Technical Report

UCAM-CL-TR-824
ISSN 1476-2986

Number 824

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Hardware synthesis from a stream-processing functional language

Simon Frankau

November 2012

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted July 2004 by the author for the degree of Doctor of Philosophy to the University of Cambridge, St. John's College.

ISSN 1476-2986

Abstract

As hardware designs grow exponentially larger, there is an increasing challenge to use transistor budgets effectively. Without higher-level synthesis tools, so much effort may be spent on low-level details that it becomes impractical to efficiently design circuits of the size that can be fabricated. This possibility of a *design gap* has been documented for some time now.

One solution is the use of domain-specific languages. This thesis covers the use of *software-like* languages to describe algorithms that are to be implemented in hardware. Hardware engineers can use the tools to improve their productivity and effectiveness in this particular domain. Software engineers can also use this approach to benefit from the parallelism available in modern hardware (such as reconfigurable systems and FPGAs), while retaining the convenience of a software description.

In this thesis a statically-allocated pure functional language, SASL, is introduced. Static allocation makes the language suited to implementation in fixed hardware resources. The I/O model is based on streams (linear lazy lists), and implicit parallelism is used in order to maintain a software-like approach. The thesis contributes constraints which allow the language to be statically-allocated, and synthesis techniques for SASL targeting both basic CSP and a graph-based target that may be compiled to a register-transfer level (RTL) description.

Further chapters examine the optimisation of the language, including the use of lenient evaluation to increase parallelism, the introduction of closures and general lazy evaluation, and the use of non-determinism in the language. The extensions are examined in terms of the restrictions required to ensure static allocation, and the techniques required to synthesise them.

Acknowledgements

I would like to thank my supervisors, Simon Moore and Alan Mycroft, without whose advice and insight this thesis would not have been written. I also gratefully acknowledge Altera for the studentship they generously provided.

Thanks to all my fellow students, for making the Computer Laboratory such an enjoyable and interesting place to (attempt to) work. Also, thanks to my family and house-mates, who have been highly supportive.

This thesis is dedicated to the memory of my mother, Patricia.

Contents

List of Figures

CHAPTER 1

Introduction and Related Work

Higher-level hardware synthesis tools are becoming increasingly necessary. The drive of exponential growth in design complexity requires that hardware designers improve their productivity similarly if they are to make efficient use of the available transistors. Increasing the level of abstraction through higher-level languages is one step towards this goal. The question is not so much whether higher-level languages will be needed, as to what they should be like.

Useful parallels may be drawn with software languages. Low-level languages may remain useful for some tasks, but an increasing workload may be taken by *domain-specific* languages, roughly equivalent to software's scripting languages. The domain we have chosen to investigate is the hardware implementation of software-like programs. In this domain, exact signal timing requirements are unimportant, although high throughput is desired.

The aim of this thesis is *not* to provide a concrete language for real-world synthesis, but to explore language features that could be used to increase abstraction. I decided to base my work upon SAFL [128], a functional language used for behavioural synthesis. SAFL is a simple language which explores the use of software-like descriptions for hardware. However, its weakness is that its only I/O model is call/return based, with no pipelining and no state held between calls. The work of my thesis is an extension of SAFL, improving the I/O model and extending the language with common functional features:

The thesis of this work is that statically-allocated pure functional languages, extended to use streams (linear lazy lists), are suitable languages for behavioural hardware synthesis of reactive systems. Furthermore, higher-level functional features such as closures and lazy evaluation may be usefully incorporated in a statically-allocated form to produce an optimising synthesis tool with a high level of abstraction.

This thesis introduces a Statically-Allocated Stream Language, SASL, in order to explore static allocation requirements, synthesis techniques and evaluation models. Further language features are examined, and optimisation techniques discussed. Static allocation is the main feature distinguishing SASL from software languages, this being a requirement for producing hardware from a SASL description without the need for external memory, which may introduce von Neumann bottlenecks. Static allocation strongly shapes the way a number of features are incorporated into the language.

The work presented in this thesis could be used directly in a functional synthesis system, or as an internal model in an imperative system, allowing functional-style program transformations to be applied. This functional approach to I/O may provide a useful means to formalise higher-level hardware synthesis.

Section 1.1 provides a background to hardware description languages (HDLs), and motivates the use of software language features in the design of higher-level HDLs. Section 1.2 sketches the HDL language space, discussing some of the more popular and interesting languages. Section 1.3 then introduces SASL itself. As a background, it refers to related work on functional languages and static allocation. Finally, Section 1.4 provides a list of contributions and an overview of the rest of this thesis.

1.1 The Need for High-Level HDLs

Moore's Law famously states that the number of transistors that can be placed on a die doubles every 18 months.[1] The "law", originally an off-hand prediction, has been remarkably accurate, perhaps becoming self-fulfilling, as integrated circuit design road-maps come to depend on it.

While this growth is often seen from the point of computer users and programmers, providing faster computers and allowing increasingly feature-rich software, the effect is probably most strongly felt by hardware designers. These are the people who must translate the growing transistor budgets into higher performance and new features. Furthermore, the growth creates a rapidly changing market where time-to-market is key, and delay is a disaster.

The effort of designers alone cannot keep up with this exponential growth in project size; increasingly powerful design tools are needed. As growth in productivity from the tools lags behind the growth in available die space, a *design gap* opens up, and new approaches are required.

Modern System on a Chip (SoC) designs can require a complex design-chain, taking large pre-designed IP cores (much like software libraries) from a number of sources, and integrating them with custom circuitry. Powerful embedded processors are common, and the development of the associated software must be integrated with the hardware design, giving the challenges of hardware/software co-design. HDLs may focus on a particular level of the design, or work over a number of levels, but they are a vital part of the modern design process. To close the design gap it is necessary to increase the productivity of HDL users by introducing higher-level HDLs.

In the following sections we will cover the genesis of hardware description languages, and similarities with the development of software programming languages. We briefly discuss the current state of hardware design, and then look at reconfigurable computing. We assume the use of digital design throughout; analogue circuits are not in the scope of this thesis.

1.1.1 A Brief History of HDLs

HDLs arose from the need to manage complex designs, and have had a fairly direct descent from the draughtsman's schematics. Computers have been used in the design of ICs (such as Fairchild's Micromosaic) and PCBs (such as those used in Cambridge's Titan computer) since the late 1960s, creating a feedback loop in the complexity of designs that can be handled. As more of the design process switched to CAD, the importance of these HDLs has increased. Note that these input formats generally do not dictate layout—separate graphical tools are used to deal with the layout of PCBs and ICs, although a few languages, such as Sheeran's Ruby [59], deal explicitly with the physical relationship between the hardware elements.

Manual Circuit Design Pen-and-paper drawings were used to represent the connections of a circuit. A variety of shapes represent circuit components, and lines between components represent wires between terminals on the components. The construction and operation of the design is not affected by the location of the symbols, which are instead arranged for intelligibility [112]. The schematic is only interpreted by humans. Adjusting the design may involve a large amount of tedious redrawing, and if the design is to

[1]The paper [105] actually gives a yearly doubling, a result which held for some time, but 18 months has provided a better long-term fit.

be processed on a computer (for simulation or PCB layout, for example), the design must be entered separately, perhaps by manually entering a netlist.

Schematic Capture Schematic capture is the creation of a schematic design using an interactive graphical program. It brings with it the advantages normally conferred by moving a form of document editing to computer—editing data becomes much simpler, tasks automated, and more complicated documents can be handled. However, there is another major change, in that the data is now given semantics. Design errors can be flagged, simulations and analysis can be performed directly from the design, and the resulting circuit can be fed directly into a toolchain for PCB layout, for example.

However, the method of design is still fundamentally the same as before. There is little in the way of abstraction, as the design process hides the physical nature of the components and their connections, but little else. The design may be hierarchical, representing whole sub-circuits as single components, but parameterisable designs are unlikely to be supported. Moreover, the designer must keep track of many details of the timing and signalling protocols used by components. As the size of designs grow, these considerations become increasingly important. For larger *IP (Intellectual Property) Cores* standardised buses may be provided, so that the designer need only wire them together, but this shifts the responsibility for timing details onto the core designers.

Despite these shortcomings, schematic capture is still a popular design method for simple circuits, where the medium helps provide an intuitive understanding of the design. For larger projects, structural HDLs may be preferred for parameterisation and complexity management, while behavioural HDLs can help to abstract away the signalling details (both of which are described below).

Netlists A *netlist* is a textual description of the connections in a circuit—it lists the components that a circuit contains, and which pins are connected to which other pins. Annotations can supply other information, such as physical layout hints and static timing analysis results. They are effectively schematics with the human-usable position information removed. Netlists may be hierarchical or flattened. Hierarchical netlists define modules, which are sub-netlists abstracted to appear as a single component, so that instances may be created. By substituting module bodies in place of their instances, a flattened netlist is produced. As with other structural entry methods, the components of a netlist may be physical (for a PCB-based design), or virtual (e.g. logic gates in a design that will be placed and routed onto an FPGA). EDIF (Electronic Design Interchange Format) is a standard format used to transfer netlist-style information between applications.

Structural Design Netlist descriptions are not particularly readable, as they lose the visual structuring of schematics, but they pave the way for *Structural design* (also known as "Register Transfer Level" (RTL) design). Structural HDLs still express the connections between components, but do so in a higher-level way.

The improvements are based on the observation that most digital systems have unidirectional signals with a single source (there are, of course, exceptions such as tri-state buses, but these may be dealt with specially). As such, components can often be syntactically represented as functions, taking signals as inputs and returning signals as outputs. By composing functions, connections can be expressed without explicit wires. *Continuous assignment* is used to represent combinatorial hardware. For example, in Verilog, the following assigns the exclusive-or of a and b to c, calculated using **and**, **or** and **not** gates:

$$\textbf{assign}\ c = (a \,\&\quad b) \quad (\quad b \,\&\, a);$$

The description of sequential functions is simplified by representing latches and registers as assignment triggered by clock events.

The level of abstraction can be raised slightly in this way, as new language features are included. For example, memories may be represented using an array-like notation. Conditional expressions (such as "**if** . . . **then** . . . **else** . . ." and "**switch**") may be used, which simply become multiplexers selecting the source of values for wires. These additions may not seriously complicate a synthesis system, but vastly improve its usability.

As well as simplifying the description of complicated control signals, finite state machines and so on, describing circuits textually can make it much simpler to automatically manipulate those circuits, and include these transformations into the circuit "source". Circuit descriptions may be made parameterisable, so that an arithmetic unit may not only be reused, but its bit-width can be tailored to the current application. More generally, macros can be used to automate the construction of complex but regular circuits.

Verilog and VHDL are the most common structural languages, although they can also be used as netlist or behavioural languages. There are IEEE standards for the RTL synthesisable subsets of Verilog and VHDL (these are 1364.1 and 1076.6, respectively). Various other languages, such as Hydra [115, 116, 117] and Ruby [59], work exclusively at the structural level, often using parameterisation and macros extensively.

Behavioural Design Behavioural descriptions focus on what the design must do, rather than its structure. Synthesis tools then create some form of structural design to match the behavioural description. The behavioural design approach can be reached in a number of ways. For example:

Structural design languages are extended until synthesis requires more than just expanding out constructs.

Constructs originally not intended for synthesis (e.g. simulation-only processes, for creating test harnesses) start to be synthesised.

Explicit behavioural features are retrofitted to existing HDLs.

Tools are created to synthesise existing software languages.

New software-like languages are created with the intention that they be synthesised.

The first two approaches are taken by both Verilog and VHDL. These are flexible languages, and have grown into the area of behavioural synthesis while retaining compatibility with structural design, although it is quite possible to write behavioural descriptions that go beyond the synthesisable subsets, and which the available synthesis tools cannot convert into hardware (by introducing impossible timing constraints, for example). The third approach is taken by System Verilog.

The fourth approach is often taken with C, using tools such as CTOV [142]. This approach provides a path to hardware/software co-design, as both sides of the design can be specified in the same language. Again, there are generally limitations as to what the synthesis tools will be able to translate. The source language may be restricted in order to make it synthesisable, but there is the danger that what will be produced is a new language sharing only some syntax with the original.

The final approach is that taken by languages like Handel-C, Lustre and SAFL. The translation process itself may be quite simple, but generally any program that meets the language's requirements will be synthesisable (subject to resource constraints). Synthesis complexity is a trade-off: a simple translation process allows the user to understand the synthesis process, and create their designs with that in mind, but can limit both the level of abstraction and the amount of optimisation that may be performed on the design. Our language, SASL, is a behavioural language that tries to raise the level of abstraction, and attempts a relatively complex synthesis approach. It tries to move hardware design to a more software-like model, and does not expect the programmer to keep a detailed mental model of the translation process.

1.1.2 A Comparison to Software Languages

The users of HDLs have traditionally been a group quite distinct from software programmers, and the origins of the languages are similarly distinct. Whereas circuit design was the domain of electrical engineers, the early uses of computers were mathematical, working on problems such as solving differential equations and breaking cryptography. This distinction can still be seen, with situations such as separate CS and EE departments in universities. However, some of these differences are starting to be eroded as HDLs pick up software-like features, and the previously distinct groups merge.

Schematic entry and low-level structural languages can be thought of as the assembly language of hardware design. While almost diametrically opposite in approach, with assembly's sequential instructions contrasting with the parallelism of a netlist, they both represent the lowest levels of abstraction, with minimal levels of compilation and optimisation being applied.

The higher-level structural languages have some correspondence with lower-level software languages. High-level structural Verilog and VHDL are the workhorses of hardware design, much as C is used in software development. Just as C is "high-level assembly", an experienced user can have a good knowledge of how these HDLs will be synthesised to hardware, and can control low-level details using simple changes to the source. However, this same low-level approach can prevent the synthesis tools from providing optimisations, as well as forcing the designer into "early binding", dealing with details that may prevent effective exploration of the design space.

Behavioural languages have similarities to high-level software languages: both language styles move away from relatively explicit instructions on how to construct the resulting program or circuit, instead trying to describe a more abstract solution to the problem. Details are left for the compilation tool to deal with, and there is a large scope for optimisation.

As with the extension of C to C++, behavioural synthesis is being added to Verilog and VHDL, since backwards-compatibility is an important issue. At the same time, new and different languages are introduced which are explicitly high-level, making different assumptions. Domain specific hardware languages can act like scripting languages, reducing development time for specialised tasks. Throughout, hardware languages have lagged behind software-based ones. Verilog and VHDL were introduced in the 1980s, while C was a product of the early 1970s. Despite this lag, these languages do not seem to have taken advantage of the lessons learnt in language design, and are rather inelegant in places. At the same time, perhaps due to a comparatively small user base combined with the scaling to exponentially larger designs, the tools seem expensive and immature compared to software compilers. The lag makes it possible to apply software language knowledge to hardware problems to obtain novel solutions.

1.1.3 Modern Hardware Development

Increasing transistor counts allow ever greater integration, so that most designs consist of only a few chips. The design complexity is therefore in the design of the silicon, creating SoC (System on a Chip) circuits. Increasingly complex microprocessors may be embedded within a design, introducing elements of hardware/software co-design. However, embedded processors may not provide the required performance, in which case a direct hardware implementation is necessary. In such situations, it is useful to have a language that can describe algorithms in a software-like manner, but is designed with hardware synthesis in mind. This is the aim of SASL. The software-like structure should allow interfacing to software systems for code that is control-flow bound or less performance critical, although that topic is not covered in this thesis.

The threat of the design gap encourages the use of increasingly complex tools; the ever-growing resources available to hardware designers should make the use of high-level design tools acceptable, despite the overhead that is introduced, just as has occurred with software languages. Domain-specific tools and languages ease the path towards higher-levels of abstraction by concentrating on particular situations. As complexity increases further, optimising synthesis tools may provide better performance than designs that can be economically produced by a human, just as programmers today avoid using assembly language.

The cost of starting new designs is also increasing, with the price for set of masks now on the order of a million dollars. A large volume is needed to break even, so there may be a move towards *platform chips* which act as a standard design containing a set of IP modules suited to a particular task, such as a mobile phone handset. Some higher-level languages may be suitable to "glue" the IP components together.

A very different approach to fabricating chips in bulk is the use of FPGAs, which consist of programmable logic blocks and interconnect. By suitably programming the logic blocks and connections between them at power-on, any circuit may be produced. The advantages of FPGAs compared to ASICs are that for small volumes they are very inexpensive compared to custom silicon, and that they are in-circuit reprogrammable, providing a great amount of flexibility. The disadvantages are that they will have inferior performance, area and power consumption characteristics, and be more expensive at large volumes.

This reprogrammability makes FPGAs uniquely suited to a number of tasks. FPGA areas could be added to platform chips to allow the implementation of glue logic, add "missing features", or even correct bugs after shipping. On-the-fly reconfiguration, where suitably-programmed FPGAs may be used as co-processors for algorithmic tasks, is an active research area, and is discussed in the next section.

1.1.4 Runtime Reconfigurable Systems

Runtime reconfigurable systems [48] are based on architectures such as FPGAs, and use the parallel execution of hardware-style designs, in place of traditional processors, to perform algorithms. Such systems may not be implemented using the fine-grained logic blocks of FPGAs, but could use coarser-grained components such as ALUs, as with the Xputer [62], PACT XPP [12] and MIT Matrix [104]. Similar synthesis techniques should apply, which may even be applicable to tiled processor arrays, like MIT's RAW project [149].

Such systems can provide a very large amount of parallelism, and may give a large speed-up over conventional processors for algorithms which are not control-flow limited, if sufficient memory bandwidth is available. Poor suitability for programs with complex control-flow means such reconfigurable arrays are often suggested as co-processors for conventional systems. These co-processors are generally well-suited to applications such as streamed media processing [57], and have been successfully used to accelerate tasks such as performing Photoshop filters [135], DTP rendering [91] and video processing [63, 86]. Reconfigurability also provides an advantage over a plain hardware implementation in that the implementation can be specialised for a particular run, for example through constant propagation [155] and partial evaluation [134], although such an optimised FPGA will still be slower than a native silicon implementation.

General approaches to reconfigurable systems are discussed in various papers [71, 87, 25, 24]. The idea of an "operating system" for reconfigurable hardware has been discussed [29, 22], while Brebner [23] and Donlin [49] have looked at approaches for modifying the system at runtime. The reconfiguration of hardware while it is running has a strong parallel in the use of overlays in software systems, providing another example of where hardware systems can use software experience. The temporal granularity at which systems are reconfigured provides one form of classification. For example, the DPGA [21, 47, 141, 46] (intended to be a cross between SIMD and FPGA approaches) caches configurations, so that it can switch between FPGA-like configurations with an extremely low overhead. Carnegie Mellon University's PipeRench project [126, 34, 27, 33] relies on incrementally reconfiguring a physical pipeline to simulate a larger virtual FPGA. Pipelined reconfiguration has also been explored by Luk [88]. More conservative designs treat reconfiguration as an expensive overhead, like task switching, to be performed at a much larger granularity than the underlying processing steps.

Another major design decision for reconfigurable systems is how they are to be integrated with the processing elements. Tightly-bound reconfigurable systems place the FPGA-like element as part of the

CPU (or, in the case of DISC [154], replace more or less the entirety of the CPU). This approach is taken by the Garp [36] and PRISC [124], while HARP [120] looks at tightly-binding a DPGA to the CPU. Loosely-bound reconfigurable systems keep the reconfigurable elements separate from the main CPU, allowing them to be triggered by the CPU but run separately. Many prototype systems take this approach, for example placing the reconfigurable elements on a PCI bus, although this is unlikely to have sufficient bandwidth for all but a few computation-bound tasks. Some reconfigurable systems may implement small processors, such as the Nano processor [156] or Altera Nios [4] in the reconfigurable fabric, to deal with control-flow bound computation. The processors may be parameterised [118] to fit the task.

Reconfigurable systems rely on producing hardware that implements user-specified algorithms. A number of these projects have synthesis tools, including co-design systems (mostly based on extracting suitable operations from C programs), but these are often relatively simple and low-level. SASL is intended to be an appropriate language for programming these kinds of systems at a higher level.

The synthesis presented in this thesis is for a fine-grain architecture, although it should be possible to port to a coarse-grained architecture by introducing relevant primitives to the language, as long as the underlying architecture is able to support the necessary back-pressure and control signalling. The language has no concept of reconfiguration, but its higher level of abstraction may make it a suitable language to add this feature to. Similarly, SASL could form part of a high-level co-design system based on functional languages, since it is not a hardware-specific language, just as Handel/Occam has been used for a CSP-based co-design system [119].

Another approach that lends itself to reconfigurable computing is the addition of computational elements to memories. The overhead of moving data to and from the processor is becoming increasingly large, and a number of projects [97, 92, 83, 84] are investigating the embedding of processing elements in individual RAM chips in order to accelerate computation in a distributed manner. Such processing elements need to be simple and efficient at dataflow processing on streams of memory items. They should be able to be programmed in a software-like way. An appropriate implementation seems to be to use reconfigurable processing elements programmed with a language such as SASL.

1.2 The Hardware Description Language Space

Having discussed HDLs in general, this section will examine specific existing HDLs. As with software languages, the HDLs vary considerably in their approach. Before covering the languages, we will consider the design assumptions that lead to such variety.

1.2.1 Language Assumptions

A wide range of assumptions has lead to a spectrum of solutions. Design choices are made on factors including:

Interfacing Requirements At the simplest level, this may require that the produced circuit runs at a specific frequency to integrate with other circuitry. Interfacing protocols may be used, ranging from complex bus interfaces such as PCI down to simple two-phase signalling. Language assumptions can make certain types of I/O very difficult; languages that assume synchronous circuit design may need low-level glue to work with asynchronous I/O, while high-level synthesis systems may not easily interface with low-level, time-constrained systems. Languages that perform low-level interfacing will lose some abstraction at the signalling level.

Circuit Performance Circuit performance generally brings to mind the ability to synthesise a system to match a required clock frequency, effectively part of the interfacing requirements. However, there can be more; pipelined systems can vary in latency and throughput, and parallelism may allow a circuit to be unfolded or duplicated to increase performance. Languages may provide such high-level optimisations, while at the other end of the scale low-level features allow tuning for maximum performance.

Resource Constraints As well as meeting the requirements for interfacing and performance, designs have to stay within budget for area, power, and other factors. To support this, languages could specify constraints, and the synthesis systems could compile the project within given constraints. More common, however, are frameworks that expect the user to adjust their designs in order to meet the constraints. For such systems, transformations that allow the user to rapidly and simply explore the design space are useful.

Rapid Development Transformations in order to meet constraints are part of a more general set of language features suited to rapid development. Rapid development systems use abstraction to accelerate design, trading control of details (and associated low-level optimisations) for speed of design. As designs can be made to work more quickly, rapid development encourages design space exploration.

Target Domain Languages specialised to a particular domain can make developing certain applications much faster, and produce very efficient designs. There are systems to generate state machines, DSP systems, and so on, as well as parameterised block generators to create specific devices, such as memories. Some languages may trade off their special advantages against generality.

Target Architecture Although the high-level and low-level synthesis stages may be split, allowing placing and routing to be separated from other stages, the target architecture may strongly affect the assumptions that can be made. Asynchronous designs may be very difficult to map onto an FPGA, while runtime-reconfigurable systems would not work with fixed hardware.

Backwards Compatibility As with software languages, compatibility with existing languages, notably Verilog and VHDL, is important. These languages have a great deal of flexibility, but work at a low level, and sufficiently different assumptions may make it impossible to embed the new features in existing languages. This introduces the next factor:

Toolchain integration Even if a language strongly differs from existing HDLs, it will need to integrate with existing tools if it is to be used. New software languages generally need to be able to be linked with existing languages and libraries, but for hardware languages this is an even more important issue. There is heavy investment in the current tool-chains, and the variety and complexity of targets makes reimplementing entire tool-chains unreasonable. Vendor-controlled languages may be integrated directly into their toolchain, while for other high-level languages it is often simplest just to synthesise to Verilog or VHDL netlists.

The choices in how to approach each issue are inter-related, although languages can give some flexibility. For example, Handel-C provides high-level channels that simplify synchronisation for rapid development, but the user can eliminate these in favour of cycle-by-cycle semantics if they later wish to optimise for performance. One of the common themes is the level of abstraction. Although working at a high level of abstraction can make low-level interfacing and optimisation more difficult, it also allows for better design space exploration, enables higher-level optimisations, and hopefully gives the compiler more flexibility to perform low-level optimisations automatically.

