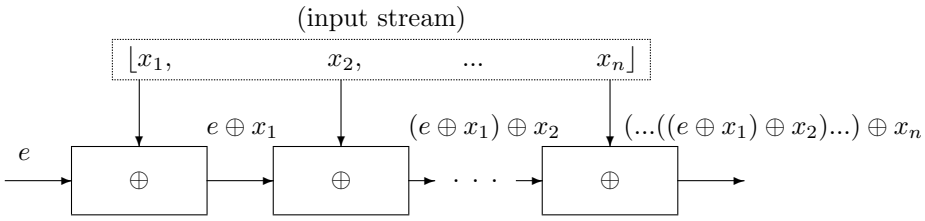


Figure 5.17: A potential communication oriented implementation of the process $SFOLDL$.

In effect we can achieve this kind of definition by composing together two processes. The first takes in a stream and outputs the values it contains as a vector. This can be achieved using $S2VMAP_n$, introduced in Section 5.2.3. Here the function applied to each item in the incoming stream is simply the identity function. The second process is then the vector implementation of a left directed fold. In other words:

$$S2VMAP_n(id) \gg VFOLDL_n(F, e)$$

Although such a definition would be mathematically pleasing in that it adheres well to the principles of functional programming, it will be hard to implement in practice. For a start, there would be a significant communication and process usage overhead, which may seem wasteful. Additionally, we'd effectively be refining this into an n staged pipeline, which would of course break one of the important principles of the stream refinement; that we do not necessarily know how many items we are dealing with.

Let us consider instead a specification which will lend itself to a more efficient implementation. In practice we are much more likely to opt for an approach where the computation proceeds by accumulation in some kind of local variable, akin to our simple case definitions. In CSP, given that we are dealing with a specification language rather than a programming language, we do not normally concern ourselves with many of the technicalities that confront us when writing code to be compiled. In this instance however, we are forced to think in slightly more detail than usual about how CSP processes may affect the state of variables.

In a version of 'pseudo CSP' where processes can return values, we can construct the following general case definition for `SFOLDL`. Here the process `F` passed as a parameter is given the current state variable `a`, and is then responsible for reading a value from the input stream, applying the operator to that value and `a`, and then returning the result.

$$\begin{aligned} SFOLDL(F, e) \quad = \quad & a := e; \\ & \mu\, X \bullet \\ & in.eot\ ?\ any \rightarrow prd\ a \rightarrow SKIP \\ & \Box \\ & a := F(a)[in.value/in]; X \end{aligned}$$

Where e is not a constant, we may benefit from manipulating it directly, allowing us to do without the local variable a. We have:

$$\begin{aligned} SFOLDL(F, e) \quad = \quad & \mu\, X \bullet \\ & in.eot\ ?\ any \rightarrow prd\ e \rightarrow SKIP \\ & \Box \\ & e := F(e)[in.value/in]; X \end{aligned}$$

In an alternative "pseudo CSP", where parameters can be passed to processes by reference rather than purely by value, we can have a definition as follows. Here the process `F` given as a parameter is passed the current state variable `a` as a reference (i.e. it can modify it). It is then responsible for reading a value from the input stream, applying the operator to that value and `a`, and storing the result in the variable `a`. Here parameter passing by reference is denoted with an ampersand (`&`).

$$\begin{aligned} SFOLDL(F, e) \quad = \quad & a := e; \\ & \mu\, X \bullet \\ & in.eot\ ?\ any \rightarrow prd\ a \rightarrow SKIP \\ & \Box \\ & F(\&a)[in.value/in]; X \end{aligned}$$

As before, we may in certain circumstances be able to do without the local variable a.

$$
\begin{aligned}
SFOLDL(F,e) \quad = \quad & \mu\, X \bullet \\
& in.eot\ ?\ any \rightarrow prd\ e \rightarrow SKIP \\
& \Box \\
& F(\&e)[in.value/in]; X
\end{aligned}
$$

Proof

To prove that $SFOLDL(F,e)$ is a valid refinement of $sfoldl\ f\ e$, we shall require the following diagram to commute:

$$
\begin{array}{ccc}
\lfloor B \rfloor & \xrightarrow{sfoldl\ f\ e} & A \\
\Big\downarrow{\scriptstyle prd} & & \Big\downarrow{\scriptstyle prd} \\
Process & \xrightarrow{\rhd\ SFOLDL(F,e)} & Process
\end{array}
$$

That is to say, we require the following equivalence to hold for any stream s, and binary operator f with base value e, where F is a valid refinement of F :

$$prd\ s\ \rhd\ SFOLDL(F,e)\ =\ prd\ (sfoldl\ f\ e\ s)$$

Let us consider our sample case definition for $SFOLDL$. To prove this is a valid refinement we wish to prove the following equivalence holds:

$$prd\ s\ \rhd\ SFOLDL(\oplus,e)\ =\ prd\ (sfoldl\ (\oplus)\ e\ s)$$

As before, in the case of the empty stream, the proof will follow that for $SMAP$. Now for the case of the non empty stream. Let us unwind the definition of $SFOLDL$ in the left hand side of the above equivalence. We shall use the recursive definition as it is easier to reason with, but hopefully it can be appreciated that the two definitions are equivalent. So, to begin with, we have:

$$prd\ s\ \rhd\ SFOLDL(\oplus,e)$$

Which becomes:

$$
prd\ (a:s)\ \rhd\ \left(\begin{array}{l}
in.eot\ ?\ any \rightarrow out\ !\ e \rightarrow SKIP \\
| \\
in.value\ ?\ x \rightarrow SFOLDL(\oplus, e \oplus x)
\end{array}\right)
$$

It is clear both sides of the feed are willing to communicate on the value conduit of the stream which connects them, so we have:

$$prd\ (s)\ \rhd\ SFOLDL(\oplus, e \oplus a)$$

Were the remaining stream s empty, we would then be reduced to:

$$out\ !\ e \oplus a$$

Which is equivalent to:

$$prd\ (e \oplus a)$$

Which, given the definition of $sfoldl$ is equivalent to:

$$prd\ (sfoldl\ (\oplus)\ e\ \lfloor a \rfloor)$$

Handel-C Implementation

In the simple case where we are dealing with a stream of items, we can construct a definition for `SFOLDL` as shown in Figure 5.18. Here the parameter `f` is an expression, perhaps a Handel-C `macro expr`.

```
macro proc SFOLDL_SIMPLE (streamin, conduitout, f, e)
{
   Bool eot;
   messagetype (streamin) x;
   typeof (e) a;
   eot  = False;
   a = e;
   do
   {
      prialt
      {
      case streamin.eot ? eot:
         conduitout ! a;
         break;
      case streamin.value ? x:
         a = f (a,x);
         break;
      }
   } while (!eot)
}
```

Figure 5.18: The simple case Handel-C definition of the process SFOLDL.

In the more general case we have the following. Note in Handel-C parameters passed to `macro proc` style definitions are implicitly done so by reference. So, in this definition, the process `F` is responsible for consuming a value from the input stream (if it is willing to communicate), then applying the binary operator to that value and `a`, and finally storing the result in `a`. The definition can be seen in Figure 5.19.

To illustrate, a sample process `F` working on simple streams of items is given in Figure 5.20.

Let us now consider the implementation of $SFOLDR$. To begin with we have the simple case where we are dealing with a basic stream of items. This is effectively a two pass process. First

```
macro proc SFOLDL (streamin, conduitout, F, e)
{
   Bool eot;
   typeof(e) a;
   eot  = False;
   a = e;
   do
   {
      prialt
      {
      case streamin.eot ? eot:
         PRODUCE (conduitout, a);
         break;
      default:
         F (streamin.value,a);
         break;
      }
   } while (!eot)
}
```

Figure 5.19: The general case Handel-C definition of the process SFOLDL.

```
macro proc F (conduitin,a)
{
   messagetype (conduitin) x;
   prialt
   {
   case conduitin ? x:
      a = f (a,x);
      break;
   default:
      break;
   }
}
```

Figure 5.20: The process F for use with stream implementations of fold.

we are required to read the entire stream into a buffer. Once EOT is signalled, we can then begin our second pass, where we step backwards through the items in the buffer, applying the operator `f` and accumulating the result as we go. When we have got to the start of the buffer, we can then output the result.

The static nature of Handel-C programs means that we are required to know the size of the buffer in advance. This definition is slightly flexible in that it can cope with any streams with length up to the specified buffer size, not just those with *exactly* that length. A simple check is made here to make sure the items input from the stream do not go beyond the end of the buffer. Where the incoming stream is larger than the buffer, the fold operation will be carried out on a truncated sequence. As such, this is only a valid refinement of $sfoldr$ where the length of the buffer is guaranteed to be greater than or equal to any incoming stream. An example implementation employing this buffering scheme is given in Figure 5.21.

```
macro proc SFOLDR_SIMPLE (buffersize, streamin, conduitout, f, e)
{
   Bool eot;
   messagetype (streamin) x;
   messagetype (streamin) xs[buffersize];
   typeof (buffersize) bufferpos;
   typeof (e) a;
   eot  = False;
   bufferpos = 0;
   do
   {
      prialt
      {
      case streamin.eot ? eot:
         a = e;
         while (bufferpos > 0)
         {
            bufferpos--;
            a = f (xs[bufferpos],a);
         }
         conduitout ! a;
         break;
      case streamin.value ? x:
         if (bufferpos < buffersize)
         {
            xs[bufferpos] = x;
            bufferpos++;
         }
         break;
      }
   } while (!eot)
}
```

Figure 5.21: The simple case Handel-C definition of the process SFOLDR.

The more general case is somewhat harder to define, given that the two pass behaviour of the simple case above has forced us to decouple the communication from the application of the binary operator. As noted previously, one way to implement a fold in stream terms is actually by means of the vector implementation, something along the lines of the following:

$$SFOLDR_n(F, e) = S2VMAP_n(id) \gg VFOLDR_n(F, e)$$

Whilst we may consider this definition wasteful for the simple case of a streams of items, it may become a more reasonable strategy when considering more complex structures.

5.3.2 Vectors

Let us consider a function $vfold$ as a possible refinement in vector terms for $fold$. Strictly speaking, we should write this $vfold_n$, however we shall find it convenient on occasion to omit the subscript where its value is clear from context. This function has the following type:

$$vfold_n \ :: \ (A \to A \to A) \to \langle A \rangle_n \to A$$

As before, the implementation could proceed informally as follows:

$$vfold_n \ (\oplus) \ \langle x_1, x_2, ..., x_n \rangle_n = x_1 \oplus x_2 \oplus ... \oplus x_n$$

To prove this is a valid refinement of $fold$, we require that the following diagram commutes.

$$\begin{array}{ccc} [A] & \xrightarrow{fold \ (\oplus)} & A \\ \uparrow{\scriptstyle abs_V} & & \uparrow{\scriptstyle id} \\ \langle A \rangle_n & \xrightarrow{vfold_n \ (\oplus)} & A \end{array}$$

We can prove this as follows:

$$\begin{array}{lll} & (fold \ (\oplus) \ \circ \ abs_V) \ \langle x_1, x_2, ..., x_n \rangle_n & \{id\} \\ = & fold \ (\oplus) \ [x_1, x_2, ..., x_n] & \{def. \ abs_V\} \\ = & x_1 \oplus x_2 \oplus ... \oplus x_n & \{def. \ fold\} \\ = & vfold_n \ (\oplus) \ \langle x_1, x_2, ..., x_n \rangle_n & \{def. \ vfold_n\} \\ = & (id \ \circ \ vfold_n \ (\oplus)) \ \langle x_1, x_2, ..., x_n \rangle_n & \{id\} \end{array}$$

Similarly, we could prove refinements in vector terms of the four $fold$ variants. These would have the following types:

$$\begin{array}{lll} vfoldr_n & :: & (A \to B \to B) \to B \to \langle A \rangle_n \to B \\ vfoldl_n & :: & (A \to B \to A) \to A \to \langle B \rangle_n \to A \\ vfoldr1_n & :: & (A \to A \to A) \to \langle A \rangle_n \to A \\ vfoldl1_n & :: & (A \to A \to A) \to \langle A \rangle_n \to A \end{array}$$

Their informal definitions are as follows:

$$\begin{array}{lll} vfoldr_n \ (\oplus) \ e \ \langle x_1, x_2, ..., x_n \rangle_n & = & x_1 \oplus (x_2 \oplus (... \oplus (x_n \oplus e))) \\ vfoldl_n \ (\oplus) \ e \ \langle x_1, x_2, ..., x_n \rangle_n & = & (((e \oplus x_1) \oplus x_2) \oplus ...) \oplus x_n \\ vfoldr1_n \ (\oplus) \ \langle x_1, x_2, ..., x_n \rangle_n & = & x_1 \oplus (x_2 \oplus (... \oplus (x_{n-1} \oplus x_n))) \\ vfoldl1_n \ (\oplus) \ \langle x_1, x_2, ..., x_n \rangle_n & = & (((x_1 \oplus x_2) \oplus x_3) \oplus ...) \oplus x_n \end{array}$$

Process Refinement

A process implementing the functionality of $fold \ (\oplus)$ in vector terms should input a vector of values, and output the result of folding the operator $\oplus$ over the values input. This is depicted in Figure 5.22.

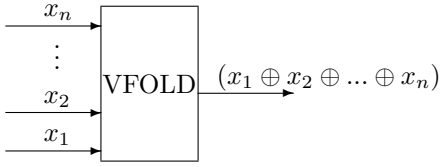


Figure 5.22: The fold process for vectors.

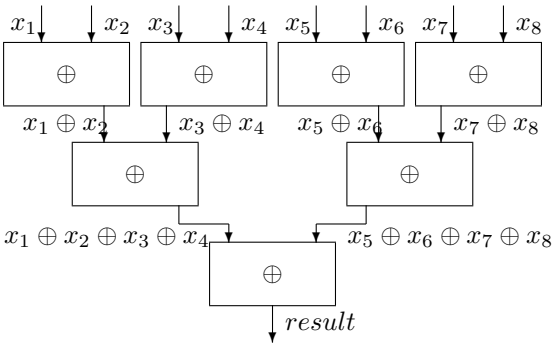


Figure 5.23: A funnel implementation of the VFOLD process.

Logarithmic Version

For an associative operator, the well known funnel network topology implements this functionality in $log(n)$ time. This is depicted in Figure 5.23. Let us consider how this process might be defined. First, recall the type of the operator $(\oplus)$ passed to $fold$ as a parameter:

$$\oplus :: A \to A \to A$$

Given a process F which refines the operator $\oplus$, with the following alphabet:

$$\alpha F = \{in_a :: \underline{A}, in_b :: \underline{A}, out :: \underline{A}\}$$

The network can be defined along the following lines.

$$VFOLD_n(F) \quad = \quad \underset{i=1}{\overset{n}{||}} \quad F[mid_{(2i)}/in_a, mid_{(2i+1)}/in_b, mid_i/out]$$

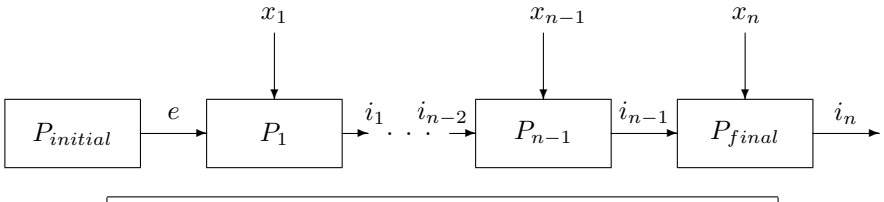
We have the following alphabet:

$$\alpha VFOLD_n(F) = \{in :: \underline{\langle A \rangle_n}, out :: \underline{A}\}$$

In practice a full definition will need a bit more complexity, to deal with the input vector and the output conduit. However the above definition gives a general idea of the topology. Given this definition, mid_1 is the output conduit, and the conduits in the range $[mid_n, \ldots, mid_{2n-1}]$ form the input vector. Thus, for n inputs, this network uses a total of $n-1$ processing elements and $2n-1$ conduits.

Linear Version

Where the characteristic operator ($\oplus$) is not associative we shall need to look to alternative implementations of fold in the vector setting. Again we shall want to consider the four different variations of fold. Let us initially consider a process candidate for $VFOLDL$, which is a potential refinement for the function $vfoldl$. This is depicted in Figure 5.24.



$$
\begin{array}{lcl}
i_1 & = & e \oplus x_1 \\
i_{n-2} & = & (...((e \oplus x_1) \oplus x_2)...) \oplus x_{n-2} \\
i_{n-1} & = & ((...((e \oplus x_1) \oplus x_2)...) \oplus x_{n-2}) \oplus x_{n-1} \\
i_n & = & (((...((e \oplus x_1) \oplus x_2)...) \oplus x_{n-2}) \oplus x_{n-1}) \oplus x_n
\end{array}
$$

Figure 5.24: A linear implementation of the VFOLDL process.

Recall that for $foldl$, the operator passed as a parameter will have the following type:

$$(\oplus) :: A \rightarrow B \rightarrow A$$

Thus, a process F which refines this operator is required, which should have the following alphabet:

$$\alpha F = \{left :: \underline{A}, right :: \underline{B}, result :: \underline{A}\}$$

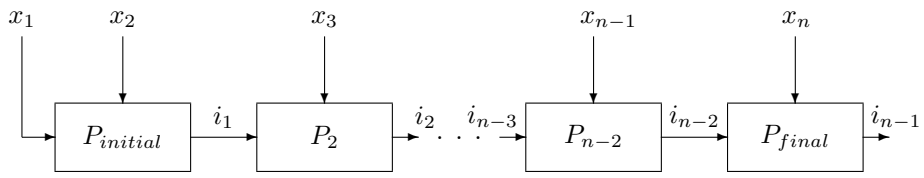
We can now construct our definition. For the most part, we are connecting each instance of F with the corresponding element of the input vector, and the output of the previous instance of F. This pattern changes slightly at either end as we have to deal with the base value, and the output conduit.

$$
\begin{array}{rcl}
VFOLDL_n(F, e) & = & \left(P_{initial} \;||\; \left(\begin{array}{c} (n-1) \\ || \quad P_i \\ i = 1 \end{array} \right) ||P_{final} \right) \backslash \{mid\} \\
 & & P_{initial} = prd\ e\ [mid_1/out] \\
where & & P_i = F[mid_i/left, in_i/right, mid_{i+1}/result] \\
 & & P_{final} = F[mid_n/left, in_n/right, out/result]
\end{array}
$$

This gives us the following alphabet:

$$\alpha VFOLDL_n(F, e) = \{in :: \underline{\langle B \rangle_n}, out :: \underline{A}\}$$

Along similar lines to the process $VFOLDL$ we may construct the variant $VFOLDL1$, depicted in Figure 5.25.



$$
\begin{array}{lcl}
i_1 & = & x_1 \oplus x_2 \\
i_2 & = & (x_1 \oplus x_2) \oplus x_3 \\
i_{n-3} & = & (...((x_1 \oplus x_2) \oplus x_3)...) \oplus x_{n-2} \\
i_{n-2} & = & ((...((x_1 \oplus x_2) \oplus x_3)...) \oplus x_{n-2}) \oplus x_{n-1} \\
i_{n-1} & = & (((...((x_1 \oplus x_2) \oplus x_3)...) \oplus x_{n-2}) \oplus x_{n-1}) \oplus x_n
\end{array}
$$

Figure 5.25: A linear implementation of the VFOLDL1 process.

We can define this as follows:

$$
\begin{array}{rcl}
VFOLDL1_n(F) & = & \left(P_{initial} \;||\; \left(\begin{array}{c} (n-2) \\ || \quad P_i \\ i=2 \end{array} \right) ||P_{final} \right) \backslash \{mid\} \\
 & P_{initial} & = F[in_1/left, in_2/right, mid_i/result] \\
where & P_i & = F[mid_{i-1}/left, in_{i+1}/right, mid_i/result] \\
 & P_{final} & = F[mid_{n-2}/left, in_n/right, out/result]
\end{array}
$$

Here the process F will take the following alphabet:

$$\alpha F = \{left :: \underline{A}, right :: \underline{A}, result :: \underline{A}\}$$

Giving $VFOLDL1_n(F)$ an alphabet like so:

$$\alpha VFOLDL1_n(F) = \{in :: \underline{\langle A \rangle_n}, out :: \underline{A}\}$$

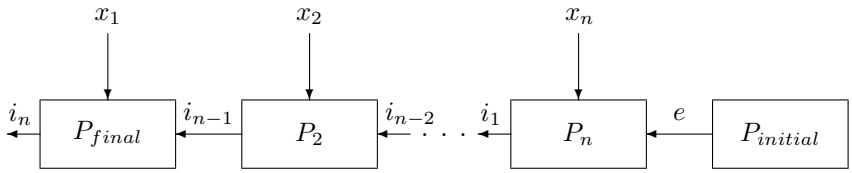
Let us now consider the right directed version of fold in vector terms. The process $VFOLDR$ is depicted in Figure 5.26.

Recall that for the function $foldr$, the operator passed as a parameter will have the following type:

$$(\oplus) :: A \to B \to B$$

As such, for $VFOLDR$, the alphabet of process F will be defined like so:

$$\alpha F = \{left :: \underline{A}, right :: \underline{B}, result :: \underline{B}\}$$



$$
\begin{array}{lcl}
i_1 & = & x_n \oplus e \\
i_{n-2} & = & x_3 \oplus (...(x_{n-1} \oplus (x_n \oplus e))...) \\
i_{n-1} & = & x_2 \oplus (x_3 \oplus (...(x_{n-1} \oplus (x_n \oplus e))...)) \\
i_n & = & x_1 \oplus (x_2 \oplus (x_3 \oplus (...(x_{n-1} \oplus (x_n \oplus e))...)))
\end{array}
$$

Figure 5.26: A linear implementation of the VFOLDR process.

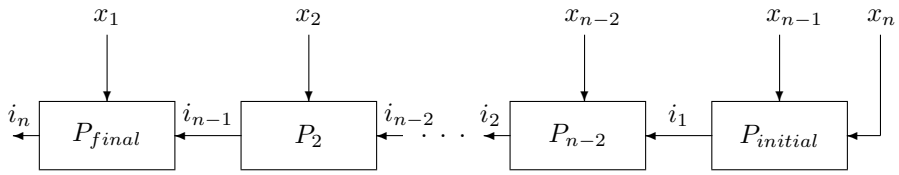
Giving $VFOLDR_n(F)$ an alphabet like so:

$$\alpha VFOLDR_n(F) = \{in :: \underline{\langle A \rangle_n}, out :: \underline{B}\}$$

We can supply a definition, similar to that for $VFOLDL$, as follows:

$$
\begin{array}{lcl}
VFOLDR_n(F,e) & = & \left(P_{final} \;||\; \left(\begin{array}{c} n \\ || \\ i=2 \end{array} \; P_i \right) ||P_{initial} \right) \backslash \{mid\} \\
 & P_{initial} & = \; prd\ e\ [mid_n/out] \\
where & P_i & = \; F[in_i/left, mid_i/right, mid_{i-1}/result] \\
 & P_{final} & = \; F[in_1/left, mid_1/right, out/result]
\end{array}
$$

Finally we shall look at $VFOLDR1$, again a simple modification of $VFOLDR$. This is depicted in Figure 5.27.



$$
\begin{array}{lcl}
i_1 & = & x_{n-1} \oplus x_n \\
i_2 & = & x_{n-2} \oplus (x_{n-1} \oplus x_n) \\
i_{n-2} & = & x_3 \oplus (...(x_{n-2} \oplus (x_{n-1} \oplus x_n))...) \\
i_{n-1} & = & x_2 \oplus (x_3 \oplus (...(x_{n-2} \oplus (x_{n-1} \oplus x_n))...)) \\
i_n & = & x_1 \oplus (x_2 \oplus (x_3 \oplus (...(x_{n-2} \oplus (x_{n-1} \oplus x_n))...)))
\end{array}
$$

Figure 5.27: A linear implementation of the VFOLDR1 process for vectors.

So, for $VFOLDR1$, the process F takes the following alphabet:

$$\alpha F = \{left :: \underline{A}, right :: \underline{A}, result :: \underline{A}\}$$

Giving $VFOLDR1_n(F)$ an alphabet like so:

$$\alpha VFOLDR1_n(F) = \{in :: \underline{\langle A \rangle_n}, out :: \underline{A}\}$$

The definition is then as follows:

$$\begin{array}{rcl} VFOLDR1_n(F,e) & = & \left(P_{final} \;||\; \left(\begin{array}{c} (n-2) \\ || \quad P_i \\ i=2 \end{array} \right) ||P_{initial} \right) \backslash \{mid\} \\ & P_{initial} & = F[in_{n-1}/left, in_n/right, mid_{n-2}/result] \\ where & P_i & = F[in_i/left, mid_i/right, mid_{i-1}/result] \\ & P_{final} & = F[in_1/left, mid_1/right, out/result] \end{array}$$

Proof

To prove that $VFOLDL_n(F,e)$ is a valid refinement of $vfoldl_n\ f\ e$, we shall require the following diagram to commute:

$$\begin{array}{ccc} \langle B \rangle_n & \xrightarrow{vfoldl_n\ f\ e} & A \\ \Big\downarrow prd & & \Big\downarrow prd \\ \triangleright\ VFOLDL_n(F,e) & & \\ Process & \longrightarrow & Process \end{array}$$

That is to say, we require the following equivalence to hold for any vector v, and binary operator f with base value e, where F is a valid refinement of f :

$$prd\ v\ \triangleright\ VFOLDL_n(F,e)\ =\ prd\ (vfoldl_n\ f\ e\ v)$$

Let us first explore our definition of $VFOLDL_n$ a little. Where the input vector is of size zero, we effectively implement just $P_{initial}$. This is equivalent to just producing the base value e:

$$VFOLDL_0(F,e)\ =\ prd\ e$$

Furthermore, where the input vector is of size one, we implement just $P_{initial}$ in parallel with P_{final}.

$$\begin{array}{rcl} VFOLDL_1(F,e) & = & (P_{initial}\ ||\ P_{final}) \backslash \{mid\} \\ & P_{initial} & = prd\ e\ [mid_1/out] \\ where & P_{final} & = F[mid_1/left, in_1/right, out/result] \end{array}$$

So, to return to the proof. In the case of the empty vector, we are attempting to prove the following equivalence:

$$prd\ \langle\rangle_0\ \triangleright\ VFOLDL_0(F,e)\ =\ prd\ (vfoldl_0\ f\ e\ \langle\rangle_0)$$

Given that $prd\ \langle\rangle_0$ results in no action, we then have:

$$VFOLDL_0(F,e)\ =\ prd\ (vfoldl_0\ f\ e\ \langle\rangle_0)$$

We know from our definition of $VFOLDL$ that:

$$VFOLDL_0(F,e)\ =\ prd\ e$$

...and thus:

$$prd\ e\ =\ prd\ (vfoldl_0\ f\ e\ \langle\rangle_0)$$

Finally, appealing to our definition of $vfoldl$ we arrive at:

$$prd\ e\ =\ prd\ e$$

Handel-C Implementation

We shall see that our Handel-C implementations follow naturally from our CSP specifications. It is important to bear in mind that, whereas CSP subscripts range from 1, those in Handel-C, in common with most programming languages, range from zero. First we shall consider the implementation of `VFOLDR`, which is given in Figure 5.28.

```
macro proc VFOLDR (n,in,cout,F,e)
{
   conduittype(cout) mid [n];
   typeof(n) i;
   par (i=0;i<=n;i++)
   {
      ifselect (n==0)
         PRODUCE (cout,e);
      else ifselect (i==n)
         PRODUCE (mid[n-1],e);
      else ifselect (i==0)
         F (in[i],mid[i],cout);
      else
         F (in[i],mid[i],mid[i-1]);
   }
}
```

Figure 5.28: The Handel-C definition of the process VFOLDR.

Next we have the implementation of `VFOLDL`, which is given in Figure 5.29.

```
macro proc VFOLDL (n,in,cout,F,e)
{
   conduittype(in[0]) mid [n];
   typeof(n) i;
   par (i=0;i<=n;i++)
   {
      ifselect (n==0)
         PRODUCE (cout,e);
      else ifselect (i==0)
         PRODUCE (mid[0],e);
      else ifselect (i==n)
         F (in[n-1],mid[n-1],cout);
      else
         F (in[i-1],mid[i-1],mid[i]);
   }
}
```

Figure 5.29: The Handel-C definition of the process VFOLDL.

5.3.3 Combined Structures

In terms of a two dimensional structure (for example a list of lists), the act of folding over the entire structure is equivalent to mapping the fold function to each part, and then folding all of the intermediate results. Thus, we have the following expression, which we shall term $tdfold$ (two-dimensional fold):

$$tdfold\ (\oplus)\ =\ fold\ (\oplus)\ \circ\ map\ (fold\ (\oplus))$$

The expression $tdfold\ (\oplus)$ has type:

$$[[A]] \rightarrow A$$

As usual, we can construct a further four variants of $tdfold$, based on each of the four $fold$ variants. These have the following type:

$$\begin{array}{lcl} tdfoldr & :: & (A \rightarrow B \rightarrow B) \rightarrow B \rightarrow [[A]] \rightarrow B \\ tdfoldl & :: & (A \rightarrow B \rightarrow A) \rightarrow A \rightarrow [[B]] \rightarrow A \\ tdfoldr1 & :: & (A \rightarrow A \rightarrow A) \rightarrow [[A]] \rightarrow A \\ tdfoldl1 & :: & (A \rightarrow A \rightarrow A) \rightarrow [[A]] \rightarrow A \end{array}$$

These can be defined as follows:

$$\begin{array}{lcl} tdfoldr\ (\oplus)\ e & = & foldr\ (\oplus)\ e\ \circ\ map\ (foldr\ (\oplus)\ e) \\ tdfoldl\ (\oplus)\ e & = & foldl\ (\oplus)\ e\ \circ\ map\ (foldl\ (\oplus)\ e) \\ tdfoldr1\ (\oplus) & = & foldr1\ (\oplus)\ \circ\ map\ (foldr1\ (\oplus)) \\ tdfoldl1\ (\oplus) & = & foldl1\ (\oplus)\ \circ\ map\ (foldl1\ (\oplus)) \end{array}$$

5.3.4 Distributed Lists

The function $tdfold$ should provide a suitable refinement for $fold$ in a distributed list setting. As a proof of this we would require the following diagram to commute:

$$\begin{array}{ccc} [A] & \xrightarrow{fold\ (\oplus)} & A \\ \uparrow{\scriptstyle abs_D} & & \uparrow{\scriptstyle id} \\ [[A]] & \xrightarrow{tdfold\ (\oplus)} & A \end{array}$$

The proof of this, taking a list of lists $[l_1, l_2, ..., l_n]$, and our first definition for abs_D, is a simple product of reduce promotion:

$$\begin{array}{lll} & (fold\ (\oplus)\ \circ\ abs_{D1})\ [l_1, l_2, ..., l_n] & \{id\} \\ = & (fold\ (\oplus)\ \circ\ fold\ (+\!\!+))\ [l_1, l_2, ..., l_n] & \{def.\ abs_{D1}\} \\ = & (fold\ (\oplus)\ \circ\ map\ (fold\ (\oplus))\ [l_1, l_2, ..., l_n] & \{reduce\ promotion\} \\ = & tdfold\ (\oplus)\ [l_1, l_2, ..., l_n] & \{def.\ tdfold\} \\ = & (id\ \circ\ tdfold\ (\oplus))\ [l_1, l_2, ..., l_n] & \{id\} \end{array}$$

Similar proofs could be constructed for the four $tdfold$ variants.

5.3.5 Vector of Streams

Let us take as an example the vector of streams. A refinement of the above function would have type:

$$\langle \lfloor A \rfloor \rangle_n \rightarrow A$$

The 'parts' in this case are streams, so we begin by applying $sfold$. These are contained in a vector, so we need to employ $vmap$ to map $sfold$ to each part. The function $vmap$ used in this way will produce a vector of intermediate results, so we should now apply $vfold$ to collect them. Thus we have:

$$vsfold_n\ (\oplus)\ =\ vfold_n\ (\oplus)\ \circ\ vmap_n\ (sfold\ (\oplus))$$

To prove this is a valid refinement of $tdfold$, we must show that the following diagram commutes:

$$\begin{array}{ccc} [[A]] & \xrightarrow{tdfold\ (\oplus)} & A \\ \uparrow{\scriptstyle abs_{VS}} & & \uparrow{\scriptstyle id} \\ \langle \lfloor A \rfloor \rangle_n & \xrightarrow{vsfold_n\ (\oplus)} & A \end{array}$$

Given a vector of streams $\langle s_1, s_2, ..., s_n \rangle_n$, we can prove this as follows:

$$
\begin{array}{lll}
 & (tdfold\ (\oplus)\ \circ\ abs_{VS})\ \langle s_1, ..., s_n \rangle_n & \{id\} \\
= & tdfold\ (\oplus)\ [l_1, ..., l_n] & \{def.\ abs_{VS}\} \\
= & (fold\ (\oplus)\ \circ\ map\ (fold\ (\oplus)))\ [l_1, ..., l_n] & \{def.\ tdfold\} \\
= & fold\ (\oplus)\ [fold\ (\oplus)\ l_1, ..., fold\ (\oplus)\ l_n] & \{def.\ map\} \\
= & fold\ (\oplus)\ [(fold\ (\oplus)\ \circ abs_S)\ s_1, ..., (fold\ (\oplus)\ \circ abs_S)\ s_n] & \{def.\ abs_S\} \\
= & fold\ (\oplus)\ [sfold\ (\oplus)\ s_1, ..., sfold\ (\oplus)\ s_n] & \{sfold\} \\
= & (fold\ (\oplus)\ \circ\ abs_V)\ \langle sfold\ (\oplus)\ s_1, ..., sfold\ (\oplus)\ s_n \rangle_n & \{def.\ abs_V\} \\
= & (fold\ (\oplus)\ \circ\ abs_V\ \circ\ vmap_n\ (sfold\ (\oplus)))\ \langle s_1, ..., s_n \rangle_n & \{def.\ vmap\} \\
= & (vfold_n\ (\oplus)\ \circ\ vmap_n\ (sfold\ (\oplus)))\ \langle s_1, ..., s_n \rangle_n & \{vfold\} \\
= & vsfold_n\ (\oplus)\ \langle s_1, ..., s_n \rangle_n & \{def.\ vsfold\} \\
= & (id\ \circ\ vsfold_n\ (\oplus))\ \langle s_1, ..., s_n \rangle_n & \{vsfold\}
\end{array}
$$

Similarly, we could prove refinements for the four variants of *tdfold* in vector of streams terms. These would then be defined as follows:

$$
\begin{array}{lcl}
vsfoldr_n\ (\oplus)\ e & = & vfoldr_n\ (\oplus)\ e\ \circ\ vmap_n\ (sfoldr\ (\oplus)\ e) \\
vsfoldl_n\ (\oplus)\ e & = & vfoldl_n\ (\oplus)\ e\ \circ\ vmap_n\ (sfoldl\ (\oplus)\ e) \\
vsfoldr1_n\ (\oplus) & = & vfoldr1_n\ (\oplus)\ \circ\ vmap_n\ (sfoldr1\ (\oplus)) \\
vsfoldl1_n\ (\oplus) & = & vfoldl1_n\ (\oplus)\ \circ\ vmap_n\ (sfoldl1\ (\oplus))
\end{array}
$$

Process Refinement

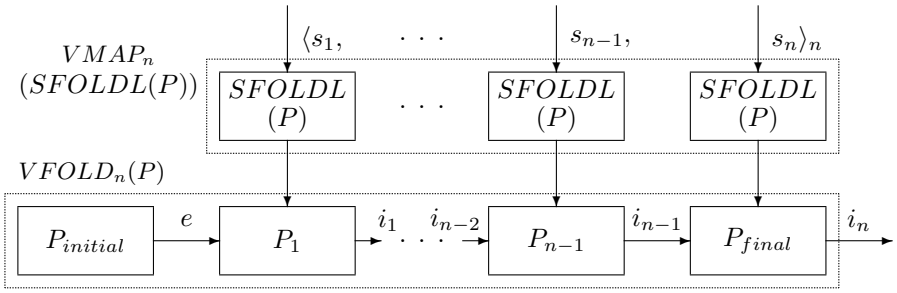


Figure 5.30: The VSFOLDL process.

Let us consider a refinement of this in process terms. In general, we have the following pattern from which our fold variants are all derived:

$$vfold_n\ (\oplus)\ \circ\ vmap_n\ (sfold\ (\oplus))$$

By now, we have already seen how all of these components can be refined to processes. So our task here is one simply of composition. Thus, we have:

$$VSFOLD_n(P)\ =\ VMAP_n\ (SFOLD\ (P)) \gg_n VFOLD_n(P)$$

Given some process P which refines the operator $\oplus$. An example of this - the specific case of $VSFOLDL$ is given in Figure 5.30. The process $VSFOLDL$ is defined as follows:

$$VSFOLDL_n(P) \;=\; VMAP_n\ (SFOLDL\ (P)) \gg_n VFOLDL_n(P)$$

One interesting issue highlighted by the diagram in Figure 5.30 is the fact that it may be possible to construct an alternative version of $VSFOLDL$ and similar processes in which the components are grouped differently, and as such, conceptually, the flow of data is transposed. This is illustrated in Figure 5.31.

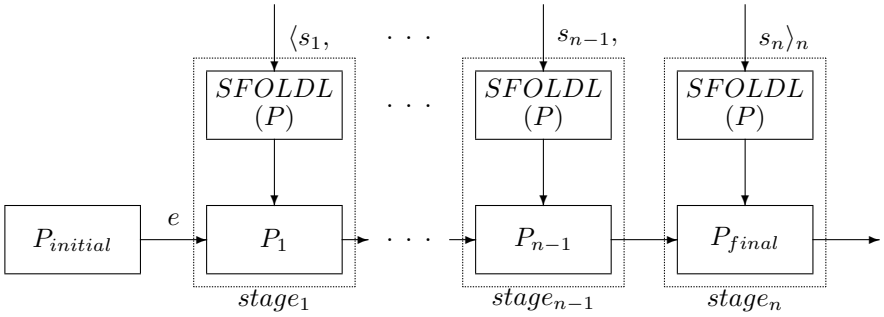


Figure 5.31: An alternative construction of the VSFOLDL process.

To understand how the process illustrated in Figure 5.31 might be constructed algorithmically, let us return to our original functional specification. The process $VSFOLDL$ is a refinement of the function *tdfoldl*. Let us recall the one of the characteristics of $fold$ given at the start of Section 5.3 - that a fold can be re-packaged as a series of steps composed together. For $foldl$, we have:

$$foldl\ (\oplus)\ e\ [x_1, x_2, ..., x_n] = ((\circ)/\ (map\ (flip\ (\oplus))\ (reverse\ [x_1, x_2, ..., x_n])))\ e$$

If we factor in our definition of *tdfoldl*, we have the following (rather unwieldy!) definition:

$$tdfoldl\ (\oplus)\ e\ [x_1, x_2, ..., x_n] = ((\circ)/\ (map\ (flip\ (\oplus))\ (reverse\ (map\ (foldl\ (\oplus)\ e)\ [x_1, x_2, ..., x_n]))))\ e$$

Appealing to map distributivity for a little simplification, we have:

$$\begin{aligned} tdfoldl\ (\oplus)\ e\ [x_1, x_2, ..., x_n] &= ((\circ)/\ (map\ stage\ (reverse\ [x_1, x_2, ..., x_n])))\ e \\ stage &= (flip\ (\oplus))\ \circ\ (foldl\ (\oplus)\ e) \end{aligned}$$

What we now have, in effect, is something more akin to a pipeline. The function *stage* in the above definition represents a single stage in the pipeline, and this corresponds to the functionality of each dashed box in Figure 5.31.

5.3.6 Stream of Streams

This should follow an analogous proof as that for the vector of streams. So, in short, we have:

$$
\begin{array}{lcl}
ssfold\ (\oplus) & = & sfold\ (\oplus)\ \circ\ smap\ (sfold\ (\oplus)) \\
ssfoldr\ (\oplus)\ e & = & sfoldr\ (\oplus)\ e\ \circ\ smap\ (sfoldr\ (\oplus)\ e) \\
ssfoldl\ (\oplus)\ e & = & sfoldl\ (\oplus)\ e\ \circ\ smap\ (sfoldl\ (\oplus)\ e) \\
ssfoldr1\ (\oplus) & = & sfoldr1\ (\oplus)\ \circ\ smap\ (sfoldr1\ (\oplus)) \\
ssfoldl1\ (\oplus) & = & sfoldl1\ (\oplus)\ \circ\ smap\ (sfoldl1\ (\oplus))
\end{array}
$$

Process Refinement

As with the previous case, our components are already proven, so all that we need to do here is a simple act of composition. Thus, in general we have:

$$SSFOLD(P)\ =\ SMAP\ (SFOLD\ (P)) \gg SFOLD(P)$$

Given some process P which refines the operator $\oplus$.

5.3.7 Stream of Vectors

Again, with a proof analogous to that for the vector of streams, we could verify that the following is a valid refinement of *tdfold*:

$$
\begin{array}{lcl}
svfold_m\ (\oplus) & = & sfold\ (\oplus)\ \circ\ smap\ (vfold_m\ (\oplus)) \\
svfoldr_m\ (\oplus)\ e & = & sfoldr\ (\oplus)\ e\ \circ\ smap\ (vfoldr_m\ (\oplus)\ e) \\
svfoldl_m\ (\oplus)\ e & = & sfoldl\ (\oplus)\ e\ \circ\ smap\ (vfoldl_m\ (\oplus)\ e) \\
svfoldr1_m\ (\oplus) & = & sfoldr1\ (\oplus)\ \circ\ smap\ (vfoldr1_m\ (\oplus)) \\
svfoldl1_m\ (\oplus) & = & sfoldl1\ (\oplus)\ \circ\ smap\ (vfoldl1_m\ (\oplus))
\end{array}
$$

Process Refinement

As before the task of process refinement is a simple one. In general we have:

$$SVFOLD_m(P)\ =\ SMAP\ (VFOLD_m\ (P)) \gg SFOLD(P)$$

Given some process P which refines the operator $\oplus$.

5.3.8 Vectors of Vectors

Finally, for the vector of vectors:

$$
\begin{array}{lcl}
vvfold_{n,m}\ (\oplus) & = & vfold_n\ (\oplus)\ \circ\ vmap_n\ (vfold_m\ (\oplus)) \\
vvfoldr_{n,m}\ (\oplus)\ e & = & vfoldr_n\ (\oplus)\ e\ \circ\ vmap_n\ (vfoldr_m\ (\oplus)\ e) \\
vvfoldl_{n,m}\ (\oplus)\ e & = & vfoldl_n\ (\oplus)\ e\ \circ\ vmap_n\ (vfoldl_m\ (\oplus)\ e) \\
vvfoldr1_{n,m}\ (\oplus) & = & vfoldr1_n\ (\oplus)\ \circ\ vmap_n\ (vfoldr1_m\ (\oplus)) \\
vvfoldl1_{n,m}\ (\oplus) & = & vfoldl1_n\ (\oplus)\ \circ\ vmap_n\ (vfoldl1_m\ (\oplus))
\end{array}
$$

Process Refinement

Once more, a simple task, we have:

$$VVFOLD_{n,m}(P) \;=\; VMAP_n\ (VFOLD_m\ (P)) \gg_n VFOLD_n(P)$$

Given some process P which refines the operator $\oplus$.

5.4 Summary

We have seen the refinement to processes of the key higher order functions - *map* and $fold$. We have looked at these in terms of the two basic settings - streams and vectors, as well as combinations of the two. We have explored the many different variations of fold. We should now be equipped with arguably the two most significant building blocks required in refining from the functional environment to the process environment.

Chapter 6

A Library of Provably Correct Re-usable Hardware Components

6.1 Introduction

In this chapter we shall present a library of components, building on those from the previous chapter (*map* and $fold$), based on higher order functions. We shall show how these highly expressive functional patterns can be refined in a provably correct manner to create re-usable hardware components. We begin by looking at homomorphisms - a class of functions which encapsulate several of the most commonly encountered higher order functions. Following on from this we explore several highly useful higher order functions - $filter$, *scan* and $unfold$. Towards the end of the chapter we examine some more specialised patterns - the functions $zfold$ and *zmap*, as well as a look at how the well known Divide and Conquer paradigm might be approached in this methodology.

6.2 Homomorphisms

Let us consider the group of functions known as homomorphisms. In list terms, a homomorphism is any function h, which can, given some binary operator $(\oplus)$ and function f, be defined as follows:

$$h \;=\; fold\ (\oplus)\ \circ\ map\ f$$

We may find it useful to explicitly state the types involved here:

$$\begin{array}{lcl} f & :: & A \to B \\ (\oplus) & :: & B \to B \to B \\ h & :: & [A] \to B \end{array}$$

Let us provide some examples. The most well known homomorphic functions are also our three most commonly encountered higher order functions - *map*, $fold$ and $filter$. Here we shall consider their definitions in line with the above template. First, some auxiliary functions. We shall find useful a simple function we shall call *single*. This takes an item and returns a list containing just that item. We shall also find useful another similar function we shall call *perhaps*. This takes an item and a predicate. If the predicate applied to that item is true, the function returns a list containing just that item, otherwise an empty list is returned. We have:

$$\begin{array}{lcl} single & :: & A \to [A] \\ perhaps & :: & (A \to Bool) \to A \to [A] \\ single\ x & = & [x] \\ perhaps\ x & = & if\ p\ x\ then\ [x]\ else\ [] \end{array}$$

Given these two simple functions, we can construct definitions for *map*, $fold$ and $filter$ which clearly demonstrate their homomorphic nature:

$$\begin{array}{lcl} map\ f & = & fold\ (+\!\!+) \circ map\ (single \circ f) \\ fold\ (\oplus) & = & fold\ (\oplus) \circ map\ id \\ filter\ p & = & fold\ (+\!\!+) \circ map\ (perhaps\ p) \end{array}$$

Any definition which is a homomorphism may take advantage of homomorphism promotion, a useful set of transformation rules. For our functions *map*, $fold$ and $filter$ we have:

$$\begin{array}{lcll} map\ f\ \circ\ fold\ (+\!\!+) & = & fold\ (+\!\!+)\ \circ\ map\ (map\ f) & \{map\ promotion\} \\ fold\ (\oplus)\ \circ\ fold\ (+\!\!+) & = & fold\ (\oplus)\ \circ\ map\ (fold\ (\oplus)) & \{fold\ promotion\} \\ filter\ p\ \circ\ fold\ (+\!\!+) & = & fold\ (+\!\!+)\ \circ\ map\ (filter\ p) & \{filter\ promotion\} \end{array}$$

Indeed, for any homomorphism in list terms, following the above template, we can state:

$$h\ \circ\ fold\ (+\!\!+)\ =\ fold\ (\oplus)\ \circ\ map\ h \quad \{homomorphism\ promotion\}$$

6.2.1 Streams

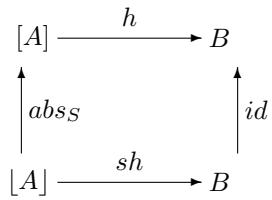
One might hope the same concept of the homomorphism in list terms will map also to the stream setting. Here, our generic homomorphism is defined as follows:

$$sh\ =\ sfold\ (\oplus)\ \circ\ smap\ f$$

This has the following type:

$$sh :: \lfloor A \rfloor \to B$$

Let us prove that this comprises a valid refinement for h, our generic list homomorphism. For this, we shall require that the following diagram commutes:



Here is the proof:

$$\begin{array}{lll}
 & (h \circ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\
= & (fold\ (\oplus) \circ map\ f \circ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ h\} \\
= & (fold\ (\oplus) \circ abs_S \circ smap\ f)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{ref.\ smap\} \\
= & (id \circ sfold\ (\oplus) \circ smap\ f)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{ref.\ sfold\} \\
= & (id \circ sh)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ sh\}
\end{array}$$

We can prove that the homomorphism promotion rule also applies in the stream setting as follows:

$$\begin{array}{lll}
 & sh \circ sfold\ (\widehat{+\!\!+}) & \{id\} \\
= & (sfold\ (\oplus) \circ smap)\ f \circ sfold\ (\widehat{+\!\!+}) & \{def.\ sh\} \\
= & id \circ fold\ (\oplus) \circ abs_S \circ smap\ f \circ sfold\ (\widehat{+\!\!+}) & \{ref.\ sfold\} \\
= & fold\ (\oplus) \circ abs_S \circ smap\ f \circ sfold\ (\widehat{+\!\!+}) & \{def.\ id\} \\
= & fold\ (\oplus) \circ map\ f \circ abs_S \circ sfold\ (\widehat{+\!\!+}) & \{ref.\ smap\} \\
= & fold\ (\oplus) \circ map\ f \circ fold\ (+\!\!+) \circ abs_{SS} & \{ref.\ sfold\ (\widehat{+\!\!+})\} \\
= & fold\ (\oplus) \circ fold\ (+\!\!+) \circ map\ (map\ f) \circ abs_{SS} & \{map\ promotion\} \\
= & fold\ (\oplus) \circ map\ (fold\ (\oplus)) \circ map\ (map\ f) \circ abs_{SS} & \{fold\ promotion\} \\
= & fold\ (\oplus) \circ map\ (fold\ (\oplus) \circ map\ f) \circ abs_{SS} & \{map\ distrib.\} \\
= & fold\ (\oplus) \circ map\ (fold\ (\oplus) \circ map\ f) \circ map\ abs_S \circ abs_S & \{def.\ abs_{SS}\} \\
= & fold\ (\oplus) \circ map\ (fold\ (\oplus) \circ map\ f \circ abs_S) \circ abs_S & \{map\ distrib.\} \\
= & fold\ (\oplus) \circ map\ (fold\ (\oplus) \circ abs_S \circ smap\ f) \circ abs_S & \{ref.\ smap\} \\
= & fold\ (\oplus) \circ map\ (id \circ sfold\ (\oplus) \circ smap\ f) \circ abs_S & \{ref.\ sfold\} \\
= & fold\ (\oplus) \circ map\ (sfold\ (\oplus) \circ smap\ f) \circ abs_S & \{def.\ id\} \\
= & fold\ (\oplus) \circ abs_S \circ smap\ (sfold\ (\oplus) \circ smap\ f) & \{ref.\ smap\} \\
= & id \circ sfold\ (\oplus) \circ smap\ (sfold\ (\oplus) \circ smap\ f) & \{ref.\ sfold\} \\
= & sfold\ (\oplus) \circ smap\ (sfold\ (\oplus) \circ smap\ f) & \{def.\ id\} \\
= & sfold\ (\oplus) \circ smap\ sh & \{def.\ sh\}
\end{array}$$

So, we have the following equivalence, our stream version of the generic homomorphism promotion rule.

$$sh\ \circ\ sfold\ (\widehat{+\!\!+})\ =\ sfold\ (\oplus)\ \circ\ smap\ sh \quad \{homomorphism\ promotion\}$$

So, given the following definitions:

$$
\begin{array}{lcl}
ssingle & :: & A \to \lfloor A \rfloor \\
sperhaps & :: & (A \to Bool) \to A \to \lfloor A \rfloor \\
ssingle\ x & = & \lfloor x \rfloor \\
sperhaps\ x & = & if\ p\ x\ then\ \lfloor x \rfloor\ else\ \lfloor \rfloor
\end{array}
$$

We can define our commonly used higher order functions in the homomorphic style:

$$
\begin{array}{lcl}
smap\ f & = & sfold\ (+\!\!+) \circ smap\ (ssingle \circ f) \\
sfold\ (\oplus) & = & sfold\ (\oplus) \circ smap\ id \\
sfilter\ p & = & sfold\ (+\!\!+) \circ smap\ (sperhaps\ p)
\end{array}
$$

We may then derive our common instances of the homomorphism promotion rule in stream terms. We have:

$$
\begin{array}{lcll}
smap\ f\ \circ\ sfold\ (\widehat{+\!\!+}) & = & sfold\ (\widehat{+\!\!+})\ \circ\ smap\ (smap\ f) & \{smap\ promotion\} \\
sfold\ (\oplus)\ \circ\ sfold\ (\widehat{+\!\!+}) & = & sfold\ (\oplus)\ \circ\ smap\ (sfold\ (\oplus)) & \{sfold\ promotion\} \\
sfilter\ p\ \circ\ sfold\ (\widehat{+\!\!+}) & = & sfold\ (\widehat{+\!\!+})\ \circ\ smap\ (sfilter\ p) & \{sfilter\ promotion\}
\end{array}
$$

6.2.2 Vectors

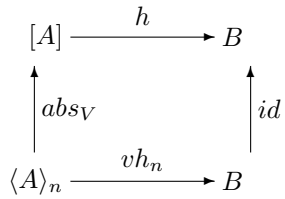
As with the stream case, one would hope the concept of the homomorphism can also be applied in the vector setting. Here, our generic homomorphism is defined as follows:

$$vh_n\ =\ vfold_n\ (\oplus)\ \circ\ vmap_n\ f$$

This has the following type:

$$vh_n :: \langle A \rangle_n \to B$$

Let us prove that this comprises a valid refinement for h, our generic list homomorphism. For this, we shall require that the following diagram commutes:



Here is the proof:

$$
\begin{array}{lll}
 & (h\ \circ\ abs_V)\ \langle x_1, x_2, ..., x_n \rangle_n & \{id\} \\
= & (fold\ (\oplus)\ \circ\ map\ f\ \circ\ abs_V)\ \langle x_1, x_2, ..., x_n \rangle_n & \{def.\ h\} \\
= & (fold\ (\oplus)\ \circ\ abs_V\ \circ\ vmap_n\ f)\ \langle x_1, x_2, ..., x_n \rangle_n & \{ref.\ vmap\} \\
= & (id\ \circ\ vfold_n\ (\oplus)\ \circ\ smap\ f)\ \langle x_1, x_2, ..., x_n \rangle_n & \{ref.\ vfold\} \\
= & (id\ \circ\ vh)\ \langle x_1, x_2, ..., x_n \rangle_n & \{def.\ vh\}
\end{array}
$$

6.3 Filter

The function $filter$ has the following type:

$$filter \ :: \ (A \rightarrow Bool) \rightarrow [A] \rightarrow [A]$$

One possible definition for $filter$ is in terms of a list comprehension, as follows:

$$filter \ p \ xs \ = \ [\ x \mid x \ \leftarrow \ xs, \ p \ x \ ]$$

We may often encounter recursive definitions for $filter$, as follows:

$$\begin{array}{lcl} filter \ p \ [] & = & [] \\ filter \ p \ (x : xs) & = & if \ p \ x \\ & & then \ x : filter \ p \ xs \\ & & else \ filter \ p \ xs \end{array}$$

Another possible definition, this time 'point-free', is as follows:

$$filter \ p \ = \ fold \ (+\!\!+) \ \circ \ map \ (\lambda x \ \bullet \ if \ p \ x \ then \ [x] \ else \ [])$$

We may find it useful to separate out the Lambda abstraction here, into a function we shall call *perhaps*. So we have:

$$\begin{array}{lcl} perhaps \ p \ x & = & if \ p \ x \ then \ [x] \ else \ [] \\ filter \ p & = & fold \ (+\!\!+) \ \circ \ map \ (perhaps \ p) \end{array}$$

As with our other higher order functions, we may find an infix binary operator version of $filter$ useful. First, the usual Haskell convention:

$$p \ `filter` \ xs \ = \ filter \ p \ xs$$

In BMF we have the following:

$$p \lhd xs \ = \ filter \ p \ xs$$

6.3.1 Streams

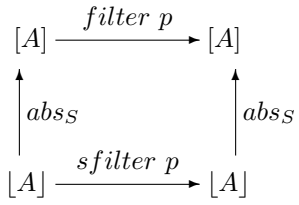
In stream terms, we have the function $sfilter$. This has the following type:

$$sfilter \ :: \ (A \rightarrow Bool) \rightarrow \lfloor A \rfloor \rightarrow \lfloor A \rfloor$$

We could define it in a similar method to $filter$, assuming we have an equivalent construct to the list comprehension in stream terms.

$$sfilter \ p \ xs \ = \lfloor \ x \mid x \ \leftarrow \ xs, \ p \ x \ \rfloor$$

To prove this is a valid refinement of $filter$, we should demonstrate that the following diagram commutes:



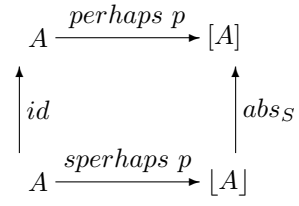
This can be proved as follows:

$$\begin{array}{lll} & (filter\ p\ \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\ = & filter\ p\ [x_1, x_2, ..., x_n] & \{def.\ abs_S\} \\ = & [\ x \mid x\ \leftarrow\ [x_1, x_2, ..., x_n],\ p\ x\ ] & \{def.\ filter\} \\ = & abs_S\ \lfloor\ x \mid x\ \leftarrow\ \lfloor x_1, x_2, ..., x_n \rfloor,\ p\ x\ \rfloor & \{def.\ abs_S\} \\ = & (abs_S\ \circ\ sfilter\ p)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ sfilter\} \end{array}$$

We may also wish to provide a second proof, for a refinement of $filter$ based on the alternative definition given earlier. We have:

$$\begin{array}{lll} sperhaps & :: & (A \to Bool) \to A \to \lfloor A \rfloor \\ sfilter & :: & (A \to Bool) \to \lfloor A \rfloor \to \lfloor A \rfloor \\ sperhaps\ p\ x & = & if\ p\ x\ then\ \lfloor x \rfloor\ else\ \lfloor \rfloor \\ sfilter\ p & = & sfold\ (\widehat{+\!\!+})\ \circ\ smap\ (sperhaps\ p) \end{array}$$

Here we make use of the stream concatenate operator (see Section 7.3). First, let us prove that *sperhaps* is a valid refinement of *perhaps*.