SASL aims to give a high level of abstraction, to be used in application areas where low-level interfacing constraints are unimportant, such as reconfigurable systems designed for computation. In such systems automated wrappers can be generated to connect the computational circuitry to underlying memories or buses.

	Imperative	Functional
Structural	Low-level VHDL/Verilog, Netlist formats . . .	HML, Lava, μFP, Ruby, Hawk . . .
Behavioural	High-level VHDL/Verilog, Handel-C, SystemC . . .	**SAFL, SAFL+, Lustre, SASL**

Figure 1.1: A selection of HDLs

1.2.2 Example Languages

Figure 1.1 shows an example set of HDLs, divided up along structural/behavioural and imperative/functional lines. While the structural/behavioural divide is quite well-known, the division between functional and behavioural languages may seem less relevant. For structural languages, imperative languages generally represent backwards compatibility and mainstream industrial use, while the functional languages are more experimental, using higher-level features to increase the level of abstraction. We argue that the functional approach may have similar advantages in the behavioural domain for enabling the creation of higher-level languages.

The behavioural-functional languages are also the most unexplored language style, as well as being an area with some of the greatest flexibility, providing the highest levels of abstraction, and allowing various functional ideas to be brought to bear on hardware synthesis. Further motivation for this kind of language is given in Section 1.3.1. SAFL and SAFL+ are simple software-like functional languages, where programs are compiled into hardware that performs that task. The languages are closely related to SASL, and are discussed in Section 2.1. The rest of this section discusses various other HDLs:

Verilog and VHDL The two main industrial HDLs, Verilog and VHDL, are both IEEE standards, with rather different roots. Verilog was a simulation language, designed by Gateway Design Automation. It was patterned on C, and made into an open standard when Cadence bought Gateway. In contrast, VHDL (VHSIC Hardware Description Language) was supported by the US Department of Defense as part of its Very High Speed IC (VHSIC) program. It has some similarities to another DoD project, Ada. To an extent these HDLs are now unified, in that they are now both being developed by the Accellera Organisation.

The languages are now used almost universally in industry for design entry, simulation *and* synthesis. For synthesis, they can be used for both netlist-level description, and structural design. High- and low-level code may be mixed together, in a way that is reminiscent of the use of assembly language inserts in languages such as C. Behavioural features are available to create test-benches, but are also increasingly used for behavioural synthesis purposes. This over-generality means that it is quite possible to write code that cannot be synthesised, or may only be synthesised very inefficiently.

Despite behavioural features, the languages provide little abstraction from low-level timing and wires, instead allowing detailed control over interfacing and circuit performance. This comes at the expense of rapid development and design-space exploration, since design details must be bound early, and the manipulation of high-level features is impeded by the need to manage low-level details.

Ruby Ruby [59, 133] is a relational language—a structural language that not only describes the connections between modules, but also their relative placement. This approach has been seen in a number of systems; such as Pebble [100], and GAMA [35], which does not explicitly allow the user to control layout, but is nonetheless designed for the generation of regular datapaths. Simple relative placement control allows systolic arrays to be efficiently floor-planned, quickly synthesising efficient circuits. This is especially useful for FPGAs, where placement and routing is often very time-consuming (preventing software-like compile-debug-edit cycles), and results in rather suboptimal layouts.

The language's syntax is quite mathematical, being based on the notation of relations. As the language connects together primitive blocks, the interfacing and timing assumptions are simply those of the underlying primitives. For regular structures, the language can provide very compact and efficient designs, but it provides little support for other structures, making it an example of a domain-specific language.

Handel-C Handel-C [39] is a commercial product formed by giving the Handel language a C-like syntax. Handel, in turn, is a CSP-based language, similar to Occam [127]. CSP [67] (Communicating Sequential Processes) models parallelism with processes that communicate through synchronising channels, providing blocking communication. Commands in a process can be executed in sequential or parallel blocks.

In Handel-C the ";" statement separator denotes sequential operation, while new syntax is introduced for parallel operation (which will then run the processes in lock-step). Assignments take a single clock cycle, while combinatorial functions are simply synthesised to combinatorial circuits. This makes compilation relatively simple, although synchronisation across different processes at the end of a parallel fork may require the production of synchronisation circuitry.

Channel-based communication is used to provide higher-level synchronisation primitives, so that parallel processes can communicate without being coded in lock-step or requiring low-level synchronisation primitives. Various features of C are omitted, to keep synthesis simple. For example, function calls are macro-expanded, and pointers are disallowed. However, it creates a relatively simple synthesis system that is usable by software programmers with little training.

The language is effectively a structural HDL (with rather more syntactic sugar than Verilog or VHDL), extended with channels for rapid development (which can later be removed for increased performance). Optimisation is performed by the user, who has cycle-level control over timing.

Lustre Lustre takes a very different "synchronous dataflow" approach. It is based on Lucid [8], a stream-based language. Streams are sequences of scalar values. Lucid was intended for use as a formal system, and describes streams using the primitives "`first`" and "`next`". Stream items are defined in terms of items from other streams, or earlier items from the same stream. Loops are described using streams to represent the intermediate results of each iteration, from which particular values can be extracted into another stream using the "`as soon as`" primitive. The stream-based paradigm is also used in languages such as Esterel [16, 17] (which takes a more imperative approach) and Signal [6]. Synchronous dataflow is also the idea behind the DSP-specific language Silage [54].

Lustre [60] is a declarative-style language which brings these concepts to hardware synthesis. Variables represent streams (or flows, or signals) of values over discrete time. These streams can take values on different clocks derived from a basic clock, with language restrictions preventing unsynchronised streams from being combined. Streams may be defined in terms of combinatorial functions on elements taken with a fixed delay from the same and other streams.

The design of Lustre is specialised towards stream-like processing without control-flow (for example, implementing loops in the language can be painful). The language provides a simple synthesis path, expecting the user to perform optimisations, and the performance is largely based on the underlying primitives. Lustre assumes a simple synchronous I/O interface. Section 2.7.2 contains a comparison between SASL and signal-based languages.

Functional Lustre [37] is an extension to the language which adds some functional-like features. However, the approach is still very different to SASL's; it is still a synchronous dataflow language, except that as well as having streams of values, it is now possible to create streams of functions.

Miscellaneous Hydra [115, 116, 117], like Lustre, treats all variables as streams of values. However, it takes a structural approach, by treating circuits as compositions of functions on these streams. Higher-order functions act like macros on the functional blocks, automatically generating the structure of systolic arrays and similar circuits. Hawk [43] also takes this approach.

Hydra is a structural HDL embedded in a higher-order functional language. This approach is also taken by HML [85] and Lava [18] (which are based on Standard ML and Haskell, respectively). Lava builds on the work of Ruby and μFP [131] (another language where higher-order functions are used to describe and manipulate structural designs). By embedding the hardware language in a software language, multiple interpretations can be used, allowing the framework to simulate or synthesise the embedded design, for example. Features of the embedding languages can be used to construct complex circuits, but the approach is still fundamentally structural.

The CSP communication model of channels, as used by Handel-C, is also used in SAFL+ [130] and the C++-embedded HDL SystemC [139]. SystemC is aimed at system-level design, approaching it by embedding hardware description into a full software language. SystemVerilog takes the opposite approach, extending Verilog into a systems language. These languages are also beginning to bring function-call style communications to HDLs. SAFL+ extends SAFL with CSP channels, creating quite a different language. These languages use channels to allow communication between processes with back-pressure, so that later elements can delay earlier stages if they are not ready. SASL uses back-pressure on all data transfers, effectively treating all data paths as channels, but hides this from the user, and restricts the channels so that deadlock is no longer possible.

CSP is not only used to provide a communications model for HDLs, but can also make a useful intermediate target language itself. CSP programs are statically allocated and can exhibit parallelism. They may be synthesised to hardware relatively simply (this being effectively what the Handel-C compiler does). Chapter 3 covers the translation of SASL to CSP. Abdallah [1, 2] has done work on streaming data through functional programs, with synthesis to CSP, but he has taken a rather different approach, where the stream processing functions are built up from scalar functions using pre-defined higher-order operators.

Various other systems [150, 28, 10, 142] create hardware from a subset of C, mostly targeting reconfigurable hardware. These systems often either have traditional processors embedded in them, or restrict the use of C to make compilation more convenient.

Johnson has done work on using software compilation technology to create hardware [74], and more recently on digital design derivation [73], where the designer formally derives a circuit from a functional style specification. General high-level synthesis from software-like languages is described in De Micheli's book [102]: the programs are converted into a data-path plus control logic, and performance trade-offs are made through *binding* and *scheduling* operations. Binding selects which functional unit performs which operation, thus limiting parallelism, and static scheduling selects which operation is performed on which unit during which clock cycle. This approach concentrates on processing scalar data, in a mostly non-pipelined fashion, in contrast to SASL.

Researchers such as Weinhardt [151, 153, 152], and Marinescu and Rinard [94, 95, 96] have worked on the automatic generation of pipelines for imperative high-level synthesis systems, but the pipelining of functional behavioural systems has not been explored in the same manner. Although similar techniques may be applied, we believe that the referential transparency provided by pure functional languages may make the pipelining of such systems simpler.

1.3 The Statically-Allocated Stream Language

SASL is a behavioural HDL that tries to raise the bar on the level of abstraction that is available. The emphasis is on performing real high-level synthesis of constructs previously only used in software languages. To do this, it takes a pure functional approach, relying on static allocation to make the language synthesisable.

In the following section, the assumptions and motivation for the language are reviewed. Subsequent sections cover the functional and statically-allocated aspects of the language, as well as briefly discussing SASL's I/O model. A final section compares SASL to SAFL+ and Lustre.

1.3.1 SASL's Niche

Most of the behavioural HDLs described above are designed to facilitate a straightforward translation to structural form. Low-level signalling and timing issues are still exposed, which means module interfacing may be complicated, and design exploration slowed. Some of the languages provide features like CSP-style channels, but such abstractions must still work with the underlying low-level assumptions, and thus the optimisations that may be performed are limited.

SASL is a software-like language intended for compilation to hardware, rather than a hardware language made to include software-like features. It is domain-specific, suitable for problems which are complex enough that a simple systolic array is unsuitable, but are not control-flow intensive enough that a software solution is preferable. For example, many graphics operations, performing structured tasks with some limited control flow, may be suitably implemented in SASL.

The language attempts to hide all details of the hardware implementation. For example, there is no explicit parallelism in the language, instead relying on the compiler to extract the parallelism from the functional descriptions. The compiler is allowed, and indeed expected, to perform a large amount of transformation and optimisation.

In an industrial language, this "black box approach" would most likely be considered impractical, as designers would wish to have a clearer view of the synthesis process, and more access to low-level details and timing. However, SASL is intended to be an experimental language, and as such is aiming at as high a level of abstraction as possible. By removing the need for backwards compatibility, new features can be experimented with, and if they turn out to be useful, they may be "back-ported" to more conventional languages. In the longer term, increasingly high levels of abstraction without programmer-level access to the internals may become acceptable, as has occurred in the software domain.

SASL is modelled on functional languages (as discussed below), an approach often used for experimental and research languages. This provides a useful framework in which to reason about language features and transformations. Functional languages often have a higher level of abstraction than similar imperative languages. They also seem more suited to hardware implementation: imperative languages impose an order on instructions, and although code analysis may remove some of these dependencies, there is generally a reliance on global state which may restrict parallelism. In comparison, pure functional languages (those that do not allow side-effects) allow any two functions to run in parallel—the lack of side-effects simplifies the extraction of parallelism. The use of functions also allows the creation of pipelines through function composition.

Approaching SASL from a software point of view, it is a statically-allocated programming language. Relatively little work seems to have been done in this area, perhaps due to the view that statically-allocated languages are simple, and thus thought to be of little practical use. However, this dissertation will show that the static allocation of complex language features (that is, the creation of O(1) storage requirements) has some subtleties, and that various useful functions can be implemented statically. In many cases, it simply suffices to synthesise a system that is large enough to deal with the expected data; the resources required for a fixed problem size are often bounded.

SASL is designed for the core implementation of stream-processing algorithms, with pipelining and parallelism limited only by the dataflow of the given program. It allows the efficient description of algorithms that would otherwise be implemented in VHDL or Verilog, where scheduling, signalling and control logic would have to be explicitly constructed.

It should be clarified that the term "streams" has traditionally had different meanings in the functional programming and hardware communities. The meaning used here comes from a functional background, where it means a lazily-evaluated (demand-driven) sequence of items, whereas in hardware it means a synchronous producer-driven sequence. While the difference can be partially abstracted away at a programming level, the distinction is important when talking about, for example, real-time streams of video data.

SASL is a language primarily targeted at FPGA-based systems, or, more generally, systems based on reconfigurable hardware (although there are also situations where it may be appropriate to use it for custom silicon designs). Such reconfigurable systems are generally associated with languages intended for rapid, high-level development. In particular, SASL should be useful as a way of writing programs that would traditionally be executed on a general-purpose CPU, but can be accelerated by hardware implementation. By using a language like SASL, a software programmer should be able to develop a hardware implementation without requiring any detailed knowledge of the underlying hardware, or hardware design techniques.

1.3.2 Functional Languages

SASL is a functional language, for the reasons described above. Functional languages are descended from LISP [99, 136], although the underlying formalism, Church's lambda calculus [42], predates general purpose computers. Standard ML [103] and Haskell [81] are representative of modern functional languages; ML is an eager and impure functional language, while Haskell is lazy and pure. Both are strongly typed with polymorphic type systems. Compared to imperative languages, functional languages are convenient to reason about as they provide a form well-suited to transformation and analysis. Higher-order operators allow a high level of abstraction to be achieved.

In a pure functional language, computation is performed by calling functions, and functions are side-effect free, with the result depending purely on the arguments provided. Variables may be bound, but not updated. Recursion is used to perform looping operations. Pure functional languages make I/O difficult. In the simplest case, a function's parameters form its input, and the value returned its output. For more complex and interactive I/O, some form of I/O state-holding object may be passed around. Haskell uses *monads* [80] to deal with state and I/O, effectively allowing imperative-like sequences of commands to be embedded within Haskell.

Other languages, such as ML, allow side effects. Reference types allow values to be updated, and I/O may be performed without resort to structures such as monads. However, this comes at some cost. Referential transparency is lost, and with it goes many of the advantages over imperative languages:

> Pure functional languages are easier to reason about, as a function's result depends solely on its arguments (so that a function can be known to only have localised effects), variables are never updated, and recursion may be reasoned about inductively.
>
> Optimisation may be simplified, as all that matters is the result of the function; if a value is not used, the call can be eliminated, without worrying about side effects.[2]
>
> The order of evaluation for subexpressions does not matter (a useful property for hardware implementations, where we wish to perform parallel evaluation).

Functional languages are generally either eager or lazy. Eager evaluation is close to the imperative view—when a variable is bound, the expression representing the value is evaluated, and only when that completes is the body evaluated. Lazy evaluation, in comparison, stores the expression, but does not evaluate it until the value is needed, so that it performs the minimal amount of computation, at the expense of a possibly very large administrative overhead.

The evaluation model can affect the type of I/O structure that is appropriate. Eager evaluation, being close to imperative ordering, is well-suited to I/O based on side-effects, as the expressions are evaluated in an intuitive order. In comparison, Haskell's monads work well with its lazy evaluation. Lenient

[2]This is the case under lazy evaluation, at least. The removal of such a call when using eager evaluation may change termination. The effect of evaluation models and optimisations on program termination is a recurring theme of this thesis.

evaluation [144] is a hybrid model which maps well to hardware, and is explored in Section 5.2 as a basic for SASL's streamed I/O.

SASL uses eager evaluation combined with lazy (later, lenient) lists. Lazy lists in an eager language is a relatively old idea [53], used in languages such as Daisy [75] and Hope [30], and even seems to predate full laziness. LiMP, the processing architecture for Daisy, allows for the possibility of speculative execution, an effect similar to SASL's lenient evaluation.

Both ML and Haskell (as well as SASL) provide Hindley-Milner polymorphic type systems: the type systems provide strong typing, so that type errors cannot occur at runtime, but at the same time a function may work over a range of types (a useful feature for generic and higher-order functions). Higher-order functions are those that take or return other functions, allowing the creation of functions like the *map* function, which, given a function f, will return a function that applies f to each element of a list. Higher-order functions are discussed in Chapter 6.

1.3.3 Static Allocation

For a design to be converted to hardware with fixed storage requirements, the original program must be statically-allocatable—that is, if it were a computer program it would be able to be run with only a pre-determined, fixed amount of memory, and as hardware it only needs a known amount of storage. SASL must meet this requirement.[3]

With a statically-allocated language the hardware design can have all storage necessary included within the circuit, reducing the bottlenecks to the input and output stages. An effect of static allocation is that the language is no longer Turing-powerful. The programs are now finite state machines, although this is of little practical effect, as the state space may be too large to apply FSM analysis (if static allocation sounds like an excessive limitation, it should be remembered that all practical computers are also FSMs, albeit with a huge state space, due to their finite address range).

Modern mainstream languages are not static allocated. Some earlier languages, such as Fortran 77 [7], do not support dynamic memory allocation or recursion, but these restrictions are now viewed as unacceptable. It is generally possible to write statically-allocated programs in modern languages, by avoiding recursion and heap allocation, but as abstraction increases it becomes increasingly difficult to statically allocate programs. Features such as linear types (see below) in research languages show how some control may be returned to the programmer.

1.3.4 Static Allocation of Functional Languages

Programming languages have traditionally expected unbounded storage for use as a stack and heap. The stack is used to record local variables and return addresses for function calls, while the heap is generally used to store data structures allocated at runtime that may be returned by a function, and thus cannot be allocated on the stack. Functional languages seem to rely on unbounded memory more than imperative ones: statically-allocated loops and explicit calls to allocate memory are replaced by recursion and implicit memory allocation.

Many existing functional languages attempt to control memory allocation, to a limited extent, in the name of efficiency. For example, a purely functional array would be updated by taking an array, an index and the new value to be placed at that index, and returning the new array. However, there may still be references to the old array, and so the entire array may need to be copied. Solutions include wrapping the object in a monad, to prevent direct access to the object, uniqueness types [3] and linear types [148] which prevent the same object from being reused. These systems do not prevent the creation of new objects, but prevent unbounded copies of existing objects being created. Linear types play an important

[3]Such bounds are not *necessary*, since a program which is not statically-allocated may be converted to a piece of hardware connected to a large piece of memory. However, this approach may lead to the sorts of von Neumann bottlenecks that destroy any advantage a parallel hardware implementation has over a general-purpose CPU.

rôle in SASL.

To create a statically-allocated language, we must eliminate both the heap and stack requirements. To eliminate the heap, it is sufficient to eliminate recursive data structures. All datatypes will then be of a bounded size, and can be stored in the stack frame of the appropriate function call. Values returned from functions will be of a bounded size, and need not be stored on the heap when returned.

The stack may then be eliminated by preventing non-tail recursive calls. If all recursive calls were eliminated each call path would be of a bounded size, and the overall storage requirements would be bounded. Recursive tail calls require no extra storage, since the information associated with the calling function does not need to be preserved across the call, and the storage it used can be reused. Although these recursive calls allow unbounded call chains, the amount of live data at any point is bounded.

A similar set of restrictions is achieved by eliminating the stack first and then disallowing recursive data structures in the heap. The stack is eliminated by converting the program to *continuation-passing form*, where all function calls become tail calls, and the remainder of the function is passed as a closure. Non-tail recursive calls become recursive data structures in continuation-passing form, and are thus not permitted.

Even if internal storage is bounded, a program may still use unlimited resources, in the form of I/O. For example, it may perform an operation on each element of an unbounded stream of data. This is acceptable, as the I/O is a necessary part of the program's operation, and it is unreasonable to limit the program to reading finite input and producing finite output. Hofmann provides a model where space is bounded and in-place updates are used [68], but this does not suit our needs as the entire data structure to be modified must be loaded simultaneously. In effect, it only bounds "extra" memory beyond the size of the input. SASL further restricts programs to allow unbounded input and output only when the state required to manage that I/O is bounded.

The need for I/O should prove less of a bottleneck than the need for memory-access in processor-based systems. As data access patterns for stream processing are much more predictable than those for general memory access (the data will be written and read in a fixed order, all intermediate results will be held internally in fixed registers), the caches may be replaced by FIFOs, and complex technologies such as out-of-order execution, normally needed to hide memory latency, can be avoided, and the area made available for computation.

Statically-allocated closures are discussed in Chapter 6, and further statically-allocated extensions to the type system, such as sized types, are discussed in Chapter 8.

1.3.5 SASL's I/O Model

SASL's I/O model relies on treating input and output as lazily evaluated lists (or *streams*). The lists are unbounded recursive data structures. More specifically, input and output "channels" are represented as lists. These lists are not synchronised to each other, so the language is not well-suited to interactive systems (where there is some form of direct feedback loop between output and input) unless the programs are carefully written to read from and write to the lists in a fashion that matches the interfacing hardware.[4] SASL, like Lustre, is intended for reactive I/O, where there is no external feedback between the input and output data.

Reads are implemented by performing a match on a list. The matched head represents an item from the input stream, and the tail represents following items. Linear typing is used to prevent unbounded buffers from being needed, since there may only be one "pointer" into each stream. Output is performed by CONS expressions, combining a head expression, representing the value to be written, with a tail expression that will generate the rest of the output stream.

These lists are not limited to I/O, but may also be used for intermediate results. A list generated by one function may be returned as a top-level result, or fed as input into another function, as if the hardware

[4]For interactive I/O, constraints between the ordering of input and output operations may be required. These can be created using linear state objects or monads, but the extra ordering constraints may reduce performance.

representing the functions were connected in a pipeline. Lists are not just a special I/O feature, but can be used throughout SASL programs, encouraging the composition of subsystems.

Unbounded data structures, such as lists, cannot be dealt with directly within statically-allocated languages, so SASL uses *lazy* lists (that is, rather than storing an unbounded, potentially infinite list, only the information required to generate the list is actually held, in bounded storage). In terms of I/O, this means that the items in a list returned by a function are not generated until they are requested, and items from argument streams are not read in until they are needed. Internal lists (i.e., others than those used for I/O) are also generated in this demand-driven fashion. This lazy stream model fits well with a request/acknowledge signalling model. It contrasts with the synchronous I/O model used in languages like Lustre, where items are consumed and produced at a regular clocked rate. Indeed, SASL may be an appropriate language for asynchronous synthesis, although this thesis concentrates on a synchronous approach, with signalling.

The calling convention for a SASL program allows for a combination of scalar values and lists as parameters. The scalar arguments are read eagerly, but input lists are only read as values are needed. Functions may return both scalar values and lists. The scalar values are computed eagerly and returned, along with “list-ready” signals. The list elements are then produced by successively requesting elements. In hardware, the arguments and results become fixed buses. For each function invocation, buses representing scalar values send a single item, while those associated with lists may send multiple items, each representing a list element. Calling conventions are explained further in Section 3.2.

1.3.6 A Comparison to Other Languages

Although SASL shares much with SAFL and SAFL+, there are some distinct differences. At the language level, SAFL enforces static allocation through the use of very restricted types, and limiting programs to tail recursion only. SASL also disallows general recursion, but extends the type of data that may be manipulated to non-recursive algebraic datatypes, and adds streams.

In synthesis, SAFL aims for some transparency in the translation process, while SASL is an attempt to explore higher levels of abstraction. SASL tries to pipeline hardware, while SAFL shares resources, saving space but reducing the ability to pipeline. The distinction between the languages is strongest in the area of I/O. SAFL is restricted to a simple function call/return model, where no state is held between calls. SAFL+ introduces CSP-style channels, producing a much more expressive HDL, at the cost of losing the pure functional approach. SASL remains a pure functional language, performing reactive I/O through lazy lists. Both function calls and stream item production may be pipelined in SASL. SASL is intended to produce efficient and highly parallel circuits.

Lustre is another language created for hardware synthesis, with streamed I/O and a programming model abstracted from the hardware. However, it is not a conventional programming language. There are no scalar values, so that all processing must be done on streams. There is no iteration construct, but instead loops must be generated by filtering streams. SASL is much closer to a conventional language. A more detailed comparison of SASL to both SAFL+ and Lustre, along with an embedding of Lustre in SASL is given in Section 2.7.

1.4 Thesis Contributions and Organisation

The main contributions of this thesis are as follows:

The creation of a statically-allocated functional language (SASL) that provides the ability to process unbounded data sets through linear lazy lists (streams), bound by *linearity* and *stability* constraints (Chapter 2).

The use of this language to algorithmically describe circuits that process streams of data in a demand-driven manner (as demonstrated in a number of example programs, such as those featured in Appendix B).

Compilation techniques to convert this language to a form suitable for low-level hardware synthesis (Chapters 3 and 4).

The use of statically-allocated lenient evaluation to improve performance of synthesised hardware (discussed mainly in Section 5.2).

The addition of closures and non-deterministic operators to this functional behavioural hardware description language (Chapters 6 and 7).

The thesis consists of the following chapters:

Chapter 2 introduces basic SASL, motivated by the limitations of SAFL. The demand-driven evaluation model is compared to SAFL+ and the synchronous dataflow model of Lustre.

Chapter 3 provides a translation of SASL to CSP. While rather sub-optimal, this translation provides a basis for further synthesis techniques, and motivates some of the techniques used in graph synthesis.

Chapter 4 shows a translation of SASL to a format based on dataflow graphs. The nodes may be implemented with a structural HDL, and an example Verilog implementation of a few nodes is given in Appendix A. *Linear SASL* is introduced as an intermediate form.

Chapter 5 covers a range of optimisations that may be applied to SASL programs. This includes static scheduling, lenient evaluation and dataflow graph optimisation.

Chapter 6 deals with the possibilities introduced by other evaluation models. Statically-allocated closures, promises and laziness are introduced.

Chapter 7 moves on to non-deterministic processing. We cover non-deterministic reading from a set of streams, and the processing of bags (multisets).

Chapter 8 concludes the thesis. Further language extensions are briefly explored, and the work is evaluated. Conclusions are given.

CHAPTER 2

The SASL Language

This chapter introduces the syntax and semantics of SASL [52, 51], a "Statically Allocated Stream Language". Only the language itself is discussed in this chapter—synthesis methods are discussed in Chapters 3 and 4. SASL, like Sharp's SAFL [128], is statically allocated, so that the resources required to run a SASL program are fixed and bounded. SASL can be viewed as a development of SAFL, introducing streamed I/O through linear lazy lists (SASL also takes a different approach to compilation, as discussed later).

Section 2.1 reviews SAFL and SAFL+, setting the scene for SASL. It also discusses functional I/O techniques. Section 2.2 discusses related work. Section 2.3 demonstrates a naïve stream processing language, which is not statically allocated. Section 2.4 adds restrictions to the language in order to make it statically allocated, and Section 2.5 provides semantics for the language. Section 2.6 discusses another static allocation technique, *deforestation*. The penultimate section in this chapter compares SASL to SAFL+ and the synchronous dataflow language Lustre. The final section provides a summary.

2.1 The Motivation: SAFL and SAFL+

SAFL stands for "Statically Allocated Functional Language". It is a first-order, strict functional language for hardware synthesis. However, it is a simple language, with poor I/O support. SAFL+ attempts to rectify the I/O situation, at the expense of creating an impure language.

2.1.1 The SAFL Language

Sharp's language, SAFL, is the basis for his PhD thesis [128], as well as a number of papers [129, 108, 109, 130, 110, 111]. A rather minimal language, its abstract syntax is given in Figure 2.1.[1] It is statically-typed, with the data types representing n-bit buses, so that all values in the language can be statically allocated. To ensure SAFL is statically allocated, it is sufficient to require that all recursive calls are tail recursive, preventing the need for a stack.

SAFL's main weakness comes from its pure functional approach. The only way to interact with a piece of hardware generated by SAFL is through a call/return mechanism that models a software function call. Parameters are provided on a bus, the hardware triggered, and if the function terminates a result will be provided on another bus. Calls are not pipelined, and no state is held between function calls. By adding external storage, output can be fed back, as shown in Figure 2.2. In this way, a SAFL program may work

[1]Throughout this thesis, the notation $\vec{x}$ is used as shorthand for a sequence of items, $x_1, \ldots, x_k$.

$p :=$	$d_1 \ \ldots \ d_n$	Program definition
$d :=$	**fun** $f(\vec{x}) = e$	Function definition
$e :=$	c	Constant
	x	Variable
	if e_1 **then** e_2 **else** e_3	Conditional expression
	let $\vec{x} = \vec{e}$ **in** e_0	Variable binding
	$a(e_1, \ldots, e_n)$	Primitive function call
	$f(e_1, \ldots, e_n)$	Function call

Figure 2.1: The abstract grammar of Sharp's SAFL

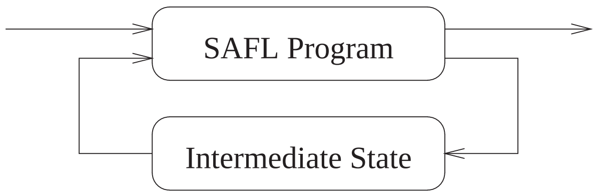


Figure 2.2: A SAFL program embedded in a system with state

as a stream processor, but this approach is rather inelegant and inflexible. SAFL+ attempts to improve on this I/O model.

2.1.2 SAFL+: An Attempt to Improve I/O

SAFL+ adds CSP-like channels to basic SAFL. These channels allow separate processes to communicate by having one process write to a channel, and another read. SAFL+'s channel model allows for multiple readers and writers, which can lead to non-determinism. As functions may now have side effects, sequential and parallel composition operators are introduced. The use of channels can also lead to deadlock.

Using channels, it is possible to create explicit pipelines: a set of processes are created in parallel that feed a stream of elements between them via channels. However, this use of channels greatly complicates such programs compared to a pipeline created through the composition of functions (which is the way pipelines are created in SASL).

Compared to SASL, the SAFL+ channels are less restricted. Data items may be passed around a loop of channels, while SASL stream dependencies are acyclic (preventing deadlock). Since both languages are statically allocated, this difference will not lead to a difference in expressive power, but only in convenience for the programmer. We argue that the restrictions of SASL are likely to improve productivity, as the advantages of making SASL programs easier to reason about and more natural to write outweigh any limitations on the programmer's flexibility.

2.1.3 Functional I/O

An introduction to I/O in functional languages is given in Section 1.3.2 and, more specifically, an introduction to SASL's I/O model is in Section 1.3.5. This section covers the I/O models of pure functional languages in more detail.

The I/O facilities of a system may be categorised into one of three types (an extension of the categorisation of [61]):

Transformational Systems take an input value, perform a calculation and return a result. A typical example would be a mathematical function.

Reactive Systems read inputs and produce outputs during processing. Reactive systems can process data continuously. However, no direct dependencies are expected between the input and output, outside of the system. This allows unbounded buffering or delays between the input and output stages. For example, reactive systems are suitable for DSP applications, but not suited to controlling an interactive user interface.

Interactive Systems provide the generalised I/O one might expect of a general-purpose computer system. There can be arbitrary external dependencies between the output and input, such as with interactive user interfaces.

This list is intended as a categorisation of the I/O model at a computational level. The underlying I/O signals could be anything from demand-driven streams with complex signalling to strictly synchronous buses.

The differences between these categories are based on how the systems deal with synchronisation. A transformational system is like a function with eagerly evaluated arguments and results. Activation and return are synchronised with the complete evaluation of the appropriate value. Infinite input and output are impossible.

A reactive system is like a function with lazily evaluated parameters and results. The start and return of the function are not necessarily synchronised with the production of I/O items. A value representing the input may be implemented as a lazily-evaluated recursive data structure which, when evaluated, returns some input along with an object to access the rest of the input through. Alternatively, it could be represented as an opaque state object, with an input function that takes a state, and returns an input item and a new state. Output can be achieved by returning a lazy data structure representing the output, or by using a state-holding object.

An important point with reactive systems is that the input and output are not synchronised. Input streams may be read ahead of time or output delayed, without affecting the results; a reactive system that reads one stream and produces another may buffer as many items as it likes. This reading of items ahead of time is the focus of the lenient evaluation optimisation of Section 5.2.

Interactive systems create dependencies between the inputs and outputs, so that a direct temporal relationship can be given between reads and writes. Interactive systems are therefore more restrictive than reactive systems. To create an interactive system, an ordering must be placed on all the I/O events, for example by passing around a single state-holding object that is used to perform all input and output, enforcing serialisation.

SASL is based on the reactive model. Its main aim is to implement algorithms, rather than perform in an interactive environment. Transformational systems are too restrictive, as static allocation would limit the size of processable data sets (hence SAFL's extension to SAFL+). The interactive model would restrain SASL too much, causing unnecessary serialisation.

Other restrictions, ignored by the above categorisation, are those to do with dependencies between sets of similar streams—how a set of writes to different streams are interleaved (and similarly for reads). SASL takes the least restricting approach, assuming that all streams are independent of each other, in the hope that this provides the most flexibility in synthesis.

An alternative way to view a functional language's I/O capacities are in terms of the way they are implemented. The rest of this section describes some pure functional I/O frameworks:

Call and Return The call-and-return model is the model used by SAFL, and matches up with transformational systems. A finite argument is passed in, and a finite result returned. Since all a pure functional language may do in terms of I/O is to take an argument and return a result, the other I/O models rely on what is basically a call-and-return model, where the values hold some I/O state information.

State-holding Objects I/O may be performed by passing in or returning values that represent I/O paths. The possibilities are:

Argument used for input This provides a simple way to receive input. A parameter contains the input state, and a function can extract items, returning a new state to read. If the input object is not linear (see below), all the input data must be kept, in case a reference to the original input state object is kept in the program, which would allow the program to re-read all its input.

Result used for output In order to write data, an object is returned that contains data to be written, and a closure to perform further computation. The closure is necessary, as the program can only produce finite output before returning.

Result used for input Perhaps surprisingly, returning a value can be used as a mechanism for input. The return value contains a closure which takes a parameter representing the value to be read, and the runtime executes the closure, supplying the appropriate argument.

Argument used for output For completeness sake, there is also this possibility. A value is passed in, and an output function takes the value, and an item to be written, returning a new state value. Linear typing is necessary to prevent an output state from being reused. Unlike the other systems mentioned, this makes it possible to have a function that produces output but never returns, providing rather odd semantics.

Linear typing [148] requires a linear value be used exactly once. This is useful for state-holding values representing input or output, in that it mirrors the state—when an operation is performed, the old state is used up, and a new one created, which in turn may only be used once. Linear typing is a major feature of the restrictions SASL uses to ensure static allocation, and is discussed in more detail later in this chapter. A similar approach is used in the *uniqueness typing* [3] of Clean.

Lazy lists are natural representations of input and output streams under a “arguments for input, results for output” model. The function to read from the input-state object becomes a stream matching, while the function to write to the output is a CONS expression. A separate lazy list is used for each input and output stream. This is the approach taken by SASL, which seems well-suited to reactive I/O, and provides an I/O model that will be familiar to functional programmers.

If returning a value is used for both input and output, the input and output operations can be performed on the same object, allowing the creation of ordering dependencies as required for interactive systems. This is effectively how monad-based I/O works:

Monads Monads are the I/O mechanism of Haskell [80]. Monads can be thought of as wrapping up an object so that it may only be accessed in a restricted manner. Once a value has been placed in a monad, it cannot be extracted. Monads can be sequenced together, with the value from the first monad passed into the second. The resulting monad cannot then be split up. This allows values held in monads to be passed around and used in a controlled fashion, but the values may never “escape” from the monad.

I/O is achieved by creating a monad that performs the program by sequencing together input and output functions. The runtime system then performs the sequenced parts of the monad, evaluating the contained closures as necessary. The monad is effectively a “list of things to do” returned by the program and evaluated by the runtime, except that since the list contains closures the exact operations performed can be data dependent.[2] The evaluation of I/O outside the program body allows the state information to be held externally to the function (preventing the need for linear state objects).

[2]A real implementation performs the I/O as the associated monad is evaluated, and relies on lazy evaluation to only evaluate the monad that is returned.

Synchronous Streams An alternative to creating a function that processes a stream of values by iterating over the elements is to make all the I/O values on a stream of data into a single value that is manipulated in its entirety. This is the approach used in languages such as Lucid [8] and Lustre [60]. The stream is the primitive datatype, and basic arithmetic and logic functions are treated as mapping functions over the entire streams. A comparison between SASL and Lustre is given in Section 2.7.

2.2 Other Related Work

A number of related languages, and the hardware background for SASL, were discussed in Chapter 1. The most strongly-related languages, SAFL and SAFL+, were discussed at the start of this chapter.

There are a number of formalisms suitable for dealing with statically-allocated parallel languages. CSP [67] is an imperative-style language with explicit parallelism, where communication occurs through channels. Data is passed when a read on a channel in one process matches up with a write in another process.

Concurrent Functional Processes (CFP) [20] provides a functional alternative to CSP, and might form the basis for a functional hardware synthesis system. However, it does not seem well-suited to generating statically-allocated systems, and the parallelism model is explicit, taking the opposite approach to SASL. Functional nets [113, 114] are another approach based on the join calculus [32], providing a way to merge petri-net style processing with functional languages. This and other approaches based around petri nets [89] depend on explicit parallelism.

Another approach is that taken by Kahn-MacQueen networks [90], where computation is achieved through a set of communicating processes which send data items to each other along the edges of a graph. The edges effectively represent streams of data, and SASL programs can be represented as fixed-topology Kahn-MacQueen networks where the processes are statically allocated. Fixed-topology Kahn-MacQueen networks are allowed unbounded queues, and so not all such networks can be translated to SASL. Fixed-topology Kahn-MacQueen networks with bounded queues and finite state processes have equivalent power to SASL programs.

Neil Jones' work on the power of CONS-less languages [77] provides a theoretical viewpoint on the expressiveness of languages with restrictions similar to static allocation. Whereas SASL allows the lazy creation of streams, Jones considers languages without the ability to create unbounded structures. Jones's "read-only tail recursive" functions are statically allocated, but allow non-linear access to their input, so that it can be "rewound" and re-read. As such, the language has the computational power of LOGSPACE, rather than just that of a finite state machine (which SASL is limited to), since the back references into the input list can act as unbounded values.

Due to issues like these, Jones' model is of limited use for our purposes, but it does raise a point of some relevance: adding higher-order functions can increase the power of some restricted languages, as data can be stored in nested closures. For example, closures can be used to convert programs to continuation-passing form, eliminating non-tail recursion, but producing a program that still cannot be statically allocated. Rather than attempt to limit a higher-order language to programs that are statically allocatable, we initially use a first-order language (although closures are discussed further in Chapter 6). SASL streams can be viewed as data structures containing closures that are guaranteed to be statically allocatable.

Linear typing allows data to be processed once and only once, which can be used for destructive array updates and so on, providing an efficient way to keep large state variables in a pure functional language. In SASL, affine linear typing is used to ensure that each stream item is read *at most* once. Linear typing means that we can ensure that unbounded buffering is not required for streams.

Wadler's listless transformation [146] allows programs with intermediate lists to be converted to a form where the intermediate lists are never fully generated.[3] In software, this can convert some programs to a

[3]Deforestation performs a similar operation to the listlessness transformation, but on *tree-like* data structures, although the

form that does not need more than a statically-allocated amount of storage. In SASL, the demand-driven nature of stream production means that programs may be statically allocated without performing listless or deforestation transformations. This transformation is discussed further in Section 2.6.

Pareto's PhD thesis [121] describes "Synchronous Haskell", a language that uses sized types [70] to guarantee that well-typed programs are free from busy loops and deadlocks, and are of bounded memory size. However, this language has a complex typing scheme that provides other information beyond static-allocatability, such as productivity. At the same time, some useful language features are restricted. For example, filtering of a stream is only possible by creating a stream with so-called *nothing* elements (known as *hiatons* in Lucid), which are used to replace elements which would have been filtered out, to maintain the original stream size. The use of sized types in SASL is discussed in Section 8.2.

2.3 A Naïve Stream Processing Language

In this section a simple first-order statically-allocated language similar to SAFL will be introduced, and then naïvely extended with stream processing constructs. Examples are then given of the problems raised by adding unconstrained stream processing to the language.

2.3.1 The Stream-less Language

We start with a strict first-order statically-allocated language. To achieve static allocation, general recursion is disallowed (as in SAFL), as are recursive datatypes. Non-recursive algebraic datatypes and tail recursion are provided (although we have not included datatype definitions in the abstract grammar presented here). Tuples can be implemented using datatypes (we take the Haskell-like approach that a constructor takes zero or more arguments, rather than ML's approach of taking either no arguments or a single tupled argument). A Hindley-Milner type system is used to type the language, and polymorphic functions are *specialised* to concrete types during synthesis.

SASL's algebraic datatypes and tuples provide much more flexibility than SAFL's buses, and attempt to bring the types expected of a high-level language to HDLs. SASL is relatively weak at expressing plain n-bit buses, as they need to be expressed as tuples of binary values in basic SASL (although buses could be added through simple language extensions).

An abstract grammar for the language is shown in Figure 2.3.[4] Subexpressions marked with the suffix "tr" are those that are in a *tail context* if the enclosing expression is in a tail context. The top level expression of a function is in a tail context. Recursive calls may only occur in tail contexts.

Only direct tail recursion and non-recursive datatypes are allowed, although the language could be extended to allow mutual tail-recursion and sized recursive datatypes [70] with upper size limits, without changing the language's expressiveness or ability to be synthesised. These and other extensions, such as lexically scoped functions, are discussed later in the thesis.

The language's semantics are strict, since lazy evaluation, in the absence of perfect strictness information (which is uncomputable in the general case), may pass closures recursively so that they build up without bound.

2.3.2 Stream-processing extensions

To add streams to the language, the ability to construct and read values from them is required. The added language features are shown in Figure 2.4. CONS nodes are evaluated lazily. When a CONS is evaluated, both the head and tail parts are evaluated, and evaluation of both must complete (the tail evaluating to another lazily-evaluated CONS node) before values are returned. This allows streams containing no items,

transformed program may still require unbounded storage.

[4]Example code may have some extensions beyond this basic grammar in order to aid readability, but conversion to this basic form is simple.

$$\begin{array}{rll}
p := & d_1 \ \ldots \ d_n & \text{Program definition} \\
d := & \textbf{fun} f x = e^{tr} & \text{Function definition} \\
e := & f e & \text{Function application} \\
 & c(e_1, \ldots, e_k) & \text{Constructor} \\
 & \textbf{case}\ e\ \textbf{of}\ m_1 \ \ldots \ m_n & \text{Case expression} \\
 & \textbf{let}\ x = e_1\ \textbf{in}\ e_2^{tr} & \text{Let expression} \\
 & x & \text{Variable access} \\
m := & c(x_1, \ldots, x_k) \quad x^{\ tr} & \text{Match}
\end{array}$$

Figure 2.3: Streamless-SASL's abstract grammar

$$\begin{array}{rll}
e := & \ldots & \\
 & e_1 \mathbf{::} e_2^{tr} & \textsc{cons}\ \text{expression} \\
 & \textbf{case}\ e_1\ \textbf{of}\ x_1\mathbf{::}x_2 \quad x\ {}_2^{tr} & \text{Stream-matching}
\end{array}$$

Figure 2.4: Grammar extensions for stream processing

by creating an infinite loop in either the head or tail expression. The tail of the stream is a tail context for the purposes of tail recursion.

The semantics of the stream-processing constructs can be defined in terms of a syntactic conversion to normal ML. The corresponding *stream* datatype is given by:

$$\textbf{datatype}\ \alpha\ \textit{stream} = \textit{cons}\ \textbf{of}\ \textit{unit} \quad (\alpha \quad (\alpha\ \textit{stream}))$$

Translation can then be performed at a syntactic level: $e_1\mathbf{::}e_2$ becomes *cons*$(\textbf{fn}() \quad (e\ _1, e_2))$ and **case** e_1 **of** $x_1\mathbf{::}x_2$ becomes **case** e_1 **of** *cons*(f) **let** $(x\ _1, x_2) = f()$ **in** e_2 **end**, where f is a fresh temporary variable.

Only infinite streams are implemented in SASL. Finite streams can be simulated by wrapping stream elements up in an *option* datatype, and treating the first *None* element as the end of the stream. Similarly, non-terminating streams can be simulated by streams with end markers by making sure that the end markers never crop up in practice, and adding never-executed pieces of code to match on end-of-stream cases.

An alternative datatype could have been used:

$$\textbf{datatype}\ \alpha\ \textit{stream'} = \textit{cons'}\ \textbf{of}\ \alpha \quad (\textit{unit} \quad (\alpha\ \textit{stream'}))$$

This definition seems inferior, as it would be impossible to represent a totally unproductive stream, and so the other was used.

2.3.3 Problems raised

Before introducing constraints to make the language statically allocated, it may be useful to look at what can go wrong if no extra limitations are applied. Unbounded storage requirements can occur if old parts of a stream need to be buffered, and the amount of data required to represent a stream can also accumulate if it is not generated carefully:

Streams that produce data at different rates could be merged, requiring unbounded buffering (case 1).

A stream may be recursively built up by repeated CONS operations in non-tail contexts, or more

```
(* 1. Streams that may require unlimited buffering *)
fun odds(x1::x2::xs) = x1::odds(xs)
fun zip(x::xs, y::ys) = (x, y)::zip(xs, ys)
fun needs-buffer(stream) = zip((odds(stream)), stream)

(* 2. A stream recursively CONS'd upon *)
fun build(item, stream) = build(item, (item::stream))

(* 3. Streams that can recursively increase storage requirements *)
(* (f is some function such as fn x    x + 1)                     *)
fun map_f(x::xs) = f(x)::map_f(xs)
fun map-iter_f(stream) = let stream' = map_f(stream)
                         in case stream' of x::xs    x::map-iter  _f(xs)
```

Figure 2.5: Programs that cause problems for static allocation

subtly have mappings recursively applied, so that the amount of information that must be held about the stream will grow unbounded (cases 2 and 3).

The next section introduces a type system and *linearity* and *stability* restrictions that make the language statically allocatable.

2.4 Restrictions for Static Allocation

In order to simplify analysis, a stratified type system is introduced. Two sets of constraints are then applied to make the language statically allocated—*linearity* prevents stream elements from being reused, and *stability* prevents the description of a stream from "blowing up", with the stream requiring more and more space to represent it on each recursive call.

2.4.1 The stratified type system

To simplify the analysis, we wish to avoid situations involving streams of streams, streams held in algebraic datatypes, and so on. At the same time, it is useful to express the type of functions without making the language higher-order. To this end, we create a stratified type system. Polymorphic types are handled in the usual manner.

The lowest layer, represented by the type variable τ, consists of *basic types*, which are the types of expressions that can occur in the simple stream-less language. All values are created using non-recursive constructors with zero or more arguments, and tuples are implemented using constructors. For example, statically-sized integers can be represented using tuples of booleans (*true* and *false* being zero-argument constructors) mirroring the binary representation. The unit datatype, a "no information" value normally represented as "()", and used for synchronisation/triggering purposes, is implemented as an algebraic datatype with a single constructor which has no parameters. Values of a basic type τ have bounded storage requirements that are a function of τ.

The next layer consists of *value types*, represented by the type variable σ:

$$\sigma := \tau \quad \tau\ stream_i \quad \sigma_1 \quad \dots \quad \sigma\ _n$$

Value types are the types associated with expressions and variables. The type may be a basic type, τ, a stream of basic type, $\tau\ stream_i$, or a tuple of value types—$\sigma_1 \quad \dots \quad \sigma\ _n$.

$$
\begin{array}{lll}
e := & \ldots & \\
 & (e_1, \ldots, e_k) & \text{Tupling} \\
 & \textbf{case } e_1 \textbf{ of } (x_1, \ldots, x_k) \quad e\,{}_2^{tr} & \text{Untupling}
\end{array}
$$

Figure 2.6: Grammar extensions for handling tuples

Each stream is given an identifier i that is used to identify the stream during stability analysis. The identifier is either a symbol from an infinite alphabet S, representing a parameter stream, or "$\star$", representing a newly created stream. We use "$S\star$" to represent the set $S \quad \star$. The function $SI(\sigma)$ is defined to return the set of stream identifiers (including $\star$) used in the typing σ. Stream type values can produce an infinite stream of items, but the amount of state required at any point to represent the stream is bounded[5], and so value types can be stored in a fixed amount of space.