The proof of this is as follows:

$$\begin{array}{lll} & (perhaps\ p\ \circ\ id)\ x & \{id\} \\ = & perhaps\ p\ x & \{def.\ id\} \\ = & if\ p\ x\ then\ [x]\ else\ [] & \{def.\ perhaps\} \\ = & abs_S\ (if\ p\ x\ then\ \lfloor x \rfloor\ else\ \lfloor \rfloor) & \{def.\ abs_S\} \\ = & abs_S\ (sperhaps\ p\ x) & \{def.\ sperhaps\} \\ = & (abs_S\ \circ\ sperhaps\ p)\ x & \{def.\ \circ\} \end{array}$$

We may now construct the proof for our alternative definition of $sfilter$ as a valid refinement of $filter$:

$$
\begin{array}{lll}
 & (filter\ p\ \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\
= & (fold\ (+\!\!+)\ \circ\ map\ (perhaps\ p)\ \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ filter\} \\
= & (fold\ (+\!\!+)\ \circ\ abs_S\ \circ\ smap\ (perhaps\ p))\ \lfloor x_1, x_2, ..., x_n \rfloor & \{ref.\ smap\} \\
= & (id\ \circ\ sfold\ (+\!\!+)\ \circ\ smap\ (perhaps\ p))\ \lfloor x_1, x_2, ..., x_n \rfloor & \{ref.\ sfold\} \\
= & (sfold\ (+\!\!+)\ \circ\ smap\ (perhaps\ p))\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ id\} \\
= & (sfold\ (+\!\!+)\ \circ\ smap\ (abs_S\ \circ\ sperhaps\ p))\ \lfloor x_1, x_2, ..., x_n \rfloor & \{ref.\ sperhaps\} \\
= & (sfold\ (+\!\!+)\ \circ\ smap\ abs_S\ \circ\ smap\ (sperhaps\ p))\ \lfloor x_1, x_2, ..., x_n \rfloor & \{smap\ distrib.\} \\
= & (abs_S\ \circ\ sfold\ (\widehat{+\!\!+})\ \circ\ smap\ (sperhaps\ p))\ \lfloor x_1, x_2, ..., x_n \rfloor & \{abs_S\ promotion\} \\
= & (abs_S\ \circ\ sfilter\ p)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ sfilter\}
\end{array}
$$

Let us clarify the use of abs_S promotion in the above proof. This provides us with the following equivalence:

$$abs_S\ \circ\ sfold\ (\widehat{+\!\!+})\ =\ sfold\ (+\!\!+)\ \circ\ smap\ abs_S$$

Close inspection should reveal that this is in fact an instance of the homomorphism promotion rule in stream terms. We can reassure ourselves that abs_S is indeed a stream homomorphism given that we can define it as follows:

$$abs_S\ =\ sfold\ (+\!\!+) \circ smap\ single$$

Process Refinement

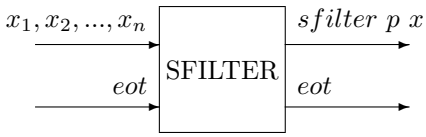


Figure 6.1: The map process for streams.

A process implementing the functionality of $filter\ p$ in stream terms should input a stream of values, and output a stream of values containing only those values that satisfy the predicate p. This is depicted in Figure 6.1.

Simple Case Definition

Let us consider a process candidate for $SFILTER$ in the simple case that the members of the stream are items. Here, at each stage, if the input *eot* channel is willing to communicate, we receive a message from it, echo the message to the output *eot* channel, and then we are finished. If the input value channel is willing to communicate, we receive a single value from it, and apply the predicate p to it. If the result is true, we may output this value and proceed to the next step. If

not we proceed to the next step without outputting the value. In this version, the predicate p can be passed as a function parameter to $SFILTER$.

$$\begin{aligned} SFILTER(p) \quad = \mu\, X \bullet \quad & in.eot\ ?\ any \rightarrow out.eot\ !\ any \rightarrow SKIP \\ & | \\ & in.value\ ?\ x \rightarrow (\ out.value\ !\ x \rightarrow X \ \ \nleftarrow p\ x \nrightarrow \ \ X\ ) \end{aligned}$$

It may help clarify this definition to include also the alphabet of the process $SFILTER$. We have:

$$\alpha SFILTER(p) \ = \ \{in :: \underline{\lfloor A \rfloor}, out :: \underline{\lfloor A \rfloor}\}$$

General Case Definition

As was the case with $SMAP$, in the more general case, we shall find the processing of the *eot* conduit will remain the same. However, the processing of the values that make up the stream will depend on their type. In the above case we assumed the values were simple items, however, the values may well be streams or vectors in themselves. In fact, we shall see a more general implementation of $SFILTER$ actually matches that of $SMAP$.

$$\begin{aligned} SFILTER(P) \quad = \mu\, X \bullet \quad & in.eot\ ?\ any \rightarrow out.eot\ !\ any \rightarrow SKIP \\ & \Box \\ & P[in.value/in, out.value/out]; X \end{aligned}$$

The process P here is not a direct refinement of the function p in the original specification. The process P has the following alphabet:

$$\alpha P \ = \ \{in :: \underline{A}, out :: \underline{A}\}$$

This process should read from its input conduit, and, in the case that the value read satisfies the predicate p, output the value to the output conduit. If the value does not satisfy the predicate, it will output nothing. In fact, the process P should be a refinement of the expression *perhaps* p, where *perhaps* is defined as follows:

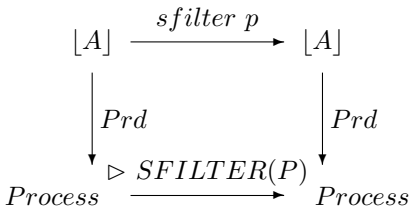
$$perhaps\ p\ x \ = if\ p\ x\ then\ x\ else\ nothing$$

The alphabet of $SFILTER(P)$ remains the same as that for our simple case definition:

$$\alpha SFILTER(P) \ = \ \{in :: \underline{\lfloor A \rfloor}, out :: \underline{\lfloor A \rfloor}\}$$

Proof

To prove that $SFILTER(P)$ is a valid refinement of $sfilter\ p$, we shall require the following diagram to commute:



That is to say, we require the following equivalence to hold for any stream s, and predicate p, where P is a valid refinement of *perhaps* p :

$$Prd\ s\ \triangleright\ SFILTER(P)\ =\ Prd\ (sfilter\ p\ s)$$

In the case of the empty stream, the proof will follow that used for $SMAP$ above. In the case of the non-empty stream, we shall need to consider the relationship of the value *nothing* with the function Prd. As previously stated, Prd applied to *nothing* returns the process $SKIP$.

$$Prd\ nothing = SKIP$$

As a consequence of this, consider producing a stream containing several items, one of which is *nothing*:

$$Prd\ (Stream\ [nothing : s])$$

Recall part of the definition of Prd when applied to streams:

$$Prd\ (Stream\ (a : s))\ =\ (Prd\ a)[out.value/out];\ Prd\ (Stream\ s)$$

As such we find the following equivalence:

$$Prd\ (Stream\ [nothing : s])\ =\ SKIP;\ Prd\ (Stream\ s)$$

Given that in CSP, the behaviour of $SKIP$ sequentially composed with any process P is equivalent to just P, we have:

$$Prd\ (Stream\ [nothing : s])\ =\ Prd\ (Stream\ s)$$

To generalise, we can state that the action of producing a stream containing any number of *nothing* values is identical to producing the same stream with all of the *nothing* values removed. Thus filtering a stream on a predicate p and producing the result is equivalent to mapping *perhaps* p to the stream and producing that result:

$$Prd\ (sfilter\ p\ s)\ =\ Prd\ (smap\ (perhaps\ p)\ s)$$

Thus *smap* (*perhaps* p) can be seen as a refinement of $sfilter\ p$ in function terms. In process terms, we have already shown $SMAP(F)$ to be a valid refinement of *smap* f. So, in turn $SMAP(P)$, where P is a valid refinement of *perhaps* p, is a valid refinement for $sfilter\ p$.

Handel-C Implementation

In the simple case where we are dealing with a stream of items, we can construct a definition for `SFILTER` as given in Figure 6.2. Here the parameter `p` is an expression, perhaps a Handel-C `macro expr`.

```
macro proc SFILTER_SIMPLE (streamin,streamout,p)
{
   Bool eot;
   messagetype (streamin) x;
   eot  = False;
   do
   {
      prialt
      {
      case streamin.eot ? eot:
         streamout.eot ! True;
         break;
      case streamin.value ? x:
         if (p(x))
         {
           streamout.value ! x;
         }
         break;
      }
   } while (!eot)
}
```

Figure 6.2: The simple case definition of the process SFILTER.

In the more general case, the definition of `SFILTER` will in fact be equivalent to that of `SMAP`. The process `F` passed as a parameter will adhere to the following scheme, the process `SPERHAPS`. This is given in Figure 6.3. Here `p` is some expression, possibly a Handel-C `macro expr` defining the required predicate functionality.

```
macro proc SPERHAPS (conduitin,conduitout)
{
   prialt
   {
   case conduitin ? x:
      if (p(x))
      {
         conduitout ! f (x);
      }
      break;
   default:
      break;
   }
}
```

Figure 6.3: The definition of the process SPERHAPS.

6.3.2 Vectors

In vector terms, let us consider a possible refinement $vfilter$.

$$vfilter_n \ :: \ (A \to Bool) \to \langle A \rangle_n \to \langle A \rangle_m$$

One notable characteristic of $filter$ is that the input and output are not necessarily the same size, and therefore the output size is not predictable in advance. We can see this in the differing subscripts in the type definition above. As such we can not implement filter directly in terms of vectors - instead we would need to look towards a combined structure.

6.3.3 Vector to Stream

One possible alternative, which may be useful in certain circumstances, is a refinement of $filter$ which takes the input as a vector, but produces the output as a stream. Here we have a function $v2sfilter_n$, with the following type:

$$v2sfilter_n \ :: \ (A \to Bool) \to \langle A \rangle_n \to \lfloor A \rfloor$$

A definition could be given similar to that for $sfilter$ above:

$$v2sfilter_n \ p \ = \ vfold_n \ (\widehat{+\!\!+}) \ \circ \ vmap_n \ (sperhaps \ p)$$

Here we are reusing the exact same definition of *sperhaps* as used in the definition of $sfilter$, above. For this definition of $v2sfilter_n$ to be considered a valid refinement of $filter$, we require that the following diagram commutes:

$$\begin{array}{ccc} [A] & \xrightarrow{filter \ p} & [A] \\ \uparrow{\scriptstyle abs_V} & & \uparrow{\scriptstyle abs_S} \\ \langle A \rangle_n & \xrightarrow{v2sfilter \ p} & \lfloor A \rfloor \end{array}$$

The proof of this is as follows:

$$\begin{array}{lll} & (filter \ p \ \circ \ abs_V) \ \langle x_1, x_2, ..., x_n \rangle_n & \{id\} \\ = & (fold \ (+\!\!+) \ \circ \ map \ (perhaps \ p) \ \circ \ abs_V) \ \langle x_1, x_2, ..., x_n \rangle_n & \{def. \ filter\} \\ = & (fold \ (+\!\!+) \ \circ \ abs_S \ \circ \ vmap_n \ (perhaps \ p)) \ \langle x_1, x_2, ..., x_n \rangle_n & \{ref. \ vmap\} \\ = & (id \ \circ \ vfold_n \ (+\!\!+) \ \circ \ vmap_n \ (perhaps \ p)) \ \langle x_1, x_2, ..., x_n \rangle_n & \{ref. \ vfold\} \\ = & (vfold_n \ (+\!\!+) \ \circ \ vmap_n \ (perhaps \ p)) \ \langle x_1, x_2, ..., x_n \rangle_n & \{def. \ id\} \\ = & (vfold_n \ (+\!\!+) \ \circ \ vmap_n \ (abs_S \ \circ \ sperhaps \ p)) \ \langle x_1, x_2, ..., x_n \rangle_n & \{ref. \ sperhaps\} \\ = & (vfold_n \ (+\!\!+) \ \circ \ vmap_n \ abs_S \ \circ \ vmap_n \ (sperhaps \ p)) \ \langle x_1, x_2, ..., x_n \rangle_n & \{vmap \ distrib.\} \\ = & (abs_S \circ vfold_n \ (\widehat{+\!\!+}) \ \circ \ vmap_n \ (sperhaps \ p)) \ \langle x_1, x_2, ..., x_n \rangle_n & \{see \ below\} \\ = & (abs_S \ \circ \ vfilter_n \ p) \ \langle x_1, x_2, ..., x_n \rangle_n & \{def. \ vfilter\} \end{array}$$

As with our earlier proof for $sfilter$, this proof hinges on the following equivalence, a form of homomorphism promotion:

$$abs_S \circ vfold_n\ (\widehat{+\!\!+})\ =\ vfold_n\ (+\!\!+)\ \circ\ vmap_n\ abs_S$$

To prove that this equivalence holds, we require that the following diagram commutes:

$$\begin{array}{ccc} \langle [A] \rangle_n & \xrightarrow{vfold_n\ (+\!\!+)} & [A] \\ \uparrow{\scriptstyle vmap_n\ abs_S} & & \uparrow{\scriptstyle abs_S} \\ \langle \lfloor A \rfloor \rangle_n & \xrightarrow{vfold_n\ (\widehat{+\!\!+})} & \lfloor A \rfloor \end{array}$$

Here comes the proof, as before, analogous to those given in Section 7.3.1:

$$\begin{array}{lll} & (vfold_n\ (+\!\!+)\ \circ\ vmap_n\ abs_S)\ \langle s_1, s_2, ..., s_n \rangle_n & \{id\} \\ = & vfold_n\ (+\!\!+)\ \lfloor abs_S\ s_1, abs_S\ s_2, ..., abs_S\ s_n \rfloor & \{def.\ vmap_n\} \\ = & abs_S\ s_1\ +\!\!+\ abs_S\ s_2\ +\!\!+\ ...\ +\!\!+\ abs_S\ s_n & \{def.\ vfold_n\} \\ = & (abs_S\ (s_1\ \widehat{+\!\!+}\ s_2))\ +\!\!+\ ...\ +\!\!+\ abs_S\ s_n & \{ref.\ \widehat{+\!\!+}\} \\ = & (abs_S\ (s_1\ \widehat{+\!\!+}\ s_2\ \widehat{+\!\!+}\ s_3))\ +\!\!+ ...\ +\!\!+\ abs_S\ s_n & \{ref.\ \widehat{+\!\!+}\} \\ & & \{...\} \\ = & abs_S\ (s_1\ \widehat{+\!\!+}\ s_2\ \widehat{+\!\!+}\ ...\ \widehat{+\!\!+}\ s_n) & \{ref.\ \widehat{+\!\!+}\} \\ = & (abs_S\ \circ\ vfold_n\ (\widehat{+\!\!+}))\ \langle s_1, s_2, ..., s_n \rangle_n & \{def.\ vfold_n\} \end{array}$$

Process Refinement

As noted previously, filter can not be implemented directly where both the input and the output is a vector.

Let us consider the specialised refinement of $filter$ which takes in a vector but outputs a stream, the function $v2sfilter_n$. Let us consider here a process refinement of this in the form of the process $V2SFILTER_n$. This is depicted in Figure 6.4.

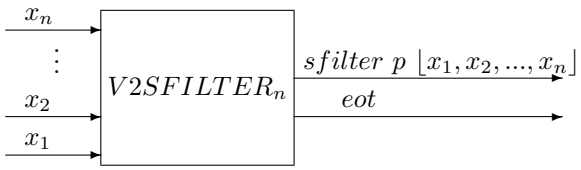


Figure 6.4: The filter process with vector input and stream output.

6.3.4 Combined Structures

In terms of a two dimensional structure (for example a list of lists), the act of filtering the entire structure is performed by mapping the filter function to each part. We shall call this function

$tdfilter$. This can be defined as follows:

$$tdfilter\ p = map\ (filter\ p)$$

This has type:

$$[[A]] \rightarrow [[A]]$$

6.3.5 Distributed Lists

The function $tdfilter$ should provide a suitable refinement for $filter$ in a distributed list setting. As a proof of this we would require the following diagram to commute:

$$\begin{array}{ccc} [A] & \xrightarrow{filter\ p} & [A] \\ \uparrow{\scriptstyle abs_D} & & \uparrow{\scriptstyle abs_D} \\ [[A]] & \xrightarrow{tdfilter\ p} & [[A]] \end{array}$$

The proof of this, taking a list of lists $[l_1, l_2, ..., l_n]$, and our first definition for abs_D, is a simple product of filter promotion:

$$\begin{array}{lll} & (filter\ p\ \circ\ abs_{D1})\ [l_1, l_2, ..., l_n] & \{id\} \\ = & (filter\ p\ \circ\ fold\ (+\!\!+))\ [l_1, l_2, ..., l_n] & \{def.\ abs_{D1}\} \\ = & (fold\ (+\!\!+)\ \circ\ map\ (filter\ p))\ [l_1, l_2, ..., l_n] & \{filter\ promotion\} \\ = & (fold\ (+\!\!+)\ \circ\ tdfilter\ p)\ [l_1, l_2, ..., l_n] & \{def.\ tdfilter\} \\ = & (abs_{D1}\ \circ\ tdfilter\ p)\ [l_1, l_2, ..., l_n] & \{def.\ abs_{D1}\} \end{array}$$

6.3.6 Stream of Streams

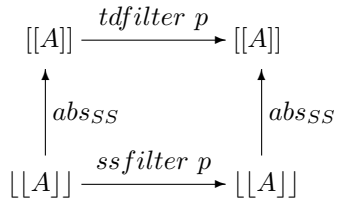
Given that we have a valid refinement of $filter$ in stream terms, we may also assume we can derive a refinement of the two dimensional filter in terms of the stream of streams. As regards the stream of streams, the function $tdfilter$ can be refined by the function $ssfilter$. This can be defined as follows, simply by replacing the $filter$ and map used in $tdfilter$ with the corresponding stream refinement:

$$ssfilter\ p = smap\ (sfilter\ p)$$

This has type:

$$\lfloor\lfloor A\rfloor\rfloor \rightarrow \lfloor\lfloor A\rfloor\rfloor$$

Proof of the validity of this refinement requires that the following diagram commutes:



The proof may proceed as follows:

$$
\begin{array}{lll}
 & (tdfilter\ p\ \circ\ abs_{SS})\ \lfloor s_1, ..., s_n \rfloor & \{id\} \\
= & tdfilter\ p\ [l_1, ..., l_n] & \{def.\ abs_{SS}\} \\
= & map\ (filter\ p)\ [l_1, ..., l_n] & \{def.\ tdfilter\} \\
= & [filter\ p\ l_1, ..., filter\ p\ l_n] & \{def.\ map\} \\
= & [(filter\ p\ \circ\ abs_S)\ s_1, ..., (filter\ p\ \circ\ abs_S)\ s_n] & \{def.\ abs_S\} \\
= & [(abs_S\ \circ\ sfilter\ p)\ s_1, ..., (abs_S\ \circ\ sfilter\ p)\ s_n] & \{def.\ sfilter\} \\
= & map\ (abs_S\ \circ\ sfilter\ p)\ [s_1, ..., s_n] & \{def.\ map\} \\
= & (map\ abs_S\ \circ\ map\ (sfilter\ p))\ [s_1, ..., s_n] & \{map - dist\} \\
= & (map\ abs_S\ \circ\ map\ (sfilter\ p)\ \circ\ abs_S)\ \lfloor s_1, ..., s_n \rfloor & \{def.\ abs_S\} \\
= & (map\ abs_S\ \circ\ abs_S\ \circ\ smap\ (sfilter\ p))\ \lfloor s_1, ..., s_n \rfloor & \{def.\ smap\} \\
= & (abs_{SS}\ \circ\ smap\ (sfilter\ p))\ \lfloor s_1, ..., s_n \rfloor & \{def.\ abs_{SS}\} \\
= & (abs_{SS}\ \circ\ ssfilter\ p)\ \lfloor s_1, ..., s_n \rfloor & \{def.\ ssfilter\}
\end{array}
$$

Process Refinement

We have shown that wherever F is a valid refinement of f, then $SMAP(F)$ is a valid refinement of $smap\ f$. Additionally we have shown that wherever P is a valid refinement of $perhaps\ p$, then $SFILTER(P)$ is a valid refinement of $sfilter\ p$. Thus if we recall the definition for $ssfilter$:

$$ssfilter\ p = smap\ (sfilter\ p)$$

A valid process refinement, $SSFILTER$, should follow intuitively:

$$SSFILTER(P)\ =\ SMAP(SFILTER(P))$$

Given a process P, with alphabet:

$$\alpha P\ =\ \{in :: \underline{A}, out :: \underline{A}\}$$

The alphabet of $SSFILTER(P)$ can be defined as follows:

$$\alpha SSFILTER(P)\ =\ \{in :: \underline{\lfloor\lfloor A \rfloor\rfloor}, out :: \underline{\lfloor\lfloor A \rfloor\rfloor}\}$$

Handel-C Implementation

As was the case with $SSMAP$, we would like to be able to do something like the following:

```
macro proc SSFILTER (streamofstreamsin,streamofstreamsout,p)
{
   SMAP (streamofstreamsin,streamofstreamsout,SFILTER(p));
}
```

However, the only definition Handel-C will natively understand requires expansion of one level of $SFILTER$ (equivalent to $SMAP$), thus we have the definition shown in Figure 6.5.

```
macro proc SSFILTER (streamofstreamsin,streamofstreamsout,p)
{
   Bool eot;
   eot  = False;
   do
   {
      prialt
      {
      case streamofstreamsin.eot ? eot:
         streamofstreamsout.eot ! True;
         break;
      default:
         SFILTER_SIMPLE (streamofstreamsin.value,
                         streamofstreamsout.value,
                         p);
         break;
      }
   } while (!eot)
}
```

Figure 6.5: The Handel-C definition of the process SSFILTER.

6.3.7 Vector of Streams

In terms of a vector of streams, the function *tdfilter* can be refined by $vsfilter$. We can define this analogously to *tdfilter*. Given that the 'parts' are streams, we shall apply $sfilter$ to each, and knowing that these are contained within a vector, we can achieve this with *vmap*.

$$vsfilter\ p = vmap\ (sfilter\ p)$$

This has type:

$$\langle \lfloor A \rfloor \rangle_n \rightarrow \langle \lfloor A \rfloor \rangle_n$$

Proof of the validity of this refinement requires that the following diagram commutes:

$$\begin{array}{ccc} [[A]] & \xrightarrow{tdfilter\ p} & [[A]] \\ \uparrow{\scriptstyle abs_{VS}} & & \uparrow{\scriptstyle abs_{VS}} \\ \langle\lfloor A\rfloor\rangle_n & \xrightarrow{vsfilter\ p} & \langle\lfloor A\rfloor\rangle_n \end{array}$$

The proof may proceed as follows, in a fashion analogous to that for the stream of streams.

$$\begin{array}{lll} & (tdfilter\ p\ \circ\ abs_{VS})\ \langle s_1, ..., s_n\rangle_n & \{id\} \\ = & tdfilter\ p\ [l_1, ..., l_n] & \{def.\ abs_{VS}\} \\ = & map\ (filter\ p)\ [l_1, ..., l_n] & \{def.\ tdfilter\} \\ = & [filter\ p\ l_1, ..., filter\ p\ l_n] & \{def.\ map\} \\ = & [(filter\ p\ \circ\ abs_S)\ s_1, ..., (filter\ p\ \circ\ abs_S)\ s_n] & \{def.\ abs_S\} \\ = & [(abs_S\ \circ\ sfilter\ p)\ s_1, ..., (abs_S\ \circ\ sfilter\ p)\ s_n] & \{def.\ sfilter\} \\ = & map\ (abs_S\ \circ\ sfilter\ p)\ [s_1, ..., s_n] & \{def.\ map\} \\ = & (map\ abs_S\ \circ\ map\ (sfilter\ p))\ [s_1, ..., s_n] & \{map - dist\} \\ = & (map\ abs_S\ \circ\ map\ (sfilter\ p)\ \circ\ abs_V)\ \langle s_1, ..., s_n\rangle_n & \{def.\ abs_V\} \\ = & (map\ abs_S\ \circ\ abs_V\ \circ\ vmap\ (sfilter\ p))\ \langle s_1, ..., s_n\rangle_n & \{def.\ vmap\} \\ = & (abs_{VS}\ \circ\ vmap\ (sfilter\ p))\ \langle s_1, ..., s_n\rangle_n & \{def.\ abs_{VS}\} \\ = & (abs_{VS}\ \circ\ vsfilter\ p)\ \langle s_1, ..., s_n\rangle_n & \{def.\ vsfilter\} \end{array}$$

Process Refinement

Let us recall our definition of $vsfilter$:

$$vsfilter\ p = vmap\ (sfilter\ p)$$

As before, given our refinements for $vmap$ and $sfilter$, we can construct the following refinement for $vsfilter\ p$ somewhat intuitively.

$$VSFILTER(P)\ =\ VMAP(SFILTER(P))$$

Given a process P, with alphabet:

$$\alpha P\ =\ \{in :: \underline{A}, out :: \underline{A}\}$$

The alphabet of $VSFILTER(P)$ can be defined as follows:

$$\alpha VSFILTER(P)\ =\ \{in :: \underline{\langle\lfloor A\rfloor\rangle_n}, out :: \underline{\langle\lfloor A\rfloor\rangle_n}\}$$

Handel-C Implementation

Again, were Handel-C to support some form of currying, we would be able to specify something like the following:

```
macro proc VSFILTER (size,vectorofstreamsin,vectorofstreamsout,p)
{
   VMAP (size,vectorofstreamsin,vectorofstreamsout,SFILTER(p));
}
```

However, we shall have to settle instead for expanding the definition of `SFILTER` into that of `VMAP`, as shown in Figure 6.6.

```
macro proc VSFILTER (size,vectorofstreamsin,vectorofstreamsout,p)
{
   typeof (size) c;
   par (c=0;c<size;c++)
   {
      SFILTER_SIMPLE(vectorofstreamsin[c],vectorofstreamsout[c],p);
   }
}
```

Figure 6.6: The Handel-C definition of the process VSFILTER.

6.3.8 Stream of Vectors

An implementation of *tdfilter* in terms of the stream of vectors would require a vector refinement of $filter$. As we have already stated that we can not supply one of these, it follows that we are also not able to provide a refinement of *tdfilter* in terms of the stream of vectors.

6.3.9 Vector of Vectors

As before, we can not supply a valid refinement of *tdfilter* in terms of the vector of vectors.

6.4 Unfold

The dual of $fold$ presents itself to us in the form of the $unfold$ function. This family of functions are also known as anamorphisms. The $unfold$ pattern is perhaps less well know than $fold$, however some interesting exploration of its properties can be found in [15]. At the simplest level, the $fold$ family of functions reduce a sequence of values to return a single value. On the other hand, the $unfold$ functions can be seen as expanding a single value into a sequence of values. Unlike $fold$, there doesn't seem to have been a widely accepted definition of $unfold$, although the general functionality should be the same regardless. The Haskell 98 standard library [37] includes a definition for $unfoldr$ which uses Haskell's $Maybe$ type:

$$\begin{array}{lcl} unfoldr & :: & (B \rightarrow Maybe\ (A,B)) \rightarrow B \rightarrow [A] \\ unfoldr\ f\ b & = & case\ f\ b\ of \\ & & \quad Nothing\ \rightarrow\ [] \\ & & \quad Just\ (a,b)\ \rightarrow\ a\ :\ unfoldr\ f\ b \end{array}$$

We shall instead use definitions which hinge on two functions passed as parameters. The first is a simple predicate function which determines if a given value is a base value. This will provide the guard for our recursion. We could instead pass a value here, and have $unfold$ compare the current value to this base value at each step, however, a function is obviously more general. The second function performs a single step - given a value, it returns a pair of values, one representing the value to be output at this step, and the other the value to calculate the next step from. As with $fold$, we have both left directed and right directed implementations, and these shall have the following type:

$$\begin{array}{l} unfoldl :: (A \rightarrow (A,B)) \rightarrow (A \rightarrow Bool) \rightarrow A \rightarrow [B] \\ unfoldr :: (A \rightarrow (B,A)) \rightarrow (A \rightarrow Bool) \rightarrow A \rightarrow [B] \end{array}$$

Definitions can be given as follows, firstly for the right directed version:

$$\begin{array}{lcl} unfoldr\ f\ p\ a & = & if\ p\ a\ then\ [] \\ & & else\ b : unfoldr\ f\ p\ c \\ & & where\ (b,c)\ =\ f\ a \end{array}$$

Similarly for the left directed version:

$$\begin{array}{lcl} unfoldl\ f\ p\ a & = & if\ p\ a\ then\ [] \\ & & else\ unfoldl\ f\ p\ b +\!\!+ [c] \\ & & where\ (b,c)\ =\ f\ a \end{array}$$

Interestingly, map can be specified in terms of $unfoldr$:

$$map\ f\ xs\ =\ unfoldr\ (\lambda(x:xs) \bullet (f\ x, xs))\ null\ xs$$

Often we shall not use $unfold$ directly - it is in fact a very general pattern which captures the behaviour of many other functions (See Sections 7.12, 7.14, and 7.15 for some examples of these). However, to give an idea of its use we shall look at two examples below.

Example - Lists

It is an important property of an unfold operation that it is capable of "undoing" a corresponding fold operation. We may illustrate this relationship between $fold$ and $unfold$ by using them together to construct the identity function on lists. That is to say, given functions g and p where:

$$\begin{aligned} g\ (f\ x\ y) &= (x, y) \\ p\ x &= (x == e) \end{aligned}$$

We can then state:

$$\begin{aligned} unfoldr\ g\ p\ (foldr\ f\ e\ xs) &= xs \\ unfoldl\ g\ p\ (foldl\ f\ e\ xs) &= xs \end{aligned}$$

Or, in other words:

$$\begin{aligned} unfoldr\ g\ p\ \circ\ foldr\ f\ e &= id_{[]} \\ unfoldl\ g\ p\ \circ\ foldl\ f\ e &= id_{[]} \end{aligned}$$

We can provide concrete examples of these. First, for our right variant, we have:

$$\begin{aligned} f &= (:) \\ g\ (x : xs) &= (x, xs) \\ e &= [] \\ p\ x &= x == [] \end{aligned}$$

Which gives us:

$$unfoldr\ (\lambda(x : xs) \bullet (x, xs))\ (== []) \circ foldr\ (:)\ []\ =\ id_{[]}$$

Secondly, for our left variant we have:

$$\begin{aligned} f &= flip\ (:) \\ g\ (x : xs) &= (xs, x) \\ e &= [] \\ p\ x &= x == [] \end{aligned}$$

Which gives us:

$$unfoldl\ (\lambda(x : xs) \bullet (xs, x))\ (== []) \circ foldl\ (flip\ (:))\ []\ =\ id_{[]}$$

Example - Binary Trees

Let us examine a slightly more sophisticated example. First, let us define a simple binary tree type.

$$data\ Tree\ a\ =\ Bin\ a\ (Tree\ a)\ (Tree\ a)\ |\ Null$$

Next a simple operator to insert a single item into the correct position of the tree. This has type:

$$(\oplus)\ ::\ Ord\ a \Rightarrow a \rightarrow Tree\ a \rightarrow Tree\ a$$

...and we can define it as follows:

$$\begin{array}{lcl} x \oplus Null & = & Bin\ x\ Null\ Null \\ x \oplus (Bin\ y\ a\ b) & = & if\ x < y \\ & & then\ Bin\ y\ (x \oplus a)\ b \\ & & else\ Bin\ y\ a\ (x \oplus b) \end{array}$$

Correspondingly, we shall also require a simple operator to remove a single item from the tree. This should take in a tree, and return a pair. The first item of the pair is the lowest value encountered in the tree, and the second item is the state of the tree after the removal. As such we have the following type:

$$(\ominus)\ ::\ Tree\ a \rightarrow (a, Tree\ a)$$

This can be defined as follows:

$$\begin{array}{lcl} \ominus\ (Bin\ x\ Null\ b) & = & (x, b) \\ \ominus\ (Bin\ x\ a\ Null) & = & (v, Bin\ x\ t\ Null) \\ & & where\ (v, t)\ =\ \ominus\ a \end{array}$$

Given these two operators, along with the $fold$ and $unfold$ functions, we can contract a binary tree sort algorithm. The algorithm consists of two phases. First, we employ $foldr$ to repeatedly apply our operator $(\oplus)$ to construct a binary tree containing every item in the input list. In the second phase, we deconstruct our intermediate tree, using $unfoldr$ and our $(\ominus)$ operator to remove each item in order and add it to a list.

$$bsort = unfoldr\ (\ominus)\ (== Null)\ \circ\ foldr\ (\oplus)\ Null$$

Hopefully this example helps to give a feeling for the relationship between $fold$ and $unfold$, as well as giving a practical demonstration of how they can be used together.

6.4.1 Streams

In stream terms, we have two functions $sunfoldl$ and $sunfoldr$ with the following types:

$$sunfoldl :: (A \to (B, A)) \to (A \to Bool) \to A \to \lfloor B \rfloor$$
$$sunfoldr :: (A \to (B, A)) \to (A \to Bool) \to A \to \lfloor B \rfloor$$

For any definition of $sunfoldr$ (or similarly for $sunfoldl$) to be considered a valid refinement of $unfoldr$ (or $unfoldl$) we shall require that the following diagram commutes:

$$\begin{array}{ccc} A & \xrightarrow{unfoldr\ f\ p} & [B] \\ \uparrow{\scriptstyle id} & & \uparrow{\scriptstyle abs_S} \\ A & \xrightarrow{sunfoldr\ f\ p} & \lfloor B \rfloor \end{array}$$

Let us consider a definition of $sunfoldr$ following the same lines as $unfoldr$.

$$\begin{array}{lcl} sunfoldr\ f\ p\ a & = & if\ p\ a\ then\ \lfloor \rfloor \\ & & else\ b \mathbin{\widehat{:}} sunfoldr\ f\ p\ c \\ & & where\ (b, c)\ =\ f\ a \end{array}$$

To prove this is a valid refinement, first for the base case (where $p\ a$ is True), we have:

$$\begin{array}{ll} (unfoldr\ f\ p \circ id)\ a & \{\} \\ = unfoldr\ f\ p\ a & \{def.\ id\} \\ = if\ p\ a\ then\ [] & \\ \quad else\ b : unfoldr\ f\ p\ c & \\ \quad where\ (b, c)\ =\ f\ a & \{def.\ unfoldr\} \\ = if\ p\ a\ then\ abs_S\ \lfloor \rfloor & \\ \quad else\ b : unfoldr\ f\ p\ c & \\ \quad where\ (b, c)\ =\ f\ a & \{def.\ abs_S\} \\ = if\ p\ a\ then\ abs_S\ \lfloor \rfloor & \\ \quad else\ (fst\ (f\ a)) : unfoldr\ f\ p\ (snd\ (f\ a)) & \{def.\ where\} \\ = (abs_S \circ sunfoldr\ f\ p)\ a & \{def.\ sunfoldr\} \end{array}$$

Process Refinement

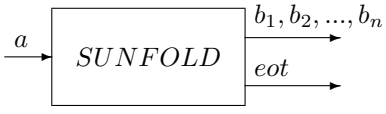


Figure 6.7: The unfold process in stream terms.

As we have done previously, we shall begin by considering the simple case where we are dealing with a 'one dimensional' stream of basic items - i.e. integers, booleans, etc. Taking first the right directed variant, we can translate our recursive functional specification $sunfoldr$ into an iterative process, as follows:

$$\begin{aligned} SUNFOLDR(f,p) \;=\; & in\ ?\ a \rightarrow \\ & \mu X \bullet \\ & \text{if } (p\ a) \\ & \text{then } out.eot\ !\ True\ \rightarrow\ SKIP \\ & \text{else } (b,a)\ :=\ f(a);\ out.value\ !\ b; X \end{aligned}$$

When we come to consider the refinement for our functional specification $sunfoldl$, we are faced with similar issues to those faced when refining $sfoldr$. This is due to the list being generated in reverse order. One way to solve this would be to pipe the stream to an instance of the $SREVERSE$ process.

$$SUNFOLDL(f,p) \;=\; SUNFOLDR(f,p) \gg SREVERSE$$

Alternatively, we could buffer the items locally, and then produce them when the predicate is finally met. As was the case with the refinement of $sfoldr$, the consequence of this is that we need to specify an upper limit in advance for the length of the outgoing stream.

For more general refinements of our unfold functions, we can move the responsibility for producing the value b in the output stream to the process F. This would give us something like the following:

$$\begin{aligned} SUNFOLDR(F,p) \;=\; & in\ ?\ a \rightarrow \\ & \mu X \bullet \\ & if\ (p\ a) \\ & then\ out.eot\ !\ True\ \rightarrow\ SKIP \\ & else\ F(\&a)[out/out]; X \end{aligned}$$

In this version, the process F is passed the current 'state' variable a. It then has the task of applying a refinement of the original function f to that value, to create a new value for the state variable a and a value b which it must output on the appropriate conduit. The ampersand `&` here denotes that the parameter is passed by reference, so changes to a within F apply within the scope of $SUNFOLDR$ also. This refinement still lacks a little generality in that the input is still restricted to being a simple item - the above definition would not support a stream or a vector as input.

Handel-C Implementation

The Handel-C implementation may then proceed as follows, considering first the simple case. Effectively, we keep the iterative behaviour of the CSP process, however, as a slight modification,

to fit with the standard scheme of an imperative `while` loop, we are in fact testing a negated version of the predicate `p`. Ideally, to keep the behaviour simple and clean, we would like to refine the function f to an expression. As Handel-C does not easily facilitate pairs of values as return types of expressions, we shall instead have to refine f to a simple Handel-C `macro proc`. The process `f` is passed the two state variables `a` and `b` by reference. It is tasked with deriving the new value for `a` and `b` from the current value of `a`. The definition is given in Figure 6.8.

```
macro proc SUNFOLDR_SIMPLE (conduitin, streamout, f, p)
{
   messagetype (conduitin) a;
   messagetype (streamout) b;
   conduitin ? a;
   while (!p(a))
   {
      f (a,b);
      streamout.valueconduit ! b;
   }
   streamout.eotconduit ! True;
}
```

Figure 6.8: The simple case definition of the process SUNFOLDR.

As noted above, for a more general version, we can move the responsibility for producing the value b in the output stream to the process F. This gives us the definition supplied in Figure 6.9.

```
macro proc SUNFOLDR (conduitin, streamout, F, p)
{
   messagetype (conduitin) a;
   messagetype (streamout) b;
   conduitin ? a;
   while (!p(a))
   {
      F (streamout.valueconduit,a,b);
   }
   streamout.eotconduit ! True;
}
```

Figure 6.9: The general case definition of the process SUNFOLDR.

6.4.2 Vectors

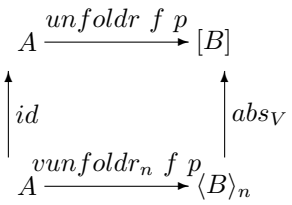
In vector terms, we have two functions $vunfoldl$ and $vunfoldr$ with the following types:

$$vunfoldl_n :: (A \to (B, A)) \to (A \to Bool) \to A \to \langle B \rangle_n$$
$$vunfoldr_n :: (A \to (B, A)) \to (A \to Bool) \to A \to \langle B \rangle_n$$

We are now working with a slightly different interpretation of our unfold refinements. The fixed size nature of vectors requires that the size of the output vector is known in advance. As such, we

will only be able to make a refinement in this way where the size can be somehow predicted from the input parameters.

For any definition of $vunfoldr$ (or similarly for $vunfoldl$) to be considered a valid refinement of $unfoldr$ (or $unfoldl$) we shall require that the following diagram commutes:



Let us consider a definition for $vunfoldr$ which follows in an analogous fashion to that provided in the stream case.

$$\begin{aligned} vunfoldr_n\ f\ p\ a &= if\ p\ a\ then\ \langle\rangle_0 \\ &\quad else\ b \mathbin{:\!\!\sim_n} vunfoldr_{n-1}\ f\ p\ c \\ &\quad where\ (b,c) = f\ a \end{aligned}$$

We could provide a similar proof here to that for $sunfoldr$, above. In practice, the predicate p will have to return true at the point where n equals zero - this is the only scenario which allows the above definition to be correctly typed. So in other words we have:

$$\begin{aligned} vunfoldr_0\ f\ p\ a &= \langle\rangle_0 \\ vunfoldr_{n+1}\ f\ p\ a &= b \mathbin{:\!\!\sim_n} vunfoldr_n\ f\ p\ c \\ &\quad where\ (b,c) = f\ a \end{aligned}$$

In this definition, the function p ends up not actually being explicitly used. However, it will factor into the decision made about the size of the output vector, along with the other two parameters.

Process Refinement

As noted above, the fixed size nature of vectors forces us to consider process refinements that differ slightly in behaviour from our original functional specifications. We will have to know somehow in advance that the characteristic predicate will return true after a certain number of steps, thus defining the size of the output vector. In effect, the predicate is refined out of the actual implementation altogether, however it does factor into our decision about the size of the output vector.

As regards a CSP specification for this process, we shall see they take a similar form to those supplied for the fold family of functions in vector terms. We again have a sequence of processes composed together in parallel. The flow of data does differ, however. In vector implementations of the fold variants, the individual component processes each took two inputs and produced one

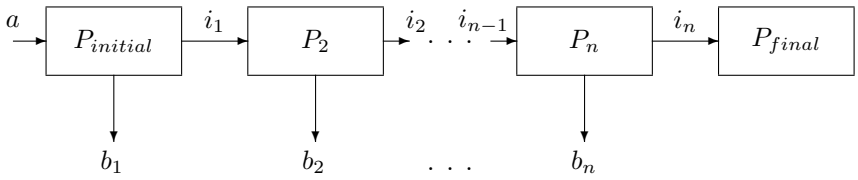


Figure 6.10: The process $VUNFOLDR$.

output. Correspondingly, for the unfold family, each component process takes one input and produces two outputs. The components of a fold can be considered as merging operations, whereas those of an unfold are splitting operations.

Recall that for $unfoldr$, the function f passed as a parameter will have the following type:

$$f :: A \rightarrow (B, A)$$

Thus, a process F which refines this function is required, which should have the following alphabet:

$$\alpha F = \{ina :: \underline{A}, outb :: \underline{B}, outa :: \underline{A}\}$$

This gives us the following alphabet for our process $VUNFOLDR$:

$$\alpha VUNFOLDR_n(F) = \{in :: \underline{A}, out :: \underline{\langle B \rangle_n}\}$$

We can now construct our definition. For the most part, we are connecting each instance of F with the corresponding element of the output vector, as well as the output of the previous instance of F and the input to the next instance of F. This pattern changes slightly at either end, where special cases are supplied.

$$\begin{array}{rlll} VUNFOLDR_n(F) & = & \left(P_{initial} \,||\, \left(\begin{array}{c} n \\ || \\ i=2 \end{array} \right. P_i \left. \right) ||P_{final} \right) \backslash \{mid\} & \\ & & P_{initial} & = & F[in/ina, out_1/outb, mid_1/outa] \\ where & & P_i & = & F[mid_{i-1}/ina, out_i/outb, mid_i/outa] \\ & & P_{final} & = & SINK[mid_n/in] \end{array}$$

Now let us consider the left directed variant, which presents itself in the form of the process $VUNFOLDL$. This is depicted in Figure 6.11.

We can provide a CSP specification for $VUNFOLDL$ analogous to that for $VUNFOLDR$ above. As before, let us recall that for $unfoldl$, the function f passed as a parameter will have the following type:

$$f :: A \rightarrow (A, B)$$

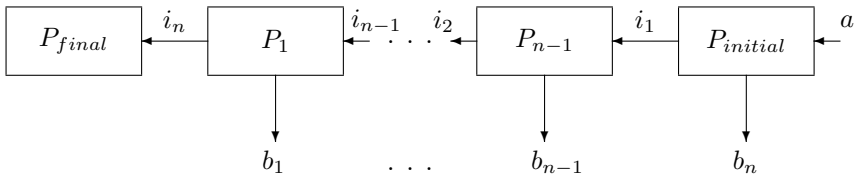


Figure 6.11: The process $VUNFOLDL$.

Thus, a process F which refines this function is required, which should have the following alphabet:

$$\alpha F = \{ina :: \underline{A}, outa :: \underline{A}, outb :: \underline{B}\}$$

This gives us the following alphabet for our process $VUNFOLDL$:

$$\alpha VUNFOLDL_n(F) = \{in :: \underline{A}, out :: \underline{\langle B \rangle_n}\}$$

We therefore have:

$$\begin{array}{rrcl} VUNFOLDL_n(F) & = & \left(P_{final} \,||\, \left(\begin{array}{c} (n-1) \\ || \\ i=1 \end{array} \right. P_i \left. \right) ||P_{initial} \right) \backslash \{mid\} \\ & P_{initial} & = & F[in/ina, out_n/outb, mid_n/outa] \\ where & P_i & = & F[mid_{i+1}/ina, out_i/outb, mid_i/outa] \\ & P_{final} & = & SINK[mid_1/in] \end{array}$$

Handel-C Implementation

Closely following the CSP specification above, we have the implementation in Handel-C for `VUNFOLDR` given in Figure 6.12.

```
macro proc VUNFOLDR (n,in,out,F)
{
   conduittype(in) mid [n];
   typeof(n) i;
   par (i=0;i<=n;i++)
   {
      ifselect (i==0)
         F (in,out[0],mid[0]);
      else ifselect (i<n)
         F (mid[i-1],out[i],mid[i]);
      else
         SINK (mid[n-1]);
   }
}
```

Figure 6.12: The Handel-C definition of the process VUNFOLDR.

We can provide a similar definition for `VUNFOLDL`, as seen in Figure 6.13.

```
macro proc VUNFOLDL (n,in,out,F)
{
   conduittype(in) mid [n];
   typeof(n) i;
   par (i=0;i<=n;i++)
   {
      ifselect (i==0)
         SINK (mid[0]);
      else ifselect (i<n)
         F (mid[i+1],out[i],mid[i]);
      else
         F (in,out[n-1],mid[n-1]);
   }
}
```

Figure 6.13: The Handel-C definition of the process VUNFOLDL.

6.5 Scan

The *scan*, or *accumulate* family of functions share certain characteristics with $fold$. They perform reduction over a list by repeated application of a binary operator, like $fold$. However, whereas a $fold$ operation results in a single value, the *scan* family produces a list of all the intermediate results. As with $fold$, we have both left and right directed variants, with and without base values. Let us first define the types of all four variants:

$$\begin{aligned}
scanl &:: (A \to B \to A) \to A \to [B] \to [A] \\
scanr &:: (A \to B \to B) \to B \to [A] \to [B] \\
scanl1 &:: (A \to A \to A) \to [A] \to [A] \\
scanr1 &:: (A \to A \to A) \to [A] \to [A]
\end{aligned}$$

Their functionality can be described informally, in terms of their corresponding $fold$ operators, as follows:

$$\begin{aligned}
&scanl\ (\oplus)\ e\ [x_1, x_2, ..., x_n] \\
&\quad = [(\oplus \not\to e)\ [],\ (\oplus \not\to e)\ [x_1],\ (\oplus \not\to e)\ [x_1, x_2],\ ...\ ,\ (\oplus \not\to e)\ [x_1, x_2, ..., x_n]] \\
&scanl1\ (\oplus)\ [x_1, x_2, ..., x_n] \\
&\quad = [\oplus \not\to [x_1],\ \oplus \not\to [x_1, x_2],\ ...\ ,\ \oplus \not\to [x_1, x_2, ..., x_n]] \\
&scanr\ (\oplus)\ e\ [x_1, x_2, ..., x_n] \\
&\quad = [(\oplus \not\leftarrow e)\ [x_1, x_2, ..., x_n],\ (\oplus \not\leftarrow e)\ [x_2, x_3, ..., x_n],\ ...\ ,\ (\oplus \not\leftarrow e)\ [x_n],\ (\oplus \not\leftarrow e)\ []] \\
&scanr1\ (\oplus)\ [x_1, x_2, ..., x_n] \\
&\quad = [\oplus \not\leftarrow [x_1, x_2, ..., x_n],\ \oplus \not\leftarrow [x_2, x_3, ..., x_n],\ ...\ ,\ \oplus \not\leftarrow [x_n]]
\end{aligned}$$

As an example, let us consider the act of summing the integers in the range 1 to 5:

$$
\begin{array}{lcl}
scanl\ (+)\ 0\ [1,2,3,4,5] & = & [0,1,3,6,10,15] \\
scanr\ (+)\ 0\ [1,2,3,4,5] & = & [15,14,12,9,5,0] \\
scanl1\ (+)\ [1,2,3,4,5] & = & [1,3,6,10,15] \\
scanr1\ (+)\ [1,2,3,4,5] & = & [15,14,12,9,5]
\end{array}
$$

As with the $fold$ operators, and our other higher order functions, we shall find binary infix versions of these functions useful on occasion. Here we have:

$$
\begin{array}{lcl}
(\oplus /\!\!/\!\!\rightarrow e)xs & = & scanl\ (\oplus)\ e\ xs \\
(\oplus \leftarrow\!\!/\!\!/ e)xs & = & scanr\ (\oplus)\ e\ xs \\
\oplus /\!\!/\!\!\rightarrow xs & = & scanl1\ (\oplus)\ xs \\
\oplus \leftarrow\!\!/\!\!/ xs & = & scanr1\ (\oplus)\ xs
\end{array}
$$

It may be noted that formal definitions for the *scan* functions can be given in terms of *inits* and *tails*, introduced in Section 7.12.

$$
\begin{array}{lcl}
scanl\ (\oplus)\ e\ xs & = & map\ (foldl\ (\oplus)\ e)\ (inits\ xs) \\
scanr\ (\oplus)\ e\ xs & = & map\ (foldr\ (\oplus)\ e)\ (tails\ xs) \\
scanl1\ (\oplus)\ xs & = & map\ (foldl1\ (\oplus))\ (inits^{+}\ xs) \\
scanr1\ (\oplus)\ xs & = & map\ (foldr1\ (\oplus))\ (tails^{+}\ xs)
\end{array}
$$

Or alternatively, we can write the above definitions in the point-free style:

$$
\begin{array}{lcl}
scanl\ (\oplus)\ e & = & map\ (foldl\ (\oplus)\ e)\ \circ\ inits \\
scanr\ (\oplus)\ e & = & map\ (foldr\ (\oplus)\ e)\ \circ\ tails \\
scanl1\ (\oplus) & = & map\ (foldl1\ (\oplus))\ \circ\ inits^{+} \\
scanr1\ (\oplus) & = & map\ (foldr1\ (\oplus))\ \circ\ tails^{+}
\end{array}
$$

In fact, it can be seen that the scan family of functions can also be expressed in terms of unfold operations. We have:

$$
\begin{array}{lcl}
scanl\ (\oplus)\ e & = & ([e]\ +\!\!+)\ \circ\ unfoldl\ (\lambda xs \bullet (init\ xs, foldl\ (\oplus)\ e\ xs))\ null \\
scanr\ (\oplus)\ e & = & (+\!\!+\ [e])\ \circ\ unfoldr\ (\lambda xs \bullet (foldr\ (\oplus)\ e\ xs, tail\ xs))\ null \\
scanl1\ (\oplus) & = & unfoldl\ (\lambda xs \bullet (init\ xs, foldl1\ (\oplus)\ xs))\ null \\
scanr1\ (\oplus) & = & unfoldr\ (\lambda xs \bullet (foldr1\ (\oplus)\ xs, tail\ xs))\ null
\end{array}
$$

Let us now consider how we may derive refinements for these functions in stream and vector terms.