The tuple type constructor allows the creation of tuples of value types (as opposed to basic types, which are tupled using constructors). A new tupling operator and its associated **case** expression are also added to the language, using the grammar extensions shown in Figure 2.6.

The type system's top level extends the type system to cover the types of functions and constructors, by adding *arrow types*. Functions have the type $\sigma_1 \quad \sigma\ _2$, while constructors have the type $\tau_1\ \ldots\ \tau_n$ $\tau\ (n \quad 0)$. These typings only apply to functions and constructors, and do not appear in the types of expressions or the typing environment, which only contain value types.

The language's typing rules are shown in Figure 2.7. The typing environment, A, is a mapping of variables to value types, and the types of functions and constructors are treated as side-conditions. The typing of stream identifiers in functions is very similar to that used for polymorphic typing. For example, if an expression has the typing:

$$x_1 : \tau_1\ \textit{stream}_\alpha, x_2 : \tau_2\ \textit{stream}_\beta \qquad e\ :\ \tau_1\ \textit{stream}_\alpha \quad \tau\ _2\ \textit{stream}_\star$$

then the typing of the function f given by **fun** $f(x_1, x_2) = e$ can be written as:

$$\alpha, \beta \quad \tau_1\ \textit{stream}_\alpha \quad \tau\ _2\ \textit{stream}_\beta \quad \tau\ _1\ \textit{stream}_\alpha \quad \tau\ _2\ \textit{stream}_\star$$

in a way analogous to polymorphic typing. As functions cannot have free stream identifiers in this language, we omit the qualifiers, as is done in ML with type variables.

In general, the type of a function **fun** $f\ x$ = e is $f : \sigma_1 \quad \sigma\ _2$, where σ_1 and σ_2 are given by the typing rules, using $x : \sigma_1 \qquad e\ :\ \sigma_2$. Fresh stream identifiers are created for all streams in σ_1; the stream identifiers in σ_1 must be distinct, with $SI(\sigma_1) \quad S$. Since the type system does not otherwise introduce new stream identifiers, $SI(\sigma_2) \quad SI(\sigma\ _1) \quad \star$. Due to linearity (explained below), each non-$\star$ identifier may occur at most once in σ_2.

The (APPLY) typing rule includes a substitution on stream identifiers, in order to match up the stream identifiers of the formal and actual parameters. The same substitution is then applied to the return type.[6] The substitution is similar to those done in calls to polymorphic functions. The use of the (APPLY) rule across recursive calls is discussed later in this chapter.

The rule (VAR) relies on a $\star$-*substitution*. This is a substitution that replaces zero or more stream identifiers with "$\star$". This substitution allows a stream derived from a parameter stream to drop its original stream identifier, in order to pass the (CONSTR-ELIM) and (CONS-INTRO) typing rules. Converting a stream identifier to $\star$ is safe, as this just throws away information.

[5]Externally provided input streams could provide an oracle supplying data that could not be generated internally, but the only internal state required would be a reference to the external data source.

[6]As mutual recursion is disallowed, a total ordering of the functions can be created so that no function requires the type of a function that has not yet been processed.

$$(\textsc{apply})\frac{A \quad e : \sigma_1 \quad f : \sigma_2 \quad \sigma_3}{A \quad f\,e : \theta(\sigma_3)}\ \theta(\sigma_2) = \sigma_1$$

$$(\textsc{constr-intro})\frac{A \quad e_1 : \tau_1 \qquad\qquad A \quad e_k : \tau_k}{A \quad c(e_1, \ldots, e_k) : \tau}\ c : \tau_1 \ldots \tau_k \quad \tau$$

$$(\textsc{tuple-intro})\frac{A \quad e_1 : \sigma_1 \qquad\qquad A \quad e_k : \sigma_k}{A \quad (e_1, \ldots, e_k) : \sigma_1 \quad \ldots \quad \sigma_k}$$

$$(\textsc{cons-intro})\frac{A \quad e_1 : \tau \quad A \quad e_2 : \tau\ \mathit{stream}_\star}{A \quad e_1 \mathbin{\mathbf{::}} e_2 : \tau\ \mathit{stream}_\star}$$

$$(\textsc{constr-elim})\frac{A \quad e : \tau \quad \left\{\begin{array}{l} A, x_1^1 : \tau_1^1, \ldots, x_{k_1}^1 : \tau_{k_1}^1 \quad e_1 : \sigma \\ \ldots \\ A, x_1^n : \tau_1^n, \ldots, x_{k_n}^n : \tau_{k_n}^n \quad e_n : \sigma \end{array}\right.}{\begin{array}{l} A \quad \textbf{case } e \textbf{ of } c_1(x_1^1, \ldots, x_{k_1}^1) \quad e_1 \\ \ldots \\ c_n(x_1^n, \ldots, x_{k_n}^n) \quad e_n : \sigma \end{array}} \left\{\begin{array}{l} c_1 : \tau_1^1 \ldots \tau_{k_1}^1 \quad \tau \\ \ldots \\ c_n : \tau_1^n \ldots \tau_{k_n}^n \quad \tau \end{array}\right.$$

$$(\textsc{tuple-elim})\frac{A \quad e_1 : \sigma_1 \quad \ldots \quad \sigma_k \quad A, x_1 : \sigma_1, \ldots, x_k : \sigma_k \quad e_2 : \sigma}{A \quad \textbf{case } e_1 \textbf{ of } (x_1, \ldots, x_k) \quad e_2 : \sigma}$$

$$(\textsc{cons-elim})\frac{A \quad e_1 : \tau\ \mathit{stream}_i \quad A, x_1 : \tau, x_2 : \tau\ \mathit{stream}_i \quad e_2 : \sigma}{A \quad \textbf{case } e_1 \textbf{ of } x_1 \mathbin{\mathbf{::}} x_2 \quad e_2 : \sigma}$$

$$(\textsc{let})\frac{A \quad e_1 : \sigma_1 \quad A, x : \sigma_1 \quad e_2 : \sigma_2}{A \quad \textbf{let } x = e_1 \textbf{ in } e_2 : \sigma_2}$$

$$(\textsc{var})\frac{}{A, x : \sigma \quad x : \theta(\sigma)}\ \theta \text{ is a } \star\text{-substitution}$$

Figure 2.7: Typing rules

$$\begin{array}{rcl}
\mathit{lin}(f\ e_1) & = & \mathit{lin}(e_1) \\
\mathit{lin}(c(e_1, \ldots, e_k)) & = & \mathit{lin}(e_1) \quad \ldots \quad \mathit{lin}(e_k) \\
\mathit{lin}((e_1, \ldots, e_k)) & = & \mathit{lin}(e_1) \quad \ldots \quad \mathit{lin}(e_k) \\
\mathit{lin}(e_1 \mathbin{\mathbf{::}} e_2) & = & \mathit{lin}(e_1) \quad \mathit{lin}(e_2) \\
\mathit{lin}(\textbf{case } e \textbf{ of } m_1 \ \ldots\ m_n) & = & \mathit{lin}(e) \quad (\mathit{lin}_m(m_1) \quad \ldots \quad \mathit{lin}_m(m_n)) \\
\mathit{lin}(\textbf{case } e_1 \textbf{ of } (x_1, \ldots, x_k) \quad e_2) & = & \mathit{lin}(e_1) \quad (\mathit{lin}(e_2) \quad x_1, \ldots, x_k\ ) \\
\mathit{lin}(\textbf{case } e_1 \textbf{ of } x_1 \mathbin{\mathbf{::}} x_2 \quad e_2) & = & \mathit{lin}(e_1) \quad (\mathit{lin}(e_2) \quad x_1, x_2\ ) \\
\mathit{lin}(\textbf{let } x = e_1 \textbf{ in } e_2) & = & \mathit{lin}(e_1) \quad (\mathit{lin}(e_2) \quad x\ ) \\
\mathit{lin}(x) & = & \left\{\begin{array}{ll} x & : \text{ if the type of } x \text{ contains a stream} \\ & : \text{ otherwise} \end{array}\right.
\end{array}$$

$$\mathit{lin}_m(c(x_1, \ldots, x_k) \quad e) = \mathit{lin}(e) \quad x_1, \ldots, x_k$$

Figure 2.8: Linearity rules

2.4.2 Linearity

The linearity constraint prevents a reference into a stream being reused, so that once an item has been read it cannot be read from the stream again. This is achieved by allowing each stream variable to be used at most once, for example being passed to only one subroutine in a function. Not using a stream variable is also permitted.

To generate a statically unbounded number of elements of a stream, a function must generate it using a CONS in a tail-call position (the alternative, using an accumulator argument to a function, is disallowed by the stability constraint below). In this case, the function's return type must be a single stream, because of this CONS in the tail-call position.

Due to linearity, one stream cannot be passed to multiple functions in parallel, and since functions that generate an unbounded amount of a stream can only return one stream, each stream can only have one other stream that directly depends on it for an unbounded number of elements. Therefore, it is not possible to generate multiple distinct streams that depend on an unbounded portion of same original stream (one stream may be used to generate another which generates a third, but linearity means the first two streams are then "used up"). Linearity similarly prevents a stream being passed to a function along with a stream it depends upon, since the original stream will have been "used up". Linearity thus prevents the synchronisation problems of Example 1 in Figure 2.5, as well as preventing "rewinding" through back-references into the stream.

Linearity can be ensured by labelling each expression with the set of linear variables it uses. The sets of variables are built in a bottom-up manner using the function lin shown in Figure 2.8. A linear value cannot be held inside a non-linear value, so any type containing a stream must be linear. The operators and are defined as follows:

$$s \quad t \quad = \begin{cases} \textbf{error} & : \text{ if } s = \textbf{error} \quad t = \textbf{error} \quad s \quad t \; = \\ s \quad t & : \text{ otherwise} \end{cases}$$

$$s \quad t \quad = \begin{cases} \textbf{error} & : \text{ if } s = \textbf{error} \quad t = \textbf{error} \\ s \quad t & : \text{ otherwise} \end{cases}$$

Merging sets that both contain the same linear variable means the value is used in multiple sub-expressions, and produces an error. A program has the required linearity property if none of its function bodies yield **error** when lin is applied.

2.4.3 Stability

In this language, stream processing is achieved using recursive functions. To produce non-stream results tail-recursion may be used, while to generate an entire new stream a tail recursive call must be performed in the *tail part* of a CONS node.

These two forms of recursion have some anti-symmetry in the forms of allowed functions, as shown in the examples of Figure 2.9. In plain tail recursive functions, expressions may be evaluated before the tail call, but not afterwards, as this would require extra storage. For tail recursion on streams, CONS operations may occur on the result of tail calls, because they will be implemented as side effects *before* the function call, but tail calls on a CONS'd version of the input could, in general, create a stream requiring unbounded space.

To forbid streams that require unbounded amounts of space to represent them, we must forbid the streams from being recursively processed in a way that increases the storage requirements each iteration.

We introduce the concept of *stability*, where streams passed to a tail call must be *substructures* of the corresponding streams in the function's formal parameter. The substructures of a stream are the stream itself, and those streams reached by repeatedly taking the tail of that stream. If the streams that are passed recursively are limited to substructures of the original parameters, the space requirements of the streams

(* Plain tail recursion. *)
fun $f_1(x) \quad = \ldots\ 1 + f_1(x)\ \ldots$ **(* Disallowed. *)**
fun $f_2(x) \quad = \ldots\ f_2(x+1)\ \ldots$ **(* Allowed. *)**

(* Stream tail recursion. *)
fun $g_1(x\mathbin{\textbf{::}}xs) = \ldots 1 \mathbin{\textbf{::}} g_1(xs)\ \ldots$ **(* Allowed. *)**
fun $g_2(xs) \quad = \ldots\ g_2(1 \mathbin{\textbf{::}} xs)\ \ldots$ **(* Disallowed. *)**

The displayed parts of the function bodies are assumed to be in tail context of a larger expression.

Figure 2.9: Examples of recursive functions

cannot build up.

In the typing system of Section 2.4.1, only a stream that is guaranteed to be a substructure of a particular parameter stream may have that stream identifier. Otherwise the stream will have the identifier "⋆".

Hence, the stability restriction is simply that for a tail recursive call the stream identifiers in the formal and actual parameters of the function must match. A modified (APPLY) rule to achieve this constraint is as follows:

$$(\textsc{apply})\,\frac{A \qquad e\ :\ \sigma_1}{A \qquad f\,e\ :\ \theta(\sigma_3)} \quad \begin{array}{c} f\ :\ \sigma_2 \quad \sigma\ \ 3 \\ \theta(\sigma_2) = \sigma_1 \\ \text{If } f\,e \text{ is a recursive tail call} \\ \text{then } \theta \text{ must be the identity} \end{array}$$

2.4.4 Static Allocation

Viewing CONS expressions as closures, with CONS-matching forcing the evaluation, the programs can be statically allocated if the closures are guaranteed to be statically allocatable (since the language without streams is statically allocatable). An informal argument to this effect is as follows: when using non-recursive datatypes, a closure may only take an unbounded amount of space if the environment of the closure contains another closure which may then recursively contain other closures in an unbounded manner. However, this is not possible, as closures are not built up across recursive calls—the recursively-passed streams can only be substructures of the original arguments. The language provides no opportunity to create such a closure.

2.4.5 Example Programs

Common operations to generate, map, filter and fold lists are simple to write in SASL, as shown in Figure 2.10.[7] Streams may be merged together (subject to linearity), but streams may not be duplicated, or multiple streams created that depend on unbounded sections of the same stream, since the resulting streams may require unbounded buffering if merged together. Example merge and duplication functions are shown in Figure 2.11. This difference to synchronous stream languages is discussed in Section 2.7.

2.5 SASL Semantics

This section presents a Structured Operational Semantics [123] for the language. The semantics, shown in Figure 2.12, are those of simple eager evaluation, except for the lazy lists. The rules show how the pair of an environment S and expression e are evaluated to produce a value. The environment is a function from variables to their values, and initially has an empty domain. The values are trees, whose nodes

[7]To avoid using higher-order functions, *f*, *g* and so on are assumed to be provided as top-level functions.

```
fun from(i) = i :: from(i + 1)
fun map_f(x::xs) = f(x) :: map_f(xs)
fun filter_p(x::xs) = if p(x) then x::filter(xs) else filter(xs)
fun fold_g(x::xs, accumulator) =
    if done(x)
    then accumulator
    else fold_g(xs, g(accumulator, x))
```

Figure 2.10: Examples of common functions

```
(* An allowed merge function. *)
fun merge_h(x::xs, y::ys) = h(x, y)::merge_h(xs, ys)

(* A disallowed duplication function. *)
fun dup-stream(stream) = (stream, stream)
```

Figure 2.11: Examples of merge and (illegal) duplication functions in SASL

are algebraic datatype constructors and tupling operators, and whose leaves are 0-place constructors and stream values. Streams are represented with a triple such as $[S, e_{hd}, e_{tl}]$. In itself, this is not a particularly useful value. To obtain the head of the stream, e_{hd} must be evaluated in S. To obtain the tail (another stream), e_{tl} must be evaluated in S. According to SASL's semantics, the value of the head of the stream is only considered valid if the tail expression also produces a non- value.

2.6 Deforestation

Our language restrictions are closely related to Wadler's treelessness [147, 50, 98, 55]. Treeless programs manipulate tree-like data structures without creating intermediate data structures, so they can work on unbounded data structures in bounded space—the same goal as SASL.[8] A (restricted) treeless term [50] is of the form:

tt	$:=$ v	Variable
	$c\ tt_1\ \ldots\ tt_n$	Constructor
	$f\ v_1\ \ldots\ v_n$	Function applied to variables
	case v **of** $p_1 : tt_1\ \ \ldots\ \ p_n : tt_n$	Pattern-matching on variables

If the body of a function is a treeless term, it is treeless. The functions and pattern matches called by a treeless term must be treeless. A term made from composed treeless functions may itself be transformed to a treeless function by the "deforestation" algorithm. The variables must be used linearly.

SASL's syntax effectively provides the same restrictions, in a different form. While treeless terms restrict all variables, in SASL we only need to restrict streams, as these are the only values which may take up unbounded space. SASL streams are linear, as required. The only relevant constructor is CONS. The restriction that only variables may be used as parameters to functions and matches in a treeless form is equivalent to the stability constraint—that a stream may not be CONS'd onto in a recursive call.

SASL's restrictions may seem weaker, but turn out to be equivalent. A SASL function may CONS an item onto a stream that it passes to another function, as long as that function call is not recursive, since

[8]In general a stack may be needed, but when restricted to lists no stack is needed.

$$(\textsc{apply}\ )\ \frac{S, e \quad v' \qquad S[x \quad v'], e' \quad v}{S, f(e) \quad v}\ \text{where } \textbf{fun } f(x) = e'$$

$$(\textsc{constr-intro}\ )\ \frac{S, e_1 \quad v_1 \quad \ldots \quad S, e_k \quad v_k}{S, c(e_1, \ldots, e_k) \quad c(v_1, \ldots, v_k)}$$

$$(\textsc{tuple-intro}\ )\ \frac{S, e_1 \quad v_1 \quad \ldots \quad S, e_k \quad v_k}{S, (e_1, \ldots, e_k) \quad (v_1, \ldots, v_k)}$$

$$(\textsc{cons-intro}\ )\ \frac{}{S, e_1 {::} e_2 \quad [S, e_1, e_2]}$$

$$(\textsc{constr-elim}\ )\ \frac{S, e \quad c_i(\vec{v}) \qquad S[\vec{x}_i \quad \vec{v}], e_i \quad v}{S, \textbf{case } e \textbf{ of } c_1(\vec{x}_1) \quad e_1 \ \ldots \ c_k(\vec{x}_k) \quad e_k \quad v}$$

$$(\textsc{tuple-elim}\ )\ \frac{S, e_1 \quad (v_1, \ldots, v_k) \qquad S[\vec{x} \quad \vec{v}], e_2 \quad v}{S, \textbf{case } e_1 \textbf{ of } (\vec{x}) \quad e_2 \quad v}$$

$$(\textsc{cons-elim}\ )\ \frac{\left\{\begin{array}{l} S, e_1 \quad [S', e_{hd}, e_{tl}] \\ \quad S', e_{hd} \quad v_1 \\ \quad S', e_{tl} \quad v_2 \end{array}\right. \qquad S[x_1 \quad v_1, x_2 \quad v_2], e_2 \quad v}{S, \textbf{case } e_1 \textbf{ of } x_1 {::} x_2 \quad e_2 \quad v}$$

$$(\textsc{let}\ )\ \frac{S, e_1 \quad v' \qquad S[x \quad v'], e_2 \quad v}{S, \textbf{let } x = e_1 \textbf{ in } e_2 \quad v}$$

$$(\textsc{var}\ )\ \frac{}{S, x \quad S(x)}\ x \quad dom(S)$$

Figure 2.12: Big step transition relation for SASL

this can be statically unfolded to a treeless form. Matching on a stream inside a function's argument expression can be transformed to treeless form by syntactically pulling the match expression outside of the function call.

Wadler's work distinguishes between treeless forms, and compositions of treeless forms (where a tree is passed between the functions), which may be made treeless, leading to a two-tiered approach. SASL does not create such a distinction, so that functions may have bodies that compose other functions together, but still be treated as treeless, since they could be transformed into a treeless form (as long as none of the functions being composed are recursive calls). Recursive calls in tail positions cannot be composed with other stream-processing functions, as this would either violate the stability constraint, or mean the call was not a tail call.

In effect, SASL is a treeless language with a more user-friendly syntax. However, the compilation techniques used are very different. Wadler's deforestation algorithm totally removes the intermediate data structures, but may cause an exponential growth in code size. SASL relies on the lazy evaluation of streams to prevent unbounded data production; data are produced only as necessary, and the treeless-style limitations guarantee that unbounded storage will never be required.

2.7 A Comparison to SAFL+ and Synchronous Dataflow

We compare SASL with the statically-allocated functional language SAFL+, and the synchronous stream language Lustre. SAFL+ was chosen as the most similar existing language, while Lustre is another stream-processing language, albeit one that takes a rather different approach.

2.7.1 SAFL+

SAFL+ was introduced in Section 2.1.2. The main difference in approach is the use of CSP channels in SAFL+, versus the use of streams in SASL. CSP channels may connect processes in almost arbitrary ways, allowing the possibility of deadlock, while SASL's streams form an acyclic network of pipelines. While the programmer is expected to deal with the details of channels in SAFL+, these are hidden in SASL. Channels require the user to deal with explicit parallel processes, and serial and parallel composition, while this is handled implicitly in SASL.

The channels in SAFL+ mean that the language is not only not pure, but can also be non-deterministic. Basic SASL is a pure, deterministic language, although non-deterministic extensions are discussed in Chapter 7. Streams are not just syntactic sugar for channels; the demand driven nature of stream item production means that a single channel cannot be used to implement a stream. If lenient evaluation is used (as described in Section 5.2), the ability to cancel active computation is required, moving further away from a CSP-like model. Channels can, however, be used to implement the passing of non-stream values, as is done in the CSP synthesis of the next chapter.

Finally, many differences show up in the synthesis approaches, explained further in the next chapter. SAFL and SAFL+ use resource sharing and do not perform automatic pipelining, while SASL is intended for pipelined hardware, and does not share resources. SAFL+ defines synchronisation points in its evaluation, whereas SASL wishes to minimise synchronisation where this allows an increase in performance.

2.7.2 Lustre

Lustre was described in Section 1.2.2. The main difference between synchronous stream processing languages and SASL is that in the synchronous stream languages *all* variables are streams with explicit clocks, and all processing is done in terms of streams, whereas in SASL the stream processing is demand-driven, not pervasive (there are non-stream variables) and not explicitly clocked, helping modular design, where the programmer need not specify the exact timing of components and signalling between them.

(* Functions may be composed to create a pipeline of stream functions. *)
fun $\textit{compose}_1(i) = \textit{fold}_g(\textit{filter}(\textit{map}_f(\textit{from}(i))), 0)$

(* Such functions can be called repeatedly. *)
fun $\textit{compose}_2(i) =$ **if** $\textit{predicate}(i)$ **then** i **else** $\textit{compose}_2(\textit{compose}_1(i))$

(* And any function from scalar to scalar can be mapped over a stream. *)
fun $\textit{compose}_3(x\text{::}xs) = \textit{compose}_2(x)\text{::}\textit{compose}_3(xs)$

Figure 2.13: Examples of function composition in SASL

The demand-driven nature of SASL streams makes the $\textit{merge}_h$ (from Figure 2.11) function simple to write, while in Lustre it is necessary to organise the clocks so that values occur on both input streams simultaneously, perhaps through explicit back-pressure if either input stream is generated by an unbounded loop. Lustre uses a *clock calculus* to describe which streams may be merged, while in SASL any pair of streams may be merged (subject to linearity constraints). While a function like *dup-stream* is allowed in Lustre, linearity prevents it in SASL (although later we give an embedding of Lustre in SASL).

The pervasive use of streams in Lustre makes conventional programming with loops difficult. While iteration in SASL can be achieved through a tail-recursive function call, Lustre requires a data-flow program where elements of a stream represent iterations, and each step either performs an iteration of the loop, or resets the loop with new values should a new request come in. Results are sent out by sending a stream value out on the cycle representing the final iteration of the loop. Loop hardware therefore takes a stream of loop initialisation requests, and returns a stream of loop results. If the loop is unbounded, some form of back-pressure will be required to prevent new requests until the current loop has finished.

SASL achieves the same result directly, hiding the implementation of back-pressure and signalling. Functions may be composed without worrying about timing or signalling, as shown in Figure 2.13. Although synchronous stream languages may be more convenient for certain classes of problems (such as hard real-time systems with exact per-cycle requirements), SASL provides familiar features from conventional software languages, presenting a higher-level interface to the programmer. Control over the cycle-based timing of streams is lost, in exchange for more abstraction and more flexible synthesis, freeing the programmer from many details.[9]

To aid comparison, we will give an example of a Lustre program translated to SASL, and vice versa.

2.7.2.1 Conversion from Lustre to SASL

The Lustre program is taken from a Lustre paper[60]. It is shown in Figure 2.14. Each item in the *alarm* stream is the **and** of the appropriate elements from the *deadline* and *is-set* steams. The stream *is-set* is defined as being the same as *set* for the first item, and then is set if the corresponding *set* item is set, reset if the corresponding *reset* item is set, and otherwise takes on the value from the previous cycle.