6.5.1 Stream to Stream

Perhaps the most obvious refinement is to both input and output a stream. Here refinements for our family of functions take on the following types:

$$
\begin{array}{lcl}
sscanl & :: & (A \to B \to A) \to A \to \lfloor B \rfloor \to \lfloor A \rfloor \\
sscanr & :: & (A \to B \to B) \to B \to \lfloor A \rfloor \to \lfloor B \rfloor \\
sscanl1 & :: & (A \to A \to A) \to \lfloor A \rfloor \to \lfloor A \rfloor \\
sscanr1 & :: & (A \to A \to A) \to \lfloor A \rfloor \to \lfloor A \rfloor
\end{array}
$$

We can provide definitions in terms of unfold operations, as follows:

$$
\begin{array}{lcl}
sscanl\ (\oplus)\ e & = & (\lfloor e \rfloor \widehat{+\!\!+}) \circ sunfoldl\ (\lambda s \bullet (sinit\ s, sfoldl\ (\oplus)\ e\ s))\ snull \\
sscanr\ (\oplus)\ e & = & (\widehat{+\!\!+} \lfloor e \rfloor) \circ sunfoldr\ (\lambda s \bullet (sfoldr\ (\oplus)\ e\ s, stail\ s))\ snull \\
sscanl1\ (\oplus) & = & sunfoldl\ (\lambda s \bullet (sinit\ s, sfoldl1\ (\oplus)\ s))\ snull \\
sscanr1\ (\oplus) & = & sunfoldr\ (\lambda s \bullet (sfoldr1\ (\oplus)\ s, stail\ s))\ snull
\end{array}
$$

Process Refinement

As has been already shown, the scan family of functions can in fact be implemented in terms of the more general unfold family. This should make their refinement to processes fairly straightforward, as we have already done a certain amount of the legwork..

In this scheme we have a family of scan processes which both input and output streams. the right directed variant, in the form of the process $SSCANR$, is depicted in Figure 6.14.

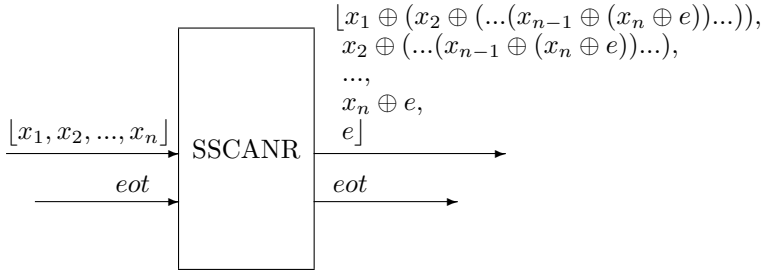


Figure 6.14: The process $SSCANR$.

Similarly, the left directed variant, which manifests itself in the form of the process $SSCANL$, is depicted in Figure 6.15.

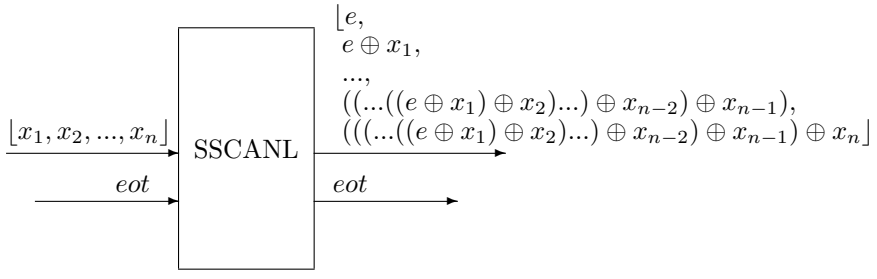


Figure 6.15: The process $SSCANL$.

Recalling the relationships between the scan and unfold family of functions given in Section 6.5.1:

$$\begin{array}{lcl} sscanl\ (\oplus)\ e & = & (\lfloor e \rfloor \widehat{+\!\!+}) \circ sunfoldl\ (\lambda\, s \bullet (sinit\ s, sfoldl\ (\oplus)\ e\ s))\ snull \\ sscanr\ (\oplus)\ e & = & (\widehat{+\!\!+} \lfloor e \rfloor) \circ sunfoldr\ (\lambda\, s \bullet (sfoldr\ (\oplus)\ e\ s, stail\ s))\ snull \\ sscanl1\ (\oplus) & = & sunfoldl\ (\lambda\, s \bullet (sinit\ s, sfoldl1\ (\oplus)\ s))\ snull \\ sscanr1\ (\oplus) & = & sunfoldr\ (\lambda\, s \bullet (sfoldr1\ (\oplus)\ s, stail\ s))\ snull \end{array}$$

We therefore have, for $SSCANR1$:

$$\begin{array}{lcl} SSCANR1\ (\oplus) & = & SUNFOLDR(F, null) \\ F(xs) & = & out\ !\ foldr1\ (\oplus)\ xs; \\ & & xs\ :=\ tail\ xs \end{array}$$

In a similar vein, for $SSCANL1$:

$$\begin{array}{lcl} SSCANL1\ (\oplus) & = & SUNFOLDL(F, null) \\ F(xs) & = & out\ !\ foldl1\ (\oplus)\ xs; \\ & & xs\ :=\ init\ xs \end{array}$$

6.5.2 Stream to Vector

Next we have the scenario where we are inputting a stream but outputting a vector. This gives us the following types:

$$\begin{array}{lcl} svscanl_n & :: & (A \to B \to A) \to A \to \lfloor B \rfloor \to \langle A \rangle_{n+1} \\ svscanr_n & :: & (A \to B \to B) \to B \to \lfloor A \rfloor \to \langle B \rangle_{n+1} \\ svscanl1_n & :: & (A \to A \to A) \to \lfloor A \rfloor \to \langle A \rangle_n \\ svscanr1_n & :: & (A \to A \to A) \to \lfloor A \rfloor \to \langle A \rangle_n \end{array}$$

Here we can supply the following definitions:

$$\begin{array}{lcl} svscanl\ (\oplus)\ e & = & (\langle e \rangle_1 \widetilde{+\!\!+_{1,n}}) \circ vunfoldl_n\ (\lambda\, s \bullet (sinit\ s, sfoldl\ (\oplus)\ e\ s))\ snull \\ svscanr\ (\oplus)\ e & = & (\widetilde{+\!\!+_{n,1}} \langle e \rangle_1) \circ vunfoldr_n\ (\lambda\, s \bullet (sfoldr\ (\oplus)\ e\ s, stail\ s))\ snull \\ svscanl1\ (\oplus) & = & vunfoldl_n\ (\lambda\, s \bullet (sinit\ s, sfoldl1\ (\oplus)\ s))\ snull \\ svscanr1\ (\oplus) & = & vunfoldr_n\ (\lambda\, s \bullet (sfoldr1\ (\oplus)\ s, stail\ s))\ snull \end{array}$$

Process Refinement

This refinement of the scan functions is possibly the most useful of those we shall consider here. The process $SVSCANR$ is depicted in Figure 6.16.

6.5.3 Vector to Stream

Finally let us consider the option of inputting a vector and outputting a stream. We shall encounter the following types:

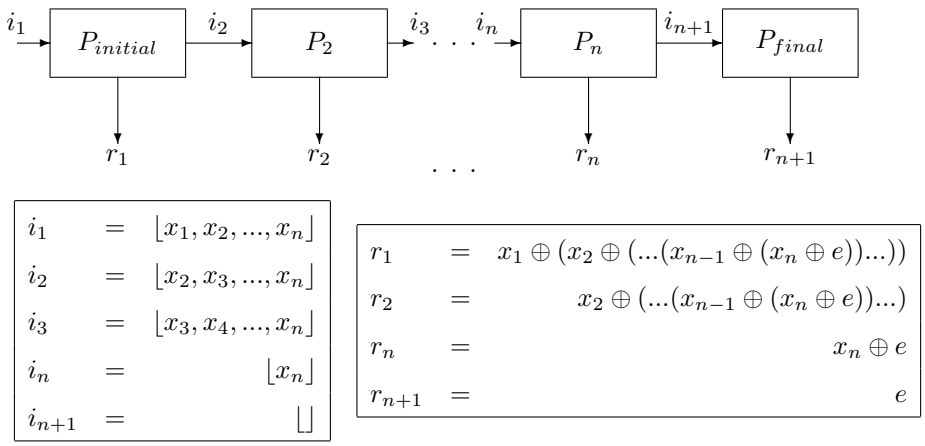


Figure 6.16: The process $SVSCANR$.

$$
\begin{array}{lcl}
vsscanl_n & :: & (A \to B \to A) \to A \to \langle B \rangle_n \to \lfloor A \rfloor \\
vsscanr_n & :: & (A \to B \to B) \to B \to \langle A \rangle_n \to \lfloor B \rfloor \\
vsscanl1_n & :: & (A \to A \to A) \to \langle A \rangle_n \to \lfloor A \rfloor \\
vsscanr1_n & :: & (A \to A \to A) \to \langle A \rangle_n \to \lfloor A \rfloor
\end{array}
$$

We would like to be able to supply definitions along the lines of the following:

$$
\begin{array}{lcl}
vsscanl_n\ (\oplus)\ e & = & (\lfloor e \rfloor \widehat{+\!\!+}) \circ sunfoldl\ (\lambda v \bullet (vinit\ v, vfoldl\ (\oplus)\ e\ v))\ vnull \\
vsscanr_n\ (\oplus)\ e & = & (\widehat{+\!\!+} \lfloor e \rfloor) \circ sunfoldr\ (\lambda v \bullet (vfoldr\ (\oplus)\ e\ v, vtail\ v))\ vnull \\
vsscanl1_n\ (\oplus) & = & sunfoldl\ (\lambda v \bullet (vinit\ v, vfoldl1\ (\oplus)\ v))\ vnull \\
vsscanr1_n\ (\oplus) & = & sunfoldr\ (\lambda v \bullet (vfoldr1\ (\oplus)\ v, vtail\ v))\ vnull
\end{array}
$$

Unfortunately there are problems with supplying these kinds of definitions in this setting. The vector handling functions require subscripts, but there are no subscripts we can provide that will allow them to work in all required cases.

Conceptually though, this ought to be able to work. What is required here is something like a *partial vector*. This could be achieved as simply as pairing a vector with a number, where the number can range from zero up to the size of the vector. This number would designate how many items in the vector are actually to be considered as active, and as such operations on these vectors could be adjusted accordingly.

6.5.4 Vector to Vector

The next scheme we shall consider is one in which we both input and output a vector. Here refinements for our family of functions take on the following types:

$$
\begin{array}{lcl}
vscanl_n & :: & (A \to B \to A) \to A \to \langle B \rangle_n \to \langle A \rangle_{n+1} \\
vscanr_n & :: & (A \to B \to B) \to B \to \langle A \rangle_n \to \langle B \rangle_{n+1} \\
vscanl1_n & :: & (A \to A \to A) \to \langle A \rangle_n \to \langle A \rangle_n \\
vscanr1_n & :: & (A \to A \to A) \to \langle A \rangle_n \to \langle A \rangle_n
\end{array}
$$

As regards definitions of these functions, we shall be faced with the same issues as those encountered for the vector to stream case above.

6.6 Zmap

The function *zmap* is similar in essence to our well known function *map*. However, whereas *map* applies the same function to every item in the list, our function *zmap* instead applies a unique function to each item in the list. Informally we have:

$$zmap_n\ (f_1, f_2, ..., f_n)\ [x_1, x_2, ..., x_n]\ =\ (f_1\ x_1, f_2\ x_2, ..., f_n\ x_n)$$

The name *zmap* here is derived from the similarity between this function and *zipwith*. A similar kind of functionality to the above can be achieved with the following expression:

$$zipwith\ (\lambda f\ x \bullet f\ x)\ [f_1, f_2, ..., f_n]\ [x_1, x_2, ..., x_n]$$

However, given that the functions to be applied to the list are contained in a list themselves, they will have to all be of the same type. For a truly generic definition of *zmap*, we may prefer to allow each of the functions to be of a different type. It is for this reason the functions are contained in a tuple of size n. More formally, we have:

$$zmap_n :: (A \to B_1, A \to B_2, ..., A \to B_n)\ [A]\ =\ (B_1, B_2, ..., B_n)$$

6.6.1 Streams

A stream refinement for *zmap* simply replaces the input list with a stream. Informally, we have:

$$szmap_n\ (f_1, f_2, ..., f_n)\ \lfloor x_1, x_2, ..., x_n \rfloor\ =\ (f_1\ x_1, f_2\ x_2, ..., f_n\ x_n)$$

To prove this is a valid refinement of $zmap_n$, we require that the following diagram commutes:

$$\begin{array}{ccc} [A] & \xrightarrow{zmap_n\ fs} & (B_1, B_2, ..., B_n) \\ \uparrow{\scriptstyle abs_S} & & \uparrow{\scriptstyle id} \\ \lfloor A \rfloor & \xrightarrow{szmap_n\ fs} & (B_1, B_2, ..., B_n) \end{array}$$

This should be straightforward to prove:

$$\begin{array}{lll} & (zmap_n\ (f_1, f_2, ..., f_n) \circ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\ = & zmap_n\ (f_1, f_2, ..., f_n)\ [x_1, x_2, ..., x_n] & \{def.\ abs_S\} \\ = & (f_1\ x_1, f_2\ x_2, ..., f_n\ x_n) & \{def.\ zmap\} \\ = & szmap_n\ (f_1, f_2, ..., f_n)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ szmap\} \\ = & (id \circ szmap_n(f_1, f_2, ..., f_n))\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ id\} \end{array}$$

6.6.2 Vectors

Our function *zmap* is well suited to vector refinement, given the inherently fixed size nature of the operation. For our generic version, we have:

$$vzmap_n\ (f_1, f_2, ..., f_n)\ \langle x_1, x_2, ..., x_n\rangle_n\ =\ (f_1\ x_1, f_2\ x_2, ..., f_n\ x_n)$$

To prove this is a valid refinement of $zmap_n$, we require that the following diagram commutes:

$$\begin{array}{ccc} [A] & \xrightarrow{zmap_n\ fs} & (B_1, B_2, ..., B_n) \\ \uparrow{\scriptstyle abs_V} & & \uparrow{\scriptstyle id} \\ \langle A\rangle_n & \xrightarrow{vzmap_n\ fs} & (B_1, B_2, ..., B_n) \end{array}$$

This should be straightforward to prove:

$$\begin{array}{lll} & (zmap_n\ (f_1, f_2, ..., f_n) \circ abs_V)\ \langle x_1, x_2, ..., x_n\rangle_n & \{id\} \\ = & zmap_n\ (f_1, f_2, ..., f_n)\ [x_1, x_2, ..., x_n] & \{def.\ abs_V\} \\ = & (f_1\ x_1, f_2\ x_2, ..., f_n\ x_n) & \{def.\ zmap\} \\ = & vzmap_n\ (f_1, f_2, ..., f_n)\ \langle x_1, x_2, ..., x_n\rangle_n & \{def.\ vzmap\} \\ = & (id \circ vzmap_n(f_1, f_2, ..., f_n))\ \langle x_1, x_2, ..., x_n\rangle_n & \{def.\ id\} \end{array}$$

6.7 Zfold

The function $zfold$ maintains a similar relationship with $fold$ to that between *zmap* and *map* above. Whereas $fold$ applies the same operator repeatedly to reduce a list of items, the function $zfold$ will apply a different operator at each juncture. Informally, we have:

$$zfold_n\ (\oplus_1, \oplus_2, ..., \oplus_{n-1})\ [x_1, x_2, ..., x_n]\ =\ x_1 \oplus_1 x_2 \oplus_2 ... \oplus_{n-1} x_n$$

As with the $fold$ family, we shall find in practice we usually implement our generic $zfold$ via one of our four variants:

$$\begin{array}{lll} zfoldr & :: & (A \to B_1 \to B_2, A \to B_2 \to B_3, ..., A \to B_n \to B_{n+1}) \\ & & \to B_1 \to [A] \to B_{n+1} \\ zfoldl & :: & (B_1 \to A \to B_2, B_2 \to A \to B_3, ..., B_n \to A \to B_{n+1}) \\ & & \to B_1 \to [A] \to B_{n+1} \\ zfoldr1 & :: & (A \to A \to B_1, A \to B_1 \to B_2, ..., A \to B_{n-2} \to B_{n-1}) \\ & & \to [A] \to B_{n-1} \\ zfoldl1 & :: & (A \to A \to B_1, B_1 \to A \to B_2, ..., B_{n-2} \to A \to B_{n-1}) \\ & & \to [A] \to B_{n-1} \end{array}$$

Their informal definitions are as follows:

$$
\begin{aligned}
&zfoldr\ (\oplus_1, \oplus_2, ..., \oplus_n)\ e\ [x_1, x_2, ..., x_n] \\
&\quad = x_1 \oplus_n (x_2 \oplus_{n-1} (... \oplus_2 (x_n \oplus_1 e))) \\
&zfoldl\ (\oplus_1, \oplus_2, ..., \oplus_n)\ e\ [x_1, x_2, ..., x_n] \\
&\quad = (((e \oplus_1 x_1) \oplus_2 x_2) \oplus_3 ...) \oplus_n x_n \\
&zfoldr1\ (\oplus_1, \oplus_2, ..., \oplus_{n-1})\ [x_1, x_2, ..., x_n] \\
&\quad = x_1 \oplus_{n-1} (x_2 \oplus_{n-2} (... \oplus_2 (x_{n-1} \oplus_1 x_n))) \\
&zfoldl1\ (\oplus_1, \oplus_2, ..., \oplus_{n-1})\ [x_1, x_2, ..., x_n] \\
&\quad = (((x_1 \oplus_1 x_2) \oplus_2 x_3) \oplus_3 ...) \oplus_{n-1} x_n
\end{aligned}
$$

6.7.1 Streams

A stream refinement for $zfold$ simply replaces the input list with a stream. Informally, we have:

$$szfold_n\ (\oplus_1, \oplus_2, ..., \oplus_{n-1})\ \lfloor x_1, x_2, ..., x_n \rfloor\ =\ x_1 \oplus_1 x_2 \oplus_2 ... \oplus_{n-1} x_n$$

To prove this is a valid refinement of $zfold_n$, we require that the following diagram commutes:

$$
\begin{array}{ccc}
[A] & \xrightarrow{zfold_n\ fs} & B \\
\uparrow{\scriptstyle abs_S} & & \uparrow{\scriptstyle id} \\
\lfloor A \rfloor & \xrightarrow{szfold_n\ fs} & B
\end{array}
$$

This should be straightforward to prove:

$$
\begin{array}{lll}
 & (zfold_n\ (\oplus_1, \oplus_2, ..., \oplus_{n-1}) \circ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\
= & zfold_n\ (\oplus_1, \oplus_2, ..., \oplus_{n-1})\ [x_1, x_2, ..., x_n] & \{def.\ abs_S\} \\
= & x_1 \oplus_1 x_2 \oplus_2 ... \oplus_{n-1} x_n & \{def.\ zfold\} \\
= & szfold_n\ (\oplus_1, \oplus_2, ..., \oplus_{n-1})\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ szfold\} \\
= & (id \circ szfold_n(\oplus_1, \oplus_2, ..., \oplus_{n-1}))\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ id\}
\end{array}
$$

Similarly we shall have refinements in stream terms for all our $zfold$ variants:

$$
\begin{aligned}
szfoldr\quad &:: (A \to B_1 \to B_2, A \to B_2 \to B_3, ..., A \to B_n \to B_{n+1}) \\
&\quad \to B_1 \to \lfloor A \rfloor \to B_{n+1} \\
szfoldl\quad &:: (B_1 \to A \to B_2, B_2 \to A \to B_3, ..., B_n \to A \to B_{n+1}) \\
&\quad \to B_1 \to \lfloor A \rfloor \to B_{n+1} \\
szfoldr1\quad &:: (A \to A \to B_1, A \to B_1 \to B_2, ..., A \to B_{n-2} \to B_{n-1}) \\
&\quad \to \lfloor A \rfloor \to B_{n-1} \\
szfoldl1\quad &:: (A \to A \to B_1, B_1 \to A \to B_2, ..., B_{n-2} \to A \to B_{n-1}) \\
&\quad \to \lfloor A \rfloor \to B_{n-1}
\end{aligned}
$$

Their informal definitions are as follows:

$$
\begin{aligned}
&szfoldr\ (\oplus_1, \oplus_2, ..., \oplus_n)\ e\ \lfloor x_1, x_2, ..., x_n \rfloor \\
&\quad =\ x_1 \oplus_n (x_2 \oplus_{n-1} (... \oplus_2 (x_n \oplus_1 e))) \\
&szfoldl\ (\oplus_1, \oplus_2, ..., \oplus_n)\ e\ \lfloor x_1, x_2, ..., x_n \rfloor \\
&\quad =\ (((e \oplus_1 x_1) \oplus_2 x_2) \oplus_3 ...) \oplus_n x_n \\
&szfoldr1\ (\oplus_1, \oplus_2, ..., \oplus_{n-1})\ \lfloor x_1, x_2, ..., x_n \rfloor \\
&\quad =\ x_1 \oplus_{n-1} (x_2 \oplus_{n-2} (... \oplus_2 (x_{n-1} \oplus_1 x_n))) \\
&szfoldl1\ (\oplus_1, \oplus_2, ..., \oplus_{n-1})\ \lfloor x_1, x_2, ..., x_n \rfloor \\
&\quad =\ (((x_1 \oplus_1 x_2) \oplus_2 x_3) \oplus_3 ...) \oplus_{n-1} x_n
\end{aligned}
$$

6.7.2 Vectors

As with *zmap*, the fixed size nature of our family of $zfold$ functions make them well suited to vector refinement. For our generic version, we have:

$$vzfold_n\ (\oplus_1, \oplus_2, ..., \oplus_{n-1})\ \langle x_1, x_2, ..., x_n \rangle_n\ =\ x_1 \oplus_1 x_2 \oplus_2 ... \oplus_{n-1} x_n$$

To prove this is a valid refinement of $zfold_n$, we require that the following diagram commutes:

$$
\begin{array}{ccc}
[A] & \xrightarrow{zfold_n\ fs} & B \\
\uparrow{\scriptstyle abs_V} & & \uparrow{\scriptstyle id} \\
\langle A \rangle_n & \xrightarrow{vzfold_n\ fs} & B
\end{array}
$$

This should be straightforward to prove:

$$
\begin{aligned}
& (zfold_n\ (\oplus_1, \oplus_2, ..., \oplus_{n-1}) \circ abs_V)\ \langle x_1, x_2, ..., x_n \rangle_n && \{id\} \\
=\ & zfold_n\ (\oplus_1, \oplus_2, ..., \oplus_{n-1})\ [x_1, x_2, ..., x_n] && \{def.\ abs_V\} \\
=\ & x_1 \oplus_1 x_2 \oplus_2 ... \oplus_{n-1} x_n && \{def.\ zfold\} \\
=\ & vzfold_n\ (\oplus_1, \oplus_2, ..., \oplus_{n-1})\ \langle x_1, x_2, ..., x_n \rangle_n && \{def.\ vzfold\} \\
=\ & (id \circ vzfold_n(\oplus_1, \oplus_2, ..., \oplus_{n-1}))\ \langle x_1, x_2, ..., x_n \rangle_n && \{def.\ id\}
\end{aligned}
$$

Similarly we shall have refinements in vector terms for all of the $zfold$ variants:

$$
\begin{aligned}
vzfoldr\quad &:: (A \to B_1 \to B_2, A \to B_2 \to B_3, ..., A \to B_n \to B_{n+1}) \\
&\quad \to B_1 \to \langle A \rangle_n \to B_{n+1} \\
vzfoldl\quad &:: (B_1 \to A \to B_2, B_2 \to A \to B_3, ..., B_n \to A \to B_{n+1}) \\
&\quad \to B_1 \to \langle A \rangle_n \to B_{n+1} \\
vzfoldr1\quad &:: (A \to A \to B_1, A \to B_1 \to B_2, ..., A \to B_{n-2} \to B_{n-1}) \\
&\quad \to \langle A \rangle_n \to B_{n-1} \\
vzfoldl1\quad &:: (A \to A \to B_1, B_1 \to A \to B_2, ..., B_{n-2} \to A \to B_{n-1}) \\
&\quad \to \langle A \rangle_n \to B_{n-1}
\end{aligned}
$$

Their informal definitions are as follows:

$$
\begin{aligned}
&vzfoldr\ (\oplus_1, \oplus_2, ..., \oplus_n)\ e\ \langle x_1, x_2, ..., x_n \rangle_n \\
&\quad = x_1 \oplus_n (x_2 \oplus_{n-1} (... \oplus_2 (x_n \oplus_1 e))) \\
&vzfoldl\ (\oplus_1, \oplus_2, ..., \oplus_n)\ e\ \langle x_1, x_2, ..., x_n \rangle_n \\
&\quad = (((e \oplus_1 x_1) \oplus_2 x_2) \oplus_3 ...) \oplus_n x_n \\
&vzfoldr1\ (\oplus_1, \oplus_2, ..., \oplus_{n-1})\ \langle x_1, x_2, ..., x_n \rangle_n \\
&\quad = x_1 \oplus_{n-1} (x_2 \oplus_{n-2} (... \oplus_2 (x_{n-1} \oplus_1 x_n))) \\
&vzfoldl1\ (\oplus_1, \oplus_2, ..., \oplus_{n-1})\ \langle x_1, x_2, ..., x_n \rangle_n \\
&\quad = (((x_1 \oplus_1 x_2) \oplus_2 x_3) \oplus_3 ...) \oplus_{n-1} x_n
\end{aligned}
$$

6.8 Divide and Conquer

Our set of higher order functions need not be limited to those commonly found implemented in the standard libraries of functional compilers and interpreters. The discipline of *Generic Programming* encourages us to consider many commonly occurring algorithmic paradigms as higher order functions.

Perhaps the most intuitive and useful of such paradigms to treat in this manner is *Divide and Conquer*. This paradigm works recursively. At each step, we split the problem up into sub-problems. Each of these is then solved separately using a recursive application of the algorithm, after which the results are combined in some manner. Clearly for any recursive algorithm to be useful in practice, the recursion has to be guarded in some fashion. For this, any instance of the Divide and Conquer algorithm must contain a 'trivial case'. Each input presented to the algorithm is first checked to see if it constitutes an instance of the trivial case. If so, a simple function is applied to it to produce the output rather than it being split, solved and combined as usual.

The Divide and Conquer paradigm is an obvious candidate for parallelisation given that we have sub-problems that can be dealt with independently, and therefore, we presume, in parallel. One treatment of this of particular note is given in [57], where a combination of conventional processors and reconfigurable logic devices are used to implement a Divide and Conquer based sorting algorithm.

This behaviour, which spans a wide range of algorithms for solving many different problems, can be encapsulated in a single higher order function:

$$
\begin{aligned}
&dc\ tc\ isTC\ split\ combine\ input \\
&\quad = \ \textbf{if}\ isTC\ input \\
&\qquad\ \ \textbf{then}\ tc\ input \\
&\qquad\ \ \textbf{else}\ (combine \circ map\ solve \circ split)\ input \\
&\qquad\ \ \textbf{where}\ solve\ =\ dc\ tc\ isTC\ split\ combine
\end{aligned}
$$

As an example, let us consider the following definition of the factorial function:

$$
\begin{aligned}
&factorial\ n\ =\ factorial'\ (1,n)\\
&factorial'\ (n,len)\\
&\quad =\ \ if\ len \leq 1\\
&\qquad\ then\ n\\
&\qquad\ else\ factorial'\ (n,h)\ *\ factorial'(n+h,len-h)\\
&\qquad\ where\ h\ =\ len\ `div`\ 2
\end{aligned}
$$

It is hopefully quite clear that this fits into the Divide and Conquer scheme. We have a trivial case (where *len* is less than or equal to one), the splitting of the input into sub-problems (n, h) and $(n + h, len - h)$, recursive application to solve these problems, and finally combination of the results, via multiplication.

We can therefore create an alternative definition for this factorial function as an instance of the Divide and Conquer higher order function:

$$
\begin{aligned}
&factorial\ n\\
&\quad =\ dc\ fst\ ((\leq 1) \circ snd)\ fsplit\ (fold\ (*))\ (1,n)\\
&fsplit\ (n,len)\ \ =\ \ [(n,h),(n+h,len-h)]\\
&\qquad\qquad\qquad\ where\ h\ =\ div\ len\ 2
\end{aligned}
$$

In order for further investigation of this higher order function to take place, it will be necessary to clearly understand its type. The function *dc* takes four functions and a value as parameters. The first function parameter is *tc*, the trivial case function. This deals with converting a value of the input type (which matches the criteria of the trivial case) into the output type. Next, the parameter $isTC$, a function which takes in a value of the input type and returns a boolean determining whether or not this constitutes an instance of the trivial case. Third we have the parameter *split*, which is required to split a single value of the input type into a list of values of the same type which represent the sub-problems at a particular stage. Fourth is the parameter *combine* which takes in a list of solved sub-problems, which are values of the output type, and returns a single value of the output type. The fifth and final parameter is the input value itself. Thus we have the following type:

$$dc :: (A \to B) \to (A \to Bool) \to (A \to [A]) \to ([B] \to B) \to A \to B$$

Process Refinement

Let us consider how we might implement the divide and conquer scheme in the process environment. The core piece of functionality in the definition of our *dc* pattern is the split / solve / combine part. We have the following expression:

$$(combine \circ map\ solve \circ split)\ input$$

Given a refinement of *split* which can produce the sub-problems as a vector and a refinement of *combine* which can accept the results as a vector, we are then in a position to exploit a vector refinement of *map* to solve our sub-problems. This is depicted in Figure 6.17.

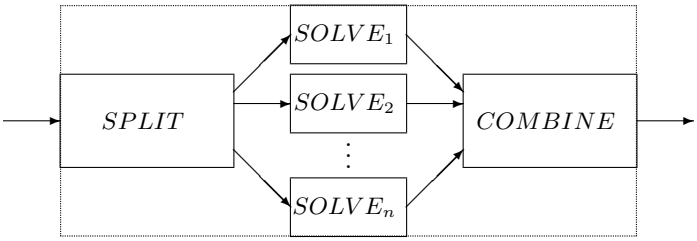


Figure 6.17: The core of the Divide and Conquer algorithm.

So we have the following data refinement of our above expression:

$$(vcombine_n \circ vmap_n\ solve \circ vsplit_n)\ input$$

Given the following types:

$$\begin{array}{lcl} vsplit_n & :: & A \rightarrow \langle A \rangle_n \\ solve & :: & A \rightarrow B \\ vcombine_n & :: & \langle B \rangle_n \rightarrow B \end{array}$$

There is a convenient level of abstraction in the function *solve*. In our definition above for the divide and conquer pattern this is implemented as a recursive application of the function *dc*. However, looking at this in terms of a specification we have instead a function *solve* which should have functionality equivalent to a recursive application of *dc* but need not necessarily be implemented in that exact manner. So, *solve* should be able to return a correct solution to any (sub) problem it may be presented with, but it need not necessarily be implemented recursively.

It is important to bear in mind that our target is a network of processes that will be implemented on an FPGA. As such we are required to construct a network which is fixed and static. At first glance we may determine that this does not suit the dynamic and recursive nature of the divide and conquer paradigm. However, bearing in mind that any instance of *solve* can be replaced with any function which is capable of solving the sub-problem, we are of course at liberty to implement some instances of *solve* sequentially rather than in parallel. In effect this enables us to expand instances of *solve* into parallel (split / solve / combine) patterns, to the extent we are allowed by available resources. Those that cannot be implemented in parallel due to lack of resources are instead implemented sequentially.

Many divide and conquer implementations are actually quite regular and predictable in their behaviour. It is quite common for the *split* function to return a constant number of sub-problems, for example. This is demonstrated in our implementation of $factorial$ above - here we always create exactly two sub-problems every time $fsplit$ is applied. Clearly this is a desirable characteristic if

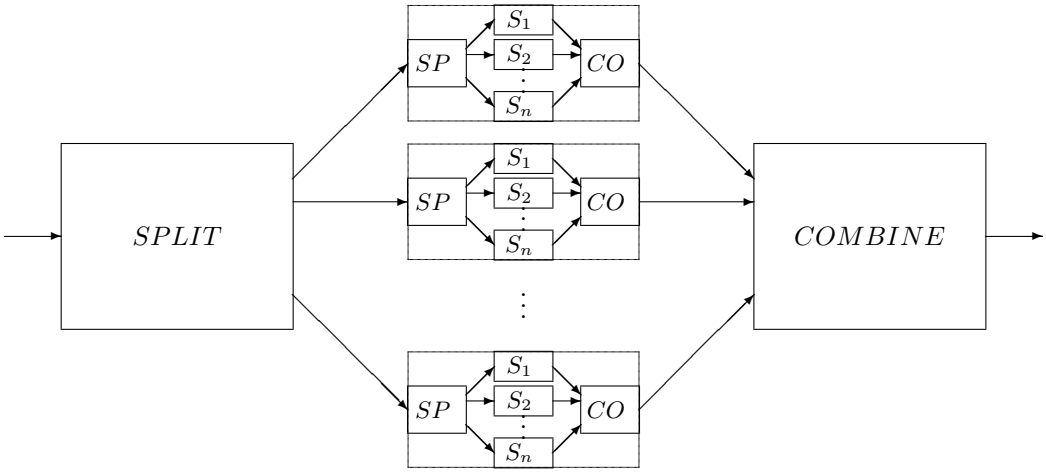


Figure 6.18: The expanded Divide and Conquer process expanded to two levels.

we are to transmit the sub-problems as a vector - at the very least we will require an upper bound. Additionally for some problems we may be able to predict, perhaps based on the size of the input, a minimal level of recursion at which the trivial case can occur. Such constraints will prove very useful in designing an appropriate process network.

To illustrate the expansion of *solve* we have two examples. In the simplest case, we expand just one level, as demonstrated in Figure 6.17. Here we can assume that each component $SOLVE_i$ is implemented sequentially. Taking this one step further, we can expand two levels, as depicted in Figure 6.18. For simplicity's sake, the treatment of the trivial case is not dealt with in these illustrations. There are several possible approaches to this. As already noted, for certain implementations it may be predictable in advance that the trivial case will not occur until the n^{th} level of expansion, in which case we can implement *solve* with just the split/solve/combine pattern. Alternatively, we may wish to add a 'bypass' onto each occurrence of the split/solve/combine pattern which checks the input for the trivial case, and acts accordingly. This is of course not always ideal as it may result in a significant of amount of redundant hardware. For certain problems it may be possible to deal with the trivial case using the split/solve/combine pattern.

To summarise, we have, in general terms, the following network of processes for the split/solve/combine phase of the divide and conquer algorithm.

$$VSPLIT_n \gg_n VMAP_n(SOLVE) \gg_n VCOMBINE$$

An implementation of a divide an conquer algorithm - quick sort - is explored further in Section 9.2.3.

The resulting implementation expresses what we naturally assume to be a dynamic paradigm in a static manner. Indeed, in this work in general we restrict ourselves to considering implementations which can be expressed statically. See the section on future work (Section 10.1) for a further discussion of these issues.

6.9 Summary

In this chapter we have presented process refinements for a number of higher order functions, which between them cover a very wide range of functionality. We have considered implementations for each of them in terms of both the data parallel and sequential schemes (i.e. vectors and streams). In effect we now have a library of 'off the shelf' components, ready to use, which may be employed as the building blocks for our parallel hardware implementations.

Chapter 7

Refinement of List Processing Functions

7.1 Introduction

In addition to the higher order functions already explored, we will often find useful a number of list processing functions usually provided in functional languages. These will be explored in this chapter. To begin with we look at some basic operators such as construct (:), concatenate (++), length and null; these are essential for expressing basic list functionality and we examine here how they may be refined to the stream and vector settings. After this we take a look at list comprehensions, and examine how this convenient form of notation can be decomposed into compositional forms as part of the refinement process. Following on from there we take a look at some simple one dimensional functions used on lists, such as *zip*, *take*, *drop*, *init* and *tail*. We examine how each of these can be refined into the stream and vector setting. Towards the end of the chapter we look at some more sophisticated combinatorial list processing functions, such as *inits* and *tails*, Cartesian product, *transpose*, *segments* and *splits*. Many of these result in quadratic sized output, and we need to think carefully about how they are to be refined if we wish to ensure an efficient resulting implementation. As we shall see in this chapter, a common pattern emerges amongst many of these combinatorial functions, and one particular higher order function - $unfold$ - proves very useful in providing scalable implementations for these components.

7.2 Construct

In list terms, the construct function (:), sometimes pronounced *cons* or *conz*, forms the basic mechanism for building lists. It takes a single item on the left hand side, and a list on the right, and returns a list with that item at the head of the given list. In other words:

$$a : [x_1, x_2, ..., x_n] = [a, x_1, x_2, ..., x_n]$$

The type of this operator is as follows:

$$(:) :: A \to [A] \to [A]$$

We shall find it useful to have a refinement of this operator in both stream and vector terms.

7.2.1 Streams

First, the stream case. Here we have an operator $\widehat{:}$ (pronounced *s-cons*), which has the following type signature:

$$(\widehat{:}) :: A \to \lfloor A \rfloor \to \lfloor A \rfloor$$

Informally, the definition is as above:

$$a \mathbin{\widehat{:}} \lfloor x_1, x_2, ..., x_n \rfloor = \lfloor a, x_1, x_2, ..., x_n \rfloor$$

To prove this is a valid refinement of *cons*, we shall require that the following diagram commutes.

$$\begin{array}{ccc} [A] & \xrightarrow{(a:)} & [A] \\ \uparrow{\scriptstyle abs_S} & & \uparrow{\scriptstyle abs_S} \\ \lfloor A \rfloor & \xrightarrow{(a\widehat{:})} & \lfloor A \rfloor \end{array}$$

The proof here is straightforward:

$$\begin{array}{lll} & ((a :) \circ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\ = & a : [x_1, x_2, ..., x_n] & \{def.\ abs_S\} \\ = & [a, x_1, x_2, ..., x_n] & \{def.\ :\} \\ = & abs_S\ \lfloor a, x_1, x_2, ..., x_n \rfloor & \{def.\ abs_S\} \\ = & abs_S\ (a \mathbin{\widehat{:}} \lfloor x_1, x_2, ..., x_n \rfloor) & \{def.\ \widehat{:}\} \\ = & (abs_S \circ (a\widehat{:}))\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ \circ\} \end{array}$$

7.2.2 Vectors

In the vector case, we follow a similar procedure. Here, our operator $:\!\widetilde{}_n$ (pronounced *v-cons-n*) has the following type:

$$(:\!\widetilde{}_n) :: A \to \langle A \rangle_n \to \langle A \rangle_{n+1}$$

We have the following informal definition:

$$a \mathbin{\widetilde{:}_n} \langle x_1, x_2, ..., x_n \rangle_n \;=\; \langle a, x_1, x_2, ..., x_n \rangle_{n+1}$$

To prove this is a valid refinement of *cons*, we shall require that the following diagram commutes.

$$\begin{array}{ccc} [A] & \xrightarrow{(a :)} & [A] \\ \uparrow{\scriptstyle abs_V} & & \uparrow{\scriptstyle abs_V} \\ \langle A \rangle_n & \xrightarrow{(a \mathbin{\widetilde{:}_n})} & \langle A \rangle_{n+1} \end{array}$$

The proof here is straightforward, and analogous to that for the stream refinement of *cons*:

$$\begin{array}{lll} & ((a :) \circ abs_V)\ \langle x_1, x_2, ..., x_n \rangle_n & \{id\} \\ = & a : [x_1, x_2, ..., x_n] & \{def.\ abs_V\} \\ = & [a, x_1, x_2, ..., x_n] & \{def.\ :\} \\ = & abs_V\ \langle a, x_1, x_2, ..., x_n \rangle_{n+1} & \{def.\ abs_V\} \\ = & abs_V\ (a \mathbin{\widetilde{:}_n} \langle x_1, x_2, ..., x_n \rangle_n) & \{def.\ \widetilde{:}_n\} \\ = & (abs_V \circ (a \mathbin{\widetilde{:}_n}))\ \langle x_1, x_2, ..., x_n \rangle_n & \{def.\ \circ\} \end{array}$$

7.3 Concatenate

The other important basic operator in list terms is the concatenate operator ($+\!\!+$). This joins two lists together as follows:

$$[x_1, x_2, ..., x_n] +\!\!+ [y_1, y_2, ..., y_n] = [x_1, x_2, ..., x_n, y_1, y_2, ..., y_n]$$

It may be useful to note also the following relationships between the concatenate operator and the empty list:

$$\begin{array}{rcl} xs +\!\!+ [] & = & xs \\ [] +\!\!+ ys & = & ys \end{array}$$

The concatenate operator has the following type:

$$(+\!\!+) :: [A] \to [A] \to [A]$$

Let us also consider a generalised version of ($+\!\!+$) which is usually called *concat*. Whereas our operator ($+\!\!+$) concatenates just two lists together, our function *concat* will concatenate an entire list of lists together. This has type:

$$concat :: [[A]] \to [A]$$

A definition is usually given in terms of $fold$, as follows:

$$concat \ = \ fold \ (+\!\!+)$$

As with the construct operator, we shall find stream and vector refinements of these functions useful.

7.3.1 Streams

In stream terms we have an operator $(\widehat{+\!\!+})$, which refines $(+\!\!+)$. This has type:

$$(\widehat{+\!\!+}) :: \lfloor A \rfloor \to \lfloor A \rfloor \to \lfloor A \rfloor$$

It can be informally defined as follows:

$$\lfloor x_1, x_2, ..., x_n \rfloor \ \widehat{+\!\!+} \ \lfloor y_1, y_2, ..., y_n \rfloor = \lfloor x_1, x_2, ..., x_n, y_1, y_2, ..., y_n \rfloor$$

To prove this is a valid refinement of $(+\!\!+)$, we shall require that the following diagram commutes. It may be useful to first recall the definition of $abs2_S$, introduced in Section 3.6.2:

$$abs2_S \ (xs, ys) \ = \ (abs_S \ xs, abs_S \ ys)$$

Now, the diagram:

$$\begin{array}{ccc} ([A],[A]) & \xrightarrow{uncurry(+\!\!+)} & [A] \\ \uparrow{\scriptstyle abs2_S} & & \uparrow{\scriptstyle abs_S} \\ (\lfloor A \rfloor, \lfloor A \rfloor) & \xrightarrow{uncurry(\widehat{+\!\!+})} & \lfloor A \rfloor \end{array}$$

Where the function *uncurry*, useful for dealing with pairs and binary operators, is defined as follows:

$$uncurry \ f \ (a, b) \ = \ f \ a \ b$$

The proof of the validity of this refinement is straightforward:

$$\begin{array}{lll} & (uncurry \ (+\!\!+) \ \circ \ abs2_S) \ (\lfloor x_1, x_2, ..., x_n \rfloor, \lfloor y_1, y_2, ..., y_m \rfloor) & \{id\} \\ = & uncurry \ (+\!\!+) \ ([x_1, x_2, ..., x_n], [y_1, y_2, ..., y_m]) & \{def. \ abs2_S\} \\ = & [x_1, x_2, ..., x_n] +\!\!+ [y_1, y_2, ..., y_m] & \{def. \ uncurry\} \\ = & [x_1, x_2, ..., x_n, y_1, y_2, ..., y_m] & \{def. \ +\!\!+\} \\ = & abs_S \ \lfloor x_1, x_2, ..., x_n, y_1, y_2, ..., y_m \rfloor & \{def. \ abs_S\} \\ = & abs_S \ (\lfloor x_1, x_2, ..., x_n \rfloor \ \widehat{+\!\!+} \ \lfloor y_1, y_2, ..., y_m \rfloor) & \{def. \ \widehat{+\!\!+}\} \\ = & abs_S \ (uncurry \ (\widehat{+\!\!+}) \ (\lfloor x_1, x_2, ..., x_n \rfloor, \lfloor y_1, y_2, ..., y_m \rfloor)) & \{def. \ uncurry\} \\ = & (abs_S \ \circ \ uncurry \ (\widehat{+\!\!+})) \ (\lfloor x_1, x_2, ..., x_n \rfloor, \lfloor y_1, y_2, ..., y_m \rfloor) & \{def. \ \circ\} \end{array}$$

The use of $uncurry$ and $abs2_S$ may slightly obfuscate the purpose of this proof. So, to clarify, we have:

$$\begin{aligned}
& (uncurry\ (+\!\!+)\ \circ\ abs2_S)\ (xs, ys)\ =\ (abs_S\ \circ\ uncurry\ (\widehat{+\!\!+}))(xs, ys) \\
& \{id\} \\
\equiv\ & uncurry\ (+\!\!+)\ (abs_S\ xs, abs_S\ ys)\ =\ (abs_S\ \circ\ uncurry\ (\widehat{+\!\!+}))(xs, ys) \\
& \{def.\ abs2_S\} \\
\equiv\ & abs_S\ xs +\!\!+ abs_S\ ys\ =\ abs_S\ (xs\ \widehat{+\!\!+}\ ys) \\
& \{def.\ uncurry\}
\end{aligned}$$

We should by now have been thoroughly reassured that $(\widehat{+\!\!+})$ is indeed a valid refinement in stream terms for $(+\!\!+)$. Let us consider how we may generalise this. Our concatenate operators ($+\!\!+$ and $\widehat{+\!\!+}$) will concatenate together two lists or two streams respectively. In the case where we wish to concatenate an entire sequence of lists or streams, we can of course make use of a $fold$ (or indeed $sfold$) operation. So, given that we have the proven equivalence in the case of two streams:

$$abs_S\ xs +\!\!+ abs_S\ ys\ =\ abs_S\ (xs\ \widehat{+\!\!+}\ ys)$$

We wish to prove this may be generalised to the following, for a list of streams[1]:

$$concat\ \circ\ map\ abs_S\ =\ abs_S\ \circ\ fold\ (\widehat{+\!\!+})$$

Or, in other words:

$$fold\ (+\!\!+)\ \circ\ map\ abs_S\ =\ abs_S\ \circ\ fold\ (\widehat{+\!\!+})$$

That is to say, we wish to prove the following diagram commutes:

$$\begin{array}{ccc}
[[A]] & \xrightarrow{fold\ (+\!\!+)} & [A] \\
\uparrow{\scriptstyle map\ abs_S} & & \uparrow{\scriptstyle abs_S} \\
[\lfloor A \rfloor] & \xrightarrow{fold\ (\widehat{+\!\!+})} & \lfloor A \rfloor
\end{array}$$

The proof of this is given below:

[1]This is a rather unusual type, however, it is useful conceptually. We are using the list here in a generic sense to represent simply a collection of streams.

$$
\begin{array}{lll}
 & (fold\ (+\!\!+)\ \circ\ map\ abs_S)\ [s_1, s_2, ..., s_n] & \{id\} \\
= & fold\ (+\!\!+)\ [abs_S\ s_1, abs_S\ s_2, ..., abs_S\ s_n] & \{def.\ map\} \\
= & abs_S\ s_1\ +\!\!+\ abs_S\ s_2\ +\!\!+\ ...\ +\!\!+\ abs_S\ s_n & \{def.\ fold\} \\
= & (abs_S\ (s_1\ \widehat{+\!\!+}\ s_2))\ +\!\!+\ ...\ +\!\!+\ abs_S\ s_n & \{ref.\ \widehat{+\!\!+}\} \\
= & (abs_S\ (s_1\ \widehat{+\!\!+}\ s_2\ \widehat{+\!\!+}\ s_3))\ +\!\!+ ...\ +\!\!+\ abs_S\ s_n & \{ref.\ \widehat{+\!\!+}\} \\
 & & \{...\} \\
= & abs_S\ (s_1\ \widehat{+\!\!+}\ s_2\ \widehat{+\!\!+}\ ...\ \widehat{+\!\!+}\ s_n) & \{ref.\ \widehat{+\!\!+}\} \\
= & (abs_S\ \circ\ fold\ (\widehat{+\!\!+}))\ [s_1, s_2, ..., s_n] & \{def.\ fold\}
\end{array}
$$

Let us also consider a similar generalisation in purely stream terms. That is to say we wish to prove that *concat* which concatenates a list of lists is correctly refined by *sconcat* which concatenates a stream of streams. We have:

$$sconcat\ =\ fold\ (\widehat{+\!\!+})$$

For this to be deemed a valid refinement of *concat*, we require the following diagram to commute:

$$
\begin{array}{ccc}
[[A]] & \xrightarrow{\ concat\ } & [A] \\
\uparrow{\scriptstyle abs_{SS}} & & \uparrow{\scriptstyle abs_S} \\
\lfloor\lfloor A\rfloor\rfloor & \xrightarrow{\ sconcat\ } & \lfloor A\rfloor
\end{array}
$$

We shall see that the proof follows similar lines to the above.

$$
\begin{array}{lll}
 & (concat\ \circ\ abs_{SS})\ \lfloor s_1, s_2, ..., s_n\rfloor & \{id\} \\
= & (fold\ (+\!\!+)\ \circ\ abs_{SS})\ \lfloor s_1, s_2, ..., s_n\rfloor & \{def.\ concat\} \\
= & fold\ (+\!\!+)\ [abs_S\ s_1, abs_S\ s_2, ..., abs_S\ s_n] & \{def.\ abs_{SS}\} \\
= & abs_S\ s_1\ +\!\!+\ abs_S\ s_2\ +\!\!+\ ...\ +\!\!+\ abs_S\ s_n & \{def.\ fold\} \\
= & (abs_S\ (s_1\ \widehat{+\!\!+}\ s_2))\ +\!\!+\ ...\ +\!\!+\ abs_S\ s_n & \{ref.\ \widehat{+\!\!+}\} \\
= & (abs_S\ (s_1\ \widehat{+\!\!+}\ s_2\ \widehat{+\!\!+}\ s_3))\ +\!\!+ ...\ +\!\!+\ abs_S\ s_n & \{ref.\ \widehat{+\!\!+}\} \\
 & & \{...\} \\
= & abs_S\ (s_1\ \widehat{+\!\!+}\ s_2\ \widehat{+\!\!+}\ ...\ \widehat{+\!\!+}\ s_n) & \{ref.\ \widehat{+\!\!+}\} \\
= & (abs_S\ \circ\ sfold\ (\widehat{+\!\!+}))\ \lfloor s_1, s_2, ..., s_n\rfloor & \{def.\ sfold\} \\
= & (abs_S\ \circ\ sconcat)\ \lfloor s_1, s_2, ..., s_n\rfloor & \{def.\ sconcat\}
\end{array}
$$

So, we now have the equivalence:

$$fold\ (+\!\!+)\ \circ\ abs_{SS}\ =\ abs_S\ \circ\ sfold\ (\widehat{+\!\!+})$$

7.3.2 Vectors

In vector terms we have an operator $(\widetilde{+\!\!+_{n,m}})$, which refines $(+\!\!+)$. This has type:

$$(\widetilde{+\!\!+_{n,m}}) :: \langle A\rangle_n \to \langle A\rangle_m \to \langle A\rangle_{n+m}$$

It can be informally defined as follows:

$$\langle x_1, x_2, ..., x_n\rangle_n \ \widetilde{+\!\!+_{n,m}} \ \langle y_1, y_2, ..., y_m\rangle_m = \langle x_1, x_2, ..., x_n, y_1, y_2, ..., y_m\rangle_{n+m}$$

The proof that this is a valid refinement will require the following diagram to commute:

$$\begin{array}{ccc} ([A],[A]) & \xrightarrow{uncurry(+\!\!+)} & [A] \\ \uparrow{\scriptstyle abs2_V} & & \uparrow{\scriptstyle abs_S} \\ (\langle A\rangle_n, \langle A\rangle_m) & \xrightarrow{uncurry(\widetilde{+\!\!+_{n,m}})} & \langle A\rangle_{n+m} \end{array}$$

We shall see the proof itself follows along the same lines as in the stream case:

$$\begin{array}{lll} & (uncurry\ (+\!\!+)\ \circ\ abs2_V)\ (\langle x_1, x_2, ..., x_n\rangle_n, \langle y_1, y_2, ..., y_m\rangle_m) & \{id\} \\ = & uncurry\ (+\!\!+)\ ([x_1, x_2, ..., x_n], [y_1, y_2, ..., y_m]) & \{def.\ abs2_V\} \\ = & [x_1, x_2, ..., x_n] +\!\!+ [y_1, y_2, ..., y_m] & \{def.\ uncurry\} \\ = & [x_1, x_2, ..., x_n, y_1, y_2, ..., y_m] & \{def.\ +\!\!+\} \\ = & abs_V\ \langle x_1, x_2, ..., x_n, y_1, y_2, ..., y_m\rangle_{n+m} & \{def.\ abs_S\} \\ = & abs_V\ (\langle x_1, x_2, ..., x_n\rangle_n\ \widetilde{+\!\!+_{n,m}}\ \langle y_1, y_2, ..., y_m\rangle_m) & \{def.\ \widetilde{+\!\!+_{n,m}}\} \\ = & abs_V\ (uncurry\ (\widetilde{+\!\!+_{n,m}})\ (\langle x_1, x_2, ..., x_n\rangle_n, \langle y_1, y_2, ..., y_m\rangle_m) & \{def.\ uncurry\} \\ = & (abs_V\ \circ\ uncurry\ (\widetilde{+\!\!+_{n,m}}))\ (\langle x_1, x_2, ..., x_n\rangle_n, \langle y_1, y_2, ..., y_m\rangle_m) & \{def.\ \circ\} \end{array}$$

The use of *uncurry* and $abs2_V$ may slightly obfuscate the purpose of this proof. Let us clarify the relationship between $+\!\!+$ and $\widetilde{+\!\!+_{n,m}}$ a little. Given two vectors xs and ys, of length n and m respectively, we have:

$$\begin{array}{ll} & (uncurry\ (+\!\!+)\ \circ\ abs2_V)\ (xs, ys)\ =\ (abs_V\ \circ\ uncurry\ (\widetilde{+\!\!+_{n,m}}))(xs, ys) \\ & \{id\} \\ \equiv & uncurry\ (+\!\!+)\ (abs_V\ xs, abs_V\ ys)\ =\ (abs_V\ \circ\ uncurry\ (\widetilde{+\!\!+_{n,m}}))(xs, ys) \\ & \{def.\ abs2_V\} \\ \equiv & abs_V\ xs +\!\!+ abs_V\ ys\ =\ abs_V\ (xs\ \widetilde{+\!\!+_{n,m}}\ ys) \\ & \{def.\ uncurry\} \end{array}$$

The task of constructing a generalised version of this relationship will not be as straightforward as it was in the stream case. This is because the vector concatenate operator has to be 'specialised' to the size of its inputs. As an example, consider the following vector of vectors:

$$\langle\langle a_1, a_2, ..., a_n\rangle_m, \langle b_1, b_2, ..., b_n\rangle_m, \langle c_1, c_2, ..., c_n\rangle_m\rangle_3$$

This would require two applications of our concatenate operator to reduce it to a single vector, which may proceed as follows (assuming evaluation proceeds from left to right - we have the same issue either way):

$$(\langle a_1, a_2, ..., a_n\rangle_m \ \widetilde{+\!\!+}_{m,m} \ \langle b_1, b_2, ..., b_n\rangle_m) \ \widetilde{+\!\!+}_{2m,m} \ \langle c_1, c_2, ..., c_n\rangle_m$$

The problem here is that we are dealing with two different concatenate operators. The left hand operator will take two m sized vectors as operands, whereas the right hand operator takes one of size m, but one of $(2 \times m)$. Given that we are in the vector setting the size of the inputs of any given function form an integral part of its definition. In other words:

$$(n \neq p \vee m \neq q) \Rightarrow \widetilde{+\!\!+}_{n,m} \neq \widetilde{+\!\!+}_{p,q}$$

So, given the obvious definition for vector of vector concatenation:

$$vconcat_{n,m} \ = \ vfold_n \ (\widetilde{+\!\!+}_{a,b})$$

There is no possible value of a and b that will work for any size n greater than two. In effect, we can not use the above definition as is. Let us consider an alternative. First, let us state clearly our requirements. For any definition of *vconcat* to be considered a valid refinement of *concat*, we require that the following diagram commutes:

$$\begin{array}{ccc} [[A]] & \xrightarrow{concat} & [A] \\ \uparrow{\scriptstyle abs_{VV}} & & \uparrow{\scriptstyle abs_V} \\ \langle\langle A\rangle_n\rangle_m & \xrightarrow{vconcat} & \langle A\rangle_{n\times m} \end{array}$$

7.4 Length

The function *length* simply returns the number of items present in a list. It has type:

$$length :: [A] \rightarrow Int$$

Its functionality can be described informally as follows:

$$length \ [x_1, x_2, ..., x_n] \ = \ n$$

More formally, definitions are often given recursively:

$$\begin{array}{lcl} length\ [] & = & 0 \\ length\ (x : xs) & = & 1\ +\ length\ xs \end{array}$$

Alternatively, in terms of $foldl$:

$$length\ =\ foldl\ (\lambda n\ x \bullet n + 1)\ 0$$

7.4.1 Streams

In stream terms we have a function *slength* with the following type:

$$slength :: \lfloor A \rfloor \rightarrow Int$$

We can provide an equivalent informal definition to that for *length*:

$$slength\ \lfloor x_1, x_2, ..., x_n \rfloor\ =\ n$$

Given a stream refinement of $foldl$ in the form of $sfoldl$, we may also supply the following definition:

$$slength\ =\ sfoldl\ (\lambda n\ x \bullet n + 1)\ 0$$

For this to be considered a valid refinement of *length*, we require the following diagram to commute:

$$\begin{array}{ccc} [A] & \xrightarrow{\ length\ } & Int \\ \uparrow{\scriptstyle abs_S} & & \uparrow{\scriptstyle id} \\ \lfloor A \rfloor & \xrightarrow{\ slength\ } & Int \end{array}$$

Given the proof that $sfoldl$ is a valid refinement of $foldl$ presented in Section 5.3.1, this proof should be fairly trivial:

$$\begin{array}{lll} & (length\ \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\ = & (foldl\ (\lambda n\ x \bullet n + 1)\ 0\ \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ length\} \\ = & (id\ \circ\ sfoldl\ (\lambda n\ x \bullet n + 1)\ 0)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{ref.\ sfoldl\} \\ = & (id\ \circ\ slength)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ slength\} \end{array}$$

7.4.2 Vectors

In vector terms we have a function *vlength* with the following type:

$$vlength_n :: \langle A \rangle_n \rightarrow Int$$

We can provide a definition in terms of a fold operation, as before:

$$vlength_n \;=\; vfoldl_n\ (\lambda n\ x \bullet n+1)\ 0$$

We can also provide a definition equivalent to the informal one given for *length*:

$$vlength_n\ \langle x_1, x_2, ..., x_n\rangle_n \;=\; n$$

In fact, given that the input vector is of fixed size, the length will be implicit from the type, so for any vector v we can write simply:

$$vlength_n\ v \;=\; n$$

Proof that this is a valid refinement of *length* requires that the following diagram commutes:

$$\begin{array}{ccc} [A] & \xrightarrow{\quad length \quad} & Int \\ \uparrow{\scriptstyle abs_V} & & \uparrow{\scriptstyle id} \\ \langle A\rangle_n & \xrightarrow{\quad vlength_n \quad} & Int \end{array}$$

We can prove this as follows:

$$\begin{array}{lll} & (length\ \circ\ abs_V)\ \langle x_1, x_2, ..., x_n\rangle_n & \{id\} \\ = & length\ [x_1, x_2, ..., x_n] & \{def.\ abs_V\} \\ = & n & \{def.\ length\} \\ = & vlength_n\ \langle x_1, x_2, ..., x_n\rangle_n & \{def.\ vlength_n\} \\ = & (id\ \circ\ vlength_n)\ \langle x_1, x_2, ..., x_n\rangle_n & \{def.\ id\} \end{array}$$

7.5 Null

The function *null* simply determines whether or not a given list is empty. It has type:

$$null :: [A] \to Bool$$

It can be defined as follows:

$$\begin{array}{lcl} null\ [] & = & True \\ null\ (x : xs) & = & False \end{array}$$

Alternatively we might define it as:

$$null\ xs \;=\; if\ (xs == [])\ then\ True\ else\ False$$

Or indeed, even more succinctly, given that $(xs == [])$ is an expression which will return a boolean, we may write simply:

$$null\ xs\ =\ (xs == [])$$

The most succinct definition of all is point-free, and is achieved simply by creating a section from the equality operator and the empty list:

$$null\ =\ (== [])$$

7.5.1 Streams

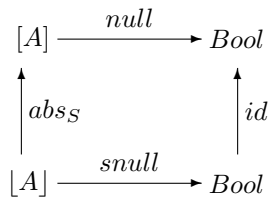
In stream terms we have a function *snull* which allows us to determine whether or not a stream is empty. This has type:

$$snull :: \lfloor A \rfloor \to Bool$$

We shall define it as follows:

$$snull\ s\ =\ (s == \lfloor\rfloor)$$

To prove this is a valid refinement of *null*, we shall require the following diagram to commute:



The proof here should be simple to construct:

$$\begin{array}{lll}
 & (null\ \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\
= & null\ [x_1, x_2, ..., x_n] & \{def.\ abs_S\} \\
= & [x_1, x_2, ..., x_n] == [] & \{def.\ null\} \\
= & [x_1, x_2, ..., x_n] == (abs_S\ \lfloor\rfloor) & \{def.\ abs_S\} \\
= & (abs_S\ \lfloor x_1, x_2, ..., x_n \rfloor) == (abs_S\ \lfloor\rfloor) & \{def.\ abs_S\} \\
= & \lfloor x_1, x_2, ..., x_n \rfloor == \lfloor\rfloor & \{def.\ ==\} \\
= & (== \lfloor\rfloor)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{\} \\
= & (id\ \circ\ (== \lfloor\rfloor))\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\
= & (id\ \circ\ snull)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ snull\}
\end{array}$$

7.5.2 Vectors

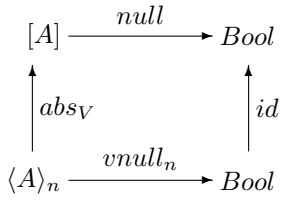
In vector terms we have a function *vnull* which allows us to determine whether or not a vector is empty. This has type:

$$vnull_n :: \langle A \rangle_n \rightarrow Bool$$

The definition shall not follow quite the same lines as that for the stream case. This is because we can not compare any arbitrary vector with the empty vector. That is to say, the type of the empty vector $\langle A \rangle_0$ is not equivalent to the type $\langle A \rangle_n$, where n is any value other than zero. Instead we can express our function as follows:

$$vnull_n\ v\ =\ vlength_n\ v\ ==\ 0$$

To prove this is indeed a valid refinement of *null*, we shall require that the following diagram commutes:



$$\begin{array}{lll}
 & (null\ \circ\ abs_V)\ \langle x_1, x_2, ..., x_n \rangle_n & \{id\} \\
= & null\ [x_1, x_2, ..., x_n] & \{def.\ abs_V\} \\
= & [x_1, x_2, ..., x_n] == [] & \{def.\ null\} \\
= & (length\ [x_1, x_2, ..., x_n]) == (length\ []) & \{see\ below\} \\
= & (length\ [x_1, x_2, ..., x_n]) == 0 & \{length\ []\} \\
= & ((length \circ abs_V)\ \langle x_1, x_2, ..., x_n \rangle_n) == 0 & \{def.\ abs_V\} \\
= & ((id \circ vlength_n)\ \langle x_1, x_2, ..., x_n \rangle_n) == 0 & \{ref.\ vlength_n\} \\
= & ((== 0) \circ id \circ vlength_n)\ \langle x_1, x_2, ..., x_n \rangle_n & \{def.\ \circ\} \\
= & (id \circ (== 0) \circ vlength_n)\ \langle x_1, x_2, ..., x_n \rangle_n & \{def.\ id\} \\
= & (id\ \circ\ vnull_n)\ \langle x_1, x_2, ..., x_n \rangle_n & \{def.\ vnull\}
\end{array}$$

To justify the fourth line of the above proof, we may wish to consider briefly the nature of equality in terms of lists. We wish to prove that given two lists where one is known to be empty, the act of comparing them for equality is equivalent comparing their lengths. A short proof should reassure us of this.

$$\begin{array}{lll}
 & (xs == ys) \wedge (ys == []) & \{id\} \\
= & (xs == []) \wedge (ys == []) & \{\} \\
= & (length\ xs == 0) \wedge (length\ ys == 0) & \{def.\ length\} \\
= & (length\ xs)\ ==\ (length\ ys) & \{\}
\end{array}$$

7.6 List Comprehensions

List comprehensions often provide a very intuitive and succinct mechanism for specifying parts of certain algorithms. We may therefore occasionally encounter such constructs, and will find it useful to have a few simple transformation rules to help deal with them. The majority of the rules presented here are well known properties of list comprehensions, and can be found in [20].

The most basic form of list comprehension is as follows:

$$[x \mid x \leftarrow [x_1, x_2, ..., x_n]]$$

This is in effect equivalent to the identity function on lists:

$$[x \mid x \leftarrow xs] \quad = \quad id\ xs \quad \{LC1\}$$

The next most basic form of list comprehension simply applies some function to a list, for example:

$$[x \times 2 \mid x \leftarrow [x_1, x_2, ..., x_n]]$$

This is clearly equivalent to *map*, and so we have our first rule:

$$[f\ x \mid x \leftarrow xs] \quad = \quad map\ f\ xs \quad \{LC2\}$$

List comprehensions can also include a predicate:

$$[x \mid x \leftarrow [x_1, x_2, ..., x_n], x \leq 10]$$

This is equivalent to a $filter$, giving us our second rule:

$$[x \mid x \leftarrow xs, p\ x] \quad = \quad filter\ p\ xs \quad \{LC3\}$$

These can of course be used in conjunction, for example:

$$[f\ x \mid x \leftarrow [x_1, x_2, ..., x_n], p\ x]$$

This corresponds to a composition of *map* and $filter$, as follows:

$$[f\ x \mid x \leftarrow xs, p\ x] \quad = \quad (map\ f\ \circ\ filter\ p)\ xs \quad \{LC4\}$$

List comprehensions may include more than one generator. Take for example:

$$[(x, y) \mid x \leftarrow [x_1, x_2, ..., x_n],\ y \leftarrow [y_1, y_2, ..., y_m]]$$

The results here will be of length $(n \times m)$. The first m values will be x_1 paired with every item in ys, The next m will be x_2 paired with every item in ys and so on. We can consider this result

instead as the concatenation of n sub-lists, each sub-list comprising an item from xs paired with every item from ys. This gives us a mechanism to decompose our list comprehension:

$$[f\ x\ y \mid x \leftarrow xs,\ y \leftarrow ys] \quad = \quad fold\ (+\!\!+)\ [[f\ x\ y \mid y \leftarrow ys] \mid x \leftarrow xs] \quad \{LC4\}$$

Where any predicates are present, these move with their associated generators:

$$\begin{aligned} &[f\ x\ y \mid x \leftarrow xs,\ y \leftarrow ys, p\ x, q\ y] \\ &\quad = \ fold\ (+\!\!+)\ [[f\ x\ y \mid y \leftarrow ys, q\ y] \mid x \leftarrow xs, p\ x]\ \{LC5\} \end{aligned}$$

We can also generalise this rule to apply to a list comprehension with any number of generators:

$$\begin{aligned} &[f\ a\ b\ c\ ... \mid a \leftarrow as,\ b \leftarrow bs,\ c \leftarrow cs, ...\ ] \\ &\quad = \ fold\ (+\!\!+)\ [[f\ a\ b\ c\ ... \mid b \leftarrow bs, c \leftarrow cs, ...\ ] \mid a \leftarrow as]\ \{LC6\} \end{aligned}$$

Note carefully how in $\{LC4\}$ above we are required to re-order the generators, in order to preserve the order of values in the output, when refactoring in this manner. Were the generators not re-ordered in this manner we would effectively be transposing the list. See Section 7.15 for a discussion of the function *transpose*. This gives rise to the following rule:

$$[f\ x\ y \mid x \leftarrow xs,\ y \leftarrow ys] \quad = \quad (fold\ (+\!\!+) \circ transpose)\ [[f\ x\ y \mid x \leftarrow xs] \mid y \leftarrow ys] \quad \{LC7\}$$

Given these rules, we can transform a fairly complex list comprehension into a collection of *map*, $filter$ and $fold$ functions. Let us give an example.

$$\begin{aligned} &[f\ x\ y \mid x \leftarrow xs,\ y \leftarrow ys, p\ x, q\ y] && \{\} \\ &= fold\ (+\!\!+)\ [[f\ x\ y \mid y \leftarrow ys, q\ y] \mid x \leftarrow xs, p\ x] && \{LC5\} \\ &= fold\ (+\!\!+)\ [(map\ (f\ x) \circ filter\ q)\ ys \mid x \leftarrow xs, p\ x] && \{LC4\} \\ &= fold\ (+\!\!+)\ (map\ (\lambda x \bullet (map\ (f\ x) \circ filter\ q)\ ys) \circ filter\ p)\ xs && \{LC4\} \\ &= fold\ (+\!\!+)\ (map\ (\lambda x \bullet map\ (f\ x)\ (filter\ q\ ys)) \circ filter\ p)\ xs && \{def.\ \circ\} \\ &= fold\ (+\!\!+)\ (map\ (\lambda x \bullet (f\ x)\ `map`\ (filter\ q\ ys)) \circ filter\ p)\ xs && \{def.\ `map`\} \\ &= fold\ (+\!\!+)\ (map\ (\lambda x \bullet (f\ x)\ *\ (filter\ q\ ys)) \circ filter\ p)\ xs && \{def.\ *\} \\ &= fold\ (+\!\!+)\ (map\ (\lambda x \bullet (*\ (filter\ q\ ys))\ (f\ x)) \circ filter\ p)\ xs && \{*\ section\} \\ &= fold\ (+\!\!+)\ (map\ (\lambda x \bullet ((*\ (filter\ q\ ys)) \circ f)\ x) \circ filter\ p)\ xs && \{def.\ \circ\} \\ &= fold\ (+\!\!+)\ (map\ ((*\ (filter\ q\ ys)) \circ f) \circ filter\ p)\ xs && \{def.\ \lambda\} \end{aligned}$$

7.7 Zip

The function *zip* is, in a sense, similar to the concatenate operator in that it provides us a mechanism for combining together the values of two lists. However, whereas the concatenate operator appends one list onto the end of another, *zip* takes each item from one list, and pairs it with the item at the same index in the other list. Additionally the two lists passed to *zip* do not necessarily have to be of the same type. Informally, we can describe this functionality as follows:

$$zip\ [x_1, x_2, ..., x_n]\ [y_1, y_2, ..., y_m]\ =\ [(x_1, y_1), (x_2, y_2), ..., (x_{n\downarrow m}, y_{n\downarrow m})]$$

Here the binary minimum ($\downarrow$) operator, pronounced *min* is used to determine the shortest of the two lengths n and m. The *zip* function has the following type:

$$zip\ ::\ [A] \rightarrow [B] \rightarrow [(A, B)]$$

A definition for *zip* is usually given recursively. For example:

$$\begin{array}{lcl} zip\ as\ [] & = & [] \\ zip\ []\ bs & = & [] \\ zip\ (a : as)\ (b : bs) & = & (a, b)\ : zip\ as\ bs \end{array}$$

Alternatively we can supply a single statement definition, along the lines of the following:

$$\begin{array}{lcl} zip\ as\ bs & = & if\ (null\ as \vee null\ bs) \\ & & then\ [] \\ & & else\ (head\ as, head\ bs)\ : zip\ (tail\ as)\ (tail\ bs) \end{array}$$

However, it is perhaps a less well known fact that *zip* can be defined in terms of an $unfold$ operation. This should become obvious after brief comparison between the above definition of *zip* and that for $unfoldr$. Thus, we have:

$$\begin{array}{lcl} zip\ xs\ ys & = & unfoldr\ f\ p\ (xs, ys) \\ & & where\quad f\ (a : as, b : bs) = ((a, b), (as, bs)) \\ & & \qquad\quad p\ (a, b) = null\ a \vee null\ b \end{array}$$

Or alternatively, without any pattern matching we have:

$$\begin{array}{lcl} zip\ xs\ ys & = & unfoldr\ f\ p\ (xs, ys) \\ & & where\quad f\ (a, b) = ((head\ a, head\ b), (tail\ a, tail\ b)) \\ & & \qquad\quad p\ (a, b) = null\ a \vee null\ b \end{array}$$

7.7.1 Streams

Considering a stream refinement of *zip*, we have a function *szip*, of the following type:

$$szip\ ::\ \lfloor A \rfloor \rightarrow \lfloor B \rfloor \rightarrow \lfloor (A, B) \rfloor$$

Informally, its functionality mimics that of *zip*:

$$szip\ \lfloor x_1, x_2, ..., x_n \rfloor\ \lfloor y_1, y_2, ..., y_m \rfloor\ =\ \lfloor (x_1, y_1), (x_2, y_2), ..., (x_{n\downarrow m}, y_{n\downarrow m}) \rfloor$$

To prove that this is a valid refinement of *zip*, we will require a diagram, similar to that used for concatenate, to commute:

$$\begin{array}{ccc} ([A],[B]) & \xrightarrow{uncurry\ zip} & [(A,B)] \\ \uparrow{\scriptstyle abs2_S} & & \uparrow{\scriptstyle abs_S} \\ (\lfloor A\rfloor,\lfloor B\rfloor) & \xrightarrow{uncurry\ szip} & \lfloor (A,B)\rfloor \end{array}$$

Proof that this is a valid refinement may then proceed as follows:

$$\begin{array}{lll} & (uncurry\ zip \circ abs2_S)\ (\lfloor x_1,x_2,...,x_n\rfloor,\lfloor y_1,y_2,...,y_m\rfloor) & \{id\} \\ = & uncurry\ zip\ ([x_1,x_2,...,x_n],[y_1,y_2,...,y_m]) & \{def.\ abs2_S\} \\ = & zip\ [x_1,x_2,...,x_n]\ [y_1,y_2,...,y_m] & \{def.\ uncurry\} \\ = & [(x_1,y_1),(x_2,y_2),...,(x_{n\downarrow m},y_{n\downarrow m})] & \{def.\ zip\} \\ = & abs_S\ \lfloor (x_1,y_1),(x_2,y_2),...,(x_{n\downarrow m},y_{n\downarrow m})\rfloor & \{def.\ abs_S\} \\ = & abs_S\ (szip\ \lfloor x_1,x_2,...,x_n\rfloor\ \lfloor y_1,y_2,...,y_m\rfloor) & \{def.\ szip\} \\ = & abs_S\ (uncurry\ szip\ (\lfloor x_1,x_2,...,x_n\rfloor,\lfloor y_1,y_2,...,y_m\rfloor)) & \{def.\ uncurry\} \\ = & (abs_S \circ uncurry\ szip)\ (\lfloor x_1,x_2,...,x_n\rfloor,\lfloor y_1,y_2,...,y_m\rfloor) & \{def.\ \circ\} \end{array}$$

Process Refinement

In process terms, we have a process which inputs two streams and outputs one stream containing items from the two input streams paired together. This is illustrated in Figure 7.1.

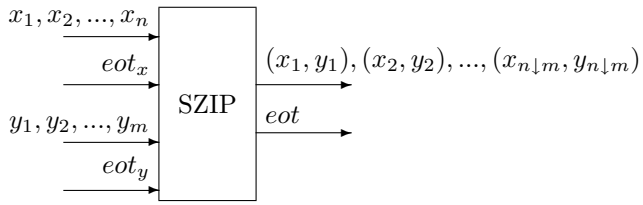


Figure 7.1: The zip process for streams.

Let us consider a CSP definition of this process. In the functional definition of *zip*, we can see that computation is terminated when either of the inputs lists is empty. This should translate fairly easily to the world of processes, as it will correspond to checking the *eot* channel. When an item remains in *both* input lists we can consume them and output a corresponding pair. In process terms, a slight complication may arise from this given that the two input streams operate independently of each other, and may be of differing lengths. Consider the definition given in Figure 7.2.

This definition is subject to an obvious flaw wherever the stream arriving on *yin* is shorter than the stream arriving on *xin*. After consuming a value from *xin* we are stuck in a one way path - and if the stream from *yin* has finished we will never receive a value from *yin*. In strict CSP terms, none of the options in this choice have precedence. So, consider the situation where *yin.eot*

$$
\begin{aligned}
SZIP \quad = \quad & xin.eot \ ? \ any \rightarrow out.eot \ ! \ True \rightarrow SKIP \\
& | \\
& yin.eot \ ? \ any \rightarrow out.eot \ ! \ True \rightarrow SKIP \\
& | \\
& xin.value \ ? \ x \rightarrow yin.value \ ? \ y \rightarrow out.value \ ! \ (x, y) \rightarrow SZIP
\end{aligned}
$$

Figure 7.2: A simple CSP definition for the process SZIP.

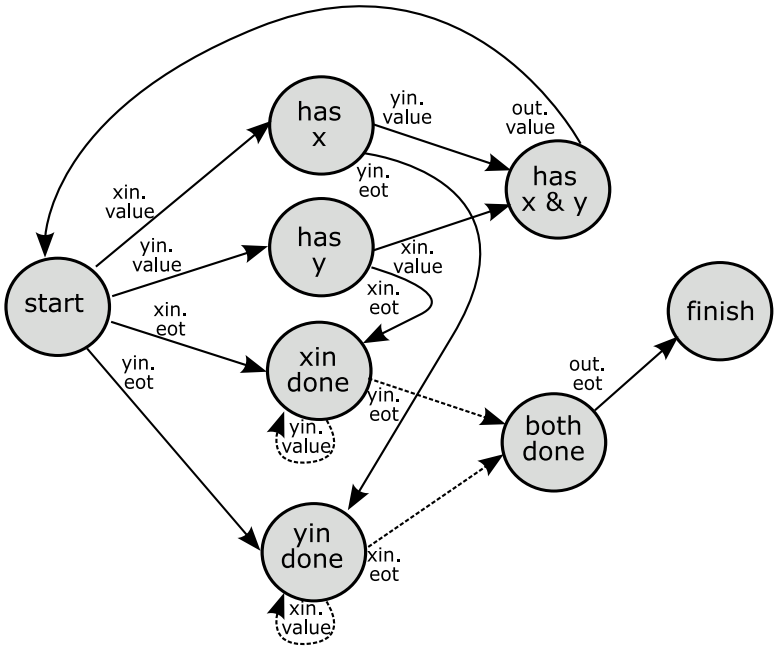


Figure 7.3: A finite state machine for the process SZIP.

and *xin.value* are willing to communicate at exactly the same point in time - this would signify that the stream arriving on *yin* has finished, but there are more values available in the stream arriving on *xin*. Given the semantics of CSP's choice operator, the decision of which branch to follow is arbitrary. In a real world implementation such as Handel-C, however, priority in a `prialt` construct is implicit based on the ordering of the alternatives. We could prioritise the *eot* channels over the *value* channels, so that if both are willing to communicate at the same time we always go along the *eot* branch.

There is however another potential problem. The processes providing the input streams may not necessarily signal EOT immediately after sending the last item in the stream. In other words there could be a period of time in which EOT is not signalled for a given stream, but no further items are going to be transmitted in that stream. We should not therefore assume simply because EOT is *not* signalled that we will definitely be able to consume another value from that stream.

One final issue we may wish to concern ourselves with is whether or not we are required to

$$
\begin{array}{lcl}
SZIP & = & xin.value\ ?\ x \rightarrow \left(\begin{array}{l} yin.value\ ?\ y \rightarrow out.value\ !\ (x,y) \rightarrow SZIP \\ | \\ yin.eot\ ?\ any \rightarrow DONEY \end{array} \right) \\
 & & | \\
 & & yin.value\ ?\ y \rightarrow \left(\begin{array}{l} xin.value\ ?\ x \rightarrow out.value\ !\ (x,y) \rightarrow SZIP \\ | \\ xin.eot\ ?\ any \rightarrow DONEX \end{array} \right) \\
 & & | \\
 & & xin.eot\ ?any\ \rightarrow DONEX \\
 & & | \\
 & & yin.eot\ ?any\ \rightarrow DONEY \\
DONEX & = & yin.value\ ?\ y \rightarrow DONEX \\
 & & | \\
 & & yin.eot\ ?\ any \rightarrow out.eot\ !\ True \rightarrow SKIP \\
DONEY & = & xin.value\ ?\ x \rightarrow DONEY \\
 & & | \\
 & & xin.eot\ ?\ any \rightarrow out.eot\ !\ True \rightarrow SKIP
\end{array}
$$

Figure 7.4: A more complex CSP definition for the process SZIP.

consume the unused portion of the longer stream, where the stream lengths are not equal. On the one hand, it may be important to us that our process network as a whole 'completes' - that at the end of computation no process is still running. If we do not consume the unused stream, we will leave the producing process hanging in a live state. On the other hand, we may envisage situations where one input stream is effectively infinite - and as such it obviously does not make sense to attempt to consume it entirely.

So, we shall require a slightly more sophisticated process to deal with all these possible eventualities. A finite state machine describing all the possible interactions is given in Figure 7.3. Note that in the finite state machine some of the transitions are depicted using dashed lines. This is to signify pieces of functionality which are optional, corresponding to the issue of whether or not to consume the unused portion of the longer stream. This finite state machine can be expressed in CSP, as illustrated in the definition in Figure 7.4.

7.7.2 Vectors

Considering a vector refinement of *zip*, we have a function *vzip*, of the following type:

$$vzip_{n,m} \ :: \ \langle A \rangle_n \rightarrow \langle B \rangle_m \rightarrow \langle (A, B) \rangle_{n \downarrow m}$$

Again, we can provide an informal definition mimicking that of *zip*:

$$vzip_{n,m} \ \langle x_1, x_2, ..., x_n \rangle_n \ \langle y_1, y_2, ..., y_m \rangle_m \ = \ \langle (x_1, y_1), (x_2, y_2), ..., (x_{n \downarrow m}, y_{n \downarrow m}) \rangle_{n \downarrow m}$$

As usual, a proof of the validity of this refinement would be analogous to that for the stream refinement of *zip*.

Process Refinement

In process terms, we have an array of processes, each of which is responsible for taking an item from each vector at the same index and pairing them together. In simple terms we have the following definition:

$$VZIP_{n,m} \ = \ \underset{i=1}{\overset{n \downarrow m}{||}} \ xin_i \ ? \ x \rightarrow yin_i \ ? \ y \rightarrow out_i \ ! \ (x, y) \rightarrow SKIP$$

In more general terms (where the elements of the vectors are not simple items) we may require some more sophisticated behaviour. This will be looked at in the refinement for *zipwith* - see Section 7.8.

7.8 ZipWith

The function *zipwith* can be considered as a generalised version of *zip*. Whereas *zip* combines two values by simply pairing them together, *zipwith* combines them using a user-supplied function which is passed as a parameter. Indeed, given the pairing function, often written $(,)$:

$$(,) \ a \ b = \ (a, b)$$

We can define *zip* as a special case of *zipwith*.

$$zip \ = \ zipwith \ (,)$$

In fact, we can also define *zipwith* in terms of *zip*, using *map* to post apply the function f:

$$zipwith \ f \ as \ bs = \ map \ (uncurry \ f) \ (zip \ as \ bs)$$

In general, the functionality of *zipwith* can be described informally as follows:

$$zipwith\ f\ [x_1, x_2, ..., x_n]\ [y_1, y_2, ..., y_m]\ =\ [f\ x_1\ y_1, f\ x_2\ y_2, ..., f\ x_{n\downarrow m}\ y_{n\downarrow m}]$$

It has the following type:

$$zipwith :: (A \to B \to C) \to [A] \to [B] \to [C]$$

A formal definition is usually given recursively, like so:

$$\begin{aligned} zipwith\ f\ (a : as)\ (b : bs) &= f\ a\ b : zipwith\ f\ as\ bs \\ zipwith\ f\ as\ bs &= [] \end{aligned}$$

We can eliminate some of the pattern matching here and arrive at a single definition:

$$\begin{aligned} zipwith\ f\ as\ bs &= if\ (null\ as \vee null\ bs) \\ &\quad then\ [] \\ &\quad else\ f\ (head\ as)\ (head\ bs)\ : \\ &\qquad zipwith\ f\ (tail\ as)\ (tail\ bs) \end{aligned}$$

We may find it helpful to introduce a couple of local definitions:

$$\begin{aligned} zipwith\ f\ as\ bs &= if\ (null\ as \vee null\ bs) \\ &\quad then\ [] \\ &\quad else\ this : next \\ &\quad where\ this = f\ (head\ as)\ (head\ bs) \\ &\qquad next\ =\ zipwith\ f\ (tail\ as)\ (tail\ bs) \end{aligned}$$

As with *zip*, we may also give a definition for *zipwith* in terms of an $unfold$ operation:

$$\begin{aligned} zipwith\ f\ xs\ ys &= unfoldr\ g\ p\ (xs, ys) \\ &\quad where\ g\ (a, b) = (f\ (head\ a)\ (head\ b), (tail\ a, tail\ b)) \\ &\qquad p\ (a, b) = null\ a \vee null\ b \end{aligned}$$

7.8.1 Streams

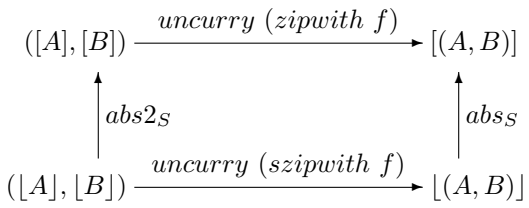
In stream terms we have the function *szipwith*, with the following type:

$$szipwith :: (A \to B \to C) \to \lfloor A \rfloor \to \lfloor B \rfloor \to \lfloor C \rfloor$$

Informally, we have:

$$\begin{aligned} &szipwith\ f\ \lfloor x_1, x_2, ..., x_n \rfloor\ \lfloor y_1, y_2, ..., y_m \rfloor \\ &= \lfloor f\ x_1\ y_1, f\ x_2\ y_2, ..., f\ x_{n\downarrow m}\ y_{n\downarrow m} \rfloor \end{aligned}$$

To prove that this is a valid refinement of *zipwith* f, we will require a diagram, similar to that used for concatenate, to commute:



Proof that this is a valid refinement may then proceed as follows:

$$
\begin{array}{lll}
 & (uncurry\ (zipwith\ f)\ \circ\ abs2_S)\ (\lfloor x_1, x_2, ..., x_n \rfloor, \lfloor y_1, y_2, ..., y_m \rfloor) & \{id\} \\
= & uncurry\ (zipwith\ f)\ ([x_1, x_2, ..., x_n], [y_1, y_2, ..., y_m]) & \{def.\ abs2_S\} \\
= & zipwith\ f\ [x_1, x_2, ..., x_n]\ [y_1, y_2, ..., y_m] & \{def.\ uncurry\} \\
= & [(x_1, y_1), (x_2, y_2), ..., (x_{n\downarrow m}, y_{n\downarrow m})] & \{def.\ zipwith\} \\
= & abs_S\ \lfloor (x_1, y_1), (x_2, y_2), ..., (x_{n\downarrow m}, y_{n\downarrow m}) \rfloor & \{def.\ abs_S\} \\
= & abs_S\ (szipwith\ f\ \lfloor x_1, x_2, ..., x_n \rfloor\ \lfloor y_1, y_2, ..., y_m \rfloor) & \{def.\ szipwith\} \\
= & abs_S\ (uncurry\ (szipwith\ f)\ (\lfloor x_1, x_2, ..., x_n \rfloor, \lfloor y_1, y_2, ..., y_m \rfloor)) & \{def.\ uncurry\} \\
= & (abs_S\ \circ\ uncurry\ (szipwith\ f))\ (\lfloor x_1, x_2, ..., x_n \rfloor, \lfloor y_1, y_2, ..., y_m \rfloor) & \{def.\ \circ\}
\end{array}
$$

Process Refinement

Our stream refinement to a process would follow that for $SZIP$ in Section 7.7.

7.8.2 Vectors

Considering a vector refinement of *zipwith*, we have a function *vzipwith*, of the following type:

$$vzipwith_{n,m}\ ::\ (A \to B \to C) \to \langle A \rangle_n \to \langle B \rangle_m \to \langle C \rangle_{n\downarrow m}$$

Informally, we have:

$$
\begin{array}{r}
vzipwith\ f\ \langle x_1, x_2, ..., x_n \rangle_n\ \langle y_1, y_2, ..., y_m \rangle_m \\
=\ \langle f\ x_1\ y_1, f\ x_2\ y_2, ..., f\ x_{n\downarrow m}\ y_{n\downarrow m} \rangle_{n\downarrow m}
\end{array}
$$

A proof here would follow the same pattern as that for *szipwith* above.

Process Refinement

Taking our definition for $VZIP$, we could provide a simple case definition for $VZIPWITH$ as follows. Here our function f is kept as a function, and we assume the elements of the input vectors are simple items.

$$
VZIPWITH_{n,m}(f) \quad = \quad \underset{i=1}{\overset{n\downarrow m}{||}} \quad xin_i\ ?\ x \to yin_i\ ?\ y \to out_i\ !\ (f\ x\ y) \to SKIP
$$

In more general terms, as with other higher order functions we have seen in the past, it may be necessary to devolve some of the communication into the refinement of f, that is to say:

$$VZIPWITH_{n,m}(F) \quad = \quad \begin{array}{c} n \downarrow m \\ || \\ i = 1 \end{array} \quad F[xin_i/xin, yin_i/yin, out_i/out]$$

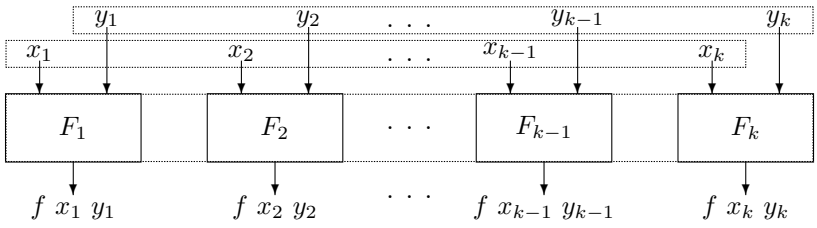
This would, of course, allow us to refine a simple function f that operated on items into the following process:

$$F \; = \; xin \; ? \; x \rightarrow yin \; ? \; y \rightarrow out \; ! \; (f \; x \; y) \rightarrow SKIP$$

As an example, to implement the functionality of *zip* we could provide the following process F:

$$F \; = \; xin \; ? \; x \rightarrow yin \; ? \; y \rightarrow out \; ! \; (x, y) \rightarrow SKIP$$

The process $VZIPWITH$ is illustrated in Figure 7.5.



Where $k \; = \; n \downarrow m$ (the length of the shorter vector).

Figure 7.5: The process $VZIPWITH$.

7.9 Head and Last

The functions *head* and *last* retrieve the first and final elements of a list respectively. Informally speaking, we have:

$$\begin{array}{lcl} head \; [x_1, x_2, ..., x_n] & = & x_1 \\ last \; [x_1, x_2, ..., x_n] & = & x_n \end{array}$$

Both have the same type, taking in a list of items, and returning a single value:

$$\begin{array}{lcl} head & :: & [A] \rightarrow A \\ tail & :: & [A] \rightarrow A \end{array}$$

Formal definitions for these functions usually rely heavily on pattern matching, for example:

$$\begin{aligned} head\ (x : xs) &= x \\ last\ [x] &= x \\ last\ (x : xs) &= last\ xs \end{aligned}$$

Both functions are undefined for the empty list. These pointwise definitions, relying on pattern matching, are not necessarily very convenient for use in transformations and proofs. Let us consider point-free alternatives. Given appropriate binary operators, it should be possible to express both *head* and *last* in terms of a fold operation. It turns out these operators are very simple indeed, for computing *head*, we just need to ensure we always take the left operand at every point, and similarly for *last* we always take the right operand. Thus we have two simple binary operators $(\leftarrow)$ and $(\rightarrow)$, pronounced $left$ and $right$:

$$\begin{aligned} a \leftarrow b &= a \\ a \rightarrow b &= b \end{aligned}$$

We then have:

$$\begin{aligned} head &= fold\ (\leftarrow) \\ last &= fold\ (\rightarrow) \end{aligned}$$

7.9.1 Streams

In stream terms, we will use definitions analogous to the point-free ones given above for *head* and *tail*:

$$\begin{aligned} shead &= sfold\ (\leftarrow) \\ slast &= sfold\ (\rightarrow) \end{aligned}$$

These have the following types respectively:

$$\begin{aligned} shead &:: \lfloor A \rfloor \rightarrow A \\ slast &:: \lfloor A \rfloor \rightarrow A \end{aligned}$$

Proofs that these are valid refinements will be more or less identical for both functions, so let us just take the example of *shead*. For this to be considered a valid refinement of *head*, we require the following diagram to commute:

$$\begin{array}{ccc} [A] & \xrightarrow{head} & A \\ \uparrow{\scriptstyle abs_S} & & \uparrow{\scriptstyle id} \\ \lfloor A \rfloor & \xrightarrow{shead} & A \end{array}$$

Given the proof that $sfold$ is a valid refinement of $fold$ presented in Section 5.3.1, this proof should be fairly trivial:

$$\begin{array}{ll}
(head \circ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\
(fold\ (\leftarrow) \circ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.head\} \\
(id \circ sfold\ (\leftarrow))\ [x_1, x_2, ..., x_n] & \{ref.\ sfold\} \\
(id \circ shead)\ [x_1, x_2, ..., x_n] & \{def.\ shead\}
\end{array}$$

7.9.2 Vectors

In vector terms, we have the functions $vhead$ and $vlast$:

$$\begin{aligned}
vhead_n &= vfold_n\ (\leftarrow) \\
vlast_n &= vfold_n\ (\rightarrow)
\end{aligned}$$

These have the following types respectively:

$$\begin{aligned}
vhead_n &:: \langle A \rangle_n \rightarrow A \\
vlast_n &:: \langle A \rangle_n \rightarrow A
\end{aligned}$$

As with the stream case, we shall just provide a proof for the validity of $vhead$ as a refinement for $head$, given that the equivalent proof for $last$ and $vlast$ would be almost identical. As usual, we require the following diagram to commute:

$$\begin{array}{ccc}
[A] & \xrightarrow{\quad head \quad} & A \\
\uparrow{\scriptstyle abs_V} & & \uparrow{\scriptstyle id} \\
\langle A \rangle_n & \xrightarrow{\quad vhead_n \quad} & A
\end{array}$$

Again, given the proof that $vfold$ is a valid refinement of $fold$ presented in Section 5.3.2, this proof should be fairly trivial:

$$\begin{array}{ll}
(head \circ abs_V)\ \langle x_1, x_2, ..., x_n \rangle_n & \{id\} \\
(fold\ (\leftarrow) \circ abs_V)\ \langle x_1, x_2, ..., x_n \rangle_n & \{def.head\} \\
(id \circ vfold_n\ (\leftarrow))\ [x_1, x_2, ..., x_n] & \{ref.\ vfold\} \\
(id \circ vhead_n)\ [x_1, x_2, ..., x_n] & \{def.\ vhead\}
\end{array}$$

7.10 Take and Drop

The functions *take* and *drop* can be considered generalisations of *head* and *tail*. To illustrate this, we have the following relationships:

$$\begin{aligned}
take\ 1\ xs &= [head\ xs] \\
drop\ 1\ xs &= tail\ xs
\end{aligned}$$

Both functions take in an integer and a list and return a list. Therefore their type signatures are the same:

$$\begin{aligned} take &:: Int \to [A] \to [A] \\ drop &:: Int \to [A] \to [A] \end{aligned}$$

Their functionality can be defined informally as follows:

$$\begin{aligned} take\ k\ [x_1, x_2, ..., x_n] &= [x_1, x_2, ..., x_{k \downarrow n}] \\ drop\ k\ [x_1, x_2, ..., x_n] &= if\ k \geq n \\ &\quad then\ [x_{k+1}, x_{k+2}, ..., x_n] \\ &\quad else\ [] \end{aligned}$$

More formal definitions are normally given recursively, for example:

$$\begin{aligned} take\ 0\ xs &= [] \\ take\ n\ [] &= [] \\ take\ n\ (x : xs) &= x\ :\ take\ (n-1)\ xs \end{aligned}$$

However, we may find a definition in terms of a list comprehension more convenient at times:

$$\begin{aligned} take\ k\ xs &= [x \mid (i, x) \leftarrow zip\ [1..]\ xs,\ i \leq k] \\ drop\ k\ xs &= [x \mid (i, x) \leftarrow zip\ [1..]\ xs,\ i > k] \end{aligned}$$

Given the close relationship between list comprehensions and the higher order functions *map* and $filter$, these definitions are of course equivalent to the following point-free definitions:

$$\begin{aligned} take\ k &= map\ snd \circ filter\ ((\leq k) \circ fst) \circ zip\ [1..] \\ drop\ k &= map\ snd \circ filter\ ((> k) \circ fst) \circ zip\ [1..] \end{aligned}$$

For *take*, we can also provide a neat definition in terms of $unfoldr$:

$$\begin{aligned} take\ n\ xs &= unfoldr\ f\ p\ (xs, n) \\ &\quad where\ \ f\ (a : as, m+1) = (a, (as, m)) \\ &\qquad\quad p\ (a, m) = null\ a \vee m \leq 0 \end{aligned}$$

However, for *drop* this is not quite so straightforward.

We may find stream and vector refinements of these functions useful.

7.10.1 Streams

Let us first consider a refinement of *take* in stream terms. The function *stake* will have type:

$$stake :: Int \to \lfloor A \rfloor \to \lfloor A \rfloor$$

We can of course supply an informal definition for its functionality:

$$stake\ k\ \lfloor x_1, x_2, ..., x_n \rfloor = \lfloor x_1, x_2, ..., x_{k \downarrow n} \rfloor$$

Given stream refinements for *map*, *filter* and *zip*, one possible definition for *stake*, analogous to the point-free definition of *take* given above, could be as follows:

$$stake\ k\ =\ smap\ snd\ \circ\ sfilter\ ((\leq k) \circ fst)\ \circ\ szip\ \lfloor 1..\rfloor$$

As usual, we will require the now familiar diagram to commute if any definition of *stake* is to be proven a valid refinement:

$$\begin{array}{ccc} [A] & \xrightarrow{take\ k} & [A] \\ \uparrow{\scriptstyle abs_S} & & \uparrow{\scriptstyle abs_S} \\ \lfloor A\rfloor & \xrightarrow{stake\ k} & \lfloor A\rfloor \end{array}$$

Let us first consider a proof in terms of our informal definition of *stake*:

$$\begin{array}{lll} & (take\ k\ \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n\rfloor & \{id\} \\ = & take\ k\ [x_1, x_2, ..., x_n] & \{def.\ abs_S\} \\ = & [x_1, x_2, ..., x_{k\downarrow n}] & \{def.\ take\} \\ = & abs_S\ \lfloor x_1, x_2, ..., x_{k\downarrow n}\rfloor & \{def.\ abs_S\} \\ = & abs_S\ (stake\ k\ \lfloor x_1, x_2, ..., x_n\rfloor) & \{def.\ stake\} \\ = & (abs_S \circ stake\ k)\ \lfloor x_1, x_2, ..., x_n\rfloor & \{def.\circ\} \end{array}$$

We may also provide a proof for our more formal, point-free definition:

$$\begin{array}{lll} & (take\ k\ \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n\rfloor & \{id\} \\ = & take\ k\ [x_1, x_2, ..., x_n] & \{def.\ abs_S\} \\ = & (map\ snd\ \circ\ filter\ ((\leq k) \circ fst)\ \circ\ zip\ [1..])\ [x_1, x_2, ..., x_n] & \{def.\ take\ k\} \\ = & (map\ snd\ \circ\ filter\ ((\leq k) \circ fst))\ [(1, x_1), (2, x_2), ..., (n, x_n)] & \{def.\ zip\} \\ = & map\ snd\ [(1, x_1), (2, x_2), ..., (k \downarrow n, x_{k\downarrow n})] & \{filter\} \\ = & [x_1, x_2, ..., x_{k\downarrow n}] & \{map\} \\ = & abs_S\ \lfloor x_1, x_2, ..., x_{k\downarrow n}\rfloor & \{def. abs_S\} \\ = & abs_S\ (smap\ snd\ \lfloor (1, x_1), (2, x_2), ..., (k \downarrow n, x_{k\downarrow n})\rfloor) & \{smap\} \\ = & abs_S\ ((smap\ snd \circ sfilter\ ((\leq k) \circ fst)))\ \lfloor (1, x_1), (2, x_2), ..., (n, x_n)\rfloor) & \{sfilter\} \\ = & abs_S\ ((smap\ snd \circ sfilter\ ((\leq k) \circ fst))\ \circ\ szip\ \lfloor 1..\rfloor)\ \lfloor x_1, x_2, ..., x_n\rfloor) & \{szip\} \\ = & abs_S\ (stake\ k\ \lfloor x_1, x_2, ..., x_n\rfloor) & \{def. stake\} \\ = & (abs_S \circ stake\ k)\ \lfloor x_1, x_2, ..., x_n\rfloor & \{def.\circ\} \end{array}$$

A similar proof could be provided for the refinement of *drop* in stream terms, given a definition closely following that of *stake*:

$$sdrop\ k\ =\ smap\ snd\ \circ\ sfilter\ ((> k) \circ fst)\ \circ\ szip\ \lfloor 1..\rfloor$$

Clearly the type would be the same as that for *stake*:

$$sdrop \quad :: \quad Int \to \lfloor A \rfloor \to \lfloor A \rfloor$$

7.10.2 Vectors

In vector terms, we have the functions *vtake* and *vdrop*. Here we need to take special consideration of the size of the resulting vectors. Clearly the parameter k affects this, and as such, this value must form part of the type.

$$\begin{aligned} vtake_n\ k &:: \langle A \rangle_n \to \langle A \rangle_{k \downarrow n} \\ vdrop_n\ k &:: \langle A \rangle_n \to \langle A \rangle_{(n-k)\uparrow 0} \end{aligned}$$

Their functionality can be defined informally as follows, assuming k is non-negative:

$$\begin{aligned} vtake_n\ k\ \langle x_1, x_2, ..., x_n \rangle_n &= \langle x_1, x_2, ..., x_{k \downarrow n} \rangle_{k \downarrow n} \\ vdrop_n\ k\ \langle x_1, x_2, ..., x_n \rangle_n &= if\ k < n\ then\ \langle x_{k+1}, x_{k+2}, ..., x_n \rangle_{n-k}\ else\ \langle \rangle_0 \end{aligned}$$

Given the issues with $filter$ in vector terms, we can not use an equivalent of the point-free definitions used for the stream case above. Furthermore, recursive definitions do not suit the vector setting, due to the fixed size nature of the structures involved. Let us assume therefore these are 'primitives'. We can check the validity of these refinements based on these informal definitions in the usual way. First the diagram which we require to commute, in this case for *vtake*:

$$\begin{array}{ccc} [A] & \xrightarrow{take\ k} & [A] \\ \uparrow{\scriptstyle abs_V} & & \uparrow{\scriptstyle abs_V} \\ \langle A \rangle_n & \xrightarrow{vtake_n\ k} & \langle A \rangle_{n \downarrow k} \end{array}$$

The proof may then proceed as follows

$$\begin{aligned} & (take\ k\ \circ\ abs_V)\ \langle x_1, x_2, ..., x_n \rangle_n && \{id\} \\ = \quad & take\ k\ [x_1, x_2, ..., x_n] && \{def.\ abs_V\} \\ = \quad & [x_1, x_2, ..., x_{k \downarrow n}] && \{def.\ take\} \\ = \quad & abs_V\ \langle x_1, x_2, ..., x_{k \downarrow n} \rangle_{k \downarrow n} && \{def.\ abs_V\} \\ = \quad & abs_V\ (vtake_n\ k\ \langle x_1, x_2, ..., x_n \rangle_n) && \{def.\ vtake_n\} \\ = \quad & (abs_V \circ vtake_n\ k)\ \langle x_1, x_2, ..., x_n \rangle_n && \{def. \circ\} \end{aligned}$$

A similar proof could, of course, be presented for *vdrop*.

7.11 Init and Tail

The functions *init* and *tail* retrieve the first and final segments of a list respectively. Their types are therefore as follows:

$$\begin{aligned} init &:: [A] \to [A] \\ tail &:: [A] \to [A] \end{aligned}$$

Informally, we have:

$$\begin{aligned} init\ [x_1, x_2, ..., x_n] &= [x_1, x_2, ..., x_{n-1}] \\ tail\ [x_1, x_2, ..., x_n] &= [x_2, x_3, ..., x_n] \end{aligned}$$

Neither function is defined in the case of the empty list. These are often seen as companions to the functions *head* and *tail*, given the following relationships (assuming xs is a non-empty list):

$$\begin{aligned} [head\ xs] +\!\!+ tail\ xs &= xs \\ init\ xs +\!\!+ [last\ xs] &= xs \end{aligned}$$

They also share a relationship with *take* and *drop*, defined as follows:

$$\begin{aligned} init\ [x_1, x_2, ..., x_n] &= take\ (n-1)\ [x_1, x_2, ..., x_n] \\ tail\ [x_1, x_2, ..., x_n] &= drop\ 1\ [x_1, x_2, ..., x_n] \end{aligned}$$

Their formal definitions are usually given via pattern matching and (in the case of *init*) recursively, as follows:

$$\begin{aligned} tail\ (x : xs) &= xs \\ init\ [x] &= [] \\ init\ (x : xs) &= x\ :\ init\ xs \end{aligned}$$

Given the relationship with *take* and *drop* we can also define them as follows:

$$\begin{aligned} init\ xs &= take\ (length\ xs\ -1)\ xs \\ tail\ xs &= drop\ 1\ xs \end{aligned}$$

7.11.1 Streams

In stream terms, we have:

$$\begin{aligned} sinit\ \lfloor x_1, x_2, ..., x_n \rfloor &= \lfloor x_1, x_2, ..., x_{n-1} \rfloor \\ stail\ \lfloor x_1, x_2, ..., x_n \rfloor &= \lfloor x_2, x_3, ..., x_n \rfloor \end{aligned}$$

These functions have the following type:

$$\begin{aligned} sinit &:: \lfloor A \rfloor \to \lfloor A \rfloor \\ stail &:: \lfloor A \rfloor \to \lfloor A \rfloor \end{aligned}$$

We can give formal definitions for these functions as follows:

$$\begin{aligned} sinit\ s &= stake\ (slength\ s\ -1)\ s \\ stail &= sdrop\ 1 \end{aligned}$$

Given that *stake* and *sdrop* have already been proven to be valid refinements for *take* and *drop* respectively, and that *sinit* and *stail* are implemented in terms of *stake* and *sdrop*, it is a trivial task to prove the validity of these refinements.

Proof that any definition of *sinit* is a valid refinement of *init* will require that the following diagram commutes:

$$\begin{array}{ccc} [A] & \xrightarrow{init} & [A] \\ \uparrow{\scriptstyle abs_S} & & \uparrow{\scriptstyle abs_S} \\ \lfloor A \rfloor & \xrightarrow{sinit} & \lfloor A \rfloor \end{array}$$

Let us consider a proof for our informal definition of *sinit*.

$$\begin{aligned} & (init\ \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor && \{id\} \\ = & \ init\ [x_1, x_2, ..., x_n] && \{def.\ abs_S\} \\ = & \ [x_1, x_2, ..., x_{n-1}] && \{def.\ init\} \\ = & \ abs_S\ \lfloor x_1, x_2, ..., x_{n-1} \rfloor && \{def.\ abs_S\} \\ = & \ abs_S\ (sinit\ \lfloor x_1, x_2, ..., x_n \rfloor) && \{def.\ sinit\} \\ = & \ (abs_S \circ sinit)\ \lfloor x_1, x_2, ..., x_n \rfloor && \{def.\circ\} \end{aligned}$$

A proof for our formal definition of *stail*, based on *sdrop*, may proceed as follows:

$$\begin{aligned} & (tail\ \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor && \{id\} \\ = & \ (drop\ 1 \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor && \{def.\ tail\} \\ = & \ (abs_S \circ\ sdrop\ 1)\ \lfloor x_1, x_2, ..., x_n \rfloor && \{ref.\ sdrop\} \\ = & \ (abs_S \circ stail)\ \lfloor x_1, x_2, ..., x_n \rfloor && \{def.\ stail\} \end{aligned}$$

A proof for the formal definition of *sinit* is a little trickier, given the pointwise style of our definition. We have:

$$
\begin{array}{lll}
 & (init \circ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\
= & ((\lambda\, s \bullet take\ (length\ s - 1)\ s) \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ init\} \\
= & ((\lambda\, s \bullet take\ (((-1) \circ length)\ s)\ s) \circ\ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ \circ\} \\
= & (\lambda\, s \bullet (take\ (((-1) \circ length \circ abs_S)\ s) \circ abs_S)\ s)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{see\ below\} \\
= & (\lambda\, s \bullet (take\ (f\ s) \circ abs_S)\ s)\ \lfloor x_1, x_2, ..., x_n \rfloor & \\
 & \quad where \quad f \ = \ (-1) \circ length \circ abs_S & \{where\} \\
= & g\ \lfloor x_1, x_2, ..., x_n \rfloor & \\
 & \quad where \quad f \ = \ (-1) \circ length \circ abs_S & \{where\} \\
 & \qquad\quad\ g\ s \ = \ (take\ (f\ s) \circ abs_S)\ s & \\
= & g\ \lfloor x_1, x_2, ..., x_n \rfloor & \\
 & \quad where \quad f \ = \ (-1) \circ id \circ slength & \{ref.\ slength\} \\
 & \qquad\quad\ g\ s \ = \ (take\ (f\ s) \circ abs_S)\ s & \\
= & g\ \lfloor x_1, x_2, ..., x_n \rfloor & \\
 & \quad where \quad f \ = \ (-1) \circ id \circ slength & \{ref.\ stake\} \\
 & \qquad\quad\ g\ s \ = \ (abs_S \circ stake\ (f\ s))\ s & \\
= & (\lambda\, s \bullet (abs_S \circ stake\ (f\ s))\ s)\ \lfloor x_1, x_2, ..., x_n \rfloor & \\
 & \quad where \quad f \ = \ (-1) \circ id \circ slength & \{where\} \\
= & (\lambda\, s \bullet (abs_S \circ stake\ (((-1) \circ id \circ slength)\ s))\ s)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{where\} \\
= & (\lambda\, s \bullet (abs_S \circ stake\ (((-1) \circ slength)\ s))\ s)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ id\} \\
= & (\lambda\, s \bullet (abs_S \circ stake\ (slength\ s - 1))\ s)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ \circ\} \\
= & (\lambda\, s \bullet (abs_S \circ sinit\ s))\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ sinit\} \\
= & (abs_S \circ sinit)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ \lambda\}
\end{array}
$$

This proof hinges on several points. The already proven refinements of *length* and *take* into *slength* and *stake* respectively are of course essential.