The most direct translation to SASL puts all the Lustre parameter streams into one tupled SASL stream, giving the program shown in Figure 2.15. In this translation, each item in the stream of tuples represents the values held by Lustre streams on that clock. Since SASL streams are produced independently, and all Lustre streams are synchronised by a clock scheme, we convert all the Lustre streams into a single SASL stream, enforcing synchronisation. Lustre streams that use other clocks can be merged with the main clock by using hiatons (“nothing” elements) to fill the gaps. The *pre* construct is implemented by passing extra scalar parameters such as *prev-is-set*. The returned stream also passes back the

[9]It may be possible to extend SASL with “pragma”s or annotations to specify timing requirements for the synthesis tool—this is a possible further area of research.

```
node WatchDog(set, reset, deadline : bool) returns (alarm : bool);
var is-set : bool;
let
    alarm = deadline and is-set;
    is-set = set
        if set then true
        else if reset then false
        else pre(is-set);
tel.
```

Figure 2.14: Lustre WatchDog program

```
fun WatchDogInt(str, is-first, prev-is-set) =
    case str of (set, reset, deadline)::rest
        let is-set = if is-first then set else
            if set then True
            else if reset then False
            else prev-is-set in
        let alarm = and(deadline, is-set) in
        (alarm, set, reset, deadline)::WatchDogInt(rest, False, is-set)

fun WatchDog(stream) =
    WatchDogInt(stream, True, False)
```

Figure 2.15: SASL WatchDog program

data passed in, since the stream passed in cannot be used again due to linearity.

In general, a Lustre node that takes streams x_1 through x_n, returning streams y_1 through y_m can be embedded in SASL as a function of the form:

$$\textbf{fun}\ \textit{example}((x_1, \ldots, x_n)\mathbf{::}tl, \textit{state}) = (f_1(\ldots), \ldots, f_m(\ldots))\mathbf{::}\textit{example}(tl, g(\ldots))$$

where the f_i and g are SAFL-like combinatorial functions of the x_i and the non-stream variable *state*. Using this translation, Lustre programs can be converted relatively easily, if not elegantly, to SASL. For example, Lustre stream duplication can be represented in SASL by duplicating the stream element-wise:

$$\textbf{fun}\ \textit{dup-elt}(x\mathbf{::}xs) = (x, x)\mathbf{::}\textit{dup-elt}(xs)$$

(in contrast to the disallowed *dup-stream* function of Figure 2.11).

The above program does not show the advantages of SASL, as it is a Lustre example, and so avoids features that are tricky to implement in that language, such as loops. Although Lustre may simplify the expression of some simple synchronous dataflow tasks, SASL is a much more appropriate tool for situations involving loops and back pressure.

2.7.2.2 Conversion from SASL to Lustre

A simple example SASL program is shown in Figure 2.16. The function *newton-raphson*$_f$ is a simple Newton-Raphson root finder used to illustrate the use of iteration in SASL. The function

```
**fun** *newton-raphson*$_f(x) =$
    **let** $y = f(x)$ **in**
    **if** $abs(y) < \epsilon$ **then** $x$ **else** *newton-raphson*$_f(x \quad y/f\,'(x))$

**fun** *map-newton-raphson*$_f(i, str) =$
    **case** *str* **of** $x$**::**$xs$
        $(i,$ *newton-raphson*$_f(x))$**::***map-newton-raphson*$_f(i+1, xs)$
```

Figure 2.16: SASL example root-finding program

map-newton-raphson$_f$ applies the root finder to a sequence of initial values. The function will return a stream of roots, with the root returned depending on the associated item from the input stream. Each item in the result stream is tagged with an integer, to show the processing of dependencies between the stream items generated. We assume mathematical primitives, the number ϵ, the function f and its derivative f' are all supplied.

As Lustre has no built-in signalling mechanism, it has to be added explicitly to the Lustre implementation. Each item of data is accompanied by a signal on a request line. To keep it simple, pipelining is not used. Each function has an input request line and a result production line, and a new request may only be made after a result has been produced. The stream is controlled with a request line, a result line and a reset line. All signalling lines are level sensitive. The details of signalling for low-level implementations of SASL programs are discussed later in this thesis.

The implementation of the function *newton-raphson*$_f$ is shown in Figure 2.17. The functions f and f' are implemented by the nodes *F* and *FD* respectively. Explicit control logic is required, using the "*Req*" variables to time tokens.

The implementation of the function *map-newton-raphson*$_f$ is given in Figure 2.18. The variable *Active* marks whether the node has been activated and will respond to stream requests. The variable *State* holds the internal state, set when the function is called, and updated each time a stream item is returned.

When the function is initially called, it does little more than store its parameter and mark itself as active. When a request arrives on the output stream, a request is made on the input stream. When the result arrives, it is sent to a copy of the node performing *newton-raphson*$_f$, and the result is combined with the current state value and returned over the output stream. The internal state is then updated.

Compared to the original SASL version, all the complexity of control flow is exposed. The example here was simple SASL, yet produced rather complex Lustre, even though only unidirectional handshaking was used. In more complex programs it will quickly become extremely difficult to manage the control flow. SASL's main advantage over Lustre is that it allows the details of control flow to be hidden and automated in such programs while still enabling the processing of streamed data.

2.8 Summary

This chapter examined a number of possible functional I/O models, before introducing a naïve stream processing langauage. Programs that could not be statically allocated were demonstrated, before adding a set of restrictions which made the language statically-allocated. Semantics were briefly discussed and SASL's approach compared to that of deforestation. Finally, the language was compared with Lustre, using example programs.

This chapter has laid the groundwork for the rest of the thesis (most directly, the synthesis chapters, 3 and 4), by introducing both the basic SASL language, and the linearity and stability constraints that are used to statically allocate more advanced features later.

```
const ϵ : real.

node NewtonRaphson(Reset, ReqIn : bool; DataIn : real) returns (ReqOut : bool; DataOut : real);
var FReqIn,   FReqOut,   FDReqIn,   FDReqOut   : bool;
var FDataIn, FDataOut, FDDataIn, FDDataOut : real;
let
    assert true    not(ReqIn and pre(FDReqOut));

    FReqIn = false    ReqIn or pre(FDReqOut);
    FDataIn = 0.    if ReqIn then DataIn else
                     if pre(FDReqOut) then
                         pre(FDataIn)    pre(FDataOut)/pre(FDDataOut) else
                     pre(FDataIn);
    (FReqOut, FDataOut) = F(Reset, FReqIn, FDataIn);

    FDDataIn = FDataIn;
    FDReqIn = FReqOut and (abs(FDataOut)    ϵ);
    (FDReqOut, FDDataOut) = FD(Reset, FDReqIn, FDDataIn);

    ReqOut = FReqOut and (abs(FDataOut) < ϵ);
    DataOut = FDataOut;
tel.
```

Figure 2.17: Lustre version of *newton-raphson*$_f$

```
node MapNewtonRaphson(Reset, ReqIn, StrInRes, StrOutReq : bool; DataIn, StrInData : real)
    returns (ReqOut, StrInReq, StrOutRes : bool; (StrOutDataA : int; StrOutDataB : real));
var Active, NRReq : bool;
var State : int;
var NRData : real;
let
    Active = false    if Reset then false else
                       if ReqIn then true else
                       pre(Active);
    State = 0    (if ReqIn then DataIn else pre(State))+
                  (if pre(StrOutRes) then 1 else 0)
    ReqOut = ReqIn;

    StrInReq = StrOutReq and Active;

    (NRReq, NRData) = NewtonRaphson(Reset, StrInRes, StrInData);

    StrOutRes = NRReq;
    (StrOutDataA, StrOutDataB) = (State, NRData);
tel.
```

Figure 2.18: Lustre version of *map-newton-raphson*$_f$

i

CHAPTER 3

Translation to CSP

CSP [67] is used as the initial synthesis target for SASL, as it is a relatively high-level language that is designed for representing parallel systems. Channel communication provides a straightforward way to specify and implement many SASL features. CSP uses explicit parallelism, so the synthesis from SASL must extract parallelism from the original program. From a CSP-level description, translation to hardware is straightforward: the CSP programs can be rewritten into lower-level HDLs that use CSP channels, such as Handel-C [39], and from there synthesised to a final hardware implementation.

The synthesis approach is like conventional compilation in many ways. The syntax tree is linearised into a set of instructions. For software, these commands would be sequential instructions, while in this synthesis they make up interacting processes. Each expression is represented by a process that triggers the processes representing its sub-expressions. Where possible, sub-processes are run in parallel, making more efficient use of the parallelism available in hardware.

Section 3.1 is a general introduction to the aims of SASL compilation, providing a backdrop to both CSP synthesis and the graph-based compilation of Chapter 4. Section 3.2 then gives an abstract hardware-oriented view of SASL's calling conventions. Section 3.3 looks at synthesising bound variable access, and Section 3.4 covers the actual translation into CSP. Section 3.5 is a brief summary.

3.1 Synthesis Aims

Our synthesis goal is to convert function definitions into hardware resources that perform those functions. Despite targeting CSP, the final goal is hardware, so we take a hardware-like approach. Functions are converted into *modules* consisting of a set of CSP commands and channels, implementing the function. Following VHDL and Verilog, the translation uses "unshared" resources: in those languages, each time a module is used, a new piece of hardware (an *instance*) is created. This approach is taken by a number of HLS systems, such as Handel-C, but contrasts with SAFL's *resource-aware* approach, where hardware is shared through arbiters, and textual duplication is required to duplicate hardware. SASL modules can therefore be treated as "black boxes", as the inputs and outputs are defined solely by the function's parameters and return type, with implementation details hidden.

Pipelining: By creating multiple instances of modules, more parallelism is made available in SASL than was available in SAFL. SAFL expects only a single caller to be active in a function at a time, so that sharing resources does not necessarily limit performance. SASL aims to allow multiple outstanding function calls, pipelining requests.[1] In such a system sharing resources could lead to much resource contention. Since hardware resources in SASL are duplicated as needed, arbiters are not required to control access from multiple external call sites, as occurs in SAFL (tail recursive functions may receive calls from internal call sites, but simpler, specialised arbiters can be used in such cases).

SASL's aim to produce pipelineable circuits is also shown in the forms of analysis that are applied. SAFL performs register placement analysis, allowing it to eliminate registers in situations where a value is guaranteed to stay constant until required. As SASL allows pipelining, further requests could change values at any point, so SAFL's "permanising registers" (which are effectively pipeline latches) are always required in SASL. Pipelining is discussed further in Section 4.1.

Scheduling: The synthesised SASL programs schedule operations dynamically, as unbounded loops require some form of dynamic scheduling. Each basic operation starts when all the data it requires is available, rather than according to a pre-determined schedule. The basic scheduling described in this chapter and the next are fully dynamic. Section 5.1 discusses the static scheduling of parts of SASL programs, which is used to reduce the overhead of full dynamic scheduling, much like *soft scheduling* in SAFL.

SASL's synthesis tries to maximise parallelism, while SAFL inserts deliberate synchronisation points. For example, in a SAFL **let** binding, the value being bound must be completely evaluated before the body. This use of "let barriers" allows certain operations to be serialised, helping the user specify the scheduling of shared resources. No such feature is needed with SASL. The CSP synthesis retains this constraint, as it uses broadcast access to variables (see Section 3.3), but the alternative unicast-based graph synthesis of Chapter 4 avoids this serialisation.

SASL's feature set makes generating fixed low-level schedules difficult—the presence of possibly unbounded loops prevents static scheduling and the I/O mechanisms rely on a demand-driven request/acknowledge scheme. In software real-time systems the programmer generally knows nothing of exact instruction timings, or the optimisations performed by the compiler, but as long as the hardware is suitably powerful it can be used to perform real-time operations, despite the compiler not being targeted to the specific real-time constraints. Such an approach may also be taken with SASL.

Streams: SASL synthesises stream operations to reads and writes on demand-driven buses that are similar to CSP channels. Whereas CSP channels send the data as a request and return no data with the acknowledgement, the SASL bus sends a data-less request and returns data with the acknowledgement. A CONS operation becomes a bus write, while performing a pattern matching on a stream is implemented as a read from that bus. Dynamically, each bus has at most a single reader and writer, so that execution is deterministic. In our CSP synthesis, these buses are implemented as pairs of CSP channels—one CSP channel is used to transmit a request token, while the other is used to transmit results. All channels in the resulting CSP program have at most a single reader and a single writer dynamically, so that execution remains deterministic.

Primitives: The basic form of SASL does not include facilities for "primitive" operations that are implemented in another language. Such primitives are not necessary to implement any pure statically-allocated function, as these functions can be described directly in SASL. Calling functions that have state or perform I/O could be problematic, as the optimisation and evaluation models may rely on the pure functional nature of the language.

However, this does not stop the possibility of implementing *external linkage* (using the term from software). Provided the primitive acts in a pure functional style, a call to an external function would look just like a normal function call in the SASL source, with similar semantics. It would be synthesised to

[1] The simple CSP synthesis of this chapter does not allow pipelining, but graph synthesis (Chapter 4) does.

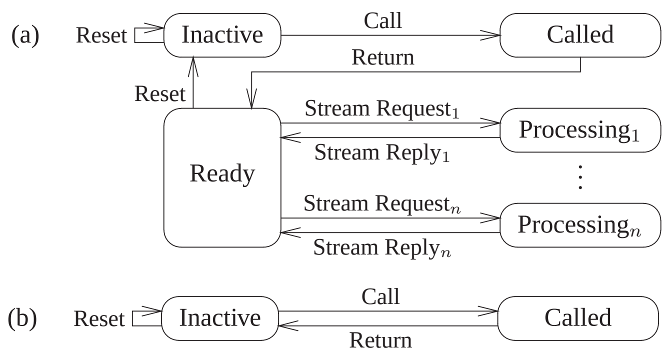


Figure 3.1: The function call state machine for (a) functions with streams and (b) stream-less functions

the instantiation of a module defined directly in, for example, Verilog. The primitive's physical interface would be highly dependent on the hardware "calling conventions" of the synthesis system, as would the signalling for the top-level calling interface.

3.2 Synthesis Outline and Function Interfacing

When synthesising SASL, a new instance is created for each non-recursive call site in the original program. Instantiation proceeds hierarchically, so that creating a new instance creates new instances of all modules it uses. Although this section describes the interface used for CSP, the ideas are broadly applicable to graph synthesis too.

The only I/O resources an instance requires are for passing in arguments and returning results—calls to sub-functions are hidden internally. In the approach taken here, this interface is split into:

A call/return mechanism for transferring basic types. This consists of an input channel to provide non-stream arguments and activate the instance, and an output channel to return non-stream results and signal that the instance is now quiescent and prepared for stream requests.

A set of *stream buses* representing input and output streams, to transfer streamed values sequentially, on demand. Stream buses consist of a pair of channels: a request channel, used to demand a new stream item, and a result channel, which returns the newly generated stream item. There may be multiple instances with read and write access to a single stream bus, but there is only ever at most one active reader and one active writer on any stream bus, due to the ordering imposed by the language—stream matches and CONSes occur in a fixed order.

A reset mechanism. This consists of a single channel which, when written to, resets the instance to a quiescent state, as described below.

A state machine representing the calling convention is shown in Figure 3.1(a). The left-hand states are "quiescent" states, and the right-hand states are "calculating" states. Input streams may only be read during a calculating state, and an instance may not move to a quiescent state while it has outstanding requests on its input streams.

The calling convention for an instance starts when a function call is requested—by sending basic type arguments (if any, or a unit datatype otherwise) to the non-stream input channel—causing a transition from the *Inactive* state to the *Called* state. In the *Inactive* state (including before the first call is initiated), the instance will not read from its input stream buses, and will not respond to requests on its output stream buses (it may receive and ignore such requests while inactive if it is sharing the stream bus with other CONS expressions). After the instance has moved to the *Called* state, the instance may read from its input streams, and the eager part of the function is evaluated before the basic-type return values are

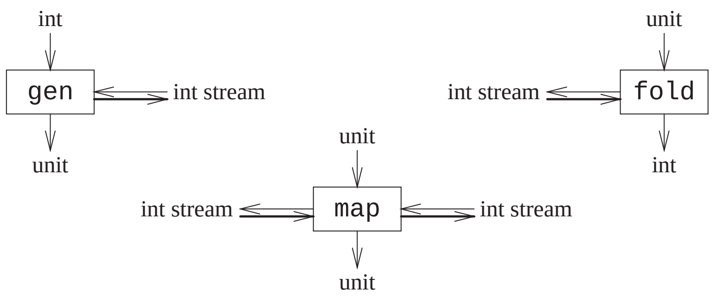


Figure 3.2: A "black box" view of the functions `gen` (*int*   *int stream*), `map` (*int stream*   *int stream*) and `fold` (*int stream*   *int*)

passed back over the non-stream output channel, causing a transition to the *Ready* state. Evaluation of lazy CONS expressions of the form $e::e'$ returns without evaluating e or e'. No stream items are returned at this stage, since stream items are generated lazily, produced only on demand, later in execution.

Matches on the streams passed in as arguments map to reads from the instance's input stream buses, and, after the hardware has signalled a return (i.e., it has entered the *Ready* state), matches on the returned streams become reads from the instance's output stream buses. An output stream may only be read when the associated stream-production hardware is quiescent; the hardware should be in the *Ready* state before any stream requests are sent, so that there are no outstanding function calls or incomplete stream requests. A read request on a stream causes the hardware to go into a *Processing* state, and it returns to the *Ready* state when the result is produced. There is one *Processing* state for each stream in the function's return type. A stream read request is implemented as a write to the stream bus's request channel. The instance generates the appropriate value for the current head of the stream, and returns it over the reply channel of the stream bus.[2]

A function instance may be called multiple times, inside a tail recursive loop, taking and returning different streams each time, so that a single stream bus may have multiple streams associated with it over the life of the program. However, only one stream may be active on a given bus at a given time. The stability constraint of Section 2.4.3 prevents a stream generated in one call from being accessible when another call to the same module occurs. Between calls to a function instance, the instance should be reset through a write to its reset channel, so that it loses any state associated with the streams it returned, going back to an "uncalled" state. The reset moves the instance from the *Ready* state to the *Inactive* state, whereupon a new call can generate new streams. A reset in the *Inactive* state does nothing. For a function that returns no streams, the *Inactive* state and the *Ready* state are the same, as shown in Figure 3.1(b). In such a case, the reset signal does nothing, and can be omitted, so that the interface reduces to SAFL's call model.

Synthesised programs do not contain any residual polymorphism; the top-level function must not have any type variables in its type, and for other functions the return type must be able to be inferred from the parameter type (for example, the function **fun** $f\ x = f\ x$ is disallowed). Hence, polymorphic values do not appear in the hardware, and no special treatment is required.

Interfaces for functions that generate, map and fold streams are shown in Figure 3.2. Vertical arrows represent basic value buses, providing call/return signals. Horizontal arrows represent stream buses (requests from right to left, results from left to right). The stream buses representing streams passed to or returned from the function are fixed channels, and do not need to be represented in the tuple of data transferred when calling or returning. If no basic types are sent or returned a unit type is used as a placeholder during the function call.

[2]In more detail, it evaluates both the head and tail of the CONS expression, and only returns the head value once both the head (the stream value) and tail (execution up to the next CONS expression) evaluation have completed.

3.3 Variable access

The CSP synthesis approach relies on converting functions into dataflow-like structures, whose nodes transfer intermediate results representing the results of sub-expressions over edges (which are implemented as channels). The implementation of variables in SASL has some subtleties. Non-stream variables can either be treated as being stored in some form of re-readable variable ("broadcast") or being transferred over edges like other intermediate values ("unicast"). These schemes, and their advantages and disadvantages, are described in the following sections. Stream variables have already been discussed informally, and their implementation is described in Section 3.3.3.

3.3.1 Broadcast variables

The "broadcast" approach is to store non-stream bound variables in a generally accessible place where they can be read as many times as is required. The location may only be overwritten when the previously held value goes out of scope. This is implemented in CSP with variables; in a direct hardware implementation, a register would be used. To implement **let** $x = e_1$ **in** e_2, e_1 would be evaluated, and the result stored before e_2 is evaluated. Any occurrence of x in e_2 becomes a CSP variable access. The **case** expressions would be dealt with similarly.

This is a relatively simple scheme, and is used in this CSP synthesis. The downside to this scheme is that e_2 cannot begin evaluation until e_1 has completed, and the function cannot be pipelined—x must be held constant until it goes out of scope, so only one call can be active in e_2 at a time. When a CONS expression is evaluated, the head must be evaluated before the tail, as the tail may contain a recursive call which overwrites variables that the head is accessing. These shortcomings can be addressed, at the expense of complexity, with unicast variables.

3.3.2 Unicast variables

An alternative which allows for more parallelism is to use channels to supply non-stream variable values. For example, in **let** $x = e_1$ **in** e_2, e_1 would be converted to a structure with its output channel representing the value x. The body expression e_2 would have an input channel representing x, and these two channels would be made identical. The two expressions can evaluate in parallel, and only need to synchronise when e_2 waits for x (rather like the *lenient evaluation* model [144]). Thus, more parallelism is available, and requests may be pipelined. Expressions involving **case** work similarly.

This model comes at some cost in complexity. Since variables are transfered over channels, and channel reads and writes must be matched up, variable access must be linear—when a program is run, any variable that goes out of scope should have been accessed exactly once (so that all variable storage is left empty once the evaluation has completed). This can be achieved by inserting expressions that explicitly copy variable values if a variable occurs more than once in an expression, and adding expressions to "use up" variables that are bound, but never accessed. Section 4.2.1 shows how to convert SASL to a linear form.

It is also necessary to make sure that even if a variable is not used we wait for its value-generating expression to terminate before returning. For example, in the **let** expression above, the whole expression should only terminate if both e_1 and e_2 complete evaluation, rather than just e_2, even if e_2 never uses x.

Although this variable usage model is more complex, it can allow the generation of more efficient hardware, and provides a more consistent synthesis model (all data is transfered over channels). We have not used this model for CSP synthesis, but it forms the basis of next chapter's graph synthesis.

3.3.3 Stream Variable Access

Streams are implemented using stream buses, as described in Section 3.2. Various details remain to be explained:

Ordering Stream Accesses: It is necessary to ensure that the reads or writes to a stream occur in order. This ordering is not enforced by the actual stream buses. To make reads (stream matches) occur in the correct order, basic-type values are used to to represent the streams. A unit value is returned by CONS expressions to signal that the stream is ready to be read from. Stream read implementations then return a unit value to show that that read has now occurred, so that the next one can take place. Passing these values around ensures read ordering dependencies are met. The write order dependencies are enforced by the implementation of CONS itself.

The Stream Activity Model: The stream bus model expects that if an expression e returns a stream, the circuitry associated with it will not listen for requests on the stream before e is evaluated, but will afterwards. In other words, each subexpression obeys the calling conventions for streams given in Section 3.2. The enclosing expression must ensure that no other circuitry is listening on the stream bus when it evaluates the expression e, so that there is only ever a single piece of circuitry waiting for stream requests on that stream.

By analysing the syntax tree, it can be shown whether the property holds for all sub-expressions of a function. For example:

> In a **let**, if the body expression does not allow a CONS to listen on the returned stream bus before the expression is evaluated, and does afterwards, the overall expression will have the same property.
>
> CONS expressions of the form e_1**::**e_2 are synthesised so that the stream bus returned by e_2 is also returned by the CONS expression. The implementation of the CONS expression will not listen on the stream bus before it is evaluated, so it will keep to the model as long as e_2 does too. Once evaluated, it will listen for a request, but not evaluate e_2, so that there is only a single active listener. When a request arrives it stops listening and evaluates e_1 and e_2, so that e_2 will manage listening for requests on the stream bus.
>
> Stream matches work by evaluating the expression that is being matched upon, to ensure that there is some circuitry listening for a request on that stream bus, and only then performing the request to read the item.
>
> Variable access expressions need to be carefully implemented to meet this model (see below).

Stream Variables: A naïve implementation of variable access does not meet the above requirements. For example, in the expression **let** $s = f()$ **in** x**::**s, s is bound to a stream which is activated in the function call to f—a CONS expression in f will be waiting for stream requests when f returns. The expression x**::**s would then be evaluated and wait for a request on the same stream bus, expecting the (as yet unexecuted) tail expression to not yet be waiting for requests on the bus. Two CONS expressions would be simultaneously waiting for requests on the same stream bus, causing an error.