Additionally, we have a transformation rule useful for dealing with pointwise definitions. Consider a function k applied to an argument x which computes its result based on passing x first to two functions g and h, and then passing the results of these to f. We have

$$k\ x = f\ (g\ x)\ (h\ x)$$

Or using Lambda notation we have:

$$k\ =\ (\lambda\, x \bullet f\ (g\ x)\ (h\ x))$$

Our function k is then composed with another function e.

$$(\lambda\, x \bullet f\ (g\ x)\ (h\ x)) \circ e$$

In such a definition, we may find it convenient to "distribute" e over the sub-expressions $(g\ x)$ and $(h\ x)$. In effect this is the factorisation rule applied to the functional composition operator. Thus we have the following equivalence:

$$(\lambda\, x \bullet f\ (g\ x)\ (h\ x)) \circ e\ =\ (\lambda\, x \bullet f\ ((g \circ e)\ x)\ ((h \circ e)\ x))$$

In fact we can generalise this up to any number of sub-expressions:

$$(\lambda\, x \bullet f\ (g_1\ x)\ (g_2\ x)...(g_n\ x)) \circ e\ =\ (\lambda\, x \bullet f\ ((g_1 \circ e)\ x)\ ((g_2 \circ e)\ x)\ ...\ ((g_n \circ e)\ x))$$

7.11.2 Vectors

In vector terms, we have:

$$\begin{array}{lcl} vinit_n\ \langle x_1, x_2, ..., x_n\rangle_n & = & \langle x_1, x_2, ..., x_{n-1}\rangle_{n-1} \\ vtail_n\ \langle x_1, x_2, ..., x_n\rangle_n & = & \langle x_2, x_3, ..., x_n\rangle_{n-1} \end{array}$$

With the following type signatures:

$$\begin{array}{lcl} vinit_n & :: & \langle A\rangle_n \to \langle A\rangle_{n-1} \\ vtail_n & :: & \langle A\rangle_n \to \langle A\rangle_{n-1} \end{array}$$

Formal definitions can follow those for *init* and *tail* given in terms of *take* and *drop*.

$$\begin{array}{lcl} vinit_n\ xs & = & vtake_n\ (vlength\ xs - 1) \\ vtail_n & = & vdrop_n\ 1 \end{array}$$

Indeed, given that we are working in the vector setting, and as such our structures are of fixed length, we can in fact simplify our definition for $vinit_n$ somewhat, to arrive at a point-free version:

$$vinit_n \quad = \quad vtake_n\ (n-1)$$

As with the stream case, given that *vtake* and *vdrop* have already been proven to be valid refinements for *take* and *drop* respectively, and that *vinit* and *vtail* are implemented in terms of *vtake* and *vdrop*, it is a trivial task to prove the validity of these refinements. Taking the example of *vtail*, we require the familiar diagram to commute:

$$\begin{array}{ccc} [A] & \xrightarrow{\ tail\ } & [A] \\ \uparrow{\scriptstyle abs_V} & & \uparrow{\scriptstyle abs_V} \\ \langle A\rangle_n & \xrightarrow{\ vtail_n\ } & \langle A\rangle_{n-1} \end{array}$$

Proof of this is again straightforward:

$$\begin{array}{lll} & (tail\ \circ\ abs_V)\ \langle x_1, x_2, ..., x_n\rangle_n & \{id\} \\ = & (drop\ 1\ \circ\ abs_V)\ \langle x_1, x_2, ..., x_n\rangle_n & \{def.\ tail\} \\ = & (abs_V \circ\ vdrop_n\ 1)\ \langle x_1, x_2, ..., x_n\rangle_n & \{ref.\ vdrop\} \\ = & (abs_V \circ vtail_n)\ \langle x_1, x_2, ..., x_n\rangle_n & \{def.\ vtail\} \end{array}$$

7.12 Inits and Tails

Two particularly useful functions are *inits* and *tails*. The first, *inits*, produces all of the initial segments of a given list in ascending order of size. We have:

$$\begin{aligned} &inits :: [A] \to [[A]] \\ &inits\ [x_1, x_2, \ldots, x_n] \ = \ [[], [x_1], \ldots, [x_1, x_2, \ldots, x_{n-1}], [x_1, x_2, \ldots, x_n]] \end{aligned}$$

As a natural companion to this, *tails* produces all of the final segments of a list, in descending order of size:

$$\begin{aligned} &tails :: [A] \to [[A]] \\ &tails\ [x_1, x_2, \ldots, x_n] \ = \ [[x_1, x_2, \ldots, x_n], [x_2, x_3, \ldots, x_n], \ldots, [x_n], []] \end{aligned}$$

More formal definitions of *inits* and *tails* are usually given recursively, for example, *inits* can be defined as follows:

$$\begin{aligned} inits[] \quad &= \quad [[]] \\ inits(x : xs) \quad &= \quad [[]] \ +\!\!+ \ map\ (x :)\ (inits\ xs) \end{aligned}$$

Correspondingly, a possible recursive definition for *tails* is as follows:

$$\begin{aligned} tails[] \quad &= \quad [[]] \\ tails(x : xs) \quad &= \quad (x : xs) \ : \ tails\ xs \end{aligned}$$

We may find it useful to consider also definitions for *inits* and *tails* in terms of higher order functions. One possible method is in terms of a *map* and a $fold$, as follows:

$$\begin{aligned} inits \quad &= \quad fold\ (\otimes_i) \ \circ \ map\ (\lambda x \ . \ [[], [x]]) \\ &\qquad where\ as \otimes_i bs \ = \ as \ +\!\!+ \ map\ (last\ as \ +\!\!+)\ (tail\ bs) \\ tails \quad &= \quad fold\ (\otimes_t) \ \circ \ map\ (\lambda x \ . \ [[x], []]) \\ &\qquad where\ as \otimes_t bs \ = \ map\ (+\!\!+ \ head\ bs)\ (init\ as) \ +\!\!+ \ bs \end{aligned}$$

These definitions clearly identify the homomorphic nature of *inits* and *tails*. Another possible definition through higher order functions is presented to us by $unfoldr$ and $unfoldl$.

$$\begin{aligned} tails\ xss \quad &= \quad unfoldr\ (\lambda xs \bullet (xs, tail\ xs))\ null\ xss +\!\!+ [[]] \\ inits\ xss \quad &= \quad [[]] +\!\!+ unfoldl\ (\lambda xs \bullet (init\ xs, xs))\ null\ xss \end{aligned}$$

Two further variations on these functions may be seen, $inits^+$ and $tails^+$. These operate in an almost identical fashion to *inits* and *tails*, except that they don't include the empty list in their output. I.e., for $inits^+$, we have:

$$\begin{aligned} &inits^+ :: [A] \to [[A]] \\ &inits^+\ [x_1, x_2, \ldots, x_n] \ = \ [[x_1], \ldots, [x_1, x_2, \ldots, x_{n-1}], [x_1, x_2, \ldots, x_n]] \end{aligned}$$

and for $tails^+$:

$$tails^+ :: [A] \to [[A]]$$
$$tails^+\ [x_1, x_2, \ldots, x_n] \ = \ [[x_1, x_2, \ldots, x_n], [x_2, x_3, \ldots, x_n], \ldots, [x_n]]$$

These variants are particularly well suited to definition in terms of $unfoldl$ and $unfoldr$:

$$\begin{aligned} tails^+ &= unfoldr\ (\lambda xs \bullet (xs, tail\ xs))\ null \\ inits^+ &= unfoldl\ (\lambda xs \bullet (init\ xs, xs))\ null \end{aligned}$$

A small amount of swapping around gives us two further variants:

$$\begin{aligned} fins^+ &= unfoldl\ (\lambda xs \bullet (tail\ xs, xs))\ null \\ begs^+ &= unfoldr\ (\lambda xs \bullet (xs, init\ xs))\ null \end{aligned}$$

Here $fins^+$ is analogous to $tails^+$ except that the lists are returned in ascending, rather than descending, order of length. Similarly $begs^+$ is analogous to $inits^+$ except that the lists are returned in descending order of length. We can of course translate between $tails^+$ and $fins^+$ (as well as $tails^+$ and $fins^+$) simply by reversing the resulting list. We have:

$$\begin{aligned} fins^+ &= reverse \circ tails^+ \\ begs^+ &= reverse \circ inits^+ \end{aligned}$$

7.12.1 Stream to Stream of Streams

The first possible refinement we shall consider refines the input list to a stream, and the output list of lists to a stream of streams. This will provide us with the least scope for parallelism, but also the least obstacles when we come to consider implementation. Considering $tails$, we shall name a refinement in these terms $ssstails$. Its definition is analogous to that of $tails$:

$$ssstails\ \lfloor x_1, x_2, \ldots, x_n \rfloor \ = \ \lfloor \lfloor x_1, x_2, \ldots, x_n \rfloor, \lfloor x_2, x_3, \ldots, x_n \rfloor, \ldots, \lfloor x_n \rfloor, \lfloor \rfloor \rfloor$$

The type of this function is as follows:

$$ssstails :: \lfloor A \rfloor \to \lfloor \lfloor A \rfloor \rfloor$$

As ever, to prove this is a valid refinement of $tails$, we shall require that the following diagram commutes:

$$\begin{array}{ccc} [A] & \xrightarrow{\ tails\ } & [[A]] \\ \uparrow{\scriptstyle abs_S} & & \uparrow{\scriptstyle abs_{SS}} \\ \lfloor A \rfloor & \xrightarrow{\ ssstails\ } & \lfloor \lfloor A \rfloor \rfloor \end{array}$$

We can prove this as follows:

$$\begin{array}{lll} & (tails \circ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\ = & tails\ [x_1, x_2, ..., x_n] & \{def.\ abs_S\} \\ = & [[x_1, x_2, \ldots, x_n], [x_2, x_3, \ldots, x_n], \ldots, [x_n], []] & \{def.\ tails\} \\ = & abs_{SS}\ \lfloor \lfloor x_1, x_2, \ldots, x_n \rfloor, \lfloor x_2, x_3, \ldots, x_n \rfloor, \ldots, \lfloor x_n \rfloor, \lfloor \rfloor \rfloor & \{def.\ abs_{SS}\} \\ = & (abs_{SS} \circ ssstails)\ \lfloor x_1, x_2, \ldots, x_n \rfloor & \{def.\ ssstails\} \end{array}$$

Considering *inits* in this setting, in short we have the following definitions:

$$\begin{array}{lcl} sssinits & :: & \lfloor A \rfloor \rightarrow \lfloor \lfloor A \rfloor \rfloor \\ sssinits^+ & :: & \lfloor A \rfloor \rightarrow \lfloor \lfloor A \rfloor \rfloor \\ sssinits\ \lfloor x_1, x_2, \ldots, x_n \rfloor & = & \lfloor \lfloor \rfloor, \lfloor x_1 \rfloor, \ldots, \lfloor x_1, x_2, \ldots, x_{n-1} \rfloor, \lfloor x_1, x_2, \ldots, x_n \rfloor \rfloor \\ sssinits^+\ \lfloor x_1, x_2, \ldots, x_n \rfloor & = & \lfloor \lfloor x_1 \rfloor, \ldots, \lfloor x_1, x_2, \ldots, x_{n-1} \rfloor, \lfloor x_1, x_2, \ldots, x_n \rfloor \rfloor \end{array}$$

Process Refinement

A process to implement the functionality of either *tails* or *init* in this setting will require buffering. The process will only receive each value from the input stream once, however, it will be required to output some of these values many times, at different positions. The ordering is, of course, all important with these functions. Let us consider therefore a process refinement for *sssinits*. We have a broad outline of the required functionality in CSP below:

$$\begin{array}{lll} SSSINITS & = & buff := [];\ Prd(buff)[out.value/out]; \\ & & \mu\, X \bullet \\ & & in.eot\ ?\ any \rightarrow out.eot\ !\ True \rightarrow SKIP \\ & & | \\ & & in.value\ ?\ x \rightarrow buff := buff +\!\!+ [x]; Prd(buff)[out.value/out]; X \end{array}$$

Similarly the process $SSSINITS^+$ could proceed along the same lines as above, but without the initial *Prd* for the empty buffer. For processes implementing the functionality of *tails* in this setting the definition is not quite as fluid. With *tails*, unlike *inits*, the first component list is the entire input list. We will therefore have had to consume the entire input stream before this can be output.

7.12.2 Stream to Vector of Streams

An improvement on the previous alternative in terms of scope for parallelism. This refinement allows the sub-lists in the output to be produced independently in parallel as a vector of streams. However, this approach has one drawback in that we will need to know the size of the input stream before we can know the size of the output vector. A refinement of *tails* in these terms shall be called *svstails*.

$$svstails_n\ \lfloor x_1, x_2, \ldots, x_n \rfloor\ =\ \langle \lfloor x_1, x_2, \ldots, x_n \rfloor, \lfloor x_2, x_3, \ldots, x_n \rfloor, \ldots, \lfloor x_n \rfloor, \lfloor \rfloor \rangle_{n+1}$$

The type of this function as as follows:

$$svstails_n :: \lfloor A \rfloor \rightarrow \langle \lfloor A \rfloor \rangle_{n+1}$$

Similarly, for our version which does not include the empty list in the output, we have:

$$svstails_n^+\ \lfloor x_1, x_2, \ldots, x_n \rfloor\ =\ \langle \lfloor x_1, x_2, \ldots, x_n \rfloor, \lfloor x_2, x_3, \ldots, x_n \rfloor, \ldots, \lfloor x_n \rfloor \rangle_n$$

This has the following type:

$$svstails_n^+ :: \lfloor A \rfloor \rightarrow \langle \lfloor A \rfloor \rangle_n$$

To prove $svstails_n$ is a valid refinement of $tails$, we shall require that the following diagram commutes:

$$\begin{array}{ccc} [A] & \xrightarrow{\ tails\ } & [[A]] \\ \uparrow{\scriptstyle abs_S} & & \uparrow{\scriptstyle abs_{VS}} \\ \lfloor A \rfloor & \xrightarrow{\ svstails_n\ } & \langle \lfloor A \rfloor \rangle_n \end{array}$$

We can prove this as follows:

$$\begin{array}{lll} & (tails \circ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\ = & tails\ [x_1, x_2, ..., x_n] & \{def.\ abs_S\} \\ = & [[x_1, x_2, \ldots, x_n], [x_2, x_3, \ldots, x_n], \ldots, [x_n], []] & \{def.\ tails\} \\ = & abs_{VS}\ \langle \lfloor x_1, x_2, \ldots, x_n \rfloor, \lfloor x_2, x_3, \ldots, x_n \rfloor, \ldots, \lfloor x_n \rfloor, \lfloor \rfloor \rangle_{n+1} & \{def.\ abs_{VS}\} \\ = & (abs_{VS} \circ svstails_n)\ \langle x_1, x_2, \ldots, x_n \rangle_{n+1} & \{def.\ svstails_n\} \end{array}$$

A similar proof could be provided for $svstails_n^+$.

Considering $inits$ in this setting, in short we have the following definitions:

$$\begin{array}{lll} svsinits_n & :: & \lfloor A \rfloor \rightarrow \langle \lfloor A \rfloor \rangle_{n+1} \\ svsinits_n^+ & :: & \lfloor A \rfloor \rightarrow \langle \lfloor A \rfloor \rangle_n \\ svsinits_n\ \lfloor x_1, x_2, \ldots, x_n \rfloor & = & \langle \lfloor \rfloor, \lfloor x_1 \rfloor, \ldots, \lfloor x_1, x_2, \ldots, x_{n-1} \rfloor, \lfloor x_1, x_2, \ldots, x_n \rfloor \rangle_{n+1} \\ svsinits_n^+\ \lfloor x_1, x_2, \ldots, x_n \rfloor & = & \langle \lfloor x_1 \rfloor, \ldots, \lfloor x_1, x_2, \ldots, x_{n-1} \rfloor, \lfloor x_1, x_2, \ldots, x_n \rfloor \rangle_n \end{array}$$

As already noted, $inits$ and $tails$ can be expressed in terms of the more general $unfold$ scheme. This expression works particularly well for the variants which do not contain the empty list in their output - $inits^+$ and $tails^+$. Correspondingly in our vector of streams output refinement, we can appeal to $vunfoldr$ and $vunfoldl$ respectively for definitions:

$$
\begin{array}{lcl}
svstails^{+} & = & vunfoldr\ tl\ snull \\
 & & where\ tl\ xs\ =\ (xs, stail\ xs) \\
svsinits^{+} & = & vunfoldl\ ini\ snull \\
 & & where\ ini\ xs\ =\ (sinit\ xs, xs)
\end{array}
$$

Process Refinement

Our process refinement shall follow the pattern of the $unfold$ functions. In our data refinement stage we have shown how we can express our derivations of $inits$ and $tails$ in this particular setting in terms of a $vunfoldl$ and a $vunfoldr$ respectively. Our main task is then to concentrate on the refinement of the characteristic function. In the case of our refinement of $tails$, we have the function tl:

$$tl\ xs\ =\ (xs, stail\ xs)$$

This has the following type:

$$tl\ ::\ \lfloor A \rfloor \rightarrow (\lfloor A \rfloor, \lfloor A \rfloor)$$

In process terms we have a component which takes one input stream and produces two output streams. One of the two output streams should contain all of the values received from the input stream. This stream will form one of the components of our overall output, and we shall name the corresponding conduit $outd$ - out 'down'. The other output stream should contain the tail of the input - all but the first item. This stream will form the input to the next component process, and we shall name the corresponding conduit $outr$ - out 'right'. As such we have the following alphabet for this process:

$$\alpha TL = \{in :: \underline{\lfloor A \rfloor}, outd :: \underline{\lfloor A \rfloor}, outr :: \underline{\lfloor A \rfloor}\}$$

This component process is illustrated in Figure 7.6.

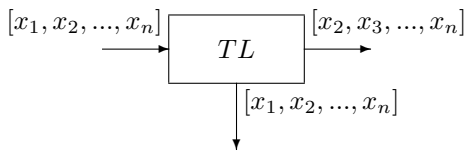


Figure 7.6: A single component of the process SVSTAILS.

As for a definition of this process, we are effectively echoing each item from the input stream to each of the two output streams at each step. The exception to this is the very first item received, which will be echoed to $outd$ but not to $outr$. Such a process can be defined in CSP as follows:

$$TL \quad = \quad in.value\ ?\ x \rightarrow outd.value\ !\ x \rightarrow \mu\, X \bullet \left(\begin{array}{l} in.eot\ ?\ any \rightarrow outd.eot\ !\ True \rightarrow outr.eot\ !\ True \rightarrow SKIP \\ | \\ in.value\ ?\ x \rightarrow outd.value\ !\ x \rightarrow outr.value\ !\ x \rightarrow X \end{array} \right)$$

We can construct an implementation of $SVSTAILS$ by composing n of these processes together in parallel. We have the following alphabet:

$$\alpha SVSTAILS_n = \{in :: \underline{\lfloor A \rfloor}, out :: \underline{\langle \lfloor A \rfloor \rangle_{n+1}}\}$$

Those components in the 'middle' of the composition will follow a standard pattern of communication, taking their input from the previous process, echoing the entire stream to their output conduit, and then passing the tail onto the next process. We shall require slightly different behaviour for the initial and final components. The initial component will have to deal with the input stream to the process as a whole. The final process will be responsible for producing two streams rather than our usual one in the output vector of streams. We shall therefore have a total of n component processes, producing $(n+1)$ output streams. We shall also require a set of $(n-1)$ intermediate stream conduits to communicate from each component process to its successor. These intermediate conduits will be named $[mid_1, mid_2, ..., mid_{n-1}]$. The definition is given below:

$$SVSTAILS_n \quad = \quad TL_{initial} \parallel \left(\begin{array}{l} n-1 \\ \quad \parallel \quad TL[mid_{i-1}/in, out_i/outd, mid_i/outr] \\ i = 2 \end{array} \right) \parallel TL_{final}$$

$$TL_{initial} \quad = \quad TL[in/in, out_1/outd, mid_1/outr]$$

$$TL_{final} \quad = \quad TL[mid_{n-1}/in, out_n/outd, out_{n+1}/outr]$$

An illustration of the process in question is given in Figure 7.7.

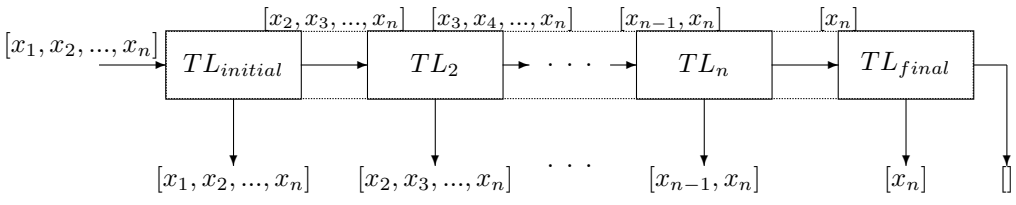


Figure 7.7: The process $SVSTAILS$.

The inclusion of the empty list (or stream) in the output is, as already noted, not always desirable - this is likely to result in wasted processing resources. So, let us consider here a process refinement for the function $svstails_n^+$. We shall see here that the process definition for $SVSTAILS^+$ follows very closely to that for $SVSTAILS$. In effect we simply need to employ one less instance

of the TL process, and slightly change the channel assignments of the final component. This is reflected first of all in the alphabet of the process (note one less output stream):

$$\alpha SVSTAILS_n^+ = \{in :: \underline{\lfloor A \rfloor}, out :: \underline{\langle \lfloor A \rfloor \rangle_n}\}$$

We shall require a total of $(n-1)$ component processes, and $(n-2)$ intermediate processes to communicate between them. As the final process produces two output streams, we will have a total of n streams in our output vector. The definition is given below, with the differences highlighted in bold:

$$\begin{aligned}
SVSTAILS_n^+ &= TL_{initial} \;||\; \left(\begin{array}{l} \mathbf{n-2} \\ \quad || \quad TL[mid_{i-1}/in, out_i/outd, mid_i/outr] \\ i = 2 \end{array} \right) \;||\; TL_{final} \\
TL_{initial} &= TL[in/in, out_1/outd, mid_1/outr] \\
TL_{final} &= TL[mid_{\mathbf{n-2}}/in, out_{\mathbf{n-1}}/outd, out_{\mathbf{n}}/outr]
\end{aligned}$$

This is depicted in Figure 7.8.

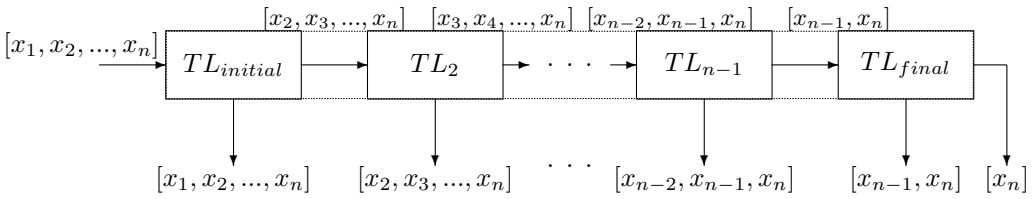


Figure 7.8: The process $SVSTAILS^+$.

Considering now a process implementing *inits* now, we shall see we can again rely on the $unfold$ pattern and build such a process by a composition of simple components. This time our component process will be a refinement of the function *ini*, given below:

$$ini\ xs \;=\; (sinit\ xs, xs)$$

This has the following type:

$$ini \;::\; \lfloor A \rfloor \rightarrow (\lfloor A \rfloor, \lfloor A \rfloor)$$

The behaviour of the component process INI is somewhat analogous to that for TL. Note here that the flow of data is in the opposite direction. This is a consequence of the head recursive nature of *inits* as opposed to the tail recursive nature of *tails*. Indeed, we have implemented *inits* in terms of $unfoldl$, but *tails* in terms of $unfoldr$. The only other major difference is, of course, that the incomplete stream output gives the *init* of the input stream rather than the *tail*. This process will deal with three conduits. The first, *in*, receives the input stream. The second, *outd* (out 'down'),

echoes the input stream verbatim. The third, *outl* (out 'left') is responsible for producing the *init* (all but the last item) of the input stream. As such the process alphabet can be defined as follows:

$$\alpha INI = \{in :: \underline{\lfloor A \rfloor}, outd :: \underline{\lfloor A \rfloor}, outl :: \underline{\lfloor A \rfloor}\}$$

Such a component is illustrated in Figure 7.9.

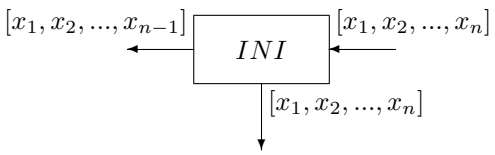


Figure 7.9: A single component of the process SVSINITS.

This component process can be implemented by means of a single item buffer. Each time an item is received, it is immediately echoed on one output stream (*outd*) and is also put in the buffer. When a subsequent item is received, the buffered item can be output on *outl*, and the newly received item will replace the previous item in the buffer. When EOT is signalled for the input stream, the buffered item is then discarded. In this way all but the last item will be output on *outl*, whereas the entirety of the input stream will be echoed on *outd*.

$$\begin{aligned} INI \quad = \quad & in.value \; ? \; b \rightarrow outd.value \; ! \; b \rightarrow \\ & \mu \, X \bullet \left(\begin{array}{l} in.eot \; ? \; any \rightarrow outd.eot \; ! \; True \rightarrow outl.eot \; ! \; True \rightarrow SKIP \\ | \\ in.value \; ? \; x \rightarrow outd.value \; ! \; x \rightarrow outl.value \; ! \; b; \; b := x; \; \rightarrow X \end{array} \right) \end{aligned}$$

As was the case with *tails*, our process implementing the functionality of *inits* can then be constructed by composing together a number of these components. As before we shall see deviation from the norm for the final and initial processes in the network. This can be specified in CSP as follows:

$$\begin{aligned} SVSINITS_n \quad &= \quad INI_{final} \; || \left(\begin{array}{l} n-1 \\ \quad || \quad INI[mid_i/in, out_{i+1}/outd, mid_{i-1}/outl] \\ i = 2 \end{array} \right) || \; INI_{initial} \\ INI_{initial} \quad &= \quad INI[in/in, out_{n+1}/outd, mid_{n-1}/outl] \\ INI_{final} \quad &= \quad INI[mid_1/in, out_2/outd, out_1/outl] \end{aligned}$$

The resulting network is depicted in Figure 7.10.

Handel-C Implementation

We can supply a Handel-C implementation which follows very closely the behaviour given in our CSP definition. First, let us look at our component process, TL. The Handel-C implementation is

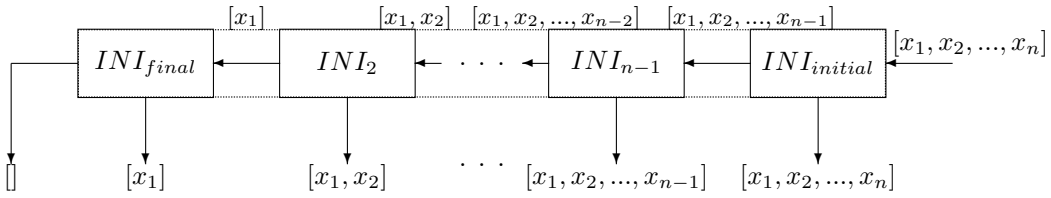


Figure 7.10: The process $SVSINITS$.

```
macro proc TL (in, outd, outr)
{
    messagetype (in) x;
    Bool eot;
    eot = False;
    in.value ? x;
    outd.value ! x;
    while (!eot)
    {
        prialt
        {
        case in.eot ? eot;
            outd.eot ! True;
            outr.eot ! True;
            break;
        case in.value ? x;
            outd.value ! x;
            outr.value ! x;
            break;
        }
    }
}
```

Figure 7.11: The Handel-C definition of the process TL.

given in Figure 7.11. We can then construct our definition for $SVSTAILS$ by composing together a number of instances of TL, in a manner analogous to the CSP definition. This is shown in Figure 7.12.

7.12.3 Stream to Stream of Vectors

This alternative is not workable as we have stated that the stream of vectors requires each sub-list to be of equal length. The output of both *inits* and *tails* by their very nature will contain sub-lists of differing lengths. This can be illustrated by an attempt to construct the type definition of this function:

$$ssvtails_n :: \lfloor A \rfloor \rightarrow \lfloor \langle A \rangle_{??} \rfloor$$

The size of the inner vector here is variable, between 0 and the size of the input stream. This

```
macro proc SVSTAILS (n, in, out)
{
    conduittype (in) mid[n];
    par (i=0;i<(n+1);i++)
    {
        ifselect (i==0)
            TL (in,out[0],mid[0]);
        else ifselect (i<n)
            TL (mid[i-1],out[i],mid[i]);
        else ifselect (i==n)
            TL (mid[n-1],out[n-1],out[n]);
    }
}
```

Figure 7.12: The Handel-C definition of the process SVSTAILS.

is not, therefore, a well typed function.

7.12.4 Stream to Vector of Vectors

Again, within a vector of vectors, each sub-vector must be of equal length, so this refinement is not possible, given the nature of the output of *inits* and *tails*. In any case, for efficiency reasons, this is probably not a very useful refinement anyway. If the input is refined to a stream, we shall require linear time to receive it. Thus there is no great benefit in being able to produce the entire output structure in constant time. As such, we shall not consider this alternative.

7.12.5 Vector to Stream of Streams

The usefulness of receiving the input as a vector (and therefore in constant time) is lost given that the output will require quadratic time to produce. Thus we shall consider this alternative only briefly here. Our informal definition for *vsstails* follows the usual pattern:

$$vsstails_n\ \langle x_1, x_2, \ldots, x_n \rangle_n\ =\ \lfloor \lfloor x_1, x_2, \ldots, x_n \rfloor, \lfloor x_2, x_3, \ldots, x_n \rfloor, \ldots, \lfloor x_n \rfloor, \lfloor \rfloor \rfloor$$

This has the following type:

$$vsstails_n :: \langle A \rangle_n \rightarrow \lfloor \lfloor A \rfloor \rfloor$$

7.12.6 Vector to Vector of Streams

We might consider this as an alternative to the stream to vector of streams refinement, given that, as mentioned, we will need to know the size of the input anyway. A refinement of *tails* in these terms shall be called *vvstails*.

$$vvstails_n\ \langle x_1, x_2, \ldots, x_n \rangle_n\ =\ \langle \lfloor x_1, x_2, \ldots, x_n \rfloor, \lfloor x_2, x_3, \ldots, x_n \rfloor, \ldots, \lfloor x_n \rfloor, \lfloor \rfloor \rangle_{n+1}$$

This has the following type:

$$vvstails_n :: \langle A \rangle_n \rightarrow \langle \lfloor A \rfloor \rangle_{n+1}$$

To prove this is a valid refinement of $tails$, we shall require that the following diagram commutes:

$$\begin{array}{ccc} [A] & \xrightarrow{\quad tails \quad} & [[A]] \\ \uparrow{\scriptstyle abs_V} & & \uparrow{\scriptstyle abs_{VS}} \\ \langle A \rangle_n & \xrightarrow{\quad vvstails_n \quad} & \langle \lfloor A \rfloor \rangle_n \end{array}$$

We can prove this as follows:

$$\begin{array}{lll} & (tails \circ abs_V)\ \langle x_1, x_2, ..., x_n \rangle_n & \{id\} \\ = & tails\ [x_1, x_2, ..., x_n] & \{def.\ abs_V\} \\ = & [[x_1, x_2, \ldots, x_n], [x_2, x_3, \ldots, x_n], \ldots, [x_n], []] & \{def.\ tails\} \\ = & abs_{VS}\ \langle \lfloor x_1, x_2, \ldots, x_n \rfloor, \lfloor x_2, x_3, \ldots, x_n \rfloor, \ldots, \lfloor x_n \rfloor, \lfloor \rfloor \rangle_{n+1} & \{def.\ abs_{VS}\} \\ = & (abs_{VS} \circ svstails_n)\ \langle x_1, x_2, \ldots, x_n \rangle_{n+1} & \{def.\ svstails_n\} \end{array}$$

7.12.7 Vector to Stream of Vectors

As before, given that the sub-lists of the output are of differing sizes, we can not employ the stream of vectors to produce the result.

7.12.8 Vector to Vector of Vectors

This refinement should, in theory, be capable of performing the entire operation in constant time. However in practice, its usefulness may be questionable given that the output will be in such an irregular form - a vector containing vectors all of differing sizes.

7.13 And, Or, Any, All

We shall occasionally find useful a handful of simple logical operations on lists, which can be defined in terms of our higher order functions *map* and $fold$. First we have two functions named *and* and *or* which can be seen as generalisations of the binary operators ($\wedge$) and ($\vee$). These functions simply fold over a list of booleans with the appropriate operator and base value. As such for a non-empty list, *and* returns True if and only if all items in the list are True, whereas *or* returns True if one or more items in the list is True.

$$\begin{array}{lcl} and & :: & [Bool] \rightarrow Bool \\ or & :: & [Bool] \rightarrow Bool \\ and & = & foldr\ (\wedge)\ True \\ or & = & foldr\ (\vee)\ False \end{array}$$

Two further simple functions related to the above two are *all* and *any*. Each takes in a list and a predicate, and then returns a boolean dictating whether or not that predicate is true for all, or any, items of that list respectively. We have:

$$\begin{aligned} any &:: (A \to Bool) \to [A] \to Bool \\ all &:: (A \to Bool) \to [A] \to Bool \\ any\ p &= or \circ map\ p \\ all\ p &= and \circ map\ p \end{aligned}$$

7.13.1 Streams

In stream terms we have the following refinements, firstly for *and* and *or*:

$$\begin{aligned} sand &:: \lfloor Bool \rfloor \to Bool \\ sor &:: \lfloor Bool \rfloor \to Bool \\ sand &= sfoldr\ (\wedge)\ True \\ sor &= sfoldr\ (\vee)\ False \end{aligned}$$

Secondly for *any* and *all*, we have:

$$\begin{aligned} sany &:: (A \to Bool) \to \lfloor A \rfloor \to Bool \\ sall &:: (A \to Bool) \to \lfloor A \rfloor \to Bool \\ sany\ p &= sor \circ smap\ p \\ sall\ p &= sand \circ smap\ p \end{aligned}$$

Proofs that these are valid refinements would be trivial given that we have already proved the refinements of *map* and $foldr$, so we shall not include them here.

7.13.2 Vectors

In vector terms we have the following refinements, firstly for *and* and *or*:

$$\begin{aligned} vand_n &:: \langle Bool \rangle_n \to Bool \\ vor_n &:: \langle Bool \rangle_n \to Bool \\ vand_n &= vfoldr_n\ (\wedge)\ True \\ vor_n &= vfoldr_n\ (\vee)\ False \end{aligned}$$

Secondly for *any* and *all*, we have:

$$\begin{aligned} vany_n &:: (A \to Bool) \to \langle A \rangle_n \to Bool \\ vall_n &:: (A \to Bool) \to \langle A \rangle_n \to Bool \\ vany_n\ p &= vor_n \circ vmap_n\ p \\ vall_n\ p &= vand_n \circ vmap_n\ p \end{aligned}$$

As with the stream case, proofs that these are valid refinements would be trivial given that we have already proved the refinements of *map* and $foldr$, so again we shall not include them here.

7.14 Cartesian Product

Many algorithms require the calculation of the Cartesian product of two sets or lists. Thus we will find this useful as a generic list processing function. Informally, we have:

$$\begin{aligned} &cp\ [x_1, x_2, ..., x_n]\ [y_1, y_2, ..., y_m] \\ &= [(x_1, y_1), (x_1, y_2), ..., (x_1, y_m), (x_2, y_1), (x_2, y_2), ..., (x_2, y_m), ..., (x_n, y_1), (x_n, y_2), ..., (x_n, y_m)] \end{aligned}$$

In list terms we have the following type:

$$cp :: [A] \rightarrow [B] \rightarrow [(A, B)]$$

This function is typically defined as a list comprehension, and this is perhaps the most succinct manner in which to present a definition:

$$cp\ xs\ ys\ =\ [(x, y)\ |\ x \leftarrow xs, y \leftarrow ys]$$

An alternative is the function *dcp*, or distributed Cartesian product, which returns the resulting pairs as a list of lists rather than all in a single list. This can be defined as follows:

$$dcp\ xs\ ys\ =\ [[(x, y)\ |\ y \leftarrow ys]\ |\ x \leftarrow xs]$$

This has the following type:

$$dcp :: [A] \rightarrow [B] \rightarrow [[(A, B)]]$$

Informally, we have:

$$\begin{aligned} &dcp\ [x_1, x_2, ..., x_n]\ [y_1, y_2, ..., y_m] \\ &= [[(x_1, y_1), (x_1, y_2), ..., (x_1, y_m)], [(x_2, y_1), (x_2, y_2), ..., (x_2, y_m)], ..., [(x_n, y_1), (x_n, y_2), ..., (x_n, y_m)]] \end{aligned}$$

The relationship between *dcp* and *cp* can be defined succinctly as follows:

$$cp\ xs\ ys\ =\ fold\ (+\!\!+)\ (dcp\ xs\ ys)$$

An alternative way to define *dcp* is in terms of *map*. Here we want to take each item from xs in turn, and pair that single item with every item in ys. So, given a function which pairs a single item to every item in a list (note the parameter order here):

$$\begin{aligned} pairtoall &:: [B] \rightarrow A \rightarrow [(A, B)] \\ pairtoall\ ys\ x &= map\ (pair\ x)\ ys \end{aligned}$$

Where *pair* is defined simply as follows:

$$pair\ a\ b\ =\ (a, b)$$

We could of course substitute pair for the default tupling function used in Haskell. That is to say:

$$pair \;=\; (,)$$

However, the word *pair* itself is probably more readable. Returning to *dcp*, given our function *pairtoall*, we can now write:

$$dcp\ xs\ ys \;=\; map\ (pairtoall\ ys)\ xs$$

We can of course bring the definition of *pairtoall* into *dcp* itself, firstly using a Lambda abstraction:

$$dcp\ xs\ ys \;=\; map\ (\lambda x \bullet map\ (pair\ x)\ ys)\ xs$$

Or alternatively, with a point-free style:

$$dcp\ xs\ ys \;=\; map\ (flip\ map\ ys \circ pair)\ xs$$

The function $flip$ simply reverses the order of another function's parameters. We have

$$flip :: (A \to B \to C) \to (B \to A \to C)$$

Which is then defined simply as:

$$flip\ f\ y\ x \;=\; f\ x\ y$$

Another alternative would be to use Haskell's convention for turning a function into an infix operator, that is:

$$f\ a\ b \;=\; a\ `f`\ b$$

As such, we have:

$$dcp\ xs\ ys \;=\; map\ ((`map`\ ys) \circ pair)\ xs$$

We may instead wish to use the binary operator version of *map*, which can be illustrated as follows:

$$map\ f\ xs \;=\; f * xs$$

This gives us one further alternative definition for *dcp*:

$$dcp\ xs\ ys \;=\; map\ ((*ys) \circ pair)\ xs$$

Or indeed, to change both to the infix version we have:

$$dcp\ xs\ ys \;=\; ((*ys) \circ pair)\ *\ xs$$

As a further alternative, we may wish to appeal to the $unfold$ family for a definition.

$$\begin{aligned} dcp\ xs\ ys &= unfoldr\ (f\ ys)\ null\ xs \\ &\quad where\ f\ ys\ (x:xs) = (map\ (pair\ x)\ ys, xs) \end{aligned}$$

However, in effect, it could be argued, all we are really doing here is using $unfoldr$ to implement the map scheme, and then using that to re-iterate one of our earlier definitions. A slight variation of this where ys is passed explicitly between instances of f may prove useful later on:

$$\begin{aligned} dcp\ xs\ ys &= unfoldr\ f\ (null \circ fst)\ (xs, ys) \\ &\quad where\ f\ (x:xs, ys) = (map\ (pair\ x)\ ys, (xs, ys)) \end{aligned}$$

Alternatively, we may wish to express this without the pattern matching:

$$\begin{aligned} dcp\ xs\ ys &= unfoldr\ f\ (null \circ fst)\ (xs, ys) \\ &\quad where\ f\ (xs, ys) = (map\ ((pair \circ head)\ xs)\ ys, (tail\ xs, ys)) \end{aligned}$$

Let us reassure ourselves formally that our function dcp is indeed a valid refinement of cp in distributed list terms.

$$\begin{array}{ccc} ([A],[B]) & \xrightarrow{uncurry\ cp} & [(A,B)] \\ \uparrow{\scriptstyle id} & & \uparrow{\scriptstyle abs_D} \\ ([A],[B]) & \xrightarrow{uncurry\ dcp} & [[(A,B)]] \end{array}$$

The proof of this is:

$$\begin{aligned} & (uncurry\ cp \circ id)\ ([x_1, x_2, ..., x_n], [y_1, y_2, ..., y_m]) && \{id\} \\ = \ & uncurry\ cp\ ([x_1, x_2, ..., x_n], [y_1, y_2, ..., y_m]) && \{def.\ id\} \\ = \ & cp\ [x_1, x_2, ..., x_n]\ [y_1, y_2, ..., y_m] && \{def.\ uncurry\} \\ = \ & [(x, y) \mid x \leftarrow [x_1, x_2, ..., x_n],\ y \leftarrow\ [y_1, y_2, ..., y_m]] && \{def.\ cp\} \\ = \ & fold\ (+\!\!+)\ [[(x, y) \mid y \leftarrow\ [y_1, y_2, ..., y_m]] \mid x \leftarrow [x_1, x_2, ..., x_n]] && \{list\ comp.\} \\ = \ & fold\ (+\!\!+)\ (dcp\ [x_1, x_2, ..., x_n]\ [y_1, y_2, ..., y_m]) && \{def.\ dcp\} \\ = \ & abs_D\ (dcp\ [x_1, x_2, ..., x_n]\ [y_1, y_2, ..., y_m]) && \{def.\ abs_D\} \\ = \ & abs_D\ (uncurry\ dcp\ ([x_1, x_2, ..., x_n], [y_1, y_2, ..., y_m])) && \{def.\ uncurry\} \\ = \ & (abs_D\ \circ\ uncurry\ dcp)\ ([x_1, x_2, ..., x_n], [y_1, y_2, ..., y_m]) && \{def.\ \circ\} \end{aligned}$$

7.14.1 Stream of Streams Output

In the first refinement of dcp we shall consider, we will output the result as a stream of streams. Here, assuming our inputs are both refined to streams, we have a function $sssdcp$, which has the following type:

$$sssdcp :: \lfloor A \rfloor \to \lfloor B \rfloor \to \lfloor \lfloor (A, B) \rfloor \rfloor$$

We have the following definition, a simple data refinement of our $unfold$ based definition for dcp:

$$\begin{aligned} sssdcp\ xs\ ys &= sunfoldr\ f\ (snull \circ fst)\ (xs, ys) \\ &\quad where\ f\ (xs, ys) = (smap\ ((pair \circ shead)\ xs)\ ys, (stail\ xs, ys)) \end{aligned}$$

A simple proof could be constructed to demonstrate that the following refinement property holds given two input streams s and t:

$$dcp\ (abs_S\ s)\ (abs_S\ t) = abs_{SS}\ (sssdcp\ s\ t)$$

Process Refinement

Looking at our $unfold$ oriented definition for the stream of streams output we can get an idea of the required behaviour for a process refinement in this setting. At each step - each application of f - we produce a stream of a value from xs paired with every value from ys. This implies we require the entirety of ys for every step of computation, and as such it will have to be read in as a preliminary stage, and then buffered locally. So in effect we have a process in two stages. The first stage is required to consume all of ys and store it locally. The second stage is to read from xs, one item at a time, and for each item output a stream containing that value from xs paired with every item from ys. In CSP terms we have something along the lines of the following:

$$\begin{aligned} SSSDCP = \ & \mu Y \bullet \ yin.eot\ ?\ True \rightarrow X \\ & \quad | \\ & \quad yin.value\ ?\ y \rightarrow ys := ys \mathbin{\widehat{+\!\!+}} \lfloor y \rfloor;\ Y \\ & \mu X \bullet \ xin.eot\ ?\ True \rightarrow out.eot\ !\ True \rightarrow SKIP \\ & \quad | \\ & \quad xin.value\ ?\ x \rightarrow Prd(smap\ (pair\ x)\ ys)[out.value/out];\ X \end{aligned}$$

Evidently, the sequential nature of this process may often prove far from ideal - it will require quadratic time - $O(nm)$ steps - to produce the output.

7.14.2 Vector of Streams Output

In the second refinement we shall consider, we shall output the result as a vector of streams. Here, assuming our inputs are both refined to streams, we have a function $svsdcp$, which has the following type:

$$svsdcp_n :: \lfloor A \rfloor \rightarrow \lfloor B \rfloor \rightarrow \langle \lfloor (A, B) \rfloor \rangle_n$$

In practice, the value n which determines the size of the output vector will be dictated by the size of the first input stream (of type $\lfloor A \rfloor$). Informally we have:

$$
\begin{aligned}
& svsdcp_n\ \lfloor x_1, x_2, ..., x_n \rfloor\ \lfloor y_1, y_2, ..., y_m \rfloor \\
= \quad & \langle \lfloor (x_1, y_1), (x_1, y_2), ..., (x_1, y_m) \rfloor, \\
& \lfloor (x_2, y_1), (x_2, y_2), ..., (x_2, y_m) \rfloor, \\
& ..., \\
& \lfloor (x_n, y_1), (x_n, y_2), ..., (x_n, y_m) \rfloor \rangle_n
\end{aligned}
$$

A simple proof could be constructed to demonstrate that the following refinement property holds given two input streams s (of length n) and t:

$$dcp\ (abs_S\ s)\ (abs_S\ t)\ =\ abs_{VS}\ (svsdcp_n\ s\ t)$$

Given that the size of the output vector of streams is fixed, based on the size of one of the inputs, we may also reasonably wish to consider refining one of the inputs to a vector also. This gives us the following alternative:

$$svsdcp'_n :: \langle A \rangle_n \rightarrow \lfloor B \rfloor \rightarrow \langle \lfloor (A, B) \rfloor \rangle_n$$

Let us consider a definition for $svsdcp_n$ in terms of the $unfoldr$ pattern. As a direct data refinement of our $unfold$ based definition for dcp we have:

$$
\begin{aligned}
svsdcp_n\ xs\ ys \quad = \quad & vunfoldr\ pr\ (snull \circ fst)\ (xs, ys) \\
& where\ pr\ (xs, ys)\ =\ (smap\ ((pair \circ shead)\ xs)\ ys, (stail\ xs, ys))
\end{aligned}
$$

Process Refinement

We shall work with our $unfold$ based definition for our function $svsdcp_n$. Here the important functionality is contained within the characteristic function pr. Let us consider this.

$$pr\ (xs, ys)\ =\ (smap\ ((pair \circ shead)\ xs)\ ys, (stail\ xs, ys))$$

The type of pr can be given as follows:

$$pr :: (\lfloor A \rfloor, \lfloor B \rfloor) \rightarrow (\lfloor (A, B) \rfloor,\ (\lfloor A \rfloor, \lfloor B \rfloor))$$

This may be a little awkward to read due to the nested tupling and the overall number of brackets, but what we have, in effect, is a function which inputs two streams and outputs three. The roles of each of these inputs will hopefully become clear once we begin to consider a process refinement for this function. This is illustrated in Figure 7.13.

Here we have a process, PR, which inputs two streams and outputs three. The alphabet of the process is as follows:

$$\alpha PR\ =\ \{xin :: \underline{\lfloor A \rfloor},\ yin :: \underline{\lfloor B \rfloor},\ xout :: \underline{\lfloor A \rfloor},\ yout :: \underline{\lfloor B \rfloor},\ out :: \underline{\lfloor (A, B) \rfloor}\}$$

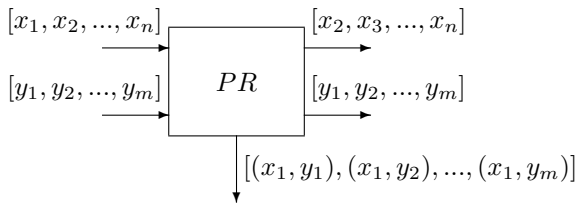


Figure 7.13: The process PR.

The conduits xin and $xout$ are used for conveying the stream xs. The process PR is responsible for taking the head of this incoming stream and storing it locally (as the value x) then passing the remainder (i.e. the tail) on unchanged. The conduits yin and $yout$ are used for conveying the stream ys. This must pass through the process in its entirety and unchanged. However, each reception of a value from this stream will prompt transmission of a value in the stream carried by conduit out. Each transmission in out will constitute the value x paired with a value y received on yin. We effectively have a process in two stages then. Firstly we have the business of acquiring our value x and then ferrying on the remainder of ys. Secondly we then have the main task of producing pairs based on this value x and incoming values from ys. Such a process can be defined in CSP as follows:

$$
\begin{array}{lll}
PR & = & xin.value\ ?\ x \rightarrow SCOPY[xin/in, xout/out]; \\
 & \mu X \bullet & in.eot\ ?\ any \rightarrow outd.eot\ !\ True \rightarrow outr.eot\ !\ True \rightarrow SKIP \\
 & & | \\
 & & in.value\ ?\ y \rightarrow outd.value\ !\ (x, y) \rightarrow outr.value\ !\ y \rightarrow X
\end{array}
$$

In practice, we may wish to provide a slightly more general version that abstracts away the manner in which the value x is received. This is depicted in Figure 7.14.

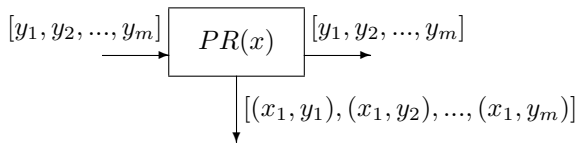


Figure 7.14: The process PR(x).

The process $PR(x)$ has the following alphabet:

$$\alpha PR(x) \ = \ \{yin :: \underline{\lfloor B \rfloor},\ yout :: \underline{\lfloor B \rfloor},\ out :: \underline{\lfloor (A, B) \rfloor}\}$$

We can supply the following definition - effectively just the second stage of PR, given above:

$$\begin{aligned} PR(x) \quad = \quad \mu X \bullet \quad & in.eot \ ? \ any \rightarrow outd.eot \ ! \ True \rightarrow outr.eot \ ! \ True \rightarrow SKIP \\ & | \\ & in.value \ ? \ y \rightarrow outd.value \ ! \ (x, y) \rightarrow outr.value \ ! \ y \rightarrow X \end{aligned}$$

Our overall network is then effectively a series of these components composed together in parallel. In our process refinement for this setting, we are provided with two options for how the first set of input (xs) is to be delivered. This may arrive as a stream, as in $svsdcp_n$, or as a vector, as in $svsdcp'_n$. We shall only examine here the definition of the process from the point at which xs has already been received, and, as such, each instance of PR has already received its corresponding x value. Our process $SVSDCP$ is illustrated in Figure 7.15.

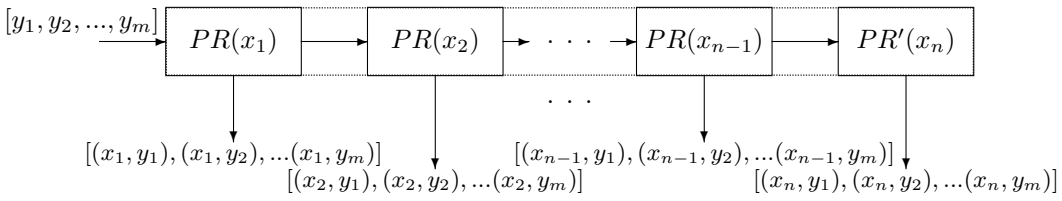


Figure 7.15: A process producing a distributed Cartesian product as a vector of streams.

Let us now consider a definition for this process in CSP. As already noted this can be constructed essentially by composing together a number of instances of our component process PR. Generally speaking, each instance of PR will be 'loaded' with x - a value from xs. It will then receive the entire stream ys, and output a stream of x paired with each item in ys, as well as echoing the input stream for the next component along to use. As usual with these kinds of definitions we shall encounter slight deviations from the norm at either end. The first instance of PR will have the responsibility of reading the input to the process as a whole. The last instance of PR will have to deal with the fact that it has no successor to echo the input to. This can proceed as follows:

$$\begin{aligned} SVSDCP_n \quad &= \quad PR_{initial} \ || \left(\begin{array}{l} n-1 \\ \quad || \quad PR[mid_{i-1}/in, out_i/outd, mid_i/outr] \\ i = 2 \end{array} \right) || \ PR_{final} \\ PR_{initial} \quad &= \quad PR[in/in, out_1/outd, mid_1/outr] \\ PR_{final} \quad &= \quad PR'[mid_{n-1}/in, out_n/outd] \end{aligned}$$

Here PR' is an adapted version of PR which is not responsible for echoing the input stream.

$$\begin{aligned} PR'(x) \quad = \quad \mu X \bullet \quad & in.eot \ ? \ any \rightarrow outd.eot \ ! \ True \rightarrow SKIP \\ & | \\ & in.value \ ? \ y \rightarrow outd.value \ ! \ (x, y) \rightarrow X \end{aligned}$$

7.14.3 Stream of Vectors Output

In this refinement the output is then produced as a stream of vectors. Assuming we refine both inputs to streams we have a function *ssvdcp*, which has the following type:

$$ssvdcp_n :: \lfloor A \rfloor \rightarrow \lfloor B \rfloor \rightarrow \lfloor \langle (A, B) \rangle_n \rfloor$$

Let us consider a definition for $svsdcp_n$ in terms of the $unfoldr$ pattern. As a direct data refinement of our $unfold$ based definition for *dcp* we have:

$$\begin{array}{lcl} ssvdcp_n\ xs\ ys & = & sunfoldr\ pr\ (snull \circ fst)\ (xs, ys) \\ & & where\ pr\ (xs, ys)\ =\ (vmap\ ((pair \circ vhead)\ xs)\ ys, (vtail\ xs, ys)) \end{array}$$

A simple proof could be constructed to demonstrate that the following refinement property holds given two input streams s (of length n) and t:

$$dcp\ (abs_S\ s)\ (abs_S\ t)\ =\ abs_{SV}\ (ssvdcp_n\ s\ t)$$

Process Refinement

Here we have a similar pattern to that for the stream of streams output. Here, however, rather than outputting a stream for each value of xs input, we instead output a vector.

$$\begin{array}{lcll} SSVDCP & = & \mu Y \bullet & yin.eot\ ?\ True \rightarrow X \\ & & & | \\ & & & yin.value\ ?\ y \rightarrow ys\ :=\ ys \widetilde{+\!\!+_{k,1}} \langle y \rangle_1;\ Y \\ & & \mu X \bullet & xin.eot\ ?\ True \rightarrow out.eot\ !\ True \rightarrow SKIP \\ & & & | \\ & & & xin.value\ ?\ x \rightarrow Prd(vmap_m\ (pair\ x)\ ys)[out.value/out];\ X \end{array}$$

7.14.4 Vector of Vectors Output

In the final refinement we have the output being produced as a vector of vectors. Here, assuming we refine both inputs to streams, we have a function *svvdcp*, which has the following type:

$$svvdcp_{n,m} :: \lfloor A \rfloor \rightarrow \lfloor B \rfloor \rightarrow \langle \langle (A, B) \rangle_m \rangle_n$$

Process Refinement

This would follow a similar patten to the vector of streams output, except that, obviously, the components of the output vector would be vectors rather than streams. As always, we may question the usefulness of a vector of vectors output, given the quadratic requirement on processing resources.