A solution to this problem is to distinguish between the stream bus S associated with the variable itself (representing the stream returned by the binding expression) and the bus T associated with the variable occurrence. In the above example, evaluation of $f()$ would cause S to be listened on, but not T. The evaluation of x**::**s would then cause T to be listened on. When a stream read occurs on T, x is returned and the variable access expression evaluated. Further requests on T should return items from S. To make this occur, the variable access expression starts a *forwarding* process that repeatedly waits for a request on T, performs a request on S, receives the item from S and returns it on T.

Eliminating Unnecessary Forwarding: Forwarding all streams at every variable access is inefficient and complicates tail recursive calls. However, variable occurrences with types containing non-$\star$ stream identifiers may be treated differently. The type system guarantees that streams with non-$\star$ stream identifiers are not CONS'd upon or merged with other streams. This means it is not necessary to forward such

```
(* (a) A function that needs to forward stream items. *)
fun select(test, stream1, stream2) = if test then stream1 else stream2
(* (b) A function that does not need to forward streams. *)
fun select2(test, stream) = if test then stream else case stream of x::xs    xs
(* (c) If modifying a stream with CONS, forwarding is required. *)
fun f1(x) = let stream = g() in x::stream
(* (d) Transforming program (c) eliminates the need for forwarding. *)
fun f2(x) = x::g()
```

Figure 3.3: Example functions that may need stream forwarding

streams in a variable access expression, as there will never be a situation where it is possible to have two CONS expressions simultaneously waiting on the associated stream bus.

A simple optimisation is to only forward streams that have a stream identifier of $\star$ in the variable occurrence's type (remember that the typing rules allow this to differ from the stream identifier in the type of the variable itself, due to the $\star$-substitution). The stability constraint requires that streams used in recursive tail calls have non-$\star$ stream identifiers, so that the streams passed into recursive calls are never forwarded, simplifying synthesis.

Examples: A selection of functions are given in Figure 3.3:

Example (a) may return either *stream1* or *stream2* conditionally, using the same stream bus. Since both binding expressions are evaluated before *select* is called, there would be two CONS expressions waiting for requests on the same stream bus, if forwarding were not used. To type the function correctly both variable access expressions must use $\star$-substitutions on the stream identifiers, so both streams are forwarded to the returned stream bus.

Example (b) conditionally selects between the stream given as an argument, and the same stream with an item read from it. The same stream bus can be used to return the result stream in both cases, without the possibility of multiple CONS expressions waiting for requests on the bus simultaneously. The variable occurrences do not need $\star$-substitutions in order to be typed, and so the streams need not be forwarded.

Example (c) calls $g()$ to create a stream, and then tries to CONS an item onto this stream. The stream must be forwarded, as the stream bus returned by $g()$ will already be active when the CONS expression is evaluated. The variable *stream* has a $\star$ stream identifier associated with it, as it is not dependent on a parameter stream, and so the variable occurrence returns a stream with a $\star$ stream identifier, causing forwarding.

Example (d) shows how the forwarding process can be eliminated by removing the variable. Since no variable is used, no forwarding is required. The function g is then lazily evaluated, and not called until after the CONS expression has responded to a request, so that the CONS expression and $g()$ can safely share a stream bus. The analysis of this chapter cannot identify that introducing a forwarder is unnecessary, but this extraneous forwarding is eliminated in the next chapter (where we distinguish between the streams currently marked $\star$).

3.4 CSP Synthesis

Our synthesis uses a form of CSP based on that in Hoare's CACM paper [67], extended with finite algebraic datatypes and tuples; we assume that tuple and constructor primitives are available, and that matching can be performed with guarded expressions. A function call becomes a write of scalar arguments on one channel, followed by a read of the result on another. Our approach was influenced by Abdallah [1]. The functions are transformed to CSP using a syntax-directed translation, and the final program constructed by composing the translated functions in parallel.

In contrast to SAFL, SASL functions must hold some state describing streams (indeed, this is part of the rationale for using lazy lists—they provide a way of dealing with state for I/O and so on). A function that returns a stream holds the information required to generate further items from that stream. Before a function is called again, the hardware that produces stream values must be reset to the *Inactive* state, as described in Section 3.2. This is done by giving each expression a "reset" channel that is written to before the proper function call is performed.

The details of synthesising non-stream elements are discussed in Section 3.4.1, and the implementation of streams is discussed later in Section 3.4.2. Before synthesis starts, the program must be unfolded so that each function has at most a single external call site (to avoid the need for arbitration), and each variable is given a unique name (to avoid scoping issues).

3.4.1 Non-stream CSP Synthesis

Each subexpression has two value-less input channels; one starts evaluation of the expression, the other performs the reset as described above. An output channel is used to return all the scalar results of the expression. All subexpressions are translated as shown in Figures 3.4(a) and 3.4(b), and the resulting commands are composed together in parallel. Each time the rules are applied, fresh names are created for the channels and temporary variables. The following channels are the exception to this:

The E^{in}, E^{out} and E^{reset} channels.

The e^{out} channel of a function definition **fun** $f\ x = e$.

The $e_1^{out}, \ldots, e_k^{out}$ channels of a constructor matching **case** e_0 **of** $c_1 \ldots \quad e\ _1 \ \ldots\ c_k \ldots \quad e\ _k$.

The e_1^{out} channel of a tuple matching **case** e_0 **of** $(x_1, \ldots, x_k) \quad e\ _1$.

The e_1^{out} channel of a **let** expression **let** $x = e_0$ **in** e_1.

The E^{in}, E^{out} and E^{reset} channels are made identical to the appropriate sub-expression channels of the enclosing expression, to allow communication between the sub-expression and enclosing expression. The other channels in the list are made identical to the E^{out} channel of that expression (or f^{out} for a function definition). This ensures that the result of the sub-expression is passed out directly, and no commands are generated to read from that channel. For expressions in tail-recursive contexts, the output channel of the tail position sub-expressions must not be made separate, since if a tail call occurs no results will be passed out.

SASL variables are implemented using the "broadcast" model of Section 3.3.1, with the values stored in CSP variables. These variables are not scoped, so variable renaming is required to ensure unique names. SASL variable names are bound in **case** matches and **let** expressions by evaluating the expression producing the variables, assigning the results to CSP variables, and then evaluating the body expression. The same CSP variables are used to store the SASL variables from different iterations of the same function instance, so the translation must be designed to avoid race conditions where CSP variables are overwritten when the old values may still be required.

Two forms of function application are given. Non-recursive call sites expect a result to be returned, which is then passed out on the *application's* output channel. Tail recursive call sites expect no result to be passed back locally, with the result instead being passed back directly through the *function's* output channel.

When a non-recursive function call expression is reset, it simply resets the function it calls, and the argument expression, as if the function being called (which is unshared) had been inlined. Recursive

(a) Function translation—see Section 3.4.1

Original function, f	Translated functions (with channels f^{in}, f^{out} and f^{reset})
fun $f\ x$ = e	$[f^{in}?x \quad e\ ^{in}!()$ $[\!]f^{reset}?() \quad e\ ^{reset}!()]$

(b) Non-stream expression translation—see Section 3.4.1

Original expression, E	Translated expression (with channels E^{in}, E^{out} and E^{reset})
$f\,e$ (non-recursive)	$[E^{in}?() \quad e\ ^{in}!(); e^{out}?t; f^{in}!t; f^{out}?t; E^{out}!t$ $[\!]E^{reset}?() \quad f\ ^{reset}!()\ \ e^{reset}!()]$
$f\,e$ (recursive)	$[E^{in}?() \quad e\ ^{in}!(); e^{out}?t; [e^{reset}!()\ \ f^{reset}!()]; E^{reset}?(); f^{in}!t$ $[\!]E^{reset}?() \quad e\ ^{reset}!()]$
$c(e_1, \ldots, e_k)$	$[E^{in}?() \quad [e\ _1^{in}!() \ \ \ldots \ \ e_k^{in}!()];$ $[e_1^{out}?t_1 \ \ \ldots \ \ e_k^{out}?t_k];$ $E^{out}!c(t_1, \ldots, t_k)$ $[\!]E^{reset}?() \quad e\ _1^{reset}!() \ \ \ldots \ \ e_k^{reset}!()]$
$(e_1, \ldots, e_k)$	$[E^{in}?() \quad [e\ _1^{in}!() \ \ \ldots \ \ e_k^{in}!()];$ $[e_1^{out}?t_1 \ \ \ldots \ \ e_k^{out}?t_k];$ $E^{out}!(t_1, \ldots, t_k)$ $[\!]E^{reset}?() \quad e\ _1^{reset}!() \ \ \ldots \ \ e_k^{reset}!()]$
case e_0 **of** $c_1(\ldots) \quad e\ _1$ $\ldots$ $c_k(\ldots) \quad e\ _k$	$[E^{in}?() \quad e\ _0^{in}!(); [e_0^{out}?c_1(\ldots) \quad e\ _1^{in}!()$ $[\!]\ldots$ $[\!]e_0^{out}?c_k(\ldots) \quad e\ _k^{in}!()]$ $[\!]E^{reset}?() \quad e\ _0^{reset}!() \ \ \ldots \ \ e_k^{reset}!()]$
case e_0 **of** $(x_1, \ldots, x_k) \quad e\ _1$	$[E^{in}?() \quad e\ _0^{in}!(); e_0^{out}?(x_1, \ldots, x_k); e_1^{in}!()$ $[\!]E^{reset}?() \quad e\ _0^{reset}!()\ \ e_1^{reset}!()]$
let x = e_0 **in** e_1	$[E^{in}?() \quad e\ _0^{in}!(); e_0^{out}?x; e_1^{in}!()$ $[\!]E^{reset}?() \quad e\ _0^{reset}!()\ \ e_1^{reset}!()]$
x	$[E^{in}?() \quad E\ ^{out}!x$ $[\!]E^{reset}?() \quad$ **skip**$]$

(c) Stream expression translation—see Section 3.4.2

Original expression, E	Translated expression (with channels E^{in}, E^{out} and E^{reset})
e_1**::**e_2 $e_2 : \tau\ \mathit{stream}_\star^S$ $E : \tau\ \mathit{stream}_\star^S$	$[E^{in}?() \quad E\ ^{out}!(); [S^{req}?() \quad e\ _1^{in}!()$ $[\!]E^{reset}?() \quad e\ _1^{reset}!()\ \ e_2^{reset}!()]$ $[\!]E^{reset}?() \quad e\ _1^{reset}!()\ \ e_2^{reset}!()]$ $[e_1^{out}?t \quad e\ _2^{in}!(); e_2^{out}?(); S^{ack}!t]$
case e_0 **of** x_1**::**$x_2 \quad e\ _1$ $e_0 : \tau\ \mathit{stream}_i^S$ $x_2 : \tau\ \mathit{stream}_i^S$	$[E^{in}?() \quad e\ _0^{in}!(); e_0^{out}?(); S^{req}!(); S^{ack}?x_1; x_2 = (); e_1^{in}!()$ $[\!]E^{reset}?() \quad e\ _1^{reset}!()\ \ e_2^{reset}!()]$
x $x : \sigma_1$ $E : \sigma_2$	$[E^{in}?() \quad \textsc{start}(\sigma\ _1, \sigma_2); E^{out}!x$ $[\!]E^{reset}?() \quad \textsc{stop}(\sigma\ _1, \sigma_2)] \quad \textsc{forward}(\sigma_1, \sigma_2)$

Figure 3.4: Syntax-directed translation to CSP

calls are more complicated. To keep the reset signalling graph acyclic, when a recursive call site is reset, it only resets its argument sub-expression. When the recursive function call itself occurs the function call expression resets the enclosing function it is calling. Since the call site is a subexpression of the function it is resetting, it must expect a reset signal and not propagate it, in order to prevent an infinite loop. Once the function has been reset, the actual function call is initiated.

3.4.2 Stream CSP Synthesis

Streams are represented using a combination of stream buses to pass actual stream values, and unit values that represent the streams on the scalar side, in order to ensure the stream reads and writes occur in order.

Stream buses are associated with stream variables using type annotations, in a similar way to how the stream identifiers are associated with streams. A stream with stream identifier i associated with the stream bus S is given a type of the form $\tau\ stream_i^S$. A distinct stream bus is created for each stream parameter. For typing, stream buses are treated in the same way as stream identifiers, except that when a variable access expression returns a stream identifier of $\star$ a different stream bus is assigned to the returned stream.[3] New stream buses are assigned where possible, but if two stream buses are joined together at the end of a conditional expression the buses are required to match by the type system, since the results should be written to the same stream bus, regardless of execution path taken (this constraint can be met using a unification-based typing system).

Figure 3.4(c) shows the translation relating to stream expressions, with a replacement translation of the variable access expression. The types of expressions are also given, as the stream buses used are held in the types. Each stream bus S is compiled to two channels: a unit request channel S^{req} and a result channel S^{ack}. Again, new names are created for the channels and temporary variables, with the exception of the output channels of sub-expressions that do occur in tail contexts:

> The e_2^{out} channel of a CONS expression e_1**::**e_2.
>
> The e_1^{out} channel of CONS-matching form **case** e_0 **of** x_1**::**x_2 $\quad e\ _1$.

These channels are made identical to the expression's E^{out}.

In the CONS translation rule, an incoming request returns immediately, setting up a process that waits for a request. If a request arrives, it triggers processing of the head expression, while if it receives a reset, it returns to an inactive state. When the head finishes computing, it evaluates the tail, and when the tail finishes evaluating the result of the head expression is passed out on the stream's result channel. It is necessary to make the stream buses for e_1^{out} and E^{out} identical so that if the tail expression causes a recursive function call the completion of the call is detected, signalling that the stream item may be returned.

The CONS-matching rule works by evaluating the stream-producing expression e_0, performing a stream request, storing the read item in the variable x_1, and activating the body expression e_1.

The variable rule in Figure 3.4(c) generates stream forwarding commands using the functions START, STOP and FORWARD, defined in Figure 3.5. These rules are complicated by the fact that a variable may be a tuple of stream and non-stream types. The rules simply break down the variable's type, extracting the stream buses of streams that are to be forwarded. For each stream with a stream identifier of $\star$ in the type of the variable occurrence:

> The FORWARD function generates a process that forwards items when an item is received on the channel $F_{on}^{S,T}$. It will continue until it receives an item on the channel $F_{\mathit{off}}^{S,T}$.
>
> The START function generates a command to start the copying process when the variable access expression is evaluated, by sending an item to $F_{on}^{S,T}$.

[3] For the CONS rule the stream bus of the whole expression is the same as the stream bus of the tail expression.

$$\begin{array}{rcl}
\textsc{start}(\sigma_1^1 \quad \ldots \quad \sigma\ {}_n^1, \sigma_1^2 \quad \ldots \quad \sigma\ {}_n^2) & = & \textsc{start}(\sigma_1^1, \sigma_1^2) \ \ldots \ \textsc{start}(\sigma_n^1, \sigma_n^2) \\
\textsc{start}(\tau, \tau) & = & \textbf{skip} \\
\textsc{start}(\tau\ \mathit{stream}_i^S, \tau\ \mathit{stream}_j^S) & = & \begin{cases} F_{on}^{S,T}!() & : \text{if } j = \star \\ \textbf{skip} & : \text{otherwise} \end{cases}
\end{array}$$

$$\begin{array}{rcl}
\textsc{stop}(\sigma_1^1 \quad \ldots \quad \sigma\ {}_n^1, \sigma_1^2 \quad \ldots \quad \sigma\ {}_n^2) & = & \textsc{stop}(\sigma_1^1, \sigma_1^2) \ \ldots \ \textsc{stop}(\sigma_n^1, \sigma_n^2) \\
\textsc{stop}(\tau, \tau) & = & \textbf{skip} \\
\textsc{stop}(\tau\ \mathit{stream}_i^S, \tau\ \mathit{stream}_j^S) & = & \begin{cases} F_{\mathit{off}}^{S,T}!() & : \text{if } j = \star \\ \textbf{skip} & : \text{otherwise} \end{cases}
\end{array}$$

$$\begin{array}{rcl}
\textsc{forward}(\sigma_1^1 \quad \ldots \quad \sigma\ {}_n^1, \sigma_1^2 \quad \ldots \quad \sigma\ {}_n^2) & = & \textsc{forward}(\sigma_1^1, \sigma_1^2) \ \ldots \ \textsc{forward}(\sigma_n^1, \sigma_n^2) \\
\textsc{forward}(\tau, \tau) & = & \textbf{skip} \\
\textsc{forward}(\tau\ \mathit{stream}_i^S, \tau\ \mathit{stream}_j^S) & = & \begin{cases} \textsc{forward-stream}(S, T) & : \text{if } j = \star \\ \textbf{skip} & : \text{otherwise} \end{cases}
\end{array}$$

---

where $\textsc{forward-stream}(S,T) =$

$$\begin{array}{l}
[F_{on}^{S,T}?() \quad \mathit{active}\ ^{S,T} := \mathit{true}; \quad [\mathit{active}^{S,T} = \mathit{true}; T^{req}?() \quad S\ ^{req}!(); S^{ack}?t; T^{ack}!t \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad [\![\mathit{active}^{S,T} = \mathit{true}; F_{\mathit{off}}^{S,T}?() \quad \mathit{active}\ ^{S,T} := \mathit{false}] \\
[\![F_{\mathit{off}}^{S,T}?() \quad \textbf{skip}]
\end{array}$$

and $\mathit{active}^{S,T}$ is a new CSP variable

Figure 3.5: Functions to generate the stream-forwarding commands

The STOP function generates a command to stop the copying process when the expression is reset, by sending an item on $F_{\mathit{off}}^{S,T}$.

Between them, these functions generate all the commands required to forward items from S to T (if the stream bus S is forwarded to T in multiple variable occurrences, a single instance of the forwarder can be shared). Streams with non-$\star$ stream identifiers in the expression's type are not forwarded, as the original stream can be used directly instead.

Example: An example program that inverts the elements of a stream, with its translation to CSP, is shown in Figure 3.6. It is annotated with numbered sub-expressions. The input and output buses are I and O respectively. As the variable occurrences of str and xs both have types whose stream identifier is non-$\star$, the streams do not need to be forwarded.

3.5 Summary

This chapter has introduced a simple synthesis to CSP, based on the "broadcast" variable model. It provides an introduction to the synthesis of the next chapter, which, while more complex, still relies on the same syntax-directed approach of creating communicating sub-processes, although in the next chapter these will be represented as graph nodes. This chapter introduces the use of stream buses, which will be refined in the next chapter so that less stream forwarding is required.

This chapter also introduced the "unicast" variable access model which will be used by graph synthesis. Graph synthesis adds a further layer of complexity by allowing the pipelining of requests.

(* Map a stream. *)
fun $mapnot(str) = ($**case** str_2 **of** x**::**$xs \quad ((not\ x\ \ {}_5)_4$**::**$(mapnot\ xs_7)_6)_3)_1$

(* The function implemented in CSP. *)
$[mapnot^{in}?str \quad e\ {}^{in}_{1} \mathbin{[\!]} mapnot^{reset}?() \quad e\ {}^{reset}_{1}]$
$[e^{in}_1?() \quad e\ {}^{in}_{2}!(); e^{out}_2?(); I^{req}!(); I^{ack}?x; xs := (); e^{in}_3!() \mathbin{[\!]} e^{reset}_1 \quad e\ {}^{reset}_{2}!() \quad e^{reset}_3!()]$
$[e^{in}_2?() \quad e\ {}^{out}_{2}!() \mathbin{[\!]} e^{reset}_2?() \quad$ **skip**$]$
$[e^{in}_3?() \quad mapnot\ \ {}^{out}!(); [O^{req}?() \quad e\ {}^{in}_{4}!() \mathbin{[\!]} e^{reset}_3 \quad e\ {}^{reset}_{4}!() \quad e^{reset}_6]$
$\mathbin{[\!]} e^{reset}_3 \quad e\ {}^{reset}_{4}!() \quad e^{reset}_6]$
$[e^{out}_4?t_1 \quad e\ {}^{in}_{6}!(); mapnot^{out}?(); O^{ack}!t_1]$
$[e^{in}_4?() \quad e\ {}^{in}_{5}!(); e^{out}_5?t_2; not^{in}!t_2; not^{out}?t_4; e^{out}_4!t_4 \mathbin{[\!]} e^{reset}_4 \quad not\ \ {}^{reset}!() \quad e^{reset}_5!()]$
$[e^{in}_5?() \quad e\ {}^{out}_{5}!x \mathbin{[\!]} e^{reset}_5?() \quad$ **skip**$]$
$[e^{in}_6 \quad e\ {}^{in}_{7}!(); e^{out}_7?t_3; [e^{reset}_7!() \quad mapnot^{reset}!()]; e^{reset}_6?(); mapnot^{in}!t_3$
$\mathbin{[\!]} e^{reset}_6?() \quad e\ {}^{reset}_{7}!()]$
$[e^{in}_7 \quad e\ {}^{out}_{7}!() \mathbin{[\!]} e^{reset}_7 \quad$ **skip**$]$

Figure 3.6: A stream function and its CSP translation

CHAPTER 4

Dataflow Graph Translation

The previous chapter deals with the translation of SASL to CSP. There are a number of weaknesses to that approach:

The broadcast variable access model (SASL variables become CSP variables) limits parallelism. Unicast variable access (SASL variables become CSP channels) could be used in CSP synthesis, at some cost to circuit complexity.

Certain SASL features are poorly matched by their CSP implementation. For example, two CSP channels are required to implement each SASL stream, providing far more low-level synchronisation than is necessary.

Non-trivial transformations are difficult, as they must be performed on either the original syntax tree, or the final CSP (which lacks high-level structure).

Certain optimisations cannot be implemented in CSP, such as lenient evaluation (see Section 5.2).

Given these limitations, we would like to transform the SASL programs to an intermediate format that suits the details of SASL, aids optimisation, and is easy to synthesise to RTL. The approach we have taken is that of dataflow graphs. Dataflow graphs can be produced from SASL relatively simply, and the hardware implementation is also quite straightforward: each node in the graph is represented by an instance of a hardware module, and the graph's edges become hardware connections (some simple example nodes are given in Appendix A, and some node schematics are distributed throughout this chapter). The dataflow graphs provide great flexibility for optimisation.

The dataflow graphs use request/acknowledge signalling to allow back-pressure. Each edge acts like a synchronous CSP channel, with a single static reader and writer. For graphs with many nodes, this can lead to a large synchronisation overhead (a common complaint against data-driven asynchronous circuits), and Section 5.1 discusses the uses of static scheduling to reduce this overhead where possible. The basic graph model shares much with Buck's token flow model [26].

Although translating the non-stream aspects of the language is relatively simple, synthesising streams is rather more complex. Each stream in the program is translated into a hardware resource called a *stream bus*, which is accessed by all nodes which read or write to that stream.

The dataflow graphs are intended to be synthesised into hardware which can deal with pipelined requests. This differs from SAFL, which deals with at most a single outstanding call. Most functional *structural* HDLs can create pipelined hardware and systolic arrays, but at the expense of support for high-level control structures such as loops. SASL *behaviourally* supports both control flow structures, and pipelined, systolic-style implementations. Issues surrounding pipelining SASL programs are examined in Section 4.1.

Despite the discussion above of "dataflow graphs", the synthesis approach in this chapter uses three graph styles, which are broadly similar, but gradually replace the higher-level (source-oriented) features with low-level (target-oriented) features:

Section 4.2 covers the conversion of SASL to a simple dataflow graph based on linear types to provide unicast variable signalling.

Section 4.3 removes tail calls from these graphs, by introducing iteration node types.

Section 4.4 brings the hardware implementation of streams into the the dataflow graph, producing graphs from which hardware may be constructed.

4.1 Pipelining SASL

The call model presented in Section 3.2 deals simply with a single caller at a time, as shown in Figure 4.1(a). SASL is intended to allow pipelined requests, and this requires a new calling convention. The call/return/stream-request/reset model (see Figure 3.1) provides a basic model for stream requests, assuming a single outstanding call. This model is extended here to allow multiple outstanding calls, to increase parallelism.