7.15 Transpose

The function *transpose* takes in a list of lists as input. Such a structure can be consider as a two dimensional array with rows and columns. It returns a list of lists where the columns in the input are rows in the output, and vice verse. This has the following type:

$$transpose \ :: \ [[A]] \rightarrow [[A]]$$

Informally, the functionality can be defined as follows:

$$\begin{aligned} &transpose\ [[x_{11}, x_{12}, x_{13}, ..., x_{1n}], [x_{21}, x_{22}, x_{23}, ..., x_{2n}], ..., [x_{m1}, x_{m2}, x_{m3}, ..., x_{mn}]] \\ &\quad = \ [[x_{11}, x_{21}, x_{31}, ..., x_{m1}], [x_{12}, x_{22}, x_{32}, ..., x_{m2}], ..., [x_{1n}, x_{2n}, x_{3n}, ..., x_{mn}]] \end{aligned}$$

Here the value n represents the width of the input list of lists, i.e. the length of each sub-list. The value m represents the height of the input list of lists, or in other words the count of sub-lists. So, the input structure has dimensions $(n \times m)$ and the output structure has dimensions $(m \times n)$.

A formal definition for this function is typically given recursively, as follows:

$$\begin{aligned} transpose\ [] &= [] \\ transpose\ ([] : xss) &= transpose\ xss \\ transpose\ ((x : xs) : xss) &= (x : [h \mid (h : t) \leftarrow xss]) : \\ &\quad transpose\ (xs : [t \mid (h : t) \leftarrow xss]) \end{aligned}$$

The pattern matching in the list comprehensions above works as a filter: only lists matching the pattern $(h : t)$ in xss are considered. In effect, empty lists are being filtered out. As such we could rewrite the above as follows:

$$\begin{aligned} transpose\ [] &= [] \\ transpose\ ([] : xss) &= transpose\ xss \\ transpose\ ((x : xs) : xss) &= (x : [head\ ys \mid ys \leftarrow xss, not\ (null\ ys)]) : \\ &\quad transpose\ (xs : [tail\ ys \mid ys \leftarrow xss, not\ (null\ ys)]) \end{aligned}$$

These list comprehensions can of course be replaced altogether with *map* and $filter$ operations:

$$\begin{aligned} transpose\ [] &= [] \\ transpose\ ([] : xss) &= transpose\ xss \\ transpose\ ((x : xs) : xss) &= (x : map\ head\ (filter\ (not \circ null)\ xss)) : \\ &\quad transpose\ (xs : map\ tail\ (filter\ (not \circ null)\ xss)) \end{aligned}$$

We can simplify this definition a little further by removing some of the pattern matching. This will also allow us to simplify the right hand side of the third line. Given that the first two lines cover all other possible alternative cases, we can make the case in the third line implicit:

$$\begin{aligned} transpose\ [] &= [] \\ transpose\ ([] : xss) &= transpose\ xss \\ transpose\ xss &= map\ head\ (filter\ (not \circ null)\ xss) : \\ &\quad transpose\ (map\ tail\ (filter\ (not \circ null)\ xss)) \end{aligned}$$

We may wish to introduce a local definition to avoid the repeated calculation of the $filter$ operation, as it is the same in both instances.

$$\begin{array}{lcl} transpose\ [] & = & [] \\ transpose\ ([] : xss) & = & transpose\ xss \\ transpose\ xss & = & map\ head\ xss' : \\ & & transpose\ (map\ tail\ xss') \\ & & where\ xss' = filter\ (not \circ null)\ xss \end{array}$$

We may wish to remove the pattern matching altogether. The main purpose for the second line of the above definition is to stop unwanted empty lists from appearing in the output - they are simply discarded altogether. Where the input consists of a mixture of empty lists and non-empty lists, this shall be dealt with by the $filter$ operation. It is only where the input consists exclusively of empty lists that we will have a problem. With a correctly constructed condition we can avoid this situation completely. The condition *all null* should achieve this. Where this applied to the input returns true, we shall simply return an empty list. Let us assure ourselves that the required conditions imposed in our pattern matching based definition shall still hold. First, we have our original base case:

$$\begin{array}{ll} & transpose\ [] = [] \\ & \{id\} \\ \equiv & if\ all\ null\ []\ then\ []\ else\ ... = [] \\ & \{def.\ transpose\} \\ \equiv & if\ True\ then\ []\ else\ ... = [] \\ & \{all\ null\ [] = True\} \\ \equiv & [] = [] \\ & \{if\} \\ \equiv & True \\ \{\} & \end{array}$$

Next, the case where we have an empty list at the head of the input. First let us assume the remainder of the list is either empty or consists only of empty lists. That is to say *all null xss* will be True.

$$\begin{array}{ll} & transpose\ ([] : xss) = transpose\ xss \\ & \{id\} \\ \equiv & if\ all\ null\ ([] : xss)\ then\ []\ else\ ... = if\ all\ null\ xss\ then\ []\ else\ ... \\ & \{def.\ transpose\} \\ \equiv & [] = [] \\ & \{all\ null\ xss = True\} \\ \equiv & True \\ & \{\} \end{array}$$

Finally we have the case where the input consist of an empty list at the head, followed by a group xss containing one or more non-empty lists. That is to say *all null xss* will be False.

$$
\begin{array}{l}
\quad transpose\ ([] : xss)\ =\ transpose\ xss \\
\quad \{id\} \\
\\
\equiv \quad \begin{array}{l} if\ all\ null\ ([] : xss) \\ then\ [] \\ else\ map\ head\ xss' : \\ transpose\ (map\ tail\ xss') \\ where\ xss' \\ \quad =\ filter\ (not \circ null)\ ([] : xss) \end{array} \quad = \quad \begin{array}{l} if\ all\ null\ xss \\ then\ [] \\ else\ map\ head\ xss' : \\ transpose\ (map\ tail\ xss') \\ where\ xss' \\ \quad =\ filter\ (not \circ null)\ xss \end{array} \\
\quad \{def.\ transpose\} \\
\\
\equiv \quad \begin{array}{l} map\ head\ xss' : \\ transpose\ (map\ tail\ xss') \\ where\ xss' \\ \quad =\ filter\ (not \circ null)\ ([] : xss) \end{array} \quad = \quad \begin{array}{l} map\ head\ xss' : \\ transpose\ (map\ tail\ xss') \\ where\ xss' \\ \quad =\ filter\ (not \circ null)\ xss \end{array} \\
\quad \{all\ null\ xss\ =\ False\} \\
\\
\equiv \quad \begin{array}{l} map\ head\ xss' : \\ transpose\ (map\ tail\ xss') \\ where\ xss' \\ \quad =\ filter\ (not \circ null)\ xss \end{array} \quad = \quad \begin{array}{l} map\ head\ xss' : \\ transpose\ (map\ tail\ xss') \\ where\ xss' \\ \quad =\ filter\ (not \circ null)\ xss \end{array} \\
\quad \{filter\ (not \circ null)\} \\
\\
\equiv \quad True \\
\quad \{\}
\end{array}
$$

Thus we arrive at the following definition:

$$
\begin{array}{lll}
transpose\ xss & = & if\ all\ null\ xss\ then\ [] \\
 & & else\ map\ head\ xss' : transpose\ (map\ tail\ xss') \\
 & & where\ xss'\ =\ filter\ (not \circ null)\ xss
\end{array}
$$

Close inspection of the above definition should reveal it is in fact an instance of the $unfoldr$ higher order function. Thus we can give an alternative definition for *transpose* as follows:

$$transpose\ =\ unfoldr\ (fork\ (map\ head, map\ tail) \circ filter\ (not \circ null))\ (all\ null)$$

We can also define *transpose* in terms of the left variant of $unfold$:

$$transpose \;=\; unfoldl\ (fork\ (map\ init, map\ last) \circ filter\ (not \circ null))\ (all\ null)$$

Given two further 'primitive' transformations on two dimensional arrays:

$$\begin{aligned} flipv &= reverse \\ fliph &= map\ reverse \end{aligned}$$

We may then define a whole set of transformation functions as follows:

$$\begin{aligned} rotate90 &= fliph \circ transpose \\ rotate180 &= fliph \circ flipv \\ rotate270 &= flipv \circ transpose \\ transverse &= fliph \circ flipv \circ transpose \end{aligned}$$

Let us now consider how our function *transpose* may be refined in our various settings. In certain settings, particularly where vectors are involved, we may want to work with a slightly different definition of *transpose*. The function $transpose'$ operates in the same way, but is only defined for the case where all sub-lists in the input list of lists are of equal length. The definition is therefore almost identical to that for *transpose*, except that it lacks the $filter$ operation to remove empty lists. We have:

$$transpose' \;=\; unfoldr\ (fork\ (map\ head, map\ tail))\ (all\ null)$$

7.15.1 Stream of Streams to Stream of Streams

In this, a fully sequential refinement, we shall both input and output a stream of streams. Thus we have the function:

$$sssstranspose :: \lfloor\lfloor A \rfloor\rfloor \rightarrow \lfloor\lfloor A \rfloor\rfloor$$

We may provide a definition for this as follows:

$$\begin{aligned} sssstranspose \;=\; & sunfoldr\ (f \circ p)\ (sall\ snull) \\ & where\ f = fork\ (smap\ shead, smap\ stail) \\ & \qquad p = sfilter\ (not \circ snull) \end{aligned}$$

7.15.2 Vector of Vectors to Vector of Vectors

Here we have both input and output as a vector of vectors. The type of this function clearly illustrates the change in dimensions of the input structure as it is processed to form the output.

$$vvvvtranspose_{n,m} :: \langle\langle A\rangle_m\rangle_n \rightarrow \langle\langle A\rangle_n\rangle_m$$

7.15.3 Vector of Streams to Vector of Streams

Here we have both input and output as a vector of streams. However, given the nature of *transpose*, the size of the output vector of streams will not necessarily be the same as the input. Given the use of vectors in this refinement, we shall actually derive a refinement for $transpose'$ rather than the more general *transpose*. We have:

$$vsvstranspose'_{n,m} :: \langle \lfloor A \rfloor \rangle_n \rightarrow \langle \lfloor A \rfloor \rangle_m$$

We may provide a definition for this as follows:

$$\begin{aligned} vsvstranspose'_{n,m} &= vunfoldr_m\ f\ (vall_n\ snull) \\ &\quad where\ f = fork\ (v2smap_n\ shead, vmap_n\ stail) \end{aligned}$$

7.15.4 Stream of Vectors to Stream of Vectors

Here we have both input and output as a stream of vectors. Again, given the nature of *transpose*, the size of the output stream of vectors will not necessarily be the same as the input.

$$svsvtranspose_{n,m} :: \lfloor \langle A \rangle_m \rfloor \rightarrow \lfloor \langle A \rangle_n \rfloor$$

7.15.5 Stream of Vectors to Vector of Streams

A further refinement we may be interested in has the input as a stream of vectors and the output as a vector of streams.

$$svvstranspose_m :: \lfloor \langle A \rangle_m \rfloor \rightarrow \langle \lfloor A \rfloor \rangle_m$$

7.15.6 Vector of Streams to Stream of Vectors

The final refinement for *transpose* we shall examine has the input as a vector of streams, and the output as a stream of vectors.

$$vssvtranspose_n :: \langle \lfloor A \rfloor \rangle_n \rightarrow \lfloor \langle A \rangle_n \rfloor$$

7.16 Segments

The function *segments* takes in a list, and returns a list of all segments of that list.

$$segments :: [A] \rightarrow [[A]]$$

Another way to describe this collection is as every initial segment of every final segment of the input list. This corresponds to mapping the function *inits* to the results of the function *tails*, and then concatenating the results. We have:

$$segments \;=\; fold\ (+\!\!+) \circ map\ inits \circ tails$$

As an example, we have:

$$segments\ [x_1, x_2, x_3] \;=\; [[], [x_1], [x_1, x_2], [x_1, x_2, x_3], [], [x_2], [x_2, x_3], [], [x_3], []]$$

To help illustrate this behaviour, let us also show the results without concatenation, so we can see where each part comes from:

$(map\ inits \circ tails)\ [x_1, x_2, x_3] =$	
$[$	
$\quad [\,], [x_1], [x_1, x_2], [x_1, x_2, x_3]\ ],$	$= inits\ [x_1, x_2, x_3]$
$\quad [\,], [x_2], [x_2, x_3]\ ],$	$= inits\ [x_2, x_3]$
$\quad [\,], [x_3]\ ],$	$= inits\ [x_3]$
$\quad [\,]\ ]$	$= inits\ []$
$]$	

For an input list of size n, the size of the output (in terms of number of lists) returned by $segments^+$ is $O(n^2)$. To be precise, given an input list of size n, we have the following number of lists in the output:

$$\frac{(n+1)^2 + n + 1}{2}$$

In many algorithms we may find the repeated occurrence of the empty list here somewhat cumbersome. As such we have a variant $segments^+$, which can be defined analogously to $segments$, but using $tails^+$ and $inits^+$ instead of $tails$ and $inits$. We have:

$$segments^+ \;=\; fold\ (+\!\!+) \circ map\ inits^+ \circ tails^+$$

This has the same type as $segments$:

$$segments^+ :: [A] \to [[A]]$$

The results when applied to a list are identical to that for $segments$, except that no empty lists appear in the output. For example:

$$segments^+\ [x_1, x_2, x_3] \;=\; [[x_1], [x_1, x_2], [x_1, x_2, x_3], [x_2], [x_2, x_3], [x_3]]$$

For an input list of size n, the size of the output (in terms of number of lists) returned by $segments^+$ is $O(n^2)$. To be precise, given an input list of size n, we have the following number of lists in the output:

$$\frac{n^2 + n}{2}$$

7.16.1 Stream to Stream of Streams

In this refinement we input a stream, and output the segments as a stream of streams.

$$ssssegments :: \lfloor A \rfloor \to \lfloor \lfloor A \rfloor \rfloor$$

We can define this as follows:

$$ssssegments \;=\; sfold\ (\widehat{+\!\!+})\circ smap\ sssinits \circ ssstails$$

7.16.2 Stream to Vector of Streams

In this refinement we input a stream, and output the segments as a vector of streams.

$$svssegments :: \lfloor A \rfloor \to \langle \lfloor A \rfloor \rangle_n$$

We can define this as follows:

$$svssegments \;=\; vfold\ (\widehat{+\!\!+})\circ vmap\ sssinits \circ svstails$$

7.17 Splits

The function *splits* takes in a list, and returns all the possible ways to partition the input list into two parts. Every pair of lists in the output should result in the input list when concatenated back together. That is to say:

$$(all\ (== xs)\circ map\ (uncurry\ (+\!\!+))\circ splits)\ xs \;=\; True$$

We have:

$$\begin{aligned} &splits\ [a_1, a_2, \cdots, a_n] \\ &\quad = [\ \ ([],\ [a_1, a_2, \cdots, a_n]),\ \ ([a_1],\ [a_2, \cdots, a_n]),\ \ ([a_1, a_2],\ [a_3, \cdots, a_n]), \\ &\qquad \cdots, \\ &\qquad ([a_1, a_2, \cdots, a_{n-1}],\ [a_n]),\ ([a_1, a_2, \cdots, a_n],\ [])\ ] \end{aligned}$$

This function has the following type:

$$splits :: [A] \to [([A], [A])]$$

We may also wish to consider a variant $splits^+$, which does not contain the empty list in its output. Informally, we have:

$$\begin{aligned} &splits^+\ [a_1, a_2, \cdots, a_n] \\ &\quad = [\ \ ([a_1],\ [a_2, \cdots, a_n]),\ \ ([a_1, a_2],\ [a_3, \cdots, a_n]), \cdots, ([a_1, a_2, \cdots, a_{n-1}],\ [a_n])\ ] \end{aligned}$$

We can provide a definition for *splits* in terms of *take* and *drop*, as follows:

$$\begin{aligned} splits\ xs &= map\ (sp\ xs)\ [0\ ..\ length\ xs] \\ &\quad where\ sp\ xs\ n\ =\ (take\ n\ xs,\ drop\ n\ xs) \end{aligned}$$

Similar we just have to modify the bounds of the enumeration in the above definition to create a definition for $splits^+$.

$$\begin{aligned} splits^+\ xs &= map\ (sp\ xs)\ [1\ ..\ (length\ xs) - 1] \\ &\quad where\ sp\ xs\ n\ =\ (take\ n\ xs,\ drop\ n\ xs) \end{aligned}$$

Our *splits* functions do in fact come under the broad umbrella of the $unfold$ family. In our definition of $unfoldr$, (see Section 6.4) however, the manner in which the end condition is dealt with does not suit the functionality of *splits* in an ideal fashion. To illustrate this consider the following potential definition:

$$\begin{aligned} splits'\ xs &= unfoldr\ f\ (null \circ fst)\ (xs, []) \\ &\quad where\ f\ (xs, ys)\ =\ ((xs, ys),\ (init\ xs,\ last\ xs\ :\ ys)) \end{aligned}$$

Let us consider applying the above definition, to the list of values $[1, 2, 3]$. We have:

$$splits'\ [1, 2, 3]\ =\ [\ ([1, 2, 3], []),\ ([1, 2], [3]),\ ([1], [2, 3])\ ]$$

Crucially we are missing the last element in the output - the pair of the empty list followed by the complete list. Of course, we can remedy this situation by adding this element explicitly into the definition:

$$\begin{aligned} splits\ xs &= (unfoldr\ f\ (null \circ fst)\ (xs, []))\ +\!\!+\ [([], xs)] \\ &\quad where\ f\ (xs, ys)\ =\ ((xs, ys),\ (init\ xs,\ last\ xs\ :\ ys)) \end{aligned}$$

Another we to approach an $unfold$ oriented definition is to pass each characteristic function a value k, as used with the original *take* / *drop* approach. Here we have the following:

$$\begin{aligned} splits\ xs &= unfoldr\ f\ p\ (0, xs) \\ &\quad where\ f\ (k, xs) &= ((take\ k\ xs,\ drop\ k\ xs),\ (k + 1, xs)) \\ &\qquad p(k, xs) &= k\ >\ length\ xs \end{aligned}$$

7.17.1 Vector of Streams Output

Assuming we take a stream as input, we have the following type definition:

$$svssplits_n :: \lfloor A \rfloor \rightarrow \langle (\lfloor A \rfloor, \lfloor A \rfloor) \rangle_{n+1}$$

Alternatively, taking a vector as input we have:

$$vvssplits_n :: \langle A \rangle_n \rightarrow \langle (\lfloor A \rfloor, \lfloor A \rfloor) \rangle_{n+1}$$

Similarly for the non empty list variant we have:

$$\begin{aligned} svssplits_n^+ &:: \lfloor A \rfloor \rightarrow \langle (\lfloor A \rfloor, \lfloor A \rfloor) \rangle_{n+1} \\ vvssplits_n^+ &:: \langle A \rangle_n \rightarrow \langle (\lfloor A \rfloor, \lfloor A \rfloor) \rangle_{n+1} \end{aligned}$$

Process Refinement

We shall base our refinement on the *take / drop* oriented specification. Let us consider a single component process $SP(k)$. This is depicted in Figure 7.16. We have the following alphabet:

$$\alpha\, SP(k) \;=\; \{in :: \underline{\lfloor A \rfloor},\; outr :: \underline{\lfloor A \rfloor},\; outda :: \underline{\lfloor A \rfloor},\; outdb :: \underline{\lfloor A \rfloor}\}$$

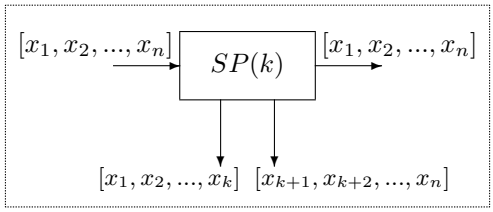


Figure 7.16: The process SP(k).

The process $SP(k)$ has the task of taking in a stream, xs, echoing it on one of its output conduits (*outr*), and then splitting it over the other two output conduits (*outda* and *outdb*). The stream should be split using the value k - the first k values to be transmitted on *outda*, and all those after the first k to be transmitted on *outdb*. In CSP terms we have the following process:

$$\begin{array}{lll} SP(k) & = & \text{if} \quad (k > 0) \\ & & \text{then} \quad in.value\ ?\ x \rightarrow outr.value\ !\ x \rightarrow outda.value\ !\ x \rightarrow SP(k-1) \\ & & \text{else} \quad outda.eot\ !\ True \rightarrow SDUP[in/in, outr/out1, outdb/out2] \end{array}$$

The process $SDUP$ used here simply echoes all values on its input stream to its two output streams. It can be defined as follows (see also Section 4.7):

$$\begin{array}{lll} SDUP & = \mu\, X \bullet & in.eot\ ?\ any \rightarrow out1.eot\ !\ True \rightarrow out2.eot\ !\ True \rightarrow SKIP \\ & & | \\ & & in.value\ ?\ x \rightarrow out1.value\ !\ x \rightarrow out2.value\ !\ x \rightarrow X \end{array}$$

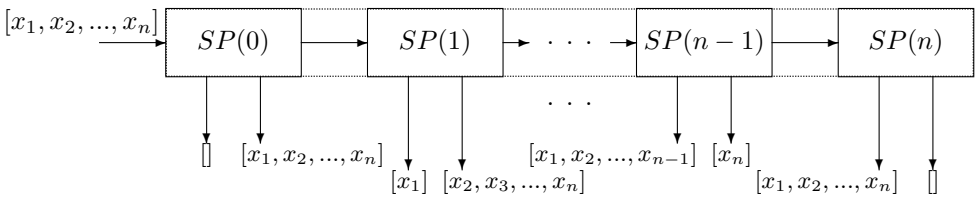


Figure 7.17: The process VSSPLITS.

We then have the now familiar task of composing together a number of instances of this component together in parallel to create our process refining the functionality of *splits*. This is depicted in Figure 7.17. In CSP terms we have the following:

$$
\begin{aligned}
VSSPLITS_n &= SP_{initial} \parallel \left(\begin{array}{c} n-1 \\ \parallel \quad SP_{normal} \\ i=1 \end{array} \right) \parallel SP_{final} \\
SP_{normal} &= SP[mid_i/in, outa_{i+1}/outda, outb_{i+1}/outdb, mid_{i+1}/outr] \\
SP_{initial} &= SP[in/in, outa_1/outda, outb_1/outdb, mid_1/outr] \\
SP_{final} &= SCOPY[mid_n/in, outa_{n+1}/outda];\ outb_{n+1}.eot\ !\ True \rightarrow SKIP
\end{aligned}
$$

7.18 Summary

We have explored a large selection of functions for processing lists, and seen how they can be implemented in stream and vector terms. One particular result of this exploration has been in the recognition of the widespread applicability of the $unfold$ pattern. In this chapter we have seen how this scheme can be used to express a number of combinatorial list processing functions, such as *inits* and *tails*, Cartesian product, *transpose* and *splits*. It is the vector interpretation of the unfold pattern which is key to providing scalable implementations for many of these components which deal in quadratic sized output.

Chapter 8

The Refinement Procedure

8.1 Introduction

In this chapter we describe the refinement procedure - the process through which a developer would go to derive an implementation using this methodology.

8.2 Procedure

There are a number of steps involved in the refinement procedure, as illustrated in Figure 8.1. We look at each of the steps in detail below.

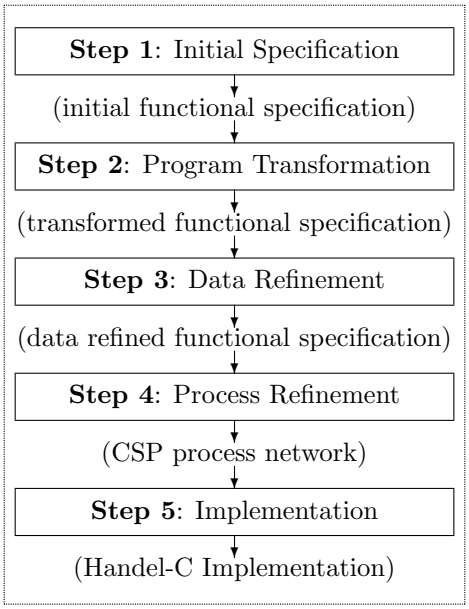


Figure 8.1: An illustration of the steps in the refinement procedure.

8.2.1 Step 1 - Initial Specification

The first task for the developer is to formulate an initial functional specification for the algorithm in Haskell. Typically this will often be an intuitive specification, which may not be the most efficient, and may not initially be in a form best suited to the remainder of the refinement process. Both of these issues can be dealt with in the next step.

It is at this stage that proofs can be constructed and properties about the specification can be asserted, if necessary. Alternatively, another approach may be to formally derive this functional specification from a specification in another framework such as Z - examples of this kind of approach can be found in [7, 12]

8.2.2 Step 2 - Program Transformation

Depending on the structure of the initial specification, some program transformation may be required. An ideal form to begin refinement from is one which is a composition of stages, wherein the maximum possible use of higher order (and list processing) functions is made. Additionally we may also strive to make efficiency improvements here, particularly where the initial specification is worse than quadratic time. BMF provides a rich framework for program transformation, and much work has been carried out in using this framework to improve the efficiency of algorithms [16, 31, 19].

8.2.3 Step 3 - Data Refinement

It is at this stage that the developer should make choices about how the types in the specification are to be refined, as discussed in Chapter 3. Single dimensional structures (i.e. lists) can be refined to either streams or vectors, two dimensional structures (lists of lists) are presented with the four combinations of the two basic types as options, and so on.

It is here that the developer begins to select components from the library presented in this work. The higher order (and list processing) functions present in the specification from the previous step will require appropriate data refined versions at this step. Indeed, the library may in fact assist the developer in making decisions about how data refinement is best to be carried out. For example, if the first stage of the algorithm employs a Cartesian product, the developer may determine, upon browsing the available refinements of *cp* in the library, that the vector of streams output is the most efficient approach. As another example, presence of $filter$ in the specification may, on consulting the library, allow us to decide that a straightforward vector approach is not appropriate for that part of the algorithm. Naturally, decisions about the first stage of the algorithm will then have an impact on the data refinement decisions made for the second stage, and so on. Each stage must accept input in the same format as the output of the previous stage. In fact we may often see the decisions about how to refine the types of the first stage determining the eventual structure of the

entire resulting network.

It is possible the developer may wish to consider several alternative data refinement strategies at this stage. This is most likely to be the case where varying degrees of parallelism can be achieved through alternative uses of vectors, streams and combinations thereof. From here on the developer would then choose one particular data refined specification with which to proceed onto the next step. If it turns out further down the line that this does not meet the efficiency or resource requirements in the implementation, then the developer may return to this stage and try another alternative. In effect we have a potentially iterative process from here on.

One particularly useful construct for controlling the extent of parallelism is the distributed list. See Section 3.6.4 for a discussion of refining distributed lists or Section 5.2.6 for an example of how they can be used in conjunction with *map*. Here we have an intermediate data refinement step, whereby lists are partitioned into lists of lists. These two dimensional structures can then be refined to efficient communication patterns which provide scope for parallelism, such as vectors of streams. This gives us an extra tuning control on the extent of parallelism - by adjusting the partition sizes we can go from modest parallelism to entirely parallel implementations.

Note also that we do not have to 'set in stone' the sizes of any fixed length structures (i.e. vectors) we are using at this point. Components in the library which work with vectors are all parameterised - typically they have a subscript n (take for example $vmap_n$). Indeed this parametrisation continues right the way through to Handel-C. So from here on we can develop an algorithm which is in effect parameterised on the problem size. It is only at the last minute - when we come to compile our resulting Handel-C implementation - that we need to set the size we are dealing with.

8.2.4 Step 4 - Process Refinement

Here we take our data refined functional specification, the result of the previous step, and refine it to a network of CSP processes. The functional, data refined versions of components from the library which appear in the specification can be refined to their already proven CSP counterparts in the library. Here function composition is refined to process piping, and function application to process feeding, all as discussed in Chapter 4. Parts of the specification which are not taken from the library - the bespoke functionality - will need to be refined by the developer into CSP here. Assuming the developer adheres to the rules for process refinement presented in this work for any such bespoke functionality, then we will be able to assert that the resulting network as a whole is correct by construction.

At this point we have a network of CSP processes. Reasoning about behaviour and concurrency should be carried out here, if necessary. For example, we may wish to visualise the network [8] or perform model checking [30] on certain aspects of it.

8.2.5 Step 5 - Implementation

Finally we take our CSP definition and implement it in Handel-C. Again, components from the library can be implemented by way of their counterpart Handel-C implementations, also provided in the library. Any bespoke functionality introduced will need to be implemented by the developer here based on the CSP definition created in the previous step.

The Handel-C implementation can now be compiled, and the result of the compiler (typically an EDIF file) can be taken and processed by a suite of place and route tools. The result of this is a programming file which can be used to configure the FPGA, and the algorithm can then be executed.

At this point the developer may wish to iterate certain parts of the refinement procedure. There are two possible scenarios which may prompt this.

Firstly, it may occur that the resulting design is in fact too large to implement on the target FPGA device. This eventuality will typically manifest itself by way of the place and route tools failing to map the design. Where the algorithm is parameterised on problem size - some value n - the developer may wish to adjust this in the Handel-C implementation and try again. The other alternative is to return to the data refinement step and consider alternatives which may result in a less complex design - typically achieved by reducing the use of parallelism. Certain parts of the algorithm which employed vectors may have to be substituted for stream implementations, although this will have an obvious impact on efficiency. Where distributed lists have been employed in the data refinement we can simply adjust the partition sizes - larger partition sizes will correspond to reduced parallelism and therefore smaller designs.

On the flip side it may also occur that the resulting design does not make full use of the resources available on the target FPGA. Again, a return to the data refinement step, or adjusting of the problem/partition sizes where applicable may allow for a more efficient algorithm with the available scope for parallelism better exploited.

8.3 Example

Let us consider a simple problem to illustrate this refinement process. Consider a function, *calc*, which is required to sum the squares of all negative numbers in a list. For example:

$$calc\ [3, -6, 2, -7, 3, -5, 2]\ =\ 110$$

8.3.1 Step 1 - Initial Specification

Our function *calc* is required to take in a list of integers and return a single integer. This gives us the following type:

$$calc\ ::\ [Int]\ \rightarrow\ Int$$

A list comprehension will allow us to calculate the squares of all the negative numbers in xs. To sum these we shall require a $fold$. This gives us the following starting specification:

$$calc\ xs\ =\ fold\ (+)\ [x^2\ |\ x\ \leftarrow\ xs,\ x\ <\ 0]$$

8.3.2 Step 2 - Program Transformation

In this case we have a fairly simple task - we wish to convert our list comprehension based specification into a compositional form. See Section 7.6 for some transformation rules to achieve this.

$$\begin{array}{rcll} calc\ xs & = & fold\ (+)\ [x^2\ |\ x\ \leftarrow\ xs,\ x\ <\ 0] & \{def.\ calc\} \\ & = & (fold\ (+)\ \circ\ map\ sq)\ [x\ |\ x\ \leftarrow\ xs,\ x\ <\ 0] & \{LC2\} \\ & = & (fold\ (+)\ \circ\ map\ sq \circ\ filter\ (<0))\ [x\ |\ x\ \leftarrow\ xs] & \{LC3\} \\ & = & (fold\ (+)\ \circ\ map\ sq \circ\ filter\ (<0))\ xs & \{LC1\} \end{array}$$

As such we arrive at the following compositional form for *calc*:

$$calc\ =\ fold\ (+)\ \circ\ map\ sq\ \circ\ filter\ (<0)$$

Alternatively, we may find it convenient to replace our operators with simple functions:

$$calc\ =\ fold\ plus\ \circ\ map\ sq\ \circ\ filter\ isneg$$

8.3.3 Step 3 - Data Refinement

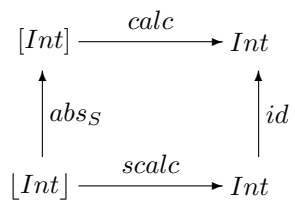
The presence of $filter$ in our specification means that we can not refine our input list directly to a vector (See Section 6.3). We shall instead consider two other alternatives here.

Stream Refinement

Our first alternative is to refine *calc* to a function *scalc*, which inputs a stream of integers and outputs a single integer. As such we have the type:

$$scalc\ ::\ \lfloor Int \rfloor\ \rightarrow\ Int$$

Such a function must allow the following diagram to commute, in order for us to state that it is a valid refinement of *calc*:



We shall find the definition presents itself somewhat mechanically during the proof:

$$\begin{array}{lll} & (calc \circ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{id\} \\ = & (fold\ (+) \circ map\ sq \circ filter\ (<0) \circ abs_S)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ calc\} \\ = & (fold\ (+) \circ map\ sq \circ abs_S \circ sfilter\ (<0))\ \lfloor x_1, x_2, ..., x_n \rfloor & \{sfilter\} \\ = & (fold\ (+) \circ abs_S \circ smap\ sq \circ sfilter\ (<0))\ \lfloor x_1, x_2, ..., x_n \rfloor & \{smap\} \\ = & (id \circ sfold\ (+) \circ smap\ sq \circ sfilter\ (<0))\ \lfloor x_1, x_2, ..., x_n \rfloor & \{sfold\} \\ = & (id \circ scalc)\ \lfloor x_1, x_2, ..., x_n \rfloor & \{def.\ scalc\} \end{array}$$

Thus *scalc* is defined as follows:

$$scalc\ =\ sfold\ (+) \circ smap\ sq \circ sfilter\ (<0)$$

This definition of *scalc* has linear complexity.

Vector of Streams Refinement

If we can partition the list, we will be able to perform much of the computational effort independently in parallel, and thus increase the efficiency. First we need to derive a distributed list refinement of *calc*. For this, we shall require the function *tdcalc*.

$$tdcalc\ ::\ [[Int]]\ \rightarrow\ Int$$

Proof that this is a valid refinement of *calc* will require the following diagram to commute:

$$\begin{array}{ccc} [Int] & \xrightarrow{\quad calc \quad} & Int \\ \uparrow{\scriptstyle abs_D} & & \uparrow{\scriptstyle id} \\ [[Int]] & \xrightarrow{\quad tdcalc \quad} & Int \end{array}$$

We can prove this in a similar manner to the above.

$$\begin{array}{lll} & (calc \circ abs_D)\ [l_1, l_2, ..., l_n] & \{id\} \\ = & (fold\ (+) \circ map\ sq \circ filter\ (<0) \circ abs_D)\ [l_1, l_2, ..., l_n] & \{def.\ calc\} \\ = & (fold\ (+) \circ map\ sq \circ abs_D \circ tdfilter\ (<0))\ [l_1, l_2, ..., l_n] & \{tdfilter\} \\ = & (fold\ (+) \circ abs_D \circ tdmap\ sq \circ tdfilter\ (<0))\ [l_1, l_2, ..., l_n] & \{tdmap\} \\ = & (id \circ tdfold\ (+) \circ tdmap\ sq \circ tdfilter\ (<0))\ [l_1, l_2, ..., l_n] & \{tdfold\} \\ = & (id \circ tdcalc)\ [l_1, l_2, ..., l_n] & \{def.\ tdcalc\} \end{array}$$

Thus *tdcalc* is defined as follows:

$$tdcalc\ =\ tdfold\ (+) \circ tdmap\ sq \circ tdfilter\ (<0)$$

From here we can decide on a communication mechanism. Given the use of $filter$, our only options from our selection of two dimensional structures are the stream of streams, and the vector

of streams. In the pursuit of increasing efficiency, we shall opt for the vector of streams. We may now attempt to refine *tdcalc* into a vector of streams definition. Such a function may be declared as follows:

$$vscalc_n \;::\; \langle \lfloor Int \rfloor \rangle_n \;\rightarrow\; Int$$

Here the subscript n denotes the *count* of partitions. We can determine the *size* of each partition by dividing the overall input size by n. Proof that $vscalc_n$ forms a valid refinement of *tdcalc* requires the following diagram to commute:

$$\begin{array}{ccc} [[Int]] & \xrightarrow{tdcalc} & Int \\ \uparrow{\scriptstyle abs_{VS}} & & \uparrow{\scriptstyle id} \\ \langle \lfloor Int \rfloor \rangle_n & \xrightarrow{vscalc} & Int \end{array}$$

The proof may proceed in a now familiar fashion:

$$\begin{array}{lll} & (tdcalc \circ abs_{VS})\ \langle s_1, s_2, ..., s_n \rangle_n & \{id\} \\ = & (tdfold\ (+) \circ tdmap\ sq \circ tdfilter\ (<0) \circ abs_{VS})\ \langle s_1, s_2, ..., s_n \rangle_n & \{def.\ tdcalc\} \\ = & (tdfold\ (+) \circ tdmap\ sq \circ abs_{VS} \circ vsfilter\ (<0))\ \langle s_1, s_2, ..., s_n \rangle_n & \{vsfilter\} \\ = & (tdfold\ (+) \circ abs_{VS} \circ vsmap\ sq \circ vsfilter\ (<0))\ \langle s_1, s_2, ..., s_n \rangle_n & \{vsmap\} \\ = & (id \circ vsfold\ (+) \circ vsmap\ sq \circ vsfilter\ (<0))\ \langle s_1, s_2, ..., s_n \rangle_n & \{vsfold\} \\ = & (id \circ vscalc)\ \langle s_1, s_2, ..., s_n \rangle_n & \{def.\ vscalc\} \end{array}$$

We are now presented with a definition for *calc*, *vscalc*, in terms of a vector of streams:

$$vscalc_n \;=\; vsfold_n\ (+) \circ vsmap_n\ sq \circ vsfilter_n\ (<0)$$

This definition gives us scope for better than linear complexity.

It may be useful to recall the relationships between *calc*, *tdcalc* and *vscalc*.

$$\begin{array}{lcl} tdcalc & = & calc \circ abs_D \\ vscalc & = & tdcalc \circ abs_{VS} \\ vscalc & = & calc \circ abs_D \circ abs_{VS} \end{array}$$

8.3.4 Step 4 - Process Refinement

Stream Refinement

Here we take the data refined functions in our specification *scalc* and refine them to processes. All of the components we have used can be found in our library, meaning that process definitions are already at hand. Given that the operator to be used with our $fold$ function is associative, we are at liberty to choose any interpretation of $fold$ we see fit - we shall opt for the left directed version

here as it removes the requirement for buffering. Finally, we simply need to know how to combine these building blocks. Composition between these functions corresponds to stream piping between processes. As a result, we have:

$$SCALC \;=\; SILTER\,(isneg) \;\gg\; SMAP(sq) \;\gg\; SFOLDL1(plus)$$

Note here that we are dealing with basic streams of items, so we can appeal to the simple case definitions of each of our higher order processes. In other words, they are parameterised by expressions rather than processes. The process $SCALC$ is depicted in Figure 8.2.

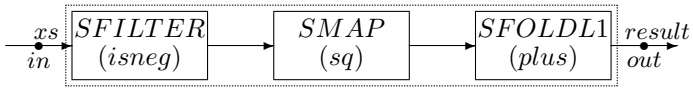


Figure 8.2: The process SCALC.

Vector of Streams Refinement

As before we now have the task of taking the components in our specification *vscalc* and refining them to processes. This is again a straightforward task of replacing functional components from our library with their CSP counterparts. Here communication between stages is achieved through vectors of streams, so we refine function composition to vector piping. We have:

$$VSCALC_n \;=\; VSFILTER_n\,(isneg) \;\gg_n\; VSMAP_n(sq) \;\gg_n\; VSFOLDL1_n\,(plus)$$

This network is depicted in Figure 8.3.

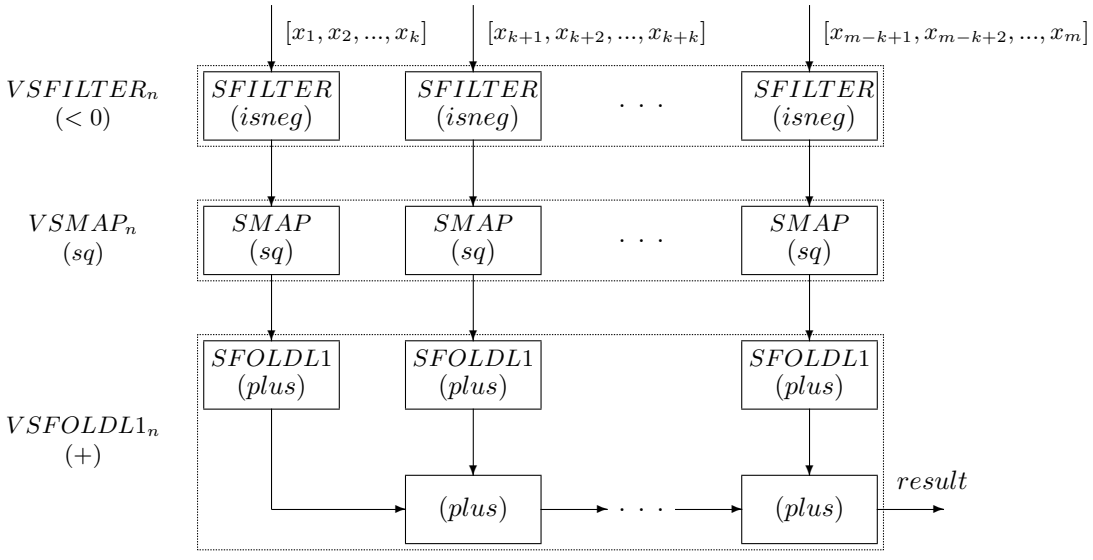


Figure 8.3: The VSCALC process network.

Here we have an input of size m split into n partitions each of size k. As the illustration highlights, the resulting network is effectively a parallel composition of n instances of our stream refinement - $SCALC$, with a $fold$ stage at the end to collate the results.

Recall from Sections 5.2, 5.3 and 6.3 that the processes we have employed here can actually be built up of other simpler processes. We have:

$$\begin{aligned} VSFOLDL1_n(f) &= VMAP_n\ (SFOLDL1\ (f)) \gg_n VSFOLDL1_n(f) \\ VSMAP_n(f) &= VMAP_n\ (SMAP\ (f)) \\ VSFILTER_n(p) &= VMAP_n\ (SFILTER\ (p)) \end{aligned}$$

As a result we can also express our network using this expanded definition:

$$\begin{aligned} VSCALC_n\ =\ VMAP_n\ (SFILTER\ (isneg))\ \gg_n\ VMAP_n\ (SMAP\ (sq))\ \gg_n \\ VMAP_n\ (SFOLDL1\ (plus)) \gg_n VSFOLDL1_n(plus) \end{aligned}$$

8.3.5 Step 5 - Implementation

Our final step is to take our networks of CSP processes and implement them in Handel-C. As before, we have relied heavily on components from our library, and as such the bulk of the implementation effort is done already. The bespoke functionality here requires just a few simple expressions to be defined:

```
macro expr isneg(x) = x<0;
macro expr sq(x) = x*x;
macro expr plus(x,y) = x+y;
```

Stream Refinement

The Handel-C definition for `SCALC` is given in Figure 8.4.

```
macro proc SCALC (streamin,itemout)
{
    Stream (Item (Int)) mida;
    Stream (Item (Int)) midb;

    par
    {
        SFILTER_SIMPLE  (streamin,mida,isneg);
        SMAP_SIMPLE     (mida,midb,sq);
        SFOLDL1_SIMPLE  (midb,itemout,plus);
    }
}
```

Figure 8.4: Handel-C definition for the process SCALC.

Vector of Streams Refinement

The Handel-C definition for `VSCALC` is given in Figure 8.5.

```
macro proc VSCALC (n,vsin,itemout)
{
    Vector (Stream (Item (Int)), mida, n);
    Vector (Stream (Item (Int)), midb, n);
    Vector ((Item (Int), midc, n);

    par
    {
        VMAP    (n,vsin,mida,SFILTER_isneg);
        VMAP    (n,mida,midb,SMAP_sq);
        VMAP    (n,midb,midc,SFOLDL1_plus);
        VFOLDL1 (n,midc,itemout,plus);
    }
}
```

Figure 8.5: Handel-C definition for the process SCALC.

Here `SFILTER_isneg`, `SMAP_sq` and `SFOLDL1_plus` are simply synonyms for the appropriate higher order process component, specialised with the appropriate expression. It is necessary to specify these in this way as Handel-C does not support currying. For example:

```
macro proc SFILTER_isneg (in, out)
{
    SFILTER_SIMPLE (in,out,isneg);
}
```

Evaluation

The developer may now wish to evaluate the relative merits of the two alternative implementations which have been derived. Compiling these and going through the place and route tools should give an indication of the resource requirements. The resulting configuration files can then be tested on the FPGA device to evaluate performance. Based on these finding the developer may then wish to repeat part of the design process, as discussed previously. For the vector of streams refinement, based on distributed lists, the extent of parallelism can be tuned by adjusting the partition size.

8.4 Summary

In this chapter we have described the full refinement procedure, from start to finish. We now have a step by step process we can follow for refining a functional specification into a hardware implementation in Handel-C. Through an example we have illustrated how this procedure achieves a number of desirable goals - improved ease and speed of development, whilst all throughout ensuring correctness and enabling flexibility in the design.

Chapter 9

Case Studies

9.1 Introduction

In this chapter we present a number of case studies which demonstrate the application of this methodology. We first look at a number of smaller, illustrative case studies - a selection of sorting algorithms. These are provided purely for illustration, and we do not provide actual results gathered from implementation on an FPGA.

Following on from this we take on two slightly more involved problems - a pair of combinatorial list processing algorithms. We follow these case studies right the way to implementation on the FPGA, and produce a set of results for varying problem sizes.

After this we move on to more substantial real world applications - a JPEG decoder and an algorithm for DNA processing, and also attempt to produce results for these from implementation on the FPGA.

9.2 Sorting

Sorting is frequently used as a case study for new or different programming techniques. There may be several reasons for this. Undoubtedly the purpose, and usually the behaviour, of any given sorting algorithm is generally clear and easy to understand. Furthermore there exists a set of widely known sort algorithms, whose relative merits are well understood and have been extensively tested. Particularly, we know already the efficiency of all the commonly used sorting algorithms.

We shall examine four basic sorting algorithms here - insert sort, merge sort, selection sort and quick sort. We shall look at how hardware implementations may be derived from each of these algorithms. We shall not, however, go so far as to actually implement these first case studies on an FPGA - these are intended purely for illustration of the design process.

9.2.1 Insert Sort

The insert sort algorithm is perhaps the most intuitive of those we will examine here, as it corresponds to the method most people will use to sort manually. We effectively maintain two lists, one of unsorted items, which is initially the input list, and one of sorted items, which is initially empty. At each step, we remove the first item from the unsorted list and traverse the sorted list until we find the correct place to insert it.

Step 1 - Intitial Specification

A functional definition of this algorithm may proceed as follows. First, considering the functionality of the *insert* function, which takes a sorted list and a single item, and returns the list with that item inserted at the correct position.

$$\begin{array}{lcl} insert\ a\ [] & = & [a] \\ insert\ a\ (x:xs) & = & if\ a < x \\ & & then\ a : x : xs \\ & & else\ x\ :\ insert\ a\ xs \end{array}$$

For the sorting algorithm as a whole, we are simply applying the function above repeatedly, once for every item in the unsorted list, in order to gradually build up the sorted list. We have:

$$\begin{array}{lcl} insertsort\ [] & = & [] \\ insertsort\ (x:xs) & = & insert\ x\ (insersort\ xs) \end{array}$$

The type of the function *insert* is defined as follows:

$$insert :: A \rightarrow [A] \rightarrow [A]$$

The type of the algorithm is, of course, defined as follows:

$$insertsort :: [A] \rightarrow [A]$$

As already mentioned, the efficiency of this algorithm is already well known - it has $O(n^2)$ efficiency. This can be confirmed somewhat intuitively. For a list of length n, we make a pass of the sorted list for each item being inserted into it. Each pass may require up to n steps.

Step 2 - Program Transformation

Our definition for the insert sort algoritm above is actually an instance of the $fold$ pattern, we have:

$$insertsort\ s \quad = \quad foldr\ insert\ []\ s$$

This is now a convenient form from which to begin refinement. As has already been shown in [2], this pattern can be refined to a pipelined implementation.

Step 3 - Data Refinement

Let us consider how we may refine the datatypes in this algorithm. We shall only examine the derivation briefly here, given that this case study has already been explored in [2]. In this case we are aiming for a process refinement of *insertsort* which both inputs and outputs a stream. First, we have the necessary data refinement:

$$sinsertsort :: \lfloor A \rfloor \rightarrow \lfloor A \rfloor$$

The implementation should follow directly:

$$sinsertsort\ s \quad = \quad sfoldr\ sinsert\ \lfloor\rfloor\ s$$

Which will also require a refinement of *insert* in stream terms:

$$sinsert :: A \rightarrow \lfloor A \rfloor \rightarrow \lfloor A \rfloor$$

As demonstrated in [2], from this definition we can derive the following which uses n instances of the function *sinsert* folded together with the composition operator:

$$sinsertsort\ s\ =\ (\circ)\ /\ (smap\ sinsert\ s)\ \lfloor\rfloor$$

Step 4 - Process Refinement

In process terms, the above definition corresponds to a pipeline of processes, as discussed in Section 4.5. In CSP we can express this pipeline as follows:

$$Prd(\lfloor\rfloor)\ \rhd \left(\begin{array}{cc} n & \\ \gg & SINSERT(x_i) \\ i=1 & \end{array} \right)$$

The process $SINSERT$ here can be defined as follows, and can be derived as a simple recursion unrolling (see Section 4.6) of our original specification *insert*:

$$\begin{array}{l} SINSERT(a) = \\ \mu\, X \bullet \\ in.eot\ ?\ any \rightarrow (out.value\ !\ a \rightarrow out.eot\ !\ True \rightarrow SKIP) \\ | \\ in.value\ ?\ x \rightarrow \left(\begin{array}{c} out.value\ !\ a \rightarrow out.value\ !\ x \rightarrow COPY \\ \nleftarrow a < x \nrightarrow \\ out.value\ !\ x \rightarrow X \end{array} \right) \end{array}$$

This insert sort process is depicted in Figure 9.1.

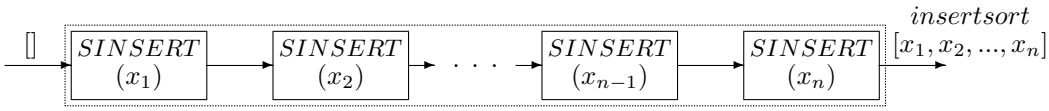


Figure 9.1: The insert sort process.

Step 5 - Handel-C Implementation

An implementation in Handel-C of the $SINSERT$ process can follow naturally from the CSP definition. We have:

```
macro proc SINSERT (in,out,x)
{
    typeof(x) a;
    Bool eot;
    eot = False;
    while (!eot)
    {
        prialt
        {
        case in.value ? x:
            if (a<x)
            {
                out.value ! a;
                out.value ! x;
                SCOPY (in,out,eot);
            }
            else
            {
                out.value ! x;
            }
            break;
        case in.eot ? eot:
            out.value ! a;
            out.eot ! eot;
            break;
        }
    }
}
```

Thus we can provide a simple Handel-C implementation for our insert sort algorithm as follows:

```
macro proc SINSERTSORT (streamout, n, xs)
{
    Stream(Item(Value)) mid[n];
    par (i=0;i<=n;i++)
    {
        ifselect (i==0)
            mid[0].eot ! True;
        else ifselect (i<n)
            SINSERT (mid[i-1],mid[i],xs[i-1]);
        else // ifselect (i==n)
            SINSERT (mid[i-1],streamout,xs[i-1]);
```

```
    }
}
```

9.2.2 Selection Sort

The selection sort (or min sort) algorithm is perhaps another based on an intuitive 'human' sorting strategy. Here we maintain two lists, one sorted and one unsorted. To begin with our sorted list is empty, and our unsorted list is the input list. At each step we scan the unsorted list to find the lowest value, and remove it, placing it at the tail of the sorted list.

Step 1 - Intitial Specification

The characteristic function here is one that extracts the lowest value from the list. That is to say, takes in a list, and returns a pair, wherein the first item of the pair is the lowest value, and the second item is the input list with the lowest item removed. We have:

$$\begin{array}{lcl} minex & :: & [A] \rightarrow (A, [A]) \\ minex\ xs & = & (x,\ remove\ x\ xs) \\ & & where\ x\ =\ fold\ (\downarrow)\ xs \end{array}$$

Where *remove* is a function which removes the first occurrence of a given element from a list (a definition is given below). Given the function *minex*, we can show that our algorithm *minsort* is in fact an instance of the $unfold$ pattern:

$$minsort\ xs\ =\ unfoldr\ minex\ null\ xs$$

The above definition for *minex* gives us a high level specification of the required functionality. It tells us that a pair is required as output, with two components as previously described - the lowest item and the remaining list. Considering an implementation that exploits some parallelism, however, we may wish to massage the above definition into a slightly different form. The issue here is one of 'interactivity'. Currently the dependencies in *minex* are such that the entirety of the input list must be consumed before any output can be produced. The remaining items are returned by *remove*, but this cannot begin to operate until ($fold\ (\downarrow)\ xs$) has been calculated, which of course requires that the entire input list has been consumed. However, this shouldn't have to be the case given the characteristics of the min ($\downarrow$) operator.

Step 2 - Program Transformation

For convenience, a definition for *remove* is given here:

$$\begin{array}{lcl} remove\ e\ [] & = & [] \\ remove\ e\ (x : xs) & = & \text{if } (e == x) \text{ then } xs \text{ else } x : (remove\ e\ xs) \end{array}$$

We should be able to create a function which combines the functionality of *remove* and $fold\ (\downarrow)$ into a single interactive pass. That is to say a function which performs the same functionality as:

$$remove\ (fold\ (\downarrow)\ xs)\ xs$$

The task we have is to remove the lowest value from the list. Where we are presented with a list containing just one value, the result is trivial - that item must by definition be the lowest value, so we return the empty list. For a list containing two or more values, we can compare the first two values. The higher of the two we know cannot be the lowest item in the list, so we can output it immediately. The lower of the two we put back into the as yet unprocessed portion of the list and recurse.

$$\begin{array}{lcl} minremove\ [x] & = & [] \\ minremove\ (x : y : xs) & = & (x \uparrow y) : minremove\ ((x \downarrow y) : xs) \end{array}$$

This function is interactive with respect to its input list - for each item consumed from the input, an item is produced in the output list (with the obvious exception of the lowest item, which is removed). It may be important to note that *minremove* is, however, not an exact equivalence of our *remove* and $fold(\downarrow)$ based definition. The output of *minremove* may be in a slightly different order, as a result of local minima. As an example:

$$\begin{array}{lcl} xs & = & [3,2,4,1] \\ remove\ (fold\ (\downarrow)\ xs)\ xs & = & [3,2,4] \\ minremove\ xs & = & [3,4,2] \end{array}$$

However we are still satisfying the criteria of our original specification - that we return a list with the lowest item removed. The actual ordering is not important.

We should recall now that our objective is to *extract* the lowest value, and not simply discard it. To this end we can provide a variant of the above function *minremove* which implements the required functionality of *minex*. We have:

$$\begin{array}{lcl} minex\ [x] & = & (x, []) \\ minex\ (x : y : xs) & = & (x \uparrow y) \oplus minex\ ((x \downarrow y) : xs) \\ & & where\ z\ \oplus (m, zs)\ =\ (m,\ z : zs) \end{array}$$

A slight variation of the above employs a 'lowest so far' value, represented as an additional parameter m, which may make for a slightly clearer definition:

$$\begin{array}{lcl} minex\ (x : xs) & = & minex'\ x\ xs \\ minex'\ m\ [] & = & (m, []) \\ minex'\ m\ (x : xs) & = & (m \uparrow x)\ \oplus\ minex'\ (m \downarrow x)\ xs \\ & & where\ y\ \oplus (a, ys)\ =\ (a,\ y : ys) \end{array}$$

Step 3 - Data Refinement

Given that our definition for *minsort* is based on $unfoldr$ we can follow this pattern through to our data refinement phase. Let us consider first refinement of the data types involved. Should we wish to input and output a stream, we might appeal to $sunfoldr$:

$$\begin{array}{lcl} sminsort & :: & \lfloor A \rfloor \to \lfloor A \rfloor \\ sminsort\ xs & = & sunfoldr\ sminex\ snull\ xs \end{array}$$

Should we wish instead to output the sorted values as a vector, we can instead appeal to $vunfoldr$:

$$\begin{array}{lcl} vminsort & :: & \lfloor A \rfloor \to \langle A \rangle_n \\ vminsort\ xs & = & vunfoldr_n\ sminex\ snull\ xs \end{array}$$

Here *sminex* is a refinement of *minex* which inputs and outputs streams. We have:

$$sminex\ ::\ \lfloor A \rfloor \to (A,\ \lfloor A \rfloor)$$

Step 4 - Process Refinement

Let us consider a CSP definition for the component process $SMINEX$. Here we have a process which inputs a stream on one conduit, and outputs on two conduits - one a stream, and another a single item.

$$\alpha SMINEX\ =\ \{in :: \underline{\lfloor A \rfloor},\ outd :: \underline{A},\ outr :: \underline{\lfloor A \rfloor}\}$$

The values to be transmitted on conduits *outd* and *outr* correspond to the first and second items respectively of the pair returned by *sminex*. So on conduit *outd*, the lowest item from the input stream received on *in* should be produced. On *outr* all items but the lowest from the input stream on *in* should be transmitted. This is depicted in Figure 9.2.

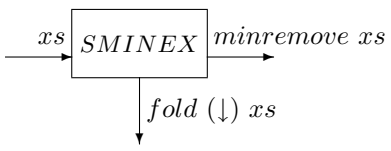


Figure 9.2: The process SMINEX.

Let us consider a CSP definition for our process $SMINEX$. This corresponds closely to our final definition for *minex* given above. We have the following:

$$\begin{array}{lcll} SMINEX & = & in.value\ ?\ m \to X & \\ & & & in.eot\ ?\ any \to outd\ !\ m \to outr.eot\ !\ True \to SKIP \\ & & \mu\, X \bullet\ | & \\ & & & in.value\ ?\ x \to outr.value\ !\ (m \uparrow x); m := (m \downarrow x); X \end{array}$$

Our vector output process $VMINSORT$ can then be created by composing together n instances of this component process following the $VUNFOLDR$ pattern. This is depicted in Figure 9.3.

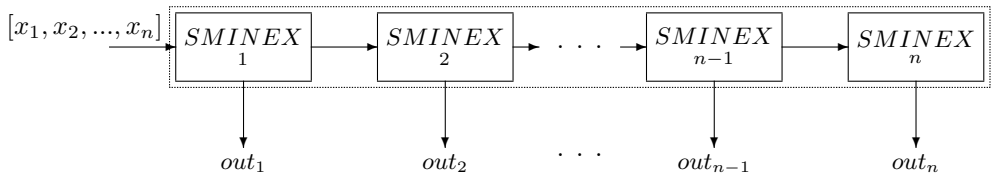


Figure 9.3: The VMINSORT process.

Step 5 - Handel-C Implementation

```
macro proc SMINEX (in,outd,outr)
{
    messagetype(outd) x,m;
    Bool eot;
    eot = False;
    in.value ? m;
    while (!eot)
    {
        prialt
        {
        case in.value ? x:
            outr.value ! max(m,x);
            m = min(m,x);
            break;
        case in.eot ? eot:
            outd ! m;
            outr.eot ! eot;
            break;
        }
    }
}
```

Figure 9.4: The Handel-C definition of the process SMINEX.

A Handel-C implementation for `SMINEX` can follow directly from the CSP definition. This is given in Figure 9.4.

9.2.3 Quick Sort

In a sequential setting, quick sort often delivers the best performance, in terms of both efficiency and memory usage, and so is very widely used. Its behaviour follows a divide and conquer pattern. At each step, we extract an item from the list (usually the head) to act as a 'pivot item'. We can then make a simple linear time pass to extract all the items lower than (or equal to) the pivot, placing these into one group, as well as all the items higher than the pivot which are placed into another group. Should both of these groups contain just zero or one item(s) each, we deem that this list is already sorted. Should one or both groups contain two or more items, we need to then recursively apply the quick sort algorithm.

Step 1 - Intitial Specification

In a functional setting, a definition is often given based on two list comprehensions.

$$\begin{array}{lcl} qsort\ [] & = & [] \\ qsort\ [x] & = & [x] \\ qsort\ (x:xs) & = & qsort\ [u|u \leftarrow xs, u \leq x] +\!\!+ [x] +\!\!+ qsort\ [v|v \leftarrow xs, v > x] \end{array}$$

Step 2 - Program Transformation

One slight variant on the above may provide small gains in efficiency where we are expecting duplicates in the list. Here we group together all items equal to the pivot and place them in an intermediate list which is, of course, sorted by definition.

$$\begin{array}{lcl} qsort\ [] & = & [] \\ qsort\ [x] & = & [x] \\ qsort\ (x:xs) & = & qsort\ [u|u \leftarrow xs, u < x] +\!\!+ [e|e \leftarrow xs, e = x] +\!\!+ qsort\ [v|v \leftarrow xs, v > x] \end{array}$$

The list comprehensions used above can also be expressed with instances of $filter$:

$$\begin{array}{lcl} qsort\ [] & = & [] \\ qsort\ [x] & = & [x] \\ qsort\ (x:xs) & = & qsort\ (filter\ (<x)\ xs) +\!\!+ (filter\ (=x)\ xs) +\!\!+ qsort\ (filter\ (>x)\ xs) \end{array}$$

Furthermore, we may also consider a definition in terms of our generic divide and conquer higher order function (see Section 6.8).

$$\begin{array}{lcl} qsort & = & dc\ id\ qistc\ qsplit\ qcombine \\ qsplit\ (x:xs) & = & [filter\ (\leq x)\ xs, [x], filter\ (>x)\ xs] \\ qcombine & = & fold\ (+\!\!+) \\ qistc\ xs & = & (length\ xs) \leq 1 \end{array}$$

That is to say, we determine that a given input corresponds to the trivial case if the length is less than or equal to 1. Where an input is an instance of the trivial case, we return it unchanged (using

the function *id*), based on the observation that singleton or empty lists are sorted by definition. The split function carries most of the interesting functionality of the quick sort algorithm. It is here, in the function *qsplit*, that the pass is made through the list to split it into two sub lists, one of items lower than the pivot and the other of items higher. The sub problems are then re-combined using a simple fold with the concatenation operator.

Step 3 - Data Refinement

One might normally expect parallel implementations of quick sort to rely heavily on shared memory. However, given the divide and conquer pattern introduced in Section 6.8, we may be able to consider an implementation based on message passing, and therefore one which is truly scalable.

Let us consider the refinement of our components. First, and perhaps most importantly, we have *qsplit*. In list terms we have the following type:

$$qsplit :: [A] \rightarrow [[A]]$$

One possible refinement for this is a version which inputs a stream, and outputs a vector of streams. Given the definition of *qsplit* we know that our output vector will contain exactly three streams. We have:

$$svqsplit :: \lfloor A \rfloor \rightarrow \langle \lfloor A \rfloor \rangle_3$$

We can define this as follows:

$$svqsplit\ (x \widehat{:} xs) \ = \ \langle sfilter\ (\leq x)\ xs, \lfloor x \rfloor, sfilter\ (> x)\ xs \rangle_3$$

Next we have the combine phase *qcombine*. Here we wish to provide a refinement which inputs a vector and outputs a stream. Again, we know here that we are dealing with a vector of size three. We have:

$$vsqcombine :: \langle \lfloor A \rfloor \rangle_3 \rightarrow \lfloor A \rfloor$$

Again we can provide a definition as follows:

$$vsqcombine \ = \ vfold_3(\widehat{+\!\!+})$$

Step 4 - Process Refinement

Let us consider a process refinement of the split function. Here we have the following alphabet:

$$\alpha SVQSPLIT \ = \ \{in :: \underline{\lfloor A \rfloor},\ out :: \langle \underline{\lfloor A \rfloor} \rangle_3\}$$

We can combine the two $filter$ expressions in the specification into a single pass, which will work particularly well as the predicates are mutually exclusive.

$$SVQSPLIT \quad = \quad in.value\ ?\ x\ \rightarrow out_2.value\ !\ x \rightarrow out_2.eot\ !\ True \rightarrow$$
$$\mu X \bullet \left(\begin{array}{ll} in.eot\ ?\ any \rightarrow & out_1.eot\ !\ True \rightarrow \\ & out_3.eot\ !\ True \rightarrow \\ & SKIP \\ | & \\ in.value\ ?\ y \rightarrow & \text{if } (y < x) \\ & \text{then } out_1.value\ !\ y \rightarrow X \\ & \text{else } out_3.value\ !\ y \rightarrow X \end{array} \right)$$

The quick sort algorithm is demonstrated in Figure 9.5.

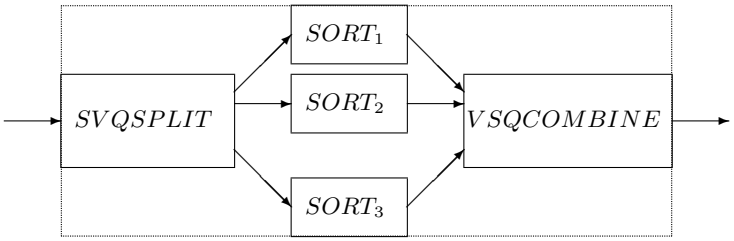


Figure 9.5: The core of the parallel Quick Sort process algorithm.

Step 5 - Handel-C Implementation

We shall not go the whole way to implementation in Handel-C for this case study; it is intended more as an illustration of how divide and conquer style problems might be approached with this methodology. Broadly though, a Handel-C implementation should be relatively straightforward - a process implementing the functionality of $SVQSPLIT$ could follow on almost directly from the CSP definition given above.

9.2.4 Merge Sort

Merge sort hinges around an operator which takes in two sorted lists and merges them together such that the resulting list is also sorted.

Step 1 - Initial Specification

The characteristic function in merge sort, *merge*, is usually defined recursively as follows:

$$
\begin{array}{lcl}
merge\ xs\ [] & = & xs \\
merge\ []\ ys & = & ys \\
merge\ (x : xs)\ (y : ys) & = & if\ x < y \\
 & & then\ x : merge\ xs\ (y : ys) \\
 & & else\ y : merge\ (x : xs)\ ys
\end{array}
$$

A list containing only a single item is of course sorted by definition. Given that we need to supply *merge* with already sorted lists, an obvious strategy is to convert the input list into a list of singletons. The merge sort algorithm can therefore be implemented with a fold.

$$mergesort\ xs\ =\ fold\ merge\ (map\ (\lambda x\ \bullet\ [x])\ xs)$$

Step 2 - Program Transformation

In this particular instance no real program transformation effort is required - the specification is already in a form suitable for refinement.

Step 3 - Data Refinement

When we come to consider data refinement, we are presented with two main options. First a stream based refinement:

$$smergesort\ =\ (sfold\ smerge)\ \circ\ smap\ (\lambda x\ \bullet\ \lfloor x \rfloor)$$

Second, a vector based refinement:

$$vmergesort\ =\ (vfold\ smerge)\ \circ\ vmap\ (\lambda x\ \bullet\ \lfloor x \rfloor)$$

Step 4 - Process Refinement

We shall consider here our vector based refinement only. Given that the characteristic operator here ($fold$) is associative, this is a candidate for a "funnel" network of processes when we come to consider implementation in parallel. This is depicted in Figure 9.6.

Step 5 - Handel-C Implementation

As with quick sdort, this is intended only as an illustrative example, and we shall not proceed the whole way to implementation in Handel-C here.

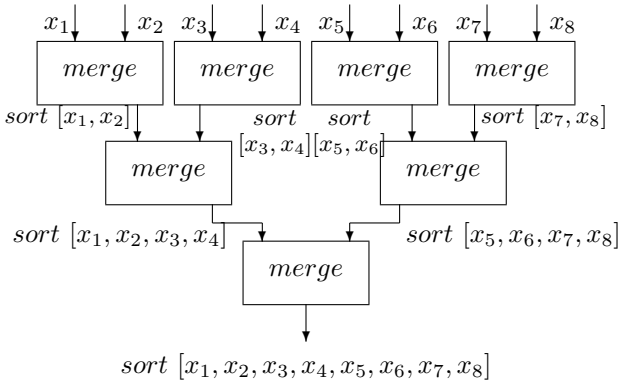


Figure 9.6: A funnel implementation of the merge sort algorithm.

9.3 Combinatorial Algorithms

We shall see that a number of algorithms, well suited to implementation through this methodology, all fall into a similar pattern. Generally speaking, these can loosely be described as three stage algorithms. The first stage takes the input and expands it into a intermediate structure. The second stage processes the intermediate structure. The third stage then collapses and combines this structure to return the result. In slightly more precise terms, we have the following pattern:

$$fold\ f\ \circ\ g\ \circ\ unfold\ h$$

The two case studies presented at in this section - *minimum distance* and *distinct elements* have been published in [40]. Here they are given a fuller treatment, with more detail on the data refinement phase, and on the final Handel-C implementation.

9.3.1 Minimum Distance

Given two lists of points in n-dimensional space, the minimum distance problem is one of finding the distance between the two points which are closest together.

Step 1 - Initial Specification

An intuitive solution to such a problem is to calculate the Cartesian product of the two lists of points, calculate the distance between each of the generated pairs, and simply find the lowest of all these distances. We have the following:

$$md\ xs\ ys = (fold\ min\ \circ\ map\ dist)\ (cp\ xs\ ys)$$

A simple function for calculating the Pythagorean distance between two points in n dimensional space can be given as follows. In this particular instance we assume we are working in three dimensions.

$$dist\ (x1, y1, z1)\ (x2, y2, z2)\ =\ \sqrt{(x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2}$$

Step 2 - Program Transformation

Our algorithm will have quadratic complexity, owing to the size of the intermediate structure generated by *cp*. In order to achieve a linear implementation in parallel, we shall have to find some means of dividing up this intermediate structure into sub-segments which can be processed independently. This can be achieved with the function *dcp* (see Section 7.14). Substituting in the definition for *dcp* we arrive at the following:

$$md\ xs\ ys = (fold\ min \circ map\ dist \circ fold\ (+\!\!+))\ (dcp\ xs\ ys)$$

Our goal of processing the intermediate results independently is only partially achieved however, as almost all of the work is being done after the list of lists is concatenated. Thankfully, we can appeal to our promotion laws to remedy this situation.

$$\begin{array}{lll} & (fold\ min \circ map\ dist \circ fold\ (+\!\!+))\ (dcp\ xs\ ys) & \{def.\} \\ = & (fold\ min \circ fold\ (+\!\!+) \circ map\ (map\ dist))\ (dcp\ xs\ ys) & \{map\ promotion\} \\ = & (fold\ min \circ map\ (fold\ min) \circ map\ (map\ dist))\ (dcp\ xs\ ys) & \{reduce\ promotion\} \end{array}$$

The resulting definition gives the scope we require for independent processing on the intermediate results. We have:

$$md\ xs\ ys\ =\ (fold\ min \circ map\ (fold\ min) \circ map\ (map\ dist))\ (dcp\ xs\ ys)$$

Step 3 - Data Refinement

We will require a refinement of the distributed Cartesian product function with output which is convenient for independent processing in parallel. See Section 7.14 for a discussion of the potential alternatives. The two prime candidates in this setting are the vector of streams and the stream of vectors, as these are both able to produce the quadratic sized output in linear time and with linear processing resource requirements. Let us consider first the case of the vector of streams outputs.

If we refine our input list xs to a stream s (that is to say $xs\ =\ abs_S\ s$), and similarly our other input list ys to a stream t (again, where $ys\ =\ abs_S\ t$), we can then present the RHS of our function *md* as follows:

$$(fold\ min \circ map\ (fold\ min) \circ map\ (map\ dist))\ (dcp\ (abs_S\ s)\ (abs_S\ t))$$

We have the following data refined version of *dcp* in this setting, as presented in Section 7.14.2:

$$svsdcp_n :: \lfloor A \rfloor \to \lfloor B \rfloor \to \langle \lfloor (A, B) \rfloor \rangle_n$$

Also in Section 7.14.2 we have the following property:

$$dcp\ (abs_S\ s)\ (abs_S\ t)\ =\ abs_{VS}\ (svsdcp_n\ s\ t)$$

Substituting this into our definition for *md* we have:

$$(fold\ min\ \circ\ map\ (fold\ min)\ \circ\ map\ (map\ dist))\ (abs_{VS}\ (svsdcp_n\ s\ t))$$

We can of course move the abstraction function into the compositional part of our definition:

$$(fold\ min\ \circ\ map\ (fold\ min)\ \circ\ map\ (map\ dist)\ \circ\ abs_{VS})\ (svsdcp_n\ s\ t)$$

From here on, we simply promote the expression abs_{VS} upwards, at each step using the relevant data refinement identity. First, for *map* in the vector of streams setting (see Section 5.2.8) we have:

$$(fold\ min\ \circ\ map\ (fold\ min)\ \circ\ abs_{VS}\ \circ\ vmap_n\ (smap\ dist))\ (svsdcp_n\ s\ t)$$

Secondly, for $fold$ in the vector of streams setting (see Section 5.3.5) we have:

$$(vfold\ min\ \circ\ vmap_n\ (fold\ min)\ \circ\ vmap_n\ (smap\ dist))\ (svsdcp_n\ s\ t)$$

Thus we arrive at our vector of streams based refinement for *md*, which we shall call *vsmd*:

$$vsmd_n\ s\ t\ =\ (vfold_n\ min\ \circ\ vmap_n\ (sfold\ min)\ \circ\ vmap_n\ (smap\ dist))\ (svsdcp_n\ s\ t)$$

We can relate between this and our original definition as follows, given two streams s and t where s is of length n:

$$md\ (abs_S\ s)\ (abs_S\ t)\ =\ vsmd_n\ s\ t$$

Considering now the stream of vectors case, an analogous derivation to the above could be produced, which would result in the following definition:

$$svmd_n\ s\ t\ =\ (sfold\ min\ \circ\ smap\ (vfold_n\ min)\ \circ\ smap\ (vmap_n\ dist))\ (ssvdcp_n\ s\ t)$$

Step 4 - Process Refinement

Let us consider refinement to processes of the data refined specification $vsmd_n$. Almost all of the functionality here can be taken from our higher order process library, requiring only trivial implementations for the refinements of the functions *min* and *dist*. Functional composition in the specification will correspond to process piping in the implementation.

We may wish at this point to decide on specific interpretations for the generic uses of $fold$ derivatives used in our specification. As the characteristic operator - *min* - is associative, either a left or right directed fold will suit our requirements. Assuming the input size n is non zero, then we are at liberty to use a refinement of either $foldr1$ or $foldl1$. There are two instances of $fold$ in our specification for $vsmd_n$ - one in a stream setting, and the other in a vector setting. Where we are required to implement the functionality of $fold$ in a stream setting, we shall use a refinement

of $foldl1$, as this can be implemented without buffering (see Section 5.3.1). In the vector setting the decision is arbitrary, so we shall again opt for a refinement of the left directed variant, $foldl1$.

The implementation can now be constructed, giving us the following network:

$$VSMD_n(xs) =$$
$$SVSDCP_n(xs) \gg_n VMAP_n(SMAP(dist)) \gg_n VMAP_n(SFOLDL1(min)) \gg_n VFOLDL1_n(min)$$

Here n represents the length of list xs. The process $VSMD_n$ depicted in Figure 9.7.

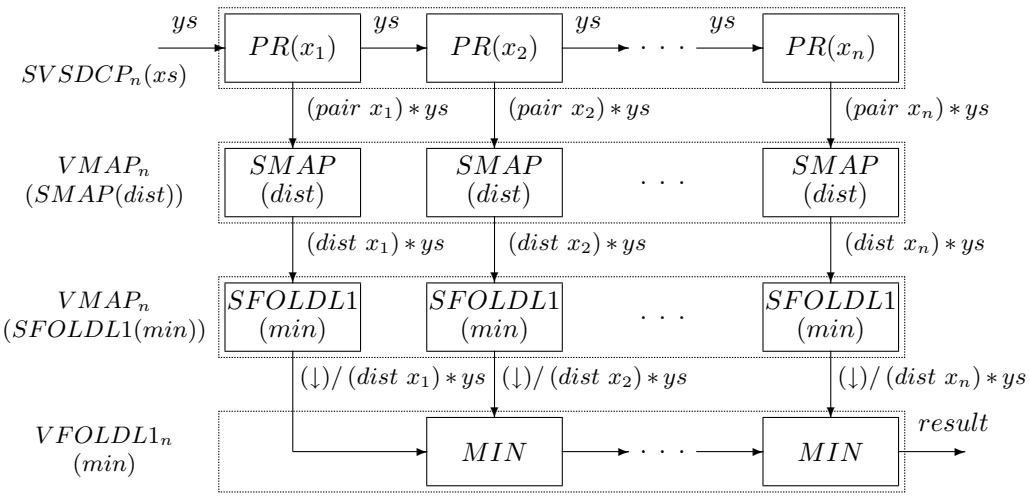


Figure 9.7: A network to solve the minimum distance problem for the lists xs and ys

Step 5 - Handel-C Implementation

The Handel-C implementation can flow naturally from the CSP definition. This is given in Figure 9.8.

```
macro proc MINDIST (n,streamin,itemout,xs)
{
    Vector (Stream (Item (CoordPair)), vectora, n);
    Vector (Stream (Item (Value)), vectorb, n);
    Vector (Item (Value), vectorc, n);

    par
    {
        SVSDCP  (n,streamin,vectora,xs);
        VMAP    (n,vectora,vectorb,SMAPDIST);
        VMAP    (n,vectorb,vectorc,SFOLDMIN);
        VFOLDL1 (n,vectorc,itemout,MIN);
    }
}
```

Figure 9.8: Handel-C definition for the minimum distance problem.

Results

Results for the minimum distance network implemented on a Xilinx XCV2000e are given in Table 9.1. In these results we make the assumption here that the length of time each cycle takes does not change significantly with respect to the problem size. Sequentially this is a quadratic algorithm, but, as these results demonstrate, our parallel hardware implementation is linear, both in resource usage and execution time.

Indeed the cycle times are very regular - we have, for a problem of size n, a cycle time in each case of precisely $5n + 6$. The slice usage is a little less regular, but follows roughly a pattern of about $250n$ slices. To this end, we may hope to be able to implement the network for a problem size of up to around 100 on an XCV2000e device.

Obviously, a larger device (or composition of multiple devices) would allow larger problem sizes to be dealt with. Alternatively, it may be possible to increase the number of items we can deal with on a single device through some optimisation techniques. This optimisation does not necessarily require us to resort to ad-hoc 'tweaking'. We can instead return to our specification and consider alternative refinements in a formal manner.

For example, we may consider condensing together the two $VMAP$ stages into a single stage, which could be proved as an equivalent with some simple map transformation laws. Although this variant implementation would be slightly less efficient in terms of computation time, it would however have a lower overhead in terms of communication, and therefore is likely to have a lower resource requirement.

items	*slices used*	(%)	*cycles*
5	1,323	6%	31
10	2,561	13%	56
15	3,817	19%	81
20	5,066	26%	106
25	6,295	32%	131
30	7,567	39%	156

Table 9.1: Results for the minimum distance network.

9.3.2 Distinct Elements

Given a list of items, the distinct elements problem is one of determining whether or not the list is free of duplicates, that is to say, no item in the list is equal to any other item.

Step 1 - Initial Specification

Instinctively we might provide a recursive algorithm to solve this problem. An empty list is distinct by definition - as it contains no items, it therefore contains no duplicates. For a non-empty list, we can take the first item and compare it to the remainder to determine whether or not there are any repetitions of that particular item. If there are not, we may then proceed to check the remainder in the same way, recursively. A typical recursive definition of this form can be given as follows:

$$\begin{array}{lcl} distinct\ [] & = & True \\ distinct\ (x : xs) & = & (and\ (map\ (\neq x)\ xs))\ \wedge (distinct\ xs) \end{array}$$

Step 2 - Program Transformation

A little inspection of this definition reveals that we are actually operating on all of the final segments of our list. For each final segment we are attempting to calculate whether or not the first element differs from the subsequent elements. Thus what we are doing is in effect equivalent to the following definition:

$$distinct \quad = \quad and \circ map\ noteq \circ tails^{+}$$

Given a function *noteq* which takes a list and dictates if the first item is different to all the others.

$$\begin{array}{lcl} noteq & = & and \circ diff \\ diff\ (x : xs) & = & map\ (\neq x)\ xs \end{array}$$

Let us expand the definition of *noteq* here to provide scope for applying the map distributivity law:

$$distinct \quad = \quad and \circ map\ (and \circ diff) \circ tails^{+}$$

This allows us to give a slight redefinition of *distinct*, which better exposes the separate processing phases.

$$distinct \quad = \quad and \circ map\ and \circ map\ diff \circ tails^{+}$$

Step 3 - Data Refinement

Let us now consider refining this specification to derive our implementation. We have an algorithm of quadratic complexity, resulting from the size of the intermediate structure created by *tails*. We would like to arrive at a linear implementation in parallel, requiring O(n) processing resources. We

therefore require each of the final segments generated by *tails* to be processed independently in our implementation. Let us assume our data will arrive initially as a stream. Our refinement of $tails^+$ will be responsible for taking in this stream and producing a vector of streams as output, with each stream in the vector representing a final segment in the list. Within each final segment we shall employ stream refinements of our functions. We then arrive at the following definition after our data refinement step.

$$vand_n \circ vmap_n\ sand \circ vmap_n\ sdiff \circ svstails_n^+$$

Expanding our definitions for $vand_n$ and *sand* will allow us to insert $fold$ derivatives in their place.

$$vfold_n(\wedge) \circ vmap_n\ (sfold(\wedge)) \circ vmap_n\ sdiff \circ svstails_n^+$$

As with our previous case study, the operator here $(\wedge)$ is associative, and we can assume our list to be non empty. So the decision as to which particular interpretation of $fold$ to opt for is fairly arbitrary. For example:

$$(vfoldl_n(\wedge))\ True \circ vmap_n\ (sfoldl(\wedge)\ True) \circ vmap_n\ sdiff \circ svstails_n^+$$

Step 4 - Process Refinement

As regards process refinement, almost all of the functionality required to implement this can be taken from our process library. The only piece of bespoke functionality is in the refinement of the $diff$ function. The definition of such a process is, however, fairly trivial:

$$SDIFF\ =\ in.value\ ?\ x\ \rightarrow\ SMAP(\neq x)$$

The above definition can therefore be refined quite straightforwardly to the following network:

$$SVSTAILS_n^+ \gg_n VMAP_n(SDIFF) \gg_n VMAP_n(SFOLDL(\wedge)) \gg_n VFOLDL_n(\wedge)$$

where n is the length of the input list. The results can be seen in Figure 9.9.

Step 5 - Handel-C Implementation

This is given in Figure 9.10.

Results

Results for the distinct elements network implemented on a Xilinx XCV2000e are given in Table 9.2. Again we make the assumption here that the length of time each cycle takes does not change significantly with respect to the problem size.

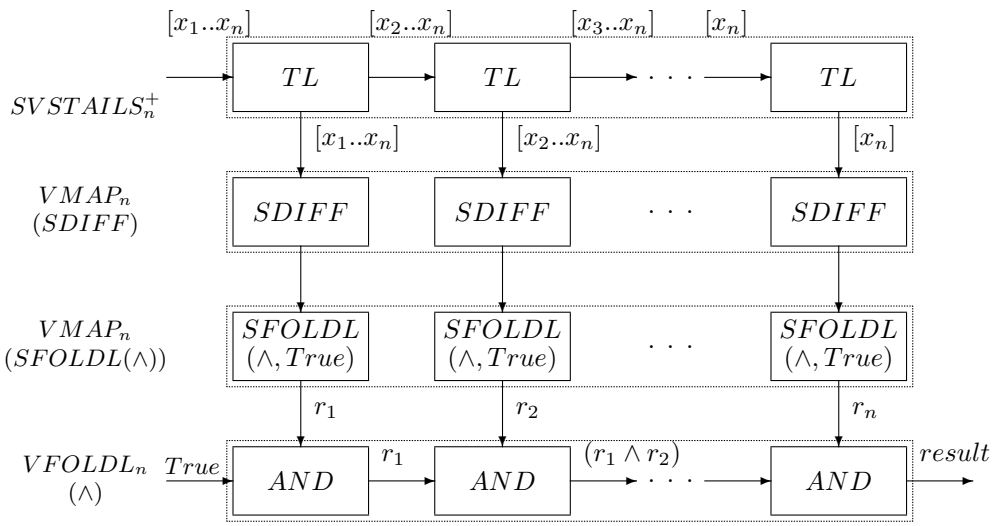


Figure 9.9: A network to solve the distinct elements problem for the list xs

```
macro proc DISTINCT (n,streamin,streamout)
{
    VectorOfStreams (Item(Value),vectora,n);
    VectorOfStreams (Item(Bool), vectorb,n);
    Vector          (Item(Bool), vectorc,n);

    par
    {
        SVSTAILSP (n,streamin,vectora);
        VMAP   (n,vectora,vectorb,SDIFF);
        VMAP   (n,vectorb,vectorc,SFOLDAND);
        VFOLDL (n,vectorc,streamout,AND,True);
    }
}
```

Figure 9.10: Handel-C definition for the distinct elements problem.

As with the minimum distance problem, sequentially we have a quadratic algorithm, but, as these results demonstrate, our parallel hardware implementation has both linear resource usage and execution time. Again the cycle times are very regular - we have, for a problem of size n, a cycle time in each case of precisely $5n + 4$. As before, the slice usage is a slightly less regular, but follows roughly a pattern of about $90n$ slices. To this end, we may hope to be able to implement the network for a problem size of just over 200 on an XCV2000e device. As before, some optimisation may allow us to increase this figure, and larger/multiple devices would of course allow for larger problem sizes to be dealt with.

items	*slices used*	(%)	*cycles*
5	492	2%	29
10	911	4%	54
15	1,345	7%	79
20	1,770	9%	104
25	2,200	11%	129
30	2,622	13%	154
40	3,467	18%	204
60	5,216	27%	304
80	6,944	36%	404
100	8,679	45%	504
120	10,410	54%	604

Table 9.2: Results for the distinct elements problem.

9.4 A JPEG Decoder

A version of this case study has been appeared in [5]. An extended version is given here, with greater emphasis on the concepts such as data refinement which are central to this work.

The JPEG format is widely accepted as the de facto standard for encoding of continuous tone real world images, and is particularly well suited to photographs. It is probably the most widely deployed image format on the internet, and is also now finding many more areas of application besides the PC.

The rise in popularity of digital photography is one such example which has made the proliferation of JPEG images even more pronounced. JPEG was the natural choice for the image format of digital cameras, and almost all such devices implement JPEG encoders and/or decoders in some form. As a predictable consequence of the increased interest in digital photography there has also been a continually increasing set of expectations in terms of image quality. This manifests itself most evidently in the form of a constant demand for higher resolution images. Certain problems are therefore presented to manufacturers of such devices. As one would expect, higher resolution images require more sophisticated technology to manage them, and particularly more time to encode and decode them. Many consumers have noticed that upon 'upgrading' to a higher specification model, they have actually ended up with a device which is slower than their previous one.

These issues are not confined to the digital camera industry. Many other handheld devices now deal with JPEG images and are subject to similar increasing requirements and expectations. In more 'traditional' fields of computing there are also similar issues. Given that more users have

access to high speed internet connections and improved quality display devices, they will naturally also expect to deal with increasingly higher resolution images. This puts an obvious strain on processing resources, one which, in certain areas, the increase in processor speeds may not be able to keep up with. Fields such as astronomy and medicine often rely on extremely high resolution images, where the speed with which they can be processed may often be an important issue.

To summarise, despite the JPEG format having been so widely implemented and deployed, there is still good cause for considering alternative strategies for implementation which may provide improved efficiency. In this section we attempt to prove that our methodology is ideally suited to the task of deriving such an implementation.

Algorithm Design

One of the problems faced when attempting the task of deriving a novel implementation of the JPEG specification is that developers are typically (and quite naturally) very reluctant to re-invent the wheel That is to say, given the widespread use of JPEG and the length of time for which it has been established, there exists already a number of tried and tested libraries. Given the amount of effort and expertise that has been poured into these libraries over the years the average developer would conclude, quite rightly, that there is little scope for improvement, at least, within a traditional software setting. However, whereas the patterns of computation used in such libraries are highly efficient for the traditional computing architectures at which they were targeted, we should not fall into the trap of using these *implementations* as specifications when starting out on a new implementation in a different setting. Imperative languages, as we have already argued, do not provide a good framework for reasoning about, and transforming algorithms. So, as always, we shall begin with a functional specification of a JPEG decoder.

We shall focus our efforts on a decoder for JPEG's baseline DCT method of compression. This is almost certainly the most commonly used method within the JPEG set of standards.

We shall require the use of restart markers in our compressed data. A JPEG decoder must maintain a set of predictors. The predictors will be modified each time a unit of data is decoded, and their values will affect the decoding of each unit. As such, for every single unit in the compressed file, we require that the previous unit has been at least partially decoded before it in turn can be decoded. This makes for a largely sequential decoding process. Thankfully, the JPEG standard recognises applications in which JPEG images might be communicated over unreliable media, and as such, data may have been lost part way through transmission. To this end, the standard includes the definition of restart markers. Whenever one of these markers is encountered, the predictors can be safely reset. This has the effect of defining a number of sections within the compressed data that can be decoded completely independently of each other.

It is important to clearly consider the hierarchy within a compressed JPEG file, when considering writing the specification for a decoder. To begin with we have a file. This can be split

into two areas, the headers and the compressed scan data. The headers contain information about the compressed data (size, format and so on) as well as tables for dequantization and Huffman decoding.

Where restart markers are used, the scan can be decomposed into a number of independent sections which we shall call intervals. An interval can be further decomposed into one or more minimum coding units (MCUs). The number of MCUs per interval is defined in the headers. The MCU is a collection of units. Each unit, when fully decompressed, will form an 8×8 matrix of samples for a given component (usually one of Y, C_b or C_r for colour images). Generally, the chrominance components will be downsampled to achieve better compression. A typical scheme has an MCU representing a 16×16 block of pixels in the fully decoded output image. Within this, there will have been a unique Y (luminance) value for every pixel. However, each chrominance value will be shared by a 2×2 pixel block. As such, an MCU in this scheme will contain four units of Y samples, followed by one of C_b samples, and one of C_r samples.

Step 1 - Initial Specification

We may find the following type definitions useful. A unit is an 8×8 matrix of coefficients (before transformation) or samples (after transformation). An MCU is a list of units. These types may therefore be defined as follows:

$$\begin{aligned} type\ UnitRow &= [Int] \\ type\ Unit &= [UnitRow] \\ type\ MCU &= [Unit] \end{aligned}$$

Now, to consider the functions that will comprise our decoder. At the highest level we require a function that will take in a list of compressed bytes representing the entire file, and will return an uncompressed image.

$$\begin{aligned} &decodeJpeg\ ::\ [Byte] \rightarrow Image \\ &decodeJpeg\ data\ =\ decodeScan\ hdrInfo\ scanData \\ &\quad where\ (scanData, hdrInfo)\ =\ decodeHeaders\ data \end{aligned}$$

An $Image$ here can be considered as a simple two dimensional array of pixel values. This definition relies on two auxiliary definitions. The first decodes the headers in the data, and returns both a $HeaderInfo$ object and a list of the remaining data in the file, following the headers.

$$decodeHeaders\ ::\ [Byte] \rightarrow ([Byte], HeaderInfo)$$

The exact definition of $decodeHeaders$ and the $HeaderInfo$ type will not be shown in full here due to lack of space. Broadly, the header information should include all the numeric parameters and structures required for decoding. The second function, $decodeScan$, is where the bulk of the decoding effort takes place.

$$
\begin{array}{lcl}
decodeScan & :: & HeaderInfo \rightarrow [Byte] \rightarrow Image \\
decodeScan\ hdrInfo & = & composeImage\ hdrInfo\ \circ \\
 & & map\ (decodeInterval\ hdrInfo)\ \circ \\
 & & readIntervals
\end{array}
$$

This function is a composition of three stages. In the first, we use the function *readIntervals* to split the compressed scan data into a list of intervals which can be decoded independently of each other. Next, we map the function *decodeInterval* to each interval in the list of decoded sections within the image. Finally we apply *composeImage* to compose these sections together, a function which we shall keep deliberately vague.

The function *readIntervals* is simple, but crucial in terms of scope for parallelism, as we shall see later. It reads through the input list of bytes, and splits it into sublists based on the occurrence of restart markers. A restart marker will be a single byte with value ff in hex, followed by a value from d0 up to d7. The *encoder* will 'pad' any byte values of ff naturally occurring in the compressed data with a single zero byte to ensure they are never confused with a restart marker. This means that *readIntervals* can safely split up the compressed data without any greater level of detail than simply examining individual byte values. As such, this task should be very fast.

$$readIntervals\ ::\ [Byte] \rightarrow [[Byte]]$$

The next function, *decodeInterval*, will take a list of compressed bytes that form a single interval, and return a list of totally decompressed MCUs that, when reconstructed, will form the corresponding section of the output image. The definition is as follows:

$$
\begin{array}{lcl}
decodeInterval & :: & HeaderInfo \rightarrow [Byte] \rightarrow [MCU] \\
decodeInterval\ hdrInfo & = & map\ (transformMCU)\ \circ \\
 & & intervalToMCUs\ hdrInfo\ \circ \\
 & & bytesToBits
\end{array}
$$

Here again we have a composition of three stages. Firstly, given that Huffman decoding works at the bit rather than byte level (due to the use of variable length codes), we employ *bytesToBits* to transform our input list of bytes into a list of bits. Next we apply $intervalToMCUs$ which should supply us with a list of MCUs, each, at this stage, containing untransformed coefficients. Finally we map $transformMCU$, such that each MCU is transformed from a list of matrices of coefficients to a list of matrices of samples (Y, C_b, and C_r values). The type of $intervalToMCUs$ is as follows:

$$intervalToMCUs\ ::\ HeaderInfo \rightarrow [Bit] \rightarrow [MCU]$$

We shall have to brush somewhat briefly over the goings on inside this function due to lack of space. Suffice to say we shall have a repeated application of a function which reads in an MCU,

Figure 9.11: A demonstration of how a JPEG image can be split into intervals.

and maintains the state of the predictors between calls. Reading an MCU is in turn a repeated application of a function which reads in units.

Let us return now to the function $transformMCU$. This takes an MCU, containing units of untransformed coefficients, and returns an MCU containing units of fully decoded sample data. It maps the function $transformUnit$ to each unit in the MCU.

$$\begin{aligned} &transformMCU \;::\; HeaderInfo \to MCU \to MCU \\ &transformMCU\ hdrInfo \;=\; map\ transformUnit \end{aligned}$$

The $transformUnit$ function performs the familiar stages of transforming an 8×8 unit of coefficients into an 8×8 unit of output sample values. Firstly it performs zig-zag reordering, then dequantization (making use of the appropriate quantization table in the $HeaderInfo$ structure), and finally applies the inverse discrete cosine transform.

$$\begin{aligned} &transformUnit \;::\; HeaderInfo \to Unit \to Unit \\ &transformUnit\ hdrInfo \;=\; idct \circ dequantize\ hdrInfo \circ zigzag \end{aligned}$$

Step 2 - Program Transformation

The majority of interesting functionality in the specification is concealed within the function $decodeInterval$. Given that an MCU is a list of units, and the number of units per MCU can be derived from the header information, it should be straightforward to flatten a list of MCUs into units and vice verse. This can be achieved with the functions $unitsToMCUs$ and $MCUsToUnits$. Thus, with a little simple program transformation, we can arrive at the following definition:

$$
\begin{aligned}
&decodeInterval'\ hdrInfo \\
&= \ unitsToMCUs\ hdrInfo \circ \\
&\quad map\ idct \circ map\ (dequantize\ hdrInfo) \circ map\ zigzag \circ \\
&\quad MCUsToUnits \circ intervalToMCUs\ hdrInfo \circ bytesToBits
\end{aligned}
$$

We may find the following 'shortcut' useful:

$$
\begin{aligned}
&intervalToUnits\ hdrInfo \\
&= \ MCUsToUnits \circ intervalToMCUs\ hdrInfo \circ bytesToBits
\end{aligned}
$$

Allowing us to instead write:

$$
\begin{aligned}
&decodeInterval'\ hdrInfo \\
&= \ unitsToMCUs\ hdrInfo \circ \\
&\quad map\ idct \circ map\ (dequantize\ hdrInfo) \circ map\ zigzag \circ \\
&\quad (intervalToUnits\ hdrInfo)
\end{aligned}
$$

Step 3 - Data Refinement

For the algorithm as a whole, we wish to be able to process each interval of the image independently in parallel. To this end, we aim to refine *readIntervals* to some form which can output a vector. We can then apply our refinement of *decodeInterval*, using a vector interpretation of *map*, to each interval. Finally we shall also require some refinement for *composeImage* which accepts input as a vector. Broadly, we have the following specification for our vector refined version of *decodeScan*:

$$
\begin{aligned}
vdecodeScan\ hdrInfo \ = \ & vcomposeImage\ hdrInfo \circ \\
& vmap\ (decodeInterval'\ hdrInfo) \circ \\
& vreadIntervals
\end{aligned}
$$

Looking now at *decodeInterval*, we should determine the communication method that will be used between subsequent stages in this part of the algorithm. Let us consider a refinement which relies on streams at the top level. A data refined version of *intervalToUnits* could produce a stream of units, which then passes through the usual zig-zag, dequantization and IDCT steps, each stage both inputttting and outputting a stream of units. We have:

$$
\begin{aligned}
&sdecodeInterval\ hdrInfo \\
&= \ sunitsToMCUs\ hdrInfo \circ \\
&\quad smap\ idct \circ smap\ (dequantize\ hdrInfo) \circ smap\ zigzag \circ \\
&\quad (sintervalToUnits\ hdrInfo)
\end{aligned}
$$

At a lower level than this we need to consider how each individual unit is to be communicated. Recall from earlier that each unit is an 8×8 block of integer values. This is represented in the specification as a list of lists. A number of option present themselves to us for how this could be

refined. A vector of vectors would effectively allow us to communicate all 64 values in a single step, but would require a large degree of parallelism. A stream of streams, at the other extreme, would require 64 steps to communicate the whole unit, but require no parallelism at all. Between the two we have the vector of streams, and the stream of vectors, each of which could communicate the unit in eight steps, but require eight independent processing resources. Let us go for one of the two in the middle - and, given the very regular dimensions of the structure to communicate, we shall employ the stream of vectors.

So we shall require a suite of data refined functions which work in terms of streams of vectors. Taking for example *idct*, we have the following data refinement:

$$svidct \quad :: \quad \lfloor \langle Int \rangle_8 \rfloor \rightarrow \lfloor \langle Int \rangle_8 \rfloor$$

Given these we can provide the following definition for *sdecodeInterval*:

$$\begin{array}{l} sdecodeInterval\ hdrInfo \\ = \quad sunitsToMCUs\ hdrInfo \circ \\ \quad smap\ svidct \circ smap\ (svdequantize\ hdrInfo) \circ smap\ svzigzag \circ \\ \quad (sintervalToUnits\ hdrInfo) \end{array}$$

Step 4 - Process Refinement

Concentrating just on the task of decoding a single interval, this compositional form for *sdecodeInterval* is now well suited to process refinement. We can refine this to the following network of CSP processes:

$$\begin{array}{l} SDECODEINTERVAL \\ = \quad SINTERVALSTOUNITS \gg \\ \quad SMAP(SVZIGZAG) \gg \\ \quad SMAP(SVDEQUANT) \gg \\ \quad SMAP(SVIDCT) \gg \\ \quad SUNITSTOMCUS \end{array}$$

We can specify process alphabets to reflect the communication types between stages, for example:

$$\alpha SVIDCT \quad :: \quad \{in :: \lfloor \langle Int \rangle_8 \rfloor,\ out :: \lfloor \langle Int \rangle_8 \rfloor\}$$

Considering the algorithm as a whole, we have:

$$\begin{array}{l} VDECODESCAN_n \\ = \quad VREADINTERVALS \gg_n \\ \quad VMAP(SDECODEINTERVAL) \gg \\ \quad VCOMPOSEIMAGE \end{array}$$

The resulting network is depicted in Figure 9.12.

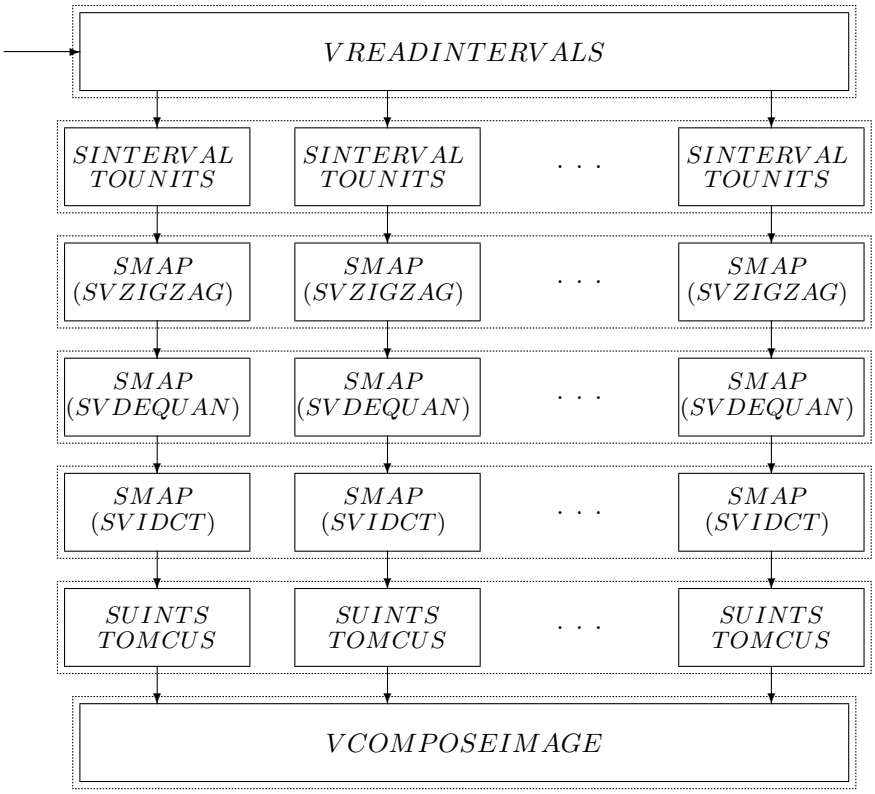


Figure 9.12: The JPEG decoder process network.

Step 5 - Handel-C Implementation

The Handel-C implementation `SDECODEINTERVAL` is given in Figure 9.13 - this can follow naturally from the CSP definition for the $SDECODEINTERVAL$ given above. Given this definition for `SDECODEINTERVAL`, we can construct our implementation for the process $VDECODESCAN$. The Handel-C implementation for `VDECODESCAN` is given in Figure 9.14.

Results

The resulting design was too large to be implemented on the available device (a Xilinx XCV2000e), despite some concerted efforts in optimisation. Even with an effectively non-parallel version (i.e. the number of intervals which can be processed concurrently reduced to just one), the place and route tools were never able to successfully map the full design. The individual components of the JPEG decoder - particularly the IDCT - result in significant resource usage. At best, the requirements were at about 125% of the available resources. It seems likely therefore that using a device twice as big as the XCV2000e, that the design could be implemented, and there may even be room for some parallelism.

Given more time, it may be possible to look at some alternative approaches here, which may allow this design to fit, even on this smaller device. It may be possible to blend VHDL with Handel-

```
macro proc SDECODEINTERVAL (in,out)
{
   StreamOfUnits smida, smidb, smidc, smidd;

   par
   {
      SINTERVALTOUNITS (in,smida);
      SMAP (smida,smidb,SVZIGZAG);
      SMAP (smidb,smidc,SVDEQUANTIZE);
      SMAP (smidc,smidd,SVIDCT);
      SUNITSTOMCUS (smidd,out);
   }
}
```

Figure 9.13: The SDECODEINTERVAL process.

```
macro proc VDECODESCAN (in,out)
{
   VectorOfStreams (n,Byte) vmida;
   StreamOfVectors (n,Byte) vmidb;
   par
   {
      VREADINTERVALS (n,in,vmida);
      VMAP(n,vmida,vmidb,SDECODEINTERVAL);
      VCOMPOSEIMAGE (vmidb, out);
   }
}
```

Figure 9.14: The VDECODESCAN process.

C, for example. It is likely that a VHDL implementation can replace some of the functionality currently implemented in Handel-C with lower overheads in terms of resource usage. This is, of course, at the price of adding complexity into the design process. Having said that, components such as the IDCT have been widely researched in reconfigurable logic, and a number of VHDL implementations exist. This may prove an interesting area of future work - looking at how the design can be composed of modules implemented in more than one language.

Another approach may be to look at a hybrid implementation - with some of the functionality performed by the FPGA, and some implemented more conventionally in software. This also suggests an interesting area of further work in looking at how this methodology can be expanded to incorporate such hybrid systems whilst still ensuring correctness.

Other Implementations

Designs for JPEG decoders have been successfully implemented elsewhere in reconfigurable logic on devices of a smiliar size - see for example [35]. This particular implementation was achieved purely using VHDL, and this highlights the overhead introduced by implementation using our methodology - particularly as we rely on Handel-C. In the section on future work we look at

how targeting other implementation languages with our methodology may help to address this issue. As noted previously, the sacrifice made in resorting to VHDL is that it introduces significant complexity and, in the absence of a formal development framework, makes asserting the correctness of the resulting implementation far more difficult.

9.5 DNA Processing

This case study was originally presented in [6]. Here we introduce an extended version of that work, taking the specification from there and including it in a fuller derivation. We add the notions of data refinement introduced in this work, and also add a full implementation in Handel-C.

The field of genetics has blossomed over the last few years, particularly as a result of the high profile Human Genome Project. To this end there is now a large and ever increasing availability of DNA information in digital formats. However, despite the ready availability of this information, the task of processing and interpreting it is by no means trivial.

To begin with, the large volumes of data to be dealt with need to be taken into account. Humans have 23 different types of chromosome, which as a diploid species is doubled to 46 (one from each parent). These chromosomes range in size from 46 million to over 240 million bases. Given that each base is one of four possible combinations, we are looking at something in the region of 90 Megabytes of data for even the smallest of chromosomes. Add to this the potential for experiments and investigations in genetic science to involve thousands of individuals and we are dealing with a very large scale of data indeed.

If the task of interpretation merely involved conventional (i.e. exact) string matching then such volumes of data may seem relatively manageable. The field of string matching has been extensively studied and highly efficient linear algorithms exist in the case that only exact matches are required [51, 25]. However, such exact matches would often not be of great use to genetic scientists. Sequences of DNA are subject to change through reproduction, and portions of a sequence may be deleted, inserted or mutated over time. As such the operation required is closer to that of the edit distance [90] or longest common subsequence. The edit distance is a measurement of the amount of operations (insertions, deletions and substitutions) required to translate one string into another. The longest common subsequence can be seen as a special case of edit distance, wherein substitutions are not allowed.