Basic Pipelining: For functions that do not take streams as parameters or return them, the calling convention for the synthesised modules is simple. Requests may be sent into an instance as fast as back-pressure allows (that is, as long as the "ready for input" line is set). Results are then produced in order, so that the n^{th} request leads to the n^{th} result, as shown in Figure 4.1(b). If one of the requests leads to non-termination, all further results are blocked.

Finer Granularity I/O: The basic calling convention expects a single argument value, and a single result value. The argument and return value are each sent as an atomic unit. If multiple values are used as parameters or results, they are collected together using tuples. This may unnecessarily restrict parallelism by enforcing synchronisation between the items where none is required. The optimisation of Section 5.3.1 deals with eliminating unnecessary tupling.

The pipelining model can be extended to deal with multiple inputs and multiple outputs by creating *logical* tuples. The n^{th} input tokens on each of the input edges are collected together to form the logical n^{th} parameter (and similarly for output edges). This model can be used for pipelining general graphs (rather than just single-in single-out functions).

One thing to note is that the multiple parameter and return elements are grouped by their arrival number on that edge, rather than the exact timing, just as the n^{th} output matches the n^{th} input, independent of how many more inputs were received before the output was produced. For example, two values may arrive on input A before the first on input B, but the value on B is still associated with the first value to arrive on A.

Pipelining Streams: The CSP reset model for streams cannot be used directly with a pipelined system. A reset now must not reset all processing involved in stream production, but instead only reset the set of streams currently being output, at which point the streams associated with the next call to the function may be accessed. In keeping with the finer-grain I/O model, a separate stream reset is provided for each stream returned, rather than a single line that resets all streams.

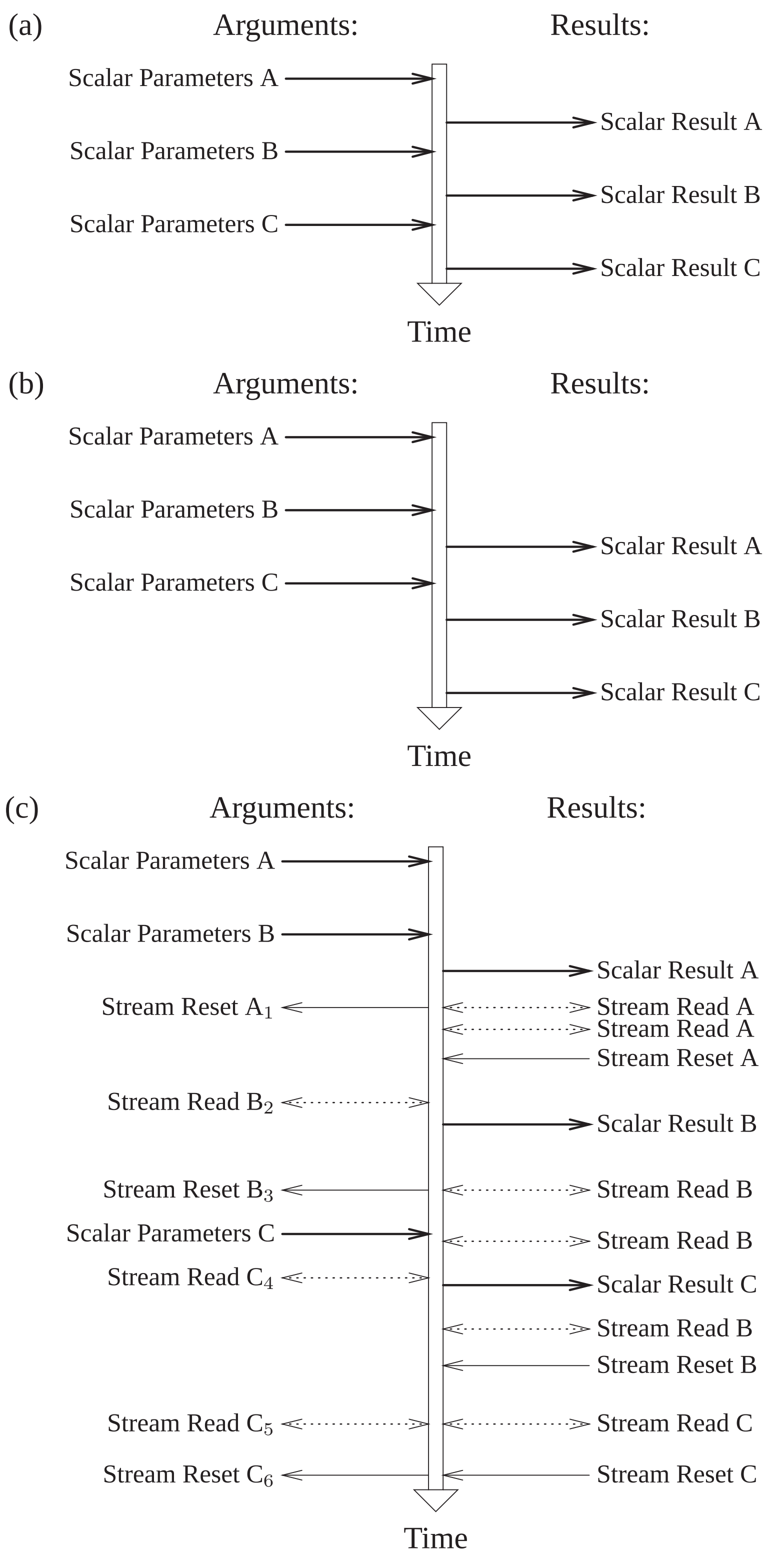


Figure 4.1: Example call sequence for (a) unpipelined access, (b) pipelined access with scalars, (c) pipelined access with streams.

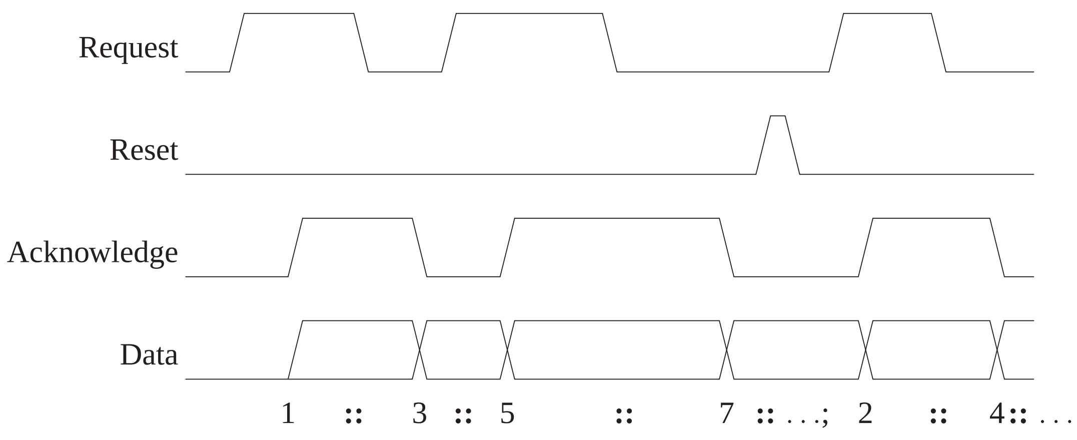


Figure 4.2: Example bus encoding of a stream.

An example calling sequence is shown in Figure 4.1(c).[1] The function has a single stream parameter and result, as well as scalar arguments and results. Stream access occurs as follows:

The stream values between the $(n \quad 1)^{\text{th}}$ reset and n^{th} reset on a particular stream bus are associated with the n^{th} set of scalar parameter and return values.

The n^{th} output stream may only be read from after the n^{th} scalar result that represents that stream is produced. This stream may be read between the stream reset of the previous stream on this bus, and this stream's reset.

The n^{th} input stream will not be read from until the associated n^{th} input token has been received. Stream reads and resets may be triggered as part of producing the scalar results (reads 2 and 4), or be triggered by the production of a stream item (read 5). Stream resets can be triggered when the stream stops being live, either during scalar result production, when a stream read occurs (resets 1 and 3), or when the output stream that uses it is reset (reset 6).

Requests for stream items are not pipelined. However, this need not limit parallelism, as the actual production of stream items can be pipelined (see Section 5.2.5), and access to stream items made through a FIFO, partially decoupling stream writing and reading.

A simple example of how the streamed values may be sent over a set of wires is shown in Figure 4.2. Two-phase signalling is used for all wires except reset. A request leads to an acknowledgement and a value being produced together at some later point. A reset drops the rest of the stream, so that the next value produced is the first item of the next stream to be transferred on that bus. For a synchronous implementation all edges are expected to coincide with an underlying clock, but the exact number of cycles between request and response can vary.

4.2 Dataflow Graph Generation

The initial intermediate format is a simple dataflow graph, where edges represent inputs, outputs and intermediate results, and vertices represent data processing operations. Eventually the vertices are implemented using standard RTL modules, and the edges are implemented as synchronised channels.

The graphs are built up from the node types shown in Figure 4.3. The graphs are created by connecting together basic nodes, call nodes, CONS constructs and conditional constructs. Basic nodes and call nodes

[1]The graph is rather simplified, showing scalar functions as taking an appreciable amount of time to process (thus making pipelining worthwhile), while presenting stream reads as instantaneous, in order to keep the diagram from becoming too large. This may well not be the case in practice.

are like the terminal symbols of a grammar, while CONS nodes and conditional nodes are like non-terminals, containing (finitely) nested subgraphs. An example dataflow graph is shown in Figure 4.9, which will be used as a translation example later.

In this section, entire streams are represented as single tokens, just like those used for scalar values. The CONS node produces a stream token, and when the stream token is matched it invokes a subgraph in order to obtain the head item and a stream token representing the next stream item. A stream is read using a stream-matching normal node that takes a stream token, and returns the head token and the tail token produced by the CONS subgraph. For the moment, the mechanism for transferring data between the CONS node and stream-reading node is hidden. The graph representation of Section 4.4 will explicitly split the stream into a unit (data-less) token used to signal that the stream is ready, and a stream bus to connect the producer to the consumer in order to transfer the actual data.

The basic form of SASL cannot be directly converted to the dataflow graph representation. SASL variables may be used more than once, or not at all, while dataflow graph edges expect one read per write (the variables must be linear). While the CSP synthesis of Chapter 3 used a "broadcast" approach to variables, the dataflow graph relies on "unicasting" variable values. The unicast approach makes reuse of storage easy, simplifying pipelining, since a storage location can be reused as soon as the item has been read once. The "broadcast" approach allows the same data value to be read repeatedly, but requires that the item is kept around until it is not live. This can limit pipelining and may require explicit reset lines.

In order to convert the program to a dataflow graph, it is first converted to Linear SASL, where *all* variables are linear. This translation is explained in the next section, while the translation from Linear SASL to a dataflow graph is given in Section 4.2.2.

4.2.1 Translation to Linear SASL

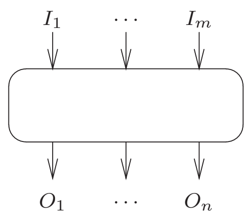
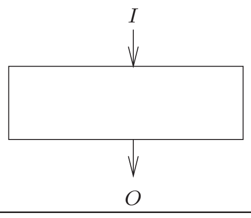
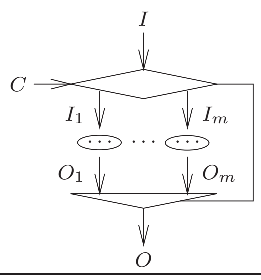
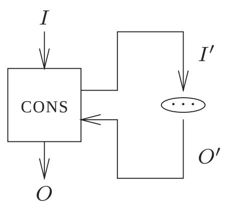
Linear SASL demands that if a variable becomes non-live, it must have been used exactly once. While normal SASL requires that a stream variable is used at most once, Linear SASL ensures that *all* variables are used *precisely* once during terminating computation. This makes it possible to convert the binding and usage of a variable into an edge that connects the output of the graph that computes the variable's value to the point at which the variable is used.

Linear SASL's grammar is given in Figure 4.4. The expressions DUP and KILL are introduced:

> The DUP expression is needed to duplicate values, since each variable may only be used once. It returns a pair containing two copies of the variable's value (variables containing streams may not be duplicated, to preserve the original linearity constraint).
>
> The KILL expression is needed to "use up" variables that are otherwise not accessed, to preserve linearity. It acts as if the variable x is read and discarded, before returning the value of the expression e.

The linearity requirements are shown in the syntax-directed rules of Figure 4.5. The rule lin_f should be applied to each function, returning the empty set if the function meets the linearity requirements, and **error** otherwise. The lin rule generates the set of free variables the expression will have used exactly once if it terminates. The rules are constructed so that the same variable may not be used in multiple subexpressions (except for conditional expressions). Variables must be used within the expression they are bound in. All cases of a conditional expression should use the same set of variables, so that the same variables are accessed irrespective of the condition.

The conversion of a SASL program to Linear SASL is achieved by a syntax-directed translation to remove non-linear variable access. The syntax tree is traversed in a top-down fashion, applying the following rules:

	**Normal** nodes are used to do basic data processing that does not require function calls—the "normal" operations that do not require special case operations. This include tupling, untupling, stream reads, the creating and unpacking of algebraic datatypes, and implementing primitives. Normal nodes take a token from each input edge I_i, perform an operation, and place a result on each output edge O_i.
	**Function call** nodes implement function calls. The function argument is put on the I edge to trigger the call, and the result is returned on the edge O. Both recursive and non-recursive function calls use function call nodes.
	**Conditional select and merge** nodes are used for control-flow. Depending on a conditional token supplied on the edge C, a token on the edge I will be supplied to one of the I_i, and a result token read from the matching O_i, and written to O.
	**CONS** nodes are are used to represent the lazily evaluated CONS expressions. They act like normal nodes with a single input I and output O. When a token is supplied to I the value of the token is stored, and a token representing the whole stream is returned from O. When the stream token returned over O is read from, the associated token that was read from I is sent out over I', and a result token is expected from O'. The result token must be a pair of the stream's head value, and the new stream tail. These are returned as the result of the stream read.

Key: $(\cdots)$ = a subgraph.

Figure 4.3: Dataflow graph nodes

$p :=$	$d_1 \;\ldots\; d_n$	Program definition
$d :=$	**fun** $f\ x$ **=** e	Function definition
$e :=$	$f\ e$	Function application
	$c(e_1, \ldots, e_k)$	Constructor
	$(e_1, \ldots, e_k)$	Tupling
	e_1**::**e_2	CONS expression
	case e **of** $m_1 \;\;\ldots\;\; m_n$	Constructor case matching
	case e_1 **of** $(x_1, \ldots, x_k) \quad e_2$	Untupling
	case e_1 **of** x_1**::**$x_2 \quad e_2$	Stream match
	let x **=** e_1 **in** e_2	Let expression
	x	Variable access
	$\text{DUP}(x)$	Variable duplication
	$\text{KILL}(x, e)$	Variable destruction
$m :=$	$c(x_1, \ldots, x_k) \quad e$	Match

Figure 4.4: Linear ("unicast") SASL's grammar

$$\begin{array}{rcl}
lin_f(\textbf{fun}\ f\ x\ \textbf{=} e) & = & lin(e)\text{-}\ \ x \\
\\
lin(f\ e) & = & lin(e) \\
lin(c(e_1, \ldots, e_k)) & = & lin(e_1) \quad \ldots \quad lin(e_k) \\
lin((e_1, \ldots, e_k)) & = & lin(e_1) \quad \ldots \quad lin(e_k) \\
lin(e_1\textbf{::}e_2) & = & lin(e_1) \quad lin(e_2) \\
lin(\textbf{case}\ e\ \textbf{of}\ m_1 \ \ldots \ m_n) & = & lin(e) \quad (lin_m(m_1) \quad \ldots \quad lin_m(m_n)) \\
lin(\textbf{case}\ e_1\ \textbf{of}\ (x_1, \ldots, x_k) \quad e_2) & = & lin(e_1) \quad (lin(e_2)\text{-}\ \ x_1, \ldots, x_k \ \ ) \\
lin(\textbf{case}\ e_1\ \textbf{of}\ x_1\textbf{::}x_2 \quad e_2) & = & lin(e_1) \quad (lin(e_2)\text{-}\ \ x_1, x_2 \ \ ) \\
lin(\textbf{let}\ x = e_1\ \textbf{in}\ e_2) & = & lin(e_1) \quad (lin(e_2)\text{-}\ \ x \ \ ) \\
lin(x) & = & x \\
lin(\text{DUP}(x)) & = & x \\
lin(\text{KILL}(x, e)) & = & x \quad lin(e) \\
\\
lin_m(c(x_1, \ldots, x_k) \quad e) & = & lin(e)\text{-}\ \ x_1, \ldots, x_k
\end{array}$$

$$s \quad t \;=\; \begin{cases} \textbf{error} & : \text{ if } s = \textbf{error} \quad t = \textbf{error} \quad s \quad t \;= \\ s \quad t & : \text{ otherwise} \end{cases}$$

$$s \quad t \;=\; \begin{cases} \textbf{error} & : \text{ if } s \quad = t \\ s & : \text{ otherwise} \end{cases}$$

$$s\text{-}t \;=\; \begin{cases} \textbf{error} & : \text{ if } s = \textbf{error} \quad t = \textbf{error} \quad s \not\supseteq t \\ s \quad t & : \text{ otherwise} \end{cases}$$

Figure 4.5: Linearity rules

```
(* (a) The non-linear function select. *)
fun select$(sel, a, b) =$
    case sel of $x$::$xs$
        if test$(x)$ then $a$ else $b$

(* (b) The linear form of select. *)
fun select$(sel, a, b) =$
    case sel of $x$::$xs$
        KILL$(xs$, if test$(x)$ then KILL$(b, a)$ else KILL$(a, b))$

(* (c) The non-linear function sum-diff. *)
fun sum-diff$(x, y) =$
    $($sum$(x, y)$, diff$(x, y))$

(* (d) The linear form of sum-diff. *)
fun sum-diff$(x, y) =$
    case DUP $x$ of$(x_1, x_2)$
        case DUP $y$ of$(y_1, y_2)$
            $($sum$(x_1, y_1)$, diff$(x_2, y_2))$

(* (e) A function already in linear form, xor. *)
fun xor$(x, y) =$
    if $x$
    then not$(y)$
    else $y$
```

Figure 4.6: The functions *select* and *sum-diff*

If the expression is a constructor matching, the sets of variables used by each conditionally-executed sub-expression are made identical by wrapping the sub-expressions in KILLs (taking care to avoid variable capture).

If a variable x occurs in multiple sub-expressions of an expression e, the expression is replaced by **case** DUP(x) **of** (x_1, x_2) e, and the variable is renamed in the sub-expressions. For constructor matchings all the conditional expressions count as a single single sub-expression.

If the expression binds a variable x that is not used in the body sub-expression e, e is replaced by KILL(x, e).

This transformation converts SASL programs to Linear SASL programs where the DUP expressions are "pushed in" as far as possible into the syntax tree, and the KILL expressions are "pulled out". Under eager evaluation, performing DUPs as late as possible and KILLs as early as possible minimises the number of live variables and streams.

Example: The function *select*, shown in Figure 4.6(a), would be rewritten in Linear SASL as shown in Figure 4.6(b). In Linear SASL, each conditional case uses both a and b, killing the variable that is not returned. The third parameter variable, *sel*, is used in the matching expression, which produces two new variables, x and xs. The variable x is used as a parameter to the function *test*, while xs is killed, as it would otherwise be unused.

The use of DUP is illustrated by converting *sum-diff*, from Figure 4.6(c) into the Linear SASL form of Figure 4.6(d). As the variables x and y are each used twice, they must be duplicated. Figure 4.6(e) shows a function with multiple conditional sub-expressions that use the same variable. This does not go against the linearity requirements, and in fact the function is already in linear form, since in all dynamic execution paths both x and y are used exactly once.

4.2.2 Translation to Dataflow Graph

A dataflow graph can be produced quite directly from a Linear SASL program, using the syntax-directed translation of Figure 4.7. Thin lines are used to represent a single graph edge, containing a single value. Thick lines represent a bundle of graph edges, containing one or more values. The edges flowing into the top of a graph represent the graph's free variables, and the edge leaving the graph represents the result of evaluation.

Constant expressions may produce graphs with no input edges. The dataflow model requires that each node N has at least one edge to trigger it, so each of these nodes is provided with a unit edge input. The edge is supplied with tokens from the node N' that most closely encloses N. N' may be a conditional or CONS node. In the later forms of graph, multiplexers (see Section 4.3.1) or iteration nodes (see Section 4.3.2) may also enclose subgraphs. If N is not enclosed within another node, the activation edge must be triggered at the top level when an external call occurs.

Example: The function *skip-blanks* of Figure 4.8 will be used as an example throughout this chapter. The dataflow graph for this function is shown in Figure 4.9.

4.2.3 Graph Properties

There are some properties of the dataflow graphs that are particularly useful, and that we wish to preserve throughout the transformation to a low-level system. For example, if one token is provided on each input edge of a graph currently containing no tokens, and each function call completes, the result will be one token on each output edge, and a graph containing no tokens. Moreover, if we design the nodes correctly requests can be pipelined, so that if new sets of tokens are sent in before all previous tokens have emerged, the result would be the same as if the graph were used in a non-pipelined manner.

In order to get correct overall pipelining behaviour, the following properties are required of the components:

Normal nodes and function calls must return results in the correct order. In other words, they should be pipelineable as described in Section 4.1.

Conditional constructs should produce results in order. At the merge stage items should be read from subgraphs in the order they were fed in, either by keeping a FIFO which contains the collection order, or by only allowing a single token into the construct at a time.

CONS nodes must only allow a single stream to be generated at a time on the associated stream bus. Later in the synthesis process, once the stream buses have been generated, locking primitives will be introduced to ensure mutual exclusion. In the mean time, we assume that each stream generated dynamically has a separate stream bus, so that all stream reads will cause a request from the correct stream.