Step 1 - Initial Specification

Let us briefly consider the issue of representation. We can represent a DNA sequence as a string of characters, where each character represents a base. There are four possible bases - A, C, G and T. In Haskell we might construct a simple data type to model this, as follows:

$$\begin{aligned} \mathit{data}\ \mathit{Base} &= A \mid C \mid G \mid T \\ \mathit{type}\ \mathit{Sequence} &= [\mathit{Base}] \end{aligned}$$

Let us consider the problem of attempting to find all matching sub-segments of two lists. The task of producing all possible sub-segments of a list is a costly one. For a list of size n, there are $O(n^2)$ possible sub-segments (see Section 7.16). Were we to apply a brute force approach here, taking the Cartesian product (see Section 7.14) of the two lists of sub-segments to compare each

and every one, we would be looking at an unmanageable $O(n^4)$ algorithm. We have something along the lines of the following:

$$ams\ xs\ ys\ =\ map\ (uncurry\ (=))\ (cp\ (segments^+\ xs)\ (segments^+\ ys))$$

Thankfully much of that effort would of course be redundant. Many of the segments of a list are naturally prefixes of other segments of that list. When comparing two lists s and t of equal length, should we determine they are not equal in value, then we can also assert that for any other two lists a and b that $(s +\!\!+ a) \neq (t +\!\!+ b)$. This allows us to 'cut short' a lot of unnecessary comparisons. This redundancy should become particularly apparent if we consider the definition for $segments^+$ (see Section 7.16 for more details):

$$segments^+\ =\ fold\ (+\!\!+) \circ map\ inits^+ \circ tails^+$$

Here we are mapping $inits^+$ to the result of $tails^+$ (see Section 7.12). That is to say for each final segment of the list we compute every initial segment. Recalling for a moment the definition of $inits^+$, we have:

$$inits^+\ [x_1, x_2, \ldots, x_n]\ =\ [[x_1], [x_1, x_2], \ldots, [x_1, x_2, \ldots, x_{n-1}], [x_1, x_2, \ldots, x_n]]$$

Where we are attempting to match the results of $inits^+\ xs$ with some sequence ys it should be clear that this task can be performed in a single linear time pass. Each list in the output of $inits^+\ xs$ contains the previous list as a prefix, and any comparison effort need not be duplicated. So, to illustrate, our 'brute force' approach for finding all initial segments of xs which match some sequence ys would be as follows:

$$ami\ xs\ ys\ =\ map\ (=ys)\ inits^+\ xs$$

The above would require quadratic time. However, we can achieve the same result with this simple linear time recursive definition:

$$\begin{array}{lcl} ami\ []\ ys & = & [] \\ ami\ (x:xs)\ [] & = & map\ (const\ False)\ (x:xs) \\ ami\ (x:xs)\ (y:ys) & = & \text{if } (x = y) \\ & & \text{then } True\ :\ (ami\ xs\ ys) \\ & & \text{else } map\ (const\ False)\ (x:xs) \end{array}$$

The output of this function will be a list of Booleans, of length n (the length of xs). The first k values will be $True$ (where k is in the range 0 to n), and the remaining $(n - k)$ values will be $False$. The regular nature of this list suggests that it may be more convenient to simply return the value k, which is in effect the length of the longest common prefix. We can define such a function, $llcp$, in terms of ami by simply summing the list - counting all $True$ values as one, and all $False$ values as zero:

$$\begin{aligned} llcp\ xs\ ys &= foldr\ (+)\ 0\ (map\ f\ (ami\ xs\ ys)) \\ &\quad where\ f\ b\ =\ \text{if } b \text{ then 1 else 0} \end{aligned}$$

Alternatively, to produce a 'stand alone' definition, we have (as seen in [6]):

$$\begin{aligned} llcp\ []\ ys &= 0 \\ llcp\ xs\ [] &= 0 \\ llcp\ (x:xs)\ (y:ys) &= \text{if } (x = y) \\ &\quad \text{then } 1\ +\ (llcp\ xs\ ys) \\ &\quad \text{else } 0 \end{aligned}$$

We can construct a definition of our previous function *ami* in terms of *llcp*, should we still require output as a list of Booleans. We have:

$$\begin{aligned} ami\ xs\ ys &= take\ \text{k}\ (repeat\ True) +\!\!+ take\ \text{(n-k)}\ (repeat\ False) \\ &\quad where\ k\ =\ llcp\ xs\ ys \\ &\quad\quad n\ =\ length\ xs \end{aligned}$$

Given *llcp* we can construct a variant of our original definition which gives us the lengths of all maximal common subsegments. We have:

$$lmcsa\ s\ t\ =\ map\ (uncurry\ llcp)\ (cp\ (tails^+\ s)\ (tails^+\ t))$$

Alternatively we may wish to replace *cp* with a list comprehension, which might provide for slightly easier reading:

$$lmcsa\ s\ t\ =\ [llcp\ a\ b\ \mid\ a \leftarrow tails^+\ s,\ b \leftarrow tails^+\ t]$$

Step 2 - Program Transformation

Our function *lmcsa* producea the results as a single list, whereas a matrix may well be of more use to us. Using the laws of list comprehension (see Section 7.6), particularly $\{LC7\}$, we can transform the above to the following:

$$lmcsa\ s\ t\ =\ (fold\ (+\!\!+) \circ transpose)\ [[llcp\ a\ b\ \mid\ a \leftarrow tails^+\ s]\ \mid\ b \leftarrow tails^+\ t]$$

We can do away with the *fold* and *transpose* here, to give us just the following. This gives us the same results, but simply in a different order and structure.

$$lmcsb\ s\ t\ =\ [[llcp\ a\ b\ \mid\ a \leftarrow tails^+\ s]\ \mid\ b \leftarrow tails^+\ t]$$

The results obtained from this function, applied to the test sequences s = `ATCCATGTCATC` and t = `CTATCTCATCG`, are shown in Figure 9.15.

		s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}
		A	T	C	C	A	T	G	T	C	A	T	C
t_1	C	0	0	1	1	0	0	0	0	1	0	0	1
t_2	T	0	1	0	0	0	1	0	1	0	0	1	0
t_3	A	3	0	0	0	2	0	0	0	0	3	0	0
t_4	T	0	2	0	0	0	1	0	2	0	0	2	0
t_5	C	0	0	1	1	0	0	0	0	1	0	0	1
t_6	T	0	2	0	0	0	1	0	5	0	0	2	0
t_7	C	0	0	1	3	0	0	0	0	4	0	0	1
t_8	A	3	0	0	0	2	0	0	0	0	3	0	0
t_9	T	0	2	0	0	0	1	0	2	0	0	2	0
t_{10}	C	0	0	1	1	0	0	0	0	1	0	0	1
t_{11}	G	0	0	0	0	0	0	1	0	0	0	0	0

Figure 9.15: The matrix output by the function *lmcsb*.

Let us take a moment to consider how the matrix depicted in Figure 9.15 should be interpreted. If the sequence s is of length n and t is of length m, then we have indices (i, j) running from $(1, 1)$ in the top left hand corner of the matrix to (n, m) in the bottom right hand corner. Each column i represents the i^{th} element of $(tails^+\ s)$ compared with each element of $(tails^+\ t)$. Similarly each row j represents the j^{th} element of $(tails^+\ t)$ compared with each element of $(tails^+\ s)$. At a given index of (i, j), a value of k represents the result of *llcp* applied to the relevant final segments from s and t at that point. In other words a value k at (i, j) means the next k items, from the i^{th} element in s, and the j^{th} element in t, are equal. The longest common subsequence begins in the matrix at position $(8, 6)$ and has a length of five. This is the sequence `TCATC`.

We are of course at liberty to re-arrange our output matrix into whichever form suits us best. See Section 7.15 for a selection of matrix transformation functions such as flipping, rotating and so on. One alternative format for our matrix would be the result of *lmcsb* rotated through 180 degrees. We have:

$$lmcsc\ xs\ ys\ =\ rotate180\ [[llcp\ x\ y\ |\ x \leftarrow tails^+\ xs]\ |\ y \leftarrow tails^+\ ys]$$

Given the definition for $rotate180$ (see Section 7.15) we can substitute the above for an alternative definition, using $fins^+$ instead of $tails^+$:

$$\begin{aligned}
lmcsc\ xs\ ys &= rotate180\ [[llcp\ x\ y\ |\ x \leftarrow tails^+\ xs]\ |\ y \leftarrow tails^+\ ys] \\
&= (fliph \circ flipv)\ [[llcp\ x\ y\ |\ x \leftarrow tails^+\ xs]\ |\ y \leftarrow tails^+\ ys] \\
&= (map\ reverse \circ reverse)\ [[llcp\ x\ y\ |\ x \leftarrow tails^+\ xs]\ |\ y \leftarrow tails^+\ ys] \\
&= map\ reverse\ [[llcp\ x\ y\ |\ x \leftarrow tails^+\ xs]\ |\ y \leftarrow tails^+\ (reverse\ ys)] \\
&= [[llcp\ x\ y\ |\ x \leftarrow tails^+\ (reverse\ xs)]\ |\ y \leftarrow tails^+\ (reverse\ ys)] \\
&= [[llcp\ x\ y\ |\ x \leftarrow fins^+\ xs]\ |\ y \leftarrow fins^+\ ys]
\end{aligned}$$

We now have a definition in a similar form to that given for *lmcs* in [6]. Let us consider the output of our new variant *lmcsc* applied again to the sequences s = `ATCCATGTCATC` and t =

`CTATCTCATCG`. The resulting matrix is given in Figure 9.16. Note that the labels used on the table are the sequences in reverse.

		s_{12}	s_{11}	s_{10}	s_9	s_8	s_6	s_6	s_5	s_4	s_3	s_2	s_1
		C	T	A	C	T	G	T	A	C	C	T	A
t_{11}	G	0	0	0	0	0	1	0	0	0	0	0	0
t_{10}	C	1	0	0	1	0	0	0	0	1	1	0	0
t_9	T	0	2	0	0	2	0	1	0	0	0	2	0
t_8	A	0	0	3	0	0	0	0	2	0	0	0	3
t_7	C	1	0	0	4	0	0	0	0	3	1	0	0
t_6	T	0	2	0	0	5	0	1	0	0	0	2	0
t_5	C	1	0	0	1	0	0	0	0	1	1	0	0
t_4	T	0	2	0	0	2	0	1	0	0	0	2	0
t_3	A	0	0	3	0	0	0	0	2	0	0	0	3
t_2	T	0	1	0	0	1	0	1	0	0	0	1	0
t_1	C	1	0	0	1	0	0	0	0	1	1	0	0

Figure 9.16: The matrix output by the function *lmcsc*.

It should hopefully be clear that the matrix presented in Figure 9.16 is the same as that in Figure 9.15, but simply rotated by 180 degrees. Viewing the matrix in this orientation is useful to us as it helps to highlight an area for improving efficiency. As our results are presented in a matrix of this form we have an output structure which has $O(nm)$ size with respect to its input lists of length n and m. Each value in the matrix requires an application of *llcp* which requires $O(n)$ time to calculate. As such we currently have an algorithm with cubic efficiency.

However, examination of the matrix in Figure 9.16 should reveal that in fact calculation of each individual value should be possible in constant time. Each value k at index (i, j) has a simple relationship with the value to the top left at index $(i-1, j-1)$. If the corresponding values from s and t at (i, j) are equal, then the value at (i, j) will be one higher than the value at $(i-1, j-1)$. If the values from s and t at that point are not equal, then the value at (i, j) will be zero.

This relationship is a direct result of our definition for *llcp*. Recall that we have:

$$\begin{aligned} llcp\ (x : xs)\ (y : ys) \quad = \quad & \text{if } (x = y) \\ & \text{then } 1 \ + \ (llcp\ xs\ ys) \\ & \text{else } 0 \end{aligned}$$

The crucial part here is the recursion. For two sequences $(x : xs)$ and $(y : ys)$, where $x = y$, we can calculate the result in constant time based on the partial result of $(llcp\ xs\ ys)$. In the matrix shown in Figure 9.16, the column i represents the final segment of s with values $[s_{(n+1)-i}, s_{(n+1)-(i-1)}, ..., s_n]$. Similarly the column j represents the final segment of t with values $[t_{(m+1)-j}, t_{(m+1)-(j-1)}, ..., t_m]$. So, given that we can calculate the value k at (i, j) as follows:

$$k_{(i,j)} \ = \ llcp\ [s_{(n+1)-i}, s_{(n+1)-(i-1)}, ..., s_n]\ [t_{(m+1)-j}, t_{(m+1)-(j-1)}, ..., t_m]$$

Then implicitly, given the following implication about the nature of $fins^+$:

$$\big((x : xs) \;=\; (fins^{+}\; s)_i\big) \;\Rightarrow\; \big(xs \;=\; (fins^{+}\; s)_{i-1}\big)$$

For k at $(i-1, j-1)$ our value is given by:

$$k_{(i-1,j-1)} \;=\; llcp\; [s_{(n+1)-(i-1)}, ..., s_n]\; [t_{(m+1)-(j-1)}, ..., t_m]$$

Thus we can say:

$$\begin{array}{lll} k_{(i,j)} & = & \text{if } (s_{(n+1)-i} \;=\; t_{(m+1)-j}) \\ & & \text{then } 1 \;+\; k_{(i-1,j-1)} \\ & & \text{else } 0 \end{array}$$

We can now compute the result for each item of the matrix in constant time, giving us an overall algorithm of quadratic efficiency. Let us consider how we may construct a complete definition for this quadratic time variant.

First we have a function *line* which generalises our above calculation for $k_{(i,j)}$ to calculate the values for an entire row or column. This is supplied with a single value c from one DNA string, and the entirety of the other string. It is also supplied with the results for the previous row/column. The parameter *pr* here is used to store the previous result, given that our point of reference is offset both horizontally and vertically.

$$\begin{array}{lll} line\; x\; pr\; []\; [] & = & [] \\ line\; x\; pr\; (y : ys)\; (r : rs) & = & (\text{if } (x = y) \text{ then } pr + 1 \text{ else } 0)\; :\; (line\; x\; r\; ys\; rs) \end{array}$$

This then allows us to give a definition for our algorithm in terms of $unfold$. Here at each stage of the evaluation we appeal to *line* to produce a row/column of results based on the previous row/column of results, a single item from one query string, and the entirety of the other query string. A simple function *stage* here is used to simply manage the correct placement of parameters and results - all the interesting functionality remains in *line*.

$$\begin{array}{lll} lmcsd\; s\; t & = & unfoldr\; (stage\; (reverse\; s))\; (null \circ snd)\; (rs, reverse\; t) \\ & & where\; rs \;=\; map\; (const\; 0)\; s \\ stage\; ys\; (rs, x : xs) & = & (ns, (ns, xs)) \\ & & where\; ns \;=\; line\; x\; 0\; ys\; rs \end{array}$$

The above will produce a matrix row by row, equivalent to the output of *lmcsc*. If we wish instead to produce the output column by column we simply need to swap the parameters, to give us the following:

$$\begin{array}{lll} lmcse\; s\; t & = & unfoldr\; (stage\; (reverse\; t))\; (null \circ snd)\; (rs, reverse\; s) \\ & & where\; rs \;=\; map\; (const\; 0)\; t \end{array}$$

We can convert between the two output formats of these two variants by means of *transpose*:

$$lmcse\ s\ t\ =\ transpose\ (lmcsd\ s\ t)$$

We may also wish to provide a variant wherein intermediate results are paired with corresponding values from the query string, and passed between successive stages in this format. We have:

$$\begin{array}{lcl} line'\ x\ pr\ [] & = & [] \\ line'\ x\ pr\ ((y,r):rs) & = & (y, \text{if } (x=y) \text{ then } pr+1 \text{ else } 0)\ :\ (line'\ x\ r\ rs) \\ lmcsf\ s\ t & = & unfoldr\ stage'\ (null \circ snd)\ (rs, reverse\ t) \\ & & where\ rs\ =\ map\ (\lambda y \bullet (y,0))\ (reverse\ s) \\ stage'\ (rs, x:xs) & = & (map\ snd\ ns, (ns, xs)) \\ & & where\ ns\ =\ line'\ x\ 0\ rs \end{array}$$

Step 3 - Data Refinement

Let us consider how our we might refine the types used in our function for finding the lengths of maximal common sub-segments. Our *lmcs* functions generally have types of the form:

$$lmcs\ ::\ [A]\ \rightarrow\ [A]\ \rightarrow\ [[Int]]$$

Or more specifically, when dealing with DNA sequences we have:

$$lmcs\ ::\ [Base]\ \rightarrow\ [Base]\ \rightarrow\ [[Int]]$$

Here we have a two dimensional output - the matrix of results. As we have seen in past examples, the best mechanisms for dealing with such two dimensional structures are either the vector of streams or the stream of vectors. Both of these should facilitate the communication of a quadratic sized structure in linear time. Assuming we wish to input both sequences as streams, a data refinement for a vector of streams output might take the following form.

$$vslmcs\ ::\ \lfloor Base \rfloor\ \rightarrow\ \lfloor Base \rfloor\ \rightarrow\ \langle \lfloor A \rfloor \rangle_n$$

Similarly a data refinement for a stream of vectors output might take the following form:

$$svlmcs\ ::\ \lfloor Base \rfloor\ \rightarrow\ \lfloor Base \rfloor\ \rightarrow\ \lfloor \langle A \rangle_m \rfloor$$

In the above the value n represents the length of the first input sequence (s) and the value m represents the length of the second input sequence (t).

Given that our final definitions for *lmcs* were in terms of $unfoldr$, the output type will be determined by how we choose to interpret $unfoldr$ in our data refinement phase. If we wish to achieve a vector of streams, we will need to appeal to $vunfoldr$. If we wish to achieve a stream of vectors we will need to appeal to $sunfoldr$. Correspondingly the manner in which the types for

our function *line* (and implicitly *stage*) are refined will also need to be carefully considered. Let us consider the types of these functions here. We have:

$$\begin{array}{lcl} line' & :: & Base \ \rightarrow \ Int \ \rightarrow \ [(Base, Int)] \ \rightarrow \ [(Base, Int)] \\ stage' & :: & ([(Base, Int)], [Base]) \ \rightarrow \ ([Int], ([(Base, Int)], [Base])) \end{array}$$

The important types to focus on here are type of the final results returned by each stage - $[Int]$ - and the types of intermediate values passed between each stage - $[(Base, Int)]$. To simplify matters, let us ignore for the time being the act of distributing values from s over each instance of *stage*, and assume instead we have a function *stage* with a pre-loaded value x. As such we have a function of the following type:

$$stage_x \quad :: \quad [(Base, Int)] \ \rightarrow \ ([Int], [(Base, Int)])$$

Here the required functionality of *stage* is clearer. It should take in a list of intermediate values from the previous stage and output a pair - a list of final results, and a list of intermediate values to pass to the next stage.

For constructing *vslmcs*, we shall require that a stream is output at each stage. Here it may well also make sense to pass the intermediate values between stages in stream form. We have:

$$sstage_x \quad :: \quad \lfloor (Base, Int) \rfloor \ \rightarrow \ (\lfloor Int \rfloor, \lfloor (Base, Int) \rfloor)$$

For constructing *svlmcs*, we shall require that a vector is output at each stage. However, given that these vectors are output in a stream of vectors, we may not require communication between individual stages - indeed they may not be manifested as separate processes. So here we leave the intermediate values in list form.

$$vstage_x \quad :: \quad [(Base, Int)] \ \rightarrow \ (\langle Int \rangle_n, [(Base, Int)])$$

Step 4 - Process Refinement

Let us consider a process $SSTAGE(x)$ which refines the functionality of the data refined $sstage_x$ presented previously. Here we have a process which inputs a stream of intermediate values (pairs of bases and integers) and outputs a stream of results, as well as a stream of intermediate values to feed to the next stage. Such a process would have the following alphabet:

$$\alpha \ SSTAGE(x) \ = \ \{in :: \lfloor (Base, Int) \rfloor, \ outd :: \lfloor Int \rfloor, \ outr :: \lfloor ((Base, Int)) \rfloor\}$$

We can define this process as follows in CSP:

$$
\begin{array}{lll}
SSTAGE(x) & = & pr := 0; \\
 & & \mu\, X \bullet \quad in.eot\ ?\ any\ \rightarrow outd.eot\ !\ True\ \rightarrow outr.eot\ !\ True \rightarrow SKIP \\
 & & \quad | \\
 & & \quad in.value\ ?\ (y,n)\ \rightarrow \left(\begin{array}{l} \text{if } (x = y) \\ \text{then } r := pr + 1 \\ \text{else } r := 0; \end{array} \right) pr := n; \\
 & & \quad\quad outd.value\ !\ r\ \rightarrow outr.value\ !\ (y,r) \rightarrow X
\end{array}
$$

The process $SSTAGE(x)$ is depicted in Figure 9.17.

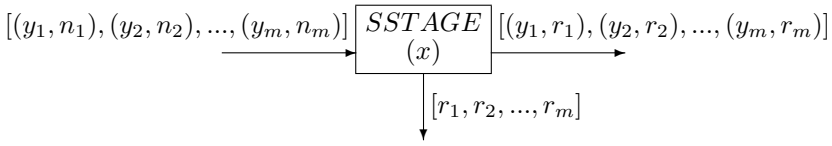


Figure 9.17: The process SSTAGE(x).

We can then compose together a number of instances of this process, following the pattern of $VUNFOLDR$, to form the process $VSLMCS$.

$$
\begin{array}{lcl}
VSLMCS_n(s) & = & \left(P_{initial} \;||\; \left(\begin{array}{c} n-1 \\ || \quad P_i \\ i = 2 \end{array} \right) \;||\; P_{final} \right) \\
P_{initial} & = & SSTAGE(s_n)[in/in, out_1/outd, mid_1/outr] \\
P_i & = & SSTAGE(s_{(n+1)-i})[mid_{i-1}/in, out_i/outd, mid_i/outr] \\
P_{final} & = & SSTAGE'(s_1)[mid_{n-1}/in, out_n/outd]
\end{array}
$$

The process $VSLMCS_n$ depicted in Figure 9.18.

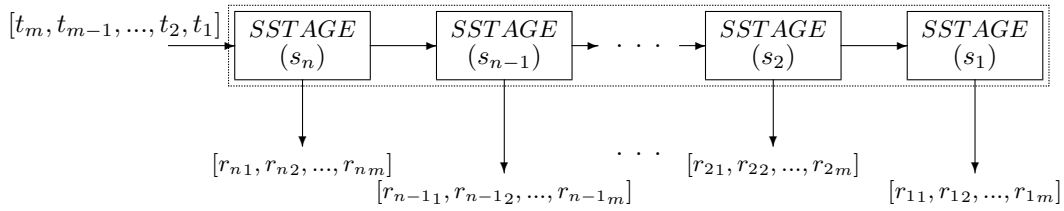


Figure 9.18: The VSLMCS process.

The process $SSTAGE'(x)$ above is a simple variant of $SSTAGE$ which, as it will be used as the final stage in the network, is not required to output intermediate results on the conduit *outr*. This has the following alphabet:

$$\alpha\ SSTAGE'(x)\ =\ \{in :: \lfloor (Base, Int) \rfloor,\ outd :: \lfloor Int \rfloor\}$$

We can define this process as follows in CSP:

$$
\begin{array}{l}
SSTAGE'(x) \\
\quad = \quad pr := 0; \\
\qquad \mu X \bullet \quad in.eot \ ?\ any \ \rightarrow outd.eot \ !\ True \ \rightarrow SKIP \\
\qquad\qquad\qquad\quad \mid \\
\qquad\qquad\qquad\quad in.value \ ?\ (y,n) \ \rightarrow \left(\begin{array}{l} \text{if } (x = y) \\ \text{then } r := pr + 1 \\ \text{else } r := 0 \end{array} \right); pr := n;\ outd.value \ !\ r \ \rightarrow X
\end{array}
$$

Step 5 - Handel-C Implementation

A Handel-C implementation can follow in a straightforward manner from the CSP definitions given above. The one technical issue we have to deal with is deciding how to represent the paired values passed between stages when it comes to communication using Handel-C channels. There are two options are. One is to use two independent channels, one for the y value (the base) and another for the n value (the intermediate result). However, given that the two values are always transmitted at the same time this may seem somewhat wasteful of resources. Perhaps a better solution would be to use Handel-C's bit manipulation operators to combine the two values before sending them on a channel, and then split them again after reception.

Our base values, as already noted, can be just one of four possible combinations, and therefore we can represent them as two bit unsigned integers. Our result values can again be represented as unsigned integers, and the required bit width will depend on the lengths of the sequences we are examining. Some simple definitions and macros may prove helpful here.

```
#define RESULTBITS 8
#define BASEBITS 2
typedef unsigned int RESULTBITS Result;
typedef unsigned int BASEBITS Base;
typedef unsigned int (BASEBITS+RESULTBITS) BRPair;
macro expr brpair(b,r) = b @ r;
macro expr brfst(br) = br[RESULTBITS+(BASEBITS-1):RESULTBITS];
macro expr brsnd(br) = br[RESULTBITS-1:0];
```

So, our CSP process $SSTAGE$ can be implemented with the Handel-C process `SSTAGE`. This is given in Figure 9.19.

Note that some small optimisations may be possible with the code presented in Figure 9.19. Local variables in Handel-C have an implicit cost in gates, and it can often be beneficial to use a single local variable for more than one purpose where possible. So, for example, a few small modifications to the above could allow us to replace the variables `r` and `pr` with just a single variable, which would result in a smaller eventual circuit. However, for clarity we shall stick to the above form.

```
macro proc SSTAGE(inl,outd,outr,x)
{
    Bool eot;
    BRPair y_n;
    Result r, pr;
    eot = False;
    pr = 0;
    do
    {
        prialt
        {
            case inl.value ? y_n:
                if (x == brfst(y_n)) r = pr + 1;
                else r = 0;
                outr.value ! brpair(brfst(y_n),r);
                outd.value ! r;
                pr = brsnd(y_n);
                break;
            case inl.eot ? eot:
                outr.eot ! True;
                outd.eot ! True;
                break;
        }
    } while (!eot);
}
```

Figure 9.19: Handel-C implementation of the SSTAGE process.

Given our implementation of `SSTAGE` we can now construct a Handel-C implementation for our overall network `VSLMCS`. This is presented in Figure 9.20. The process `SSTAGEb` is a variant of `SSTAGE` which does not output intermediate values, analogous to the CSP process $SSTAGE'$.

```
macro proc VSLMCS(n,streamin,out,s)
{
    typeof(streamin) mid[n];
    par (c=0;c<n;c++)
    {
        ifselect (c==0)
            SSTAGE (streamin,out[0],mid[0],s[n-1]);
        else ifselect (c<(n-1))
            SSTAGE (mid[c-1],out[c],mid[c],s[n-(c+1)]);
        else // ifselect (c==n-1)
            SSTAGEb (mid[c-1],out[c],s[0]);
    }
}
```

Figure 9.20: Handel-C implementation of the VSLMCS process.

Results

Results for the DNA processing network implemented on a Xilinx XCV2000e are given in Table 9.3. Here two randomly generated sequences of equal lengths were used in each case. The actual algorithm implemented was akin to that given in Figure 9.21. Here we wish to produce the matrix in full, but do not wish to store the entirety as this would create quadratic storage retirements.

```
macro proc TESTVSLMCS (n,streamin,xs)
{
    VectorOfStreams (Item(Result),vectora,n);
    par
    {
        VSLMCS (n,streamin,vectora,xs);
        VMAP (n,vectora,SINK);
    }
}
```

Figure 9.21: Test harness for the VSLMCS process.

As always we make the assumption here that the length of time each cycle takes does not change significantly with respect to the problem size. Here we have derived a linear (both in time and resource usage) hardware implementation based on a quadratic specification. The cycle times are very regular as we have seen in previous case studies - we have precisely $7n + 1$ cycles for a problem size of n. The slice usage is something in the region of $75n$, which would suggest we can deal with up to about 250 items on an XCV2000e.

items	*slices used*	(%)	*cycles*
5	281	1%	36
10	612	3%	71
15	973	5%	106
20	1,332	6%	141
25	1,713	8%	176
30	2,078	10%	211
35	2,593	13%	246
40	2,885	15%	281
45	3,339	17%	316
50	3,704	19%	351
60	4,393	22%	421
70	5,269	27%	491
80	5,901	30%	561
90	6,768	35%	631
100	7,463	38%	701

Table 9.3: Results for the DNA processing network.

9.6 Summary

In this chapter we hope to have demonstrated how the methodology presented in this work facilitates the development of elegant, provably correct and efficient hardware implementations from functional specifications. Not only have we demonstrated this for some 'academic' style problems such as sorting, but also for the development of implementations that are genuinely usable in the real world.

Chapter 10

Discussion

10.1 Future Work

10.1.1 Automation

The methodology could benefit from a tool to aid in the refinement process. Such a tool would not only make the application of the methodology easier for the user, but could also serve to reduce human errors which may inevitably occur when deriving an implementation by hand. Initial thoughts about how such a tool could manifest itself might lead one to suggest a straightforward compiler, which would input a specification in Haskell, and output an implementation in Handel-C. However, deeper consideration of the required functionality, given the nature of the methodology, is likely to instead lead to considering a tool with a more interactive workflow. The main reason for this requirement of interactivity is that the refinement process is not a purely mechanical one - at many stages the user is required to make decisions about the manner in which to proceed. The most obvious area in which user interaction is required is in the data refinement phase. Choices between stream and vector refinements will require knowledge from the user about the resource availability and efficiency requirements of the implementation. Furthermore, such decisions will be necessary for each component in the system - so in effect a number of these decisions will be required during the refinement process.

This leads us to consider a potential tool which assists and guides the user through the refinement process, rather than attempting to completely mechanise it in its entirety. Ideally, such a tool should provide scope for transformation as well as refinement. The user should be able to start with an intuitive specification, which may not initially be in a form best suited to refinement. A guided transformation environment should allow the user to manipulate this specification, such that it is better suited for refinement. For this we would require a term rewriting engine, supported by a database of transformation laws. The database should contain all of the standard functional transformation laws, such as those introduced by BMF [19], and the user should also be able to add

their own. Engines capable of this sort of transformation task already exist - one good example is MAG [60]. In short, this transformation phase should effectively allow the sorts of transformations presented in the case studies to be performed in a semi-automated fashion.

An ideal starting point for the refinement process is where the function can be expressed as a composition of components, wherein at least some of these components can be expressed in terms of higher order functions. Given this kind of compositional specification, the refinement process could then proceed step-by-step, refining each component in turn. For each component the user should be presented with choices for data refinement, bearing in mind the input and output types of that compositional phase. As usual, for single dimensional data structures (lists) the user should choose between streams or vectors. Additionally partitioning may also be presented as an option. For two dimensional structures (lists of lists) the standard set of alternatives should be offered - streams of streams, streams of vectors, vectors of streams and vectors of vectors.

After the data refinement phase, the tool is then required to perform process refinement. Higher order functions in the specification can be refined using processes from the library presented in this work. The user may also wish to define additional processes of their own, and associate them with higher order functions. Functionality in the specification which cannot be expressed using higher order functions will of course have to be tackled separately. Often such functionality is fairly trivial to refine - simple arithmetic expressions can, of course, be translated from Haskell to Handel-C almost verbatim, and any changes may be purely syntactic. A fairly simple engine can also be constructed to cope with such things as recursion unrolling - taking recursive definitions in the specification and attempting to express them iteratively. Occasionally, of course, it may be necessary as a last resort to leave some of the refinement effort to the user - perhaps by supplying some annotated template code for the user to 'fill in the blanks'.

10.1.2 Other Target Languages

In this work we have chosen to use Handel-C as the target implementation language. Handel-C presents many benefits to us in this setting, given that it facilitates direct implementation on an FPGA, and also as it uses the communication and parallelism models of CSP. However, this methodology is not necessarily limited to working with Handel-C, and one potential area of future work could be in looking at targeting other hardware description languages and environments.

CSP has been chosen as an intermediate stage in the refinement process, because it has a strong formal basis and provides a very good framework in which to reason about processes and behaviour. As already noted, Handel-C implements the communication and parallelism models of CSP, and as such makes an ideal target language. However, any other language in which we can also implement similar communication and parallelism operators may also prove a good candidate. In fact, for the majority of refinements looked at in this work we require only a small subset of CSP's many constructs and operators. Specifically, the input and output operators (? and !), the parallelism

operator (||) and the choice operator (|). Any language/environment in which we can implement these operators with the same semantics becomes a potential target for this methodology.

In general terms, one obvious potential alternative target language is occam [55, 56], as this is widely known for its strong correspondence with CSP. This is, of course, focused more on parallel processing rather than targeting reconfigurable devices (accepting of course Handel-Occam [76], a predecessor of Handel-C). However, much of the work in this thesis remains useful and relevant in the wider field of parallel processing.

Returning to reconfigurable computing, we may also wish to consider the possibility of using some of the frameworks discussed in section 1.5 as potential target environments. Pebble, Lava and SAFL all facilitate, either directly or indirectly, the generation of circuit designs suitable for implementation on FPGA devices. However, despite the high level nature of these languages, they are still semantically closer to conventional hardware description languages such as VHDL than they are to Handel-C. As such more effort will be required to move between CSP and these frameworks than is required to move from CSP to Handel-C.

Furthermore, in the field of asynchronous circuit design, the Tangram [13, 14] and Balsa [11] systems may also be interesting to explore as target languages. Both of these take CSP style specifications as input, making use of parallelism and synchronous communication.

Finally, HardwareC [52] shares a number of similarities with Handel-C, and as such represents a good candidate for a potential alternative target. HardwareC supports parallelism and communication in a similar fashion to Handel-C.

10.1.3 Wider Application Area

Another obvious avenue for future work is to look at a broader application area, given that additional case studies may highlight the need to consider new patterns and constructs.

In the JPEG decoder case study given in this work we have touched upon the field of compression of multimedia data. This is a wide field and many other applications could prove interesting to investigate. The work done on JPEG could naturally be expanded to looking at MPEG and other audio/video compression standards. The ever increasing use of such technologies makes the requirement for efficient and reliable implementations more relevant than ever - particularly given the recent proliferation of video applications in handheld devices (such as mobile phones). These sorts of "gadgets" introduce requirements of low-cost, low-power operation wherein efficiency and ease of upgrade are also very important factors. The FPGA is extremely well placed to meet all of these demands. Moreover, such devices often work with unreliable communication mechanisms - typically wireless or mobile networks. Loss or corruption of incoming data is commonplace, meaning reliability is a key factor. The design methodology proposed in this work would help significantly to ensure that such systems continue to operate correctly, even when the input to the system is incorrect or unpredictable.

In [28], Damaj looked at applying the methodology presented in this work to a number of cryptographic algorithms. Encryption is becoming an ever more widely used technology, not only because of the requirement to maintain privacy in today's heavily connected world, but also because of the demands of content protection. Digital multimedia formats have made the lossless reproduction of content (particularly music and films) extremely easy, which is an obvious problem for the copyright owners. Encryption in the form of DRM (digital rights management) is the only real way to proceed if we wish to continue to benefit from digital media whilst ensuring that copyright is respected. Already we can see encryption going hand-in-hand with almost any device which deals in multimedia data - from personal computers, though MP3 players and mobile phones to digital televisions. This relationship between digital content and encryption is likely to become even stronger in the future. As such we again have the same demands for low-cost, efficient, reliable and upgradeable implementations of encryption algorithms. As before, the FPGA, coupled with this methodology, is ideally placed to meet these demands.

10.1.4 Control Applications

In this work we have focused mainly on algorithmics, and have not concerned ourselves greatly with the issues of external control and interfacing. These control issues may involve such phenomena as interrupts, time outs and triggers based on sensors. In general we have derived implementations which solve a specific problem, and the interaction with the 'outside world' has been quite simplistic. Given some input our implementations produce the corresponding output. In real-world applications, such as avionics and robotics [77, 79], such algorithms may form just one part of a larger system, and the interactions with external systems may be quite complex.

As such, one area of future work could be to look more closely at the control and interfacing aspects of these systems. As part of this it would be important to look at how the specification of a large system can be factorised into the functional aspects, which can be treated with the methodology as presented here, and the interfacing/control aspects, which would require a wider treatment such as timed CSP. Conveniently, as the methodology has a strong foundation in CSP, we are given very good scope for reasoning about events and interaction in an integrated framework. Tools such as FDR [30] may prove useful in proving the correctness of these aspects of a system.

10.1.5 Alternative Communication Strategies

In Section 4.7 we discussed some of the issues which may arise from the interpretation of lazy evaluation in the process environment. This led us to consider how, in certain situations, we may wish to consider alternative strategies for communication. The standard interpretation of the stream in this work puts the producer in charge; it is the responsibility of the producer to signal when the transmission has ended, such that the consumer knows to stop consuming. Whilst this model works well for all situations where the stream is demanded, it may occasionally occur that

we actually require the consumer to be in charge of the transmission. Particularly this may be the case where the algorithm relies on lazy evaluation. The consumer may only require a small subset of the values that a producer is going to output. Moreover, a producer may actually output an infinite quantity of values. Without some means for a consumer to end the transmission this sort of situation may result in producing processes not terminating properly - waiting indefinitely to output a value that the consumer will never be willing to receive. In certain situations, as illustrated in Section 4.7, this could even result in deadlock.

One possibility, as discussed in Section 4.7, would be to introduce an EOR signal to go with the EOT signal. This would give the consumer a means of indicating that it has stopped receiving from the corresponding stream. The producer would therefore be responsible to check for EOR each time it attempts to output a value, and stop producing if it is signalled.

For a completely faithful interpretation of lazy evaluation in the process environment, we would require the consumer to request each and every value in the stream from the producer in turn. As such we would have two messages for every value communicated in the stream - the first a request message from the consumer, and the second the actual value (or perhaps an EOT) from the producer in return. The producer should not do any work whatsoever until it receives a request for a value - only then should it perform any required calculation and send the value. Whilst this strategy would solve the termination problems discussed above, for general use it is unnecessary, and introduces an additional communication overhead which may be somewhat undesirable.

Yet more of these 'special purpose' communication strategies can be envisaged, and these could be explored further in future work. For example, an alternative interpretation of the stream wherein the producer does not signal EOT, but instead indicates the number of values it intends to communicate at the start.

10.2 Conclusion

In this work we have set out to present a framework which reduces the complexity in developing parallel hardware implementations from functional specifications. We shall now discuss and evaluate how well this objective has been achieved. A number of objectives and requirements for the methodology were set out in the introduction to this work, and we shall examine here how these have been addressed.

In general, we began by stating that the methodology should facilitate the refinement from a specification in Haskell, a high level functional language to Handel-C, which can then be compiled directly onto an FPGA. This objective will be discussed in part in our analysis of each of the following requirements.

We required that the methodology should be able to expose parallelism through the refinement process and provide alternatives for the type of parallelism used. This is achieved through the data

refinement phase of the methodology. Here we presented two basic refinement strategies for list types in the specification. The stream corresponds to sequential communication, and the vector to parallel communication. We also examined combinations thereof - streams of streams, streams of vectors and so on. This, coupled with the option to partition lists in many cases, gives us a wide scope for implementation alternatives with different performance characteristics.

These data refinement alternatives can be illustrated when we consider the refinement of polynomial time algorithms, as demonstrated by the case studies presented here. Typically, quadratic time algorithms which employ two dimensional data structures (for example, lists of lists) can be refined in a number of different ways depending on our requirements. At one extreme, we have a vector of vectors which can communicate a quadratic sized structure in constant time, but requiring quadratic processing resources. At the other extreme, a stream of streams will communicate a quadratic structure in quadratic time but requiring constant resources. In the middle we have the vector of streams and the stream of vectors which both implement linear time and linear resource usage schemes. The data refinement aspect of the framework allows us to consider all of these alternatives, tailoring an implementation to suit our needs, whilst all the time ensuring correctness of the implementation.

In this work we have presented a substantial library of powerful components for hardware design, which are based on higher order functions and combinatorial list processing functions. We have looked extensively at how these can be interpreted and refined into different settings - streams, vectors and combinations thereof. We have provided proofs that the CSP definitions for our components (and in turn, their Handel-C implementations) correctly refine the functionality of their original functional specifications. Given the expressiveness of these components, we can use them in the definition of a number of algorithms. Having a library such as this which has already been well defined and proven is of great advantage when developing implementations. The often long compilation times inherent in development tools for hardware design can make mistakes very costly. The benefits of using a library in this way manifest themselves not only in easing the development effort itself, but also in proving the correctness of the overall implementation. As the library components have already been proven to be correct, for a new implementation we need only prove the correctness of the 'bespoke' items of functionality. As has been demonstrated in the case studies - particularly the combinatorial algorithms - often this bespoke functionality is quite trivial.

Some of the patterns of computation represented by these components will already be familiar, and have been studied extensively elsewhere - take, for example, the vector interpretation of *map*, or the 'funnel' network which refines $fold$ with an associative operator. Importantly though here we have given these a formal treatment, and provided a proof that such interpretations do correctly refine their specifications. Other components examined here are perhaps less well known, and their usefulness to parallel processing and hardware design has not before been identified as significantly

as it has in this work. Of particular note is the function $unfold$. This is a rich pattern which captures the functionality of many other functions, particularly list processing functions such as $tails$, cp (Cartesian product), $transpose$ and so on. It is useful in terms of parallelism because of the way in which it 'opens up' a structure. An interpretation of this function with the output refined as a vector provides scope for the parts to be processed independently in parallel. We have seen this pattern used in a number of different forms in the case studies, often used in tandem with a vector interpretation of map to process intermediate results, and a vector $fold$ to collate them. This combination of components works particularly well for algorithms which have intermediate structures of quadratic size. Such quadratic algorithms can be implemented with linear efficiency and resource usage in this way, and this has been demonstrated with several examples in the case studies chapter.

Whilst efficiency of the implementations is of course important, a stronger emphasis in this work is attributed to the issue of correctness. Between our Haskell implementations and our CSP definitions, correctness is assured based on our refinement laws, both in terms of process and data refinement. The data refinement step comes first, and this can be performed whilst still in the world of functions - we can represent our concepts of vectors and streams as functional types. It is by use of abstraction functions that we initially assert the correctness of our data refinements. It is through Prd, a function which returns a process, that we translate from the functional types of vectors and streams to the concrete communication behaviour they correspond to in the world of processes. Here Prd is interpreted differently in the vector setting to how it is in the stream setting. This special function Prd, in conjunction with our feed operator, also forms the basis to our process refinement rule. With just these simple refinement laws, and the wealth of transformational laws offered to us by functional programming, along with the process algebra of CSP, we have a combined framework which facilitates the provably correct refinement from functions to processes.

Scalability has been a keyword throughout. This is important in a number of aspects. In terms of the derived implementations, we strive for linear performance in both execution time and resource usage, and the case studies show a number of examples where this is achieved from a starting point of quadratic time algorithms. A basis of message passing has ensured our implementations are truly scalable to any size of problem. In this way we can continue to benefit from increased availability of larger reconfigurable devices without added complexity to the engineer, and with the reassurance that we continue to maintain correctness.

Bibliography

[1] A. E. Abdallah. Derivation of parallel algorithms from functional specifications to csp processes. In Bernhard Möller, editor, *Mathematics of Program Construction*, number 947 in LNCS, pages 67–96. Springer Verlag, 1995.

[2] A. E. Abdallah. Synthesis of massively pipelined algorithms for list manipulation. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proceedings of the European Conference on Parallel Processing, EuroPar'96*, number 1024 in LNCS, pages 911–920. Springer Verlag, 1996.

[3] A. E. Abdallah. Functional process modelling. In K Hammond and G. Michealson, editors, *Research Directions in Parallel Functional Programming*, pages 339–360. Springer Verlag, October 1999.

[4] A. E. Abdallah and John Hawkins. Calculational design of special purpose parallel algorithms. In *Proceedings of the 7th IEEE International Conference on Electronics Circuits and Systems, Jounieh (ICECS'2K)*, pages 261–267. IEEE Computer Society Press, December 2000.

[5] A. E. Abdallah and John Hawkins. Formal behavioural synthesis of handel-c parallel hardware implementations from functional specifications. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS-36 2003)*, page 278, 2003.

[6] A. E. Abdallah, G. Simiakakis, and T. Theoharis. Formal Development of a Reconfigurable Ttool for Parallel dna Matching. In *Proceedings of 7th IEEE International Conference on Electronics, Circuits and Systems (IEEE/ICECS)*, pages 268–272. IEEE Computer Society Press, December 2000.

[7] A.E. Abdallah. Filter Promotion Transformation Strategies for Deriving Efficient Programs from z Specifications. In *Proceedings of the 3rd IEEE International Conference on Formal Engineering Methods (IEEE/ICFEM)*, pages 157–167. IEEE Computer Society Press, September 2000.

[8] Ali E. Abdallah and Mark Green. An integrated csp-based tool for the visualisation, animation and performance evaluation of message passing algorithms. In *ICFEM*, pages 189–, 2000.

[9] David Agnew, Luc J. M. Claesen, and Raul Camposano, editors. *Computer Hardware Description Languages and their Applications, Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications - CHDL '93, sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC, Ottawa, Ontario, Canada, 26-28 April, 1993*, volume A-32 of *IFIP Transactions*. North-Holland, 1993.

[10] J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Comm. ACM*, 21(8), 1978.

[11] A. Bardsley and D. A. Edwards. The balsa asynchronous circuit synthesis system. In *Forum on Design Languages (2000)*. European Electronic Chips and Systems design Initiative (ECSI), 2000.

[12] Alexandra Barros. *Provably correct refinement of Z specifications into functional prototypes*. PhD thesis, University of Reading, 1998.

[13] K. Van Berkel. *Handshake circuits: An Intermediary Between Communicating Processes and VLSI*. PhD thesis, Eindhoven University of Technology, 1992.

[14] K. Van Berkel. Handshake circuits: an asynchronous architecture for VLSI programming. *International Series on Parallel Computation.*, 5, 1993.

[15] R. Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall, 1997.

[16] R. S. Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discreet Design*, pages 3–42. Springer, 1987.

[17] R. S. Bird. *Constructive functional programming*, pages 13–70. Springer-Verlag, 1988.

[18] R. S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 1998.

[19] R. S. Bird and L.G. Meertens. *Two Exercices Found in a Book on Algorithmics*. North Holland, 1986.

[20] R. S. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

[21] Richard S. Bird, Jeremy Gibbons, and Geraint Jones. Formal derivation of a pattern matching algorithm. *Sci. Comput. Program*, 12(2):93–104, 1989.

[22] Richard S. Bird, Carroll Morgan, and Jim Woodcock, editors. *Mathematics of Program Construction, Second International Conference, Oxford, U.K., June 29 - July 3, 1992, Proceedings*, volume 669 of *Lecture Notes in Computer Science*. Springer, 1993.

[23] P. Bjesse. Automatic verification of combinational and pipelined fft circuits. *Computer Aided Verification*, July 1999.

[24] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava - hardware design in haskell. In *International Conference on Functional Programming*, ACM SigPlan, page 278, Sept 1998.

[25] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[26] S. D. Brookes, C. A. R. Hoare, A. W., and Roscoe. A theory for communicating sequential processes. *J ACM*, 31(7), 1984.

[27] I. Damaj, J. Hawkins, and A. Abdallah. Mapping high level algorithms onto massively parallel reconfigurable hardware. In *IEEE/ACS International Conference on Computer Systems and Applications, Tunisia*, pages 14 – 22, July 2003.

[28] Issam W. Damaj. *Synthesis of Parallel Algorithms for Field Programmable Gate Arrays with Applications from Cryptography.* PhD thesis, London South Bank University, 2004.

[29] E. W. Dijkstra. *A Discipline of Programming*, volume 1 of *LNCS*. Springer Verlag, 1976.

[30] Failures–divergence refinement – fdr2 user manual. Formal Systems (Europe) Ltd., http://www.formal.demon.co.uk/FDR2.html.

[31] Jeremy Gibbons. Calculating functional programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction International Summer School and Workshop*. Springer, 2002.

[32] Mike Gordon. The semantic challenge of verilog hdl. In *Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 136–145. IEEE Computer Society Press, June 1995.

[33] Shaori Guo and Wayne Luk. Compiling ruby into FPGAs. In Moore and Luk [67], pages 188–197.

[34] The handel-c language. Celoxica, http://www.celoxica.com.

[35] Mohamed Khalil Hani, Zulkalnain Mohd Yusof, and Koay Kah Hoe. A JPEG decompression ASIC module designed using a VHDL module generator. *Jurnal Teknologi*, 33(D):19 – 34, 2000.

[36] Reiner W. Hartenstein and Manfred Glesner, editors. *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers, 6th International Workshop on Field-Programmable Logic, FPL '96, Darmstadt, Germany, September 23-25, 1996, Proceedings*, volume 1142 of *Lecture Notes in Computer Science*. Springer, 1996.

[37] The haskell 98 report, 1998. http://www.haskell.org.

[38] John Hawkins and A. E. Abdallah. An overview of systematic development of parallel systems for reconfigurable hardware. In Burkhard Monien and Rainer Feldmann, editors, *Proceedings of Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference Paderborn, Germany*, number 2400 in LNCS, pages 615–619. Springer Verlag, August 2002.

[39] John Hawkins and Ali E. Abdallah. A generic functional genetic algorithm. In *2001 ACS / IEEE International Conference on Computer Systems and Applications (AICCSA 2001), 26-29 June 2001, Beirut, Lebanon*, pages 11–17. IEEE Computer Society, 2001.

[40] John Hawkins and Ali E. Abdallah. Derivation of scalable message-passing algorithms using parallel combinatorial list generator functions. In *WOTUG Commmunicating Process Architectures, Oxford (CPA 2004)*, September 2004.

[41] John Hawkins and Ali E. Abdallah. Hardware synthesis of a parallel JPEG decoder from its functional specification. In *Proceeding of IFIP Working Conference on Distributed and Parallel Embedded Systems, Toulouse, France (DIPES 2004)*, pages 197–206, August 2004.

[42] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[43] C. A. R. Hoare, Ian J. Hayes, Jifeng He, Carroll Morgan, A. W. Roscoe, Jeff W. Sanders, I. H. Sorensen, J. Michael Spivey, and Bernard Sufrin. Laws of programming. *Communication. ACM*, 30(8):672–686, 1987.

[44] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communication. ACM*, 20(5), 1977.

[45] The jazz synthesis system. Improvsys Inc, http://www.improvsys.com.

[46] He Jifeng. *From CSP to hybrid systems*, pages 171–189. International Series in Computer Science. Prentice Hall, 1994.

[47] G. Jones and M. Sheeran. *Circuit design in Ruby*, pages 13–70. North-Holland, 1990.

[48] G. Jones and M. Sheeran. Relations and refinement in circuit design. In *Third Refinement Workshop*. Springer Verlag, 1991.

[49] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in ruby. In Bird et al. [22], pages 208–232.

[50] Mark B. Josephs. Receptive process theory. *Acta Inf.*, 29(1):17–31, 1992.

[51] D. E. Knuth, J.H.Morris (Jr), and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

[52] D. Ku and G. De Micheli. Hardwarec: a language for hardware design (version 2.0). Technical Report CSL-TR-90-419, Stanford University, 1990.

[53] CIP language group. *The Munich project CIP*, volume 1 of *LNCS*. Springer-Verlag, 1984.

[54] R. Lazic, T. Newcomb, , and B. Roscoe. On model checking data-independent systems with arrays with whole-array operations. In *Twenty five Years of Communicating Sequential Processes*, volume 3525 of *LNCS*. Springer, 2005.

[55] INMOS Ltd. *The Occam Programming Manual*. Prentice-Hall, 1984.

[56] Inmos Ltd. *Occam 2 Reference Manual*. Prentice Hall, 1998.

[57] W. Luk, V. Lok, and I. Page. Hardware acceleration of divide-and-conquer paradigms: a case study. In D.A. Buell and K.L. Pocek, editors, *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pages 192–201. IEEE Computer Society Press, 1993.

[58] W. Luk and S. McKeever. Pebble: a language for parametrised and reconfigurable hardware design. In *Field-Programmable Logic and Applications*, LNCS 1482, pages 9–18. Springer, 1998.

[59] Wayne Luk, S. R. Guo, Nabeel Shirazi, and N. Zhuang. A framework for developing parameterised fpga libraries. In Hartenstein and Glesner [36], pages 24–33.

[60] MAG - a tranformation tool for Haskell. Oxford University Computing Laboratory, http://web.comlab.ox.ac.uk/oucl/research/pdt/progtools/mag/.

[61] Manju Manjunathaiah and Graham M. Megson. Compositional technique for synthesising multi-phase regular arrays. In *ASAP*, pages 7–16, 2002.

[62] Manju Manjunathaiah and Graham M. Megson. Tools for regularizing array designs. *Parallel Algorithms Appl*, 19(1):51–75, 2004.

[63] Tiziana Margaria and Thomas F. Melham, editors. *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001, Livingston, Scotland, UK, September 4-7, 2001, Proceedings*, volume 2144 of *Lecture Notes in Computer Science*. Springer, 2001.

[64] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specifcation in hawk. In *Proceedings of the IEEE International Conference on Computer Languages*, 1998.

[65] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Defnition of Standard ML (Revised)*. MIT Press, 1997.

[66] Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors. *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9-15, 2000, Proceedings*, volume 1853 of *Lecture Notes in Computer Science*. Springer, 2000.

[67] Will Moore and Wayne Luk, editors. *Field-Programmable Logic and Applications, 5th International Workshop, FPL '95, Oxford, UK, August 29 - September 1, 1995, Proceedings*, volume 975 of *Lecture Notes in Computer Science*. Springer, 1995.

[68] C. Morgan, K. Robinson, and P. Gardinar. On the refinement calculus. Technical Report PRG-70, Oxford University, Programming Research Group, 1988.

[69] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[70] Carroll Morgan and Paul H. B. Gardiner. Data refinement by calculation. *Acta Inf*, 27(6):481–503, 1989.

[71] Alan Mycroft and Richard Sharp. A statically allocated parallel functional language. In Montanari et al. [66], pages 37–48.

[72] Zainalabdein Navabi. *VHDL: Analysis and Modeling of Digital Systems*. McGraw-Hill, 1993.

[73] V. Niculescu. Unbounded and bounded parallelism in BMF. case study: Rank sorting. *Studia Universitatis Babes-Bolyai, Informatica*, XLIX(1):91–98, 2004.

[74] T. S. Novell. SMALL: A programming language for state machine design. In *Proceedings of the 1997 Canadian Conference on Electrical and Computer Engineering (CCECE/CCGEI) in St. John's, Newfoundland.*, May 1997.

[75] John W. O'Leary, Mark H. Linderman, Miriam Leeser, and Mark Aagaard. Hml: A hardware description language based on standard ml. In Agnew et al. [9], pages 327–334.

[76] I. Page and W. Luk. *Compiling occam into Field-Programmable Gate Arrays*, pages 271–283. Abingdon EE and CS Books, 1991.

[77] J. Peleska. Test automation for safety-critical systems: Industrial application and future developments. In *FME '96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *LNCS*, pages 39–59. Springer-Verlag, 1996.

[78] Jan Peleska. Design and verification of fault tolerant systems with csp. *Distributed Computing*, 5(1):95–106, 1991.

[79] Jan Peleska. Formal methods for test automation - hard real-time testing of controllers for the airbus aircraft family. In *Proc. of the Sixth Biennial World Conference on Integrated Design and Process Technology (IDPT2002)*. Society for Design and Process Science, 2002.

[80] Jan Peleska. Applied formal methods - from csp to executable hybrid specifications. In *Communicating Sequential Processes, The First 25 Years*, volume 3525 of *LNCS*, pages 293–320. Springer, 2005.

[81] R. Sharp. *Higher-Level Hardware Synthesis*. PhD thesis, University of Cambridge, 2000-2002.

[82] R. Sharp. *Higher Level Hardware Synthesis*, volume 2693. Springer-Verlag, 2004.

[83] Richard Sharp and Alan Mycroft. A higher-level language for hardware synthesis. In Margaria and Melham [63], pages 228–243.

[84] Mary Sheeran. Describing butterfly networks in ruby. In *Functional Programming*, pages 182–205, 1989.

[85] Ying Shen and Theodore S. Norvell. Translating SMALL programs into FPGA configurations. In *Newfoundland Electrical and Computer Engineering Conference*, October 1999.

[86] M. Sheran. *μFP, an algebraic VLSI design language*. PhD thesis, Programming Research Group, Oxford University, 1983.

[87] Satnam Singh. Designing reconfigurable systems in lava. In *VLSI Design*, pages 299–306, 2004.

[88] Standard VHDL reference manual, 1993. IEEE Standard 1076-1993.

[89] Verilog HDL language reference manual, October 1995. IEEE Draft Standard 1364.

[90] R.A Wagner and M.J. Fisher. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[91] L Yongjian and J He. Towards a theory of bisimulation for a fragment of verilog. In *IPDPS*, 2003.