We use the term *normality* to describe the property of graphs that behave like normal nodes—they operate by taking a single token on each input, producing a single token on each output and are quiescent between requests. *Total normality* (TN) describes graphs that will always produce a full set of output tokens when given a set of input tokens, while *partial normality* (PN) is the name given to graphs that may become trapped in an infinite loop, but are otherwise TN. A PN graph that is not TN is normally a programmer

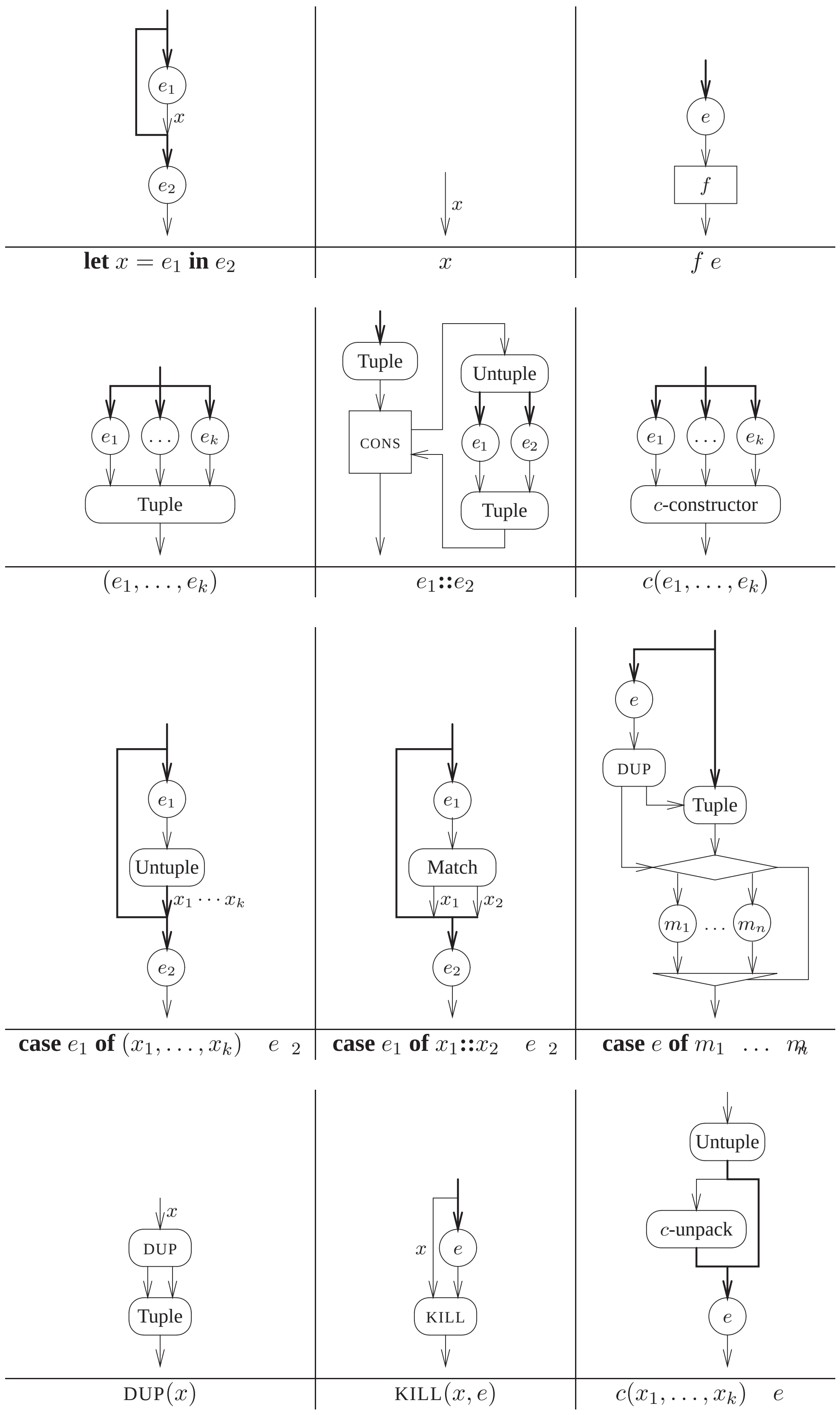


Figure 4.7: Syntax-directed translation to dataflow graph form

```
fun skip-blanks stream =
    case stream of x::xs
    case x of
        Blank       skip-blanks xs
        Symbol s    s::skip-blanks xs
```

Figure 4.8: A function to demonstrate CDFG conversion

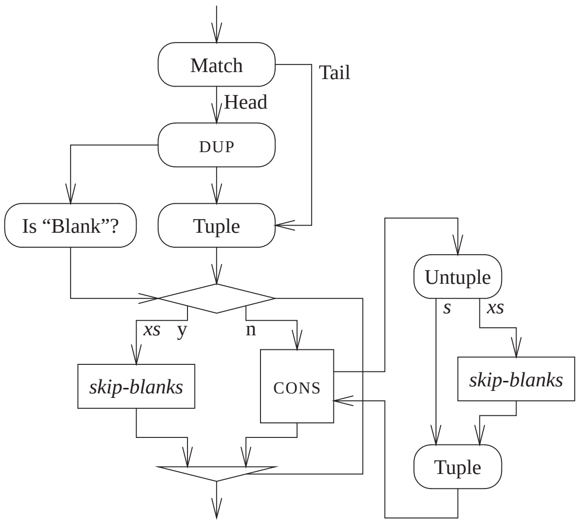


Figure 4.9: The dataflow graph for the function *skip-blanks*

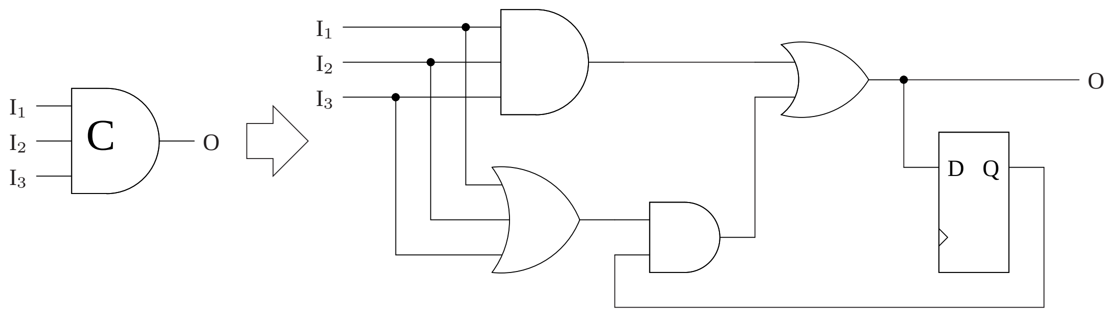


Figure 4.10: Schematic for a synchronous Muller C element

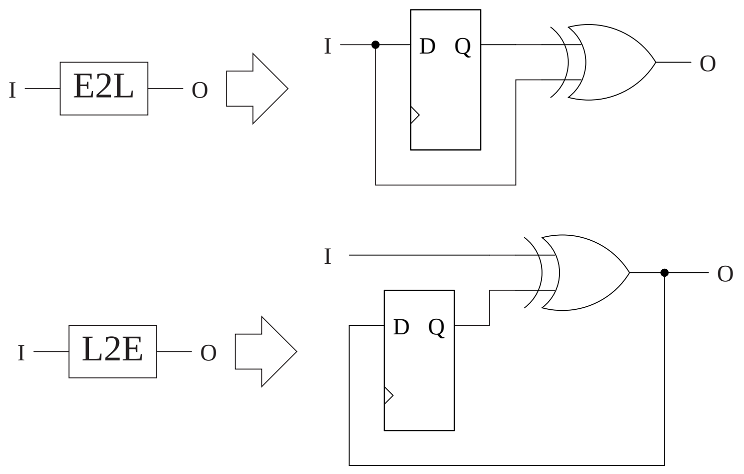


Figure 4.11: Schematics for edge-to-level and level-to-edge signal conversion

error, as we expect programs not to go into unproductive loops, but as such programs can be written in SASL they must be dealt with. A graph that can correctly deal with multiple outstanding requests is *pipelineable*. The dataflow graphs are PN and pipelineable, by construction (ignoring the details of stream buses, which will be covered later).

4.2.4 Node Implementation

Of the kinds of node in Figure 4.3, the function call nodes and CONS nodes will be transformed during later parts of the graph synthesis process into other node types. However, it is possible at this stage to give example implementations of normal and conditional nodes, in order to provide a better intuition of how the graphs are actually synthesised to hardware. Sections 4.3.3 and 4.4.7 provide further details of node implementations.

For the implementation given here, each edge is split into three parts: a request line, R, an acknowledgement line, A, and a set of data lines, D. The request and acknowledgement lines perform two-phase signalling, and the data is guaranteed to be stable between the time the request is signalled and the acknowledgement is given.

In order to deal with these two-phase signals, a number of common elements are introduced. The synchronous Muller C element shown in Figure 4.10 switches its output to the same value as its inputs when all inputs match. This is used to detect when, for example, a set of requests have all arrived, so the next stage can commence. The "E2L" and "L2E" circuits of Figure 4.11 are used to convert between the edges of two-phase signalling and level-sensitive triggering, which is more convenient for certain parts of the control logic. An assumption of these blocks is that there will not be more than one transition per

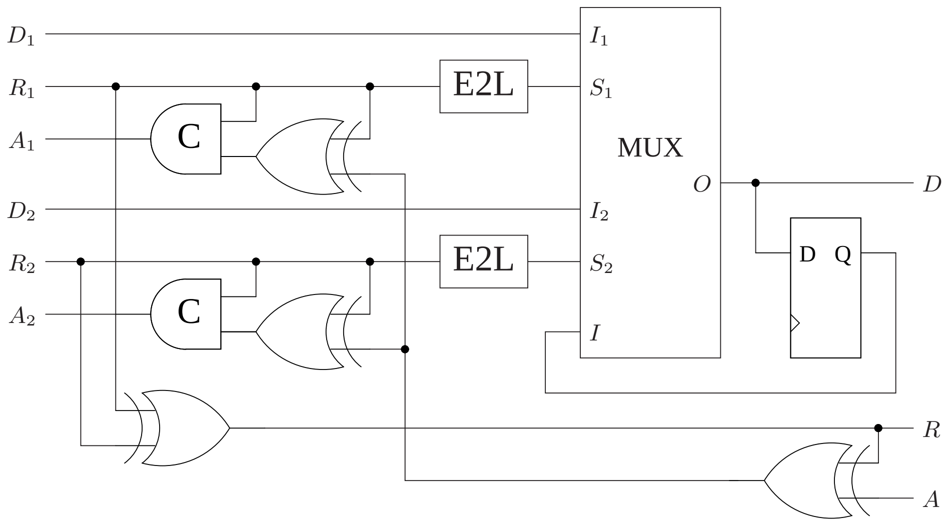


Figure 4.12: Schematics for a "Join" construct

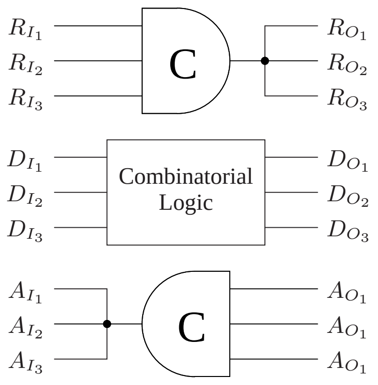


Figure 4.13: Schematics for a normal node

clock cycle.

The Join block shown in Figure 4.12 is used in the implementation of the conditional and multiplexer nodes. It takes a set of wires representing a number of graph edges, and merges them—when there is a request event on one of the inputs the associated data is transmitted on the data output wires, and the outputs request line is triggered. When an acknowledgement edge is received the input acknowledgement lines are made to match the associated input request lines, acknowledging the original request. The Join block is designed with the assumption that there will only ever be one unacknowledged request passing through it at a time.

The design may require some explanation. The MUX block puts input I on its output O unless one of the S_i is asserted, at which point it places I_i on its output. The construction of the acknowledgement logic depends on generating a signal that is high while the output has an unacknowledged request. Only when the request is acknowledged is the appropriate input acknowledged by setting the acknowledgement line to the same state as the request line.

Using these building blocks, example implementations of normal nodes and conditional nodes are given in Figures 4.13 and 4.14. The names of the inputs and outputs use the same scheme as used in Figure 4.3. The details of these schematics are as follows:

A normal node simply wraps up a piece of combinatorial logic. When a request has been signalled

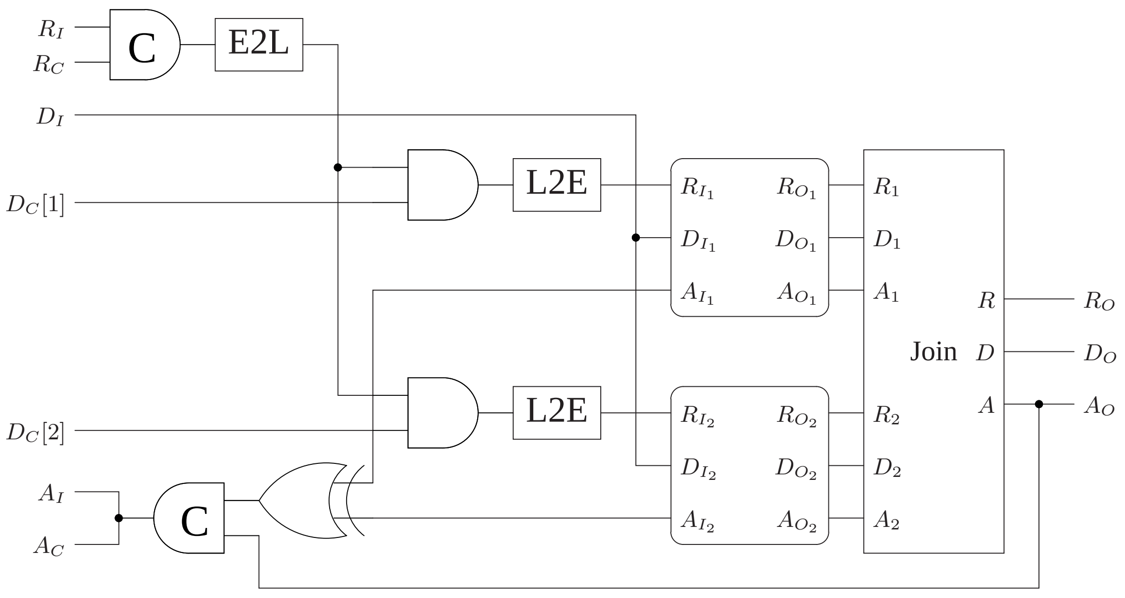


Figure 4.14: Schematics for a conditional node

on all inputs, all data is now present, so the output will be valid (after a combinatorial delay), and the output requests are signalled. Similarly, the acknowledgements are not sent back until all outputs have sent their acknowledgements, so that the outputs are held constant. Note that a sequence of normal node implemented like this will not perform *pipelined* processing without the explicit insertion of buffering stages.

The conditional node triggers when both edges I and C have had request events. Depending on which bit of C is set, the event is passed on to the appropriate subgraph. The output request is passed on to a Join block which passes on the request and routes the acknowledgement back to the appropriate subgraph. When both the subgraph has acknowledged its input, and the output has been acknowledged, the input is acknowledged. This prevents multiple requests passing through different conditional subgraphs simultaneously and overtaking each other.

4.2.5 Other Dataflow Architectures

The dataflow graphs used here are similar to those used by dataflow processor architectures. These systems have functional units like conventional processor systems, but instead of having an instruction stream, they have a set of instructions which are triggered when the relevant pieces of data are available (modern out-of-order superscalar processors are effectively limited dataflow processors, allowing them to hide memory latency).

Such dataflow systems have various problems which can be solved relatively simply in hardware. These include:

Excess parallelism may be produced—loops may be spawned faster than they can be evaluated. Traub's thesis [144] covers the use of *k-bounded* loops to limit parallelism. A hardware implementation avoids this problem, since the parallelism is limited by available hardware.

Different iterations of a loop must keep their tokens separate. A number of approaches have been tried to distinguish tokens from different iterations, such as code copying, "coloured" tokens (each token is given a tag) and activation records for each iteration. In hardware the tokens for a given iteration are kept synchronised by preventing tokens overtaking each other in the hardware.

Not all the tokens may be used, leading to the need for garbage collection. An explicit garbage collector can be used, or if activation frames are used the space is reclaimed when the variables go out of scope. In hardware, the lack of time penalty for DUP means linearity can be used, ensuring that each value is used exactly once, so GC is not required.

4.3 The Control/Dataflow Graph

Function call nodes can be viewed as placeholders for actual function implementations. The Control/Dataflow Graph (CDFG) eliminates these nodes, replacing them with the actual implementation of control flow, as described below. This transformation is a kind of linking stage, replacing symbolic representations of function calls with actual function calls. The normality and pipelineability properties should be preserved when performing this transformation.

Non-recursive function calls are achieved by simply replacing the function call node with a copy of the graph of the function being called. Recursive tail calls are more complex, and are dealt with in two stages: removing recursive calls that are enclosed within CONS expressions (CONS-*enclosed tail recursion*), and removing recursive calls that are not enclosed in CONS expressions (*direct tail recursion*). It is necessary to distinguish between these two cases, as in the first case the result of the tail call is returned to the CONS node, and in the other it is passed directly to the enclosing function.

Transforming recursive calls introduces loops to graphs. If the hardware is heavily pipelined with insufficient buffering, it may be possible to produce deadlock if tokens cannot loop back to the top of the loop because they are blocked by back-pressure from tokens earlier in the loop. The implementation of iteration nodes must prevent this (e.g., by ensuring adequate buffering on loops).

4.3.1 Removing CONS-enclosed Tail Recursion

When a tail-recursive call occurs, we wish to reactivate the original hardware, rather than invoke the call on a new instance (which is wasteful of hardware, and will not work with unbounded loops). Instead of replacing the function call node with a new instance, it is replaced with a call to the current instance. This section examines the replacement of recursive calls that occur within the tail of a CONS expression; the next section covers all other recursive calls.

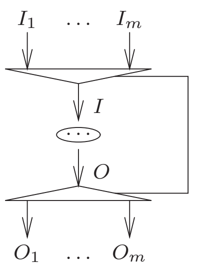
To implement CONS-enclosed recursive calls, a multiplexer node is placed around the function graph to allow calls from both the external call site and internal recursive call sites, and the edges that went into the function call node are now connected to this multiplexer node. The multiplexer node is described in Figure 4.15. The transformed dataflow graph for the example *skip-blanks* is shown in Figure 4.16.

Recursive calls enclosed in CONS are now implemented as calls back to the same piece of hardware. As long as the original graph was pipelineable, the new graph is too, since the recursive call just becomes another pipelined request. Since CONS nodes do not directly execute their subgraphs (instead waiting for a stream request before execution starts), the completion of one call to the hardware will not depend on the completion of a recursive call to the same hardware, so no deadlock problems are introduced.

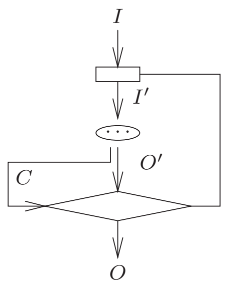
This transformation breaks the earlier structuring conventions, in that there are now edges that *directly* connect nodes inside the CONS subgraph with nodes outside. These graphs can no longer be built up using just concatenation and composition. However, normality properties are still preserved, although it is now necessary to design CONS nodes so that they can be reactivated between the time they receive a stream request and send the corresponding reply.

The multiplexer only needs to store a finite amount of state, since each recursive call will return without performing a further CONS-enclosed recursive call. This is because there is a CONS node interrupting every path from the top level of the multiplexer's subgraph to the recursive call site. Once the CONS node is reached, evaluation returns immediately.

This transformation cannot, in general, be applied to direct (not CONS-enclosed) recursive calls. If the other recursive tail call in *skip-blanks* were made into a connection into the multiplexer, the multiplexer



Multiplexer/demultiplexer nodes share access to a resource. They are used to implement CONS-enclosed tail calls in stream-producing functions (see Section 4.3.1), where the function may be called from either an external call site or a recursive call site inside the tail portion of a CONS expression. A request token is accepted from any of the top inputs I_i, and passed on through edge I, (and the edge the token was received from is recorded). When a token is received on edge O, it is passed out the arc O_i matching the corresponding input arc the request token came from.



Iteration nodes are used for loops, acting like a `do ... while` loop. A token is taken in on edge I, and passed through on edge I'. The subgraph produces tokens on edges C and O'. When the O' token reaches the diamond node, the binary condition token on C selects whether to pass the token back for another iteration through the subgraph, or to produce a result token on edge O. This is used to convert direct tail calls to iteration (see Section 4.3.2).

Key: ⋯ = a subgraph

Figure 4.15: Dataflow diagram looping node types

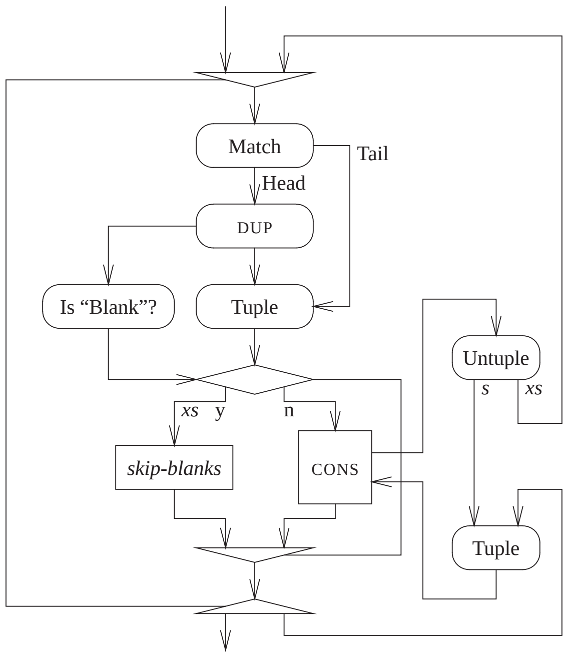


Figure 4.16: Removal of CONS-enclosed tail recursion

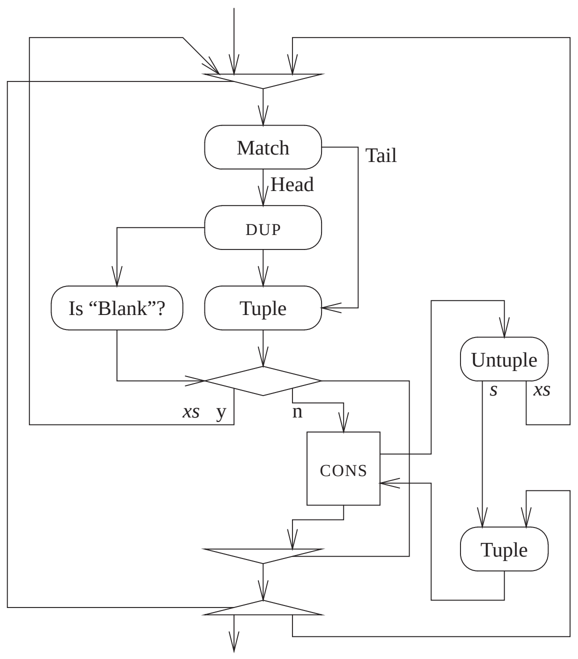


Figure 4.17: Naïve removal of direct tail recursion

node could be called an unbounded number of times before returning, needing unbounded storage. A different approach is required.

4.3.2 Removing Direct Tail Recursion

As mentioned above, direct tail recursion cannot be removed by conversion to a recursive call in the graph, since the calls may nest. However, the list of call sites does not actually need to be stored; since the recursive calls are in tail position, we can just return directly. This approach is shown in Figure 4.17 (the multiplexer's new (diagonal) input arrow does not update its recorded state, so the return goes to the last caller).

Unfortunately, this approach means that the conditional node loses its normality property—the recursive call is rather like a `goto` out of a subroutine. The hardware implementation may fail, as nodes with subgraphs may expect that a token exits for each token that enters in order to operate correctly.

A better approach disallows direct tail recursion, and introduces an iteration operator. The iteration operator is used to construct a *trampoline* [140], which repeatedly calls a supplied function. Functions with direct tail calls are rewritten to use the trampoline, returning parameters for the next call instead of performing the call directly.

An ML definition of the *trampoline* function representing the iteration node is given in Figure 4.18, along with a version of *skip-blanks* rewritten to use *trampoline*. The program is effectively being rewritten in a structured style [45]—the recursive calls in tail positions act rather like `goto`s, and are replaced with a loop with a single entry and a single exit point. This structured style can then be mapped to a dataflow graph.

The function *trampoline* is implemented using the iteration node type shown in Figure 4.15. The function itself can be implemented using the graph shown in Figure 4.19, with the function to be repeatedly called substituted for f.

The actual function transformation is achieved by rewriting the tail expressions as follows:

1. If the expression e contains no direct tail calls, return *Done*(e).

```
datatype(α, β) trampoline = Repeat of α    Done of β

fun trampoline f param =
    case f(param) of
        Repeat(new-param)    trampoline f new-param
        Done(result)         result

fun skip-blanks-2 stream =
    case stream of x::xs
    case x of
        Blank       Repeat(xs)
        Symbol s    Done(s::skip-blanks xs)

fun skip-blanks stream =
    trampoline skip-blanks-2 stream
```

Figure 4.18: The ML function *trampoline*

2. If the expression is a tail call $f(e)$, return *Repeat*(e).

3. Otherwise, recurse on each sub-expression in a tail position, substitute the resulting expressions into the original expression, and return the result.

The graph of *skip-blanks* after all recursive calls have been removed is shown in Figure 4.20. Note that the multiplexer must be placed outside of the iteration node, since the CONS-enclosed recursive calls wish to call *skip-blanks*, rather than *skip-blanks-2*. The graph looks somewhat complicated, but most of the normal nodes are performing trivial operations, and optimisation could remove much of the complexity. The bar on the left identifies the implementation of *skip-blanks-2*, the one on the right marks *skip-blanks*.

The graph is similar to Figure 4.17, except that it has been rearranged so that the point at which control returns to the top of the loop is moved outside the conditional. Although this section has explained the transformation syntactically, it can also be applied directly to dataflow graphs, making it possible to optimise the dataflow graph, and then later remove direct tail recursion.

The transformation is not technically source-to-source in that the resultant program may not adhere to SASL's type system. If the original function returns any streams, or takes any streams as parameters, the algebraic datatype used to signal whether to return or iterate will contain a stream, which is disallowed by the typing system. However, this restriction is only used to simplify the stability and linearity constraints, and this transformation does not cause any real synthesis problems.

The transformation maintains the normality properties of the original graph, but the iteration node type needs to be carefully designed to work in a pipelined environment. If the node is incorrectly implemented, two tokens could enter the loop, the second complete in fewer iterations, and leave the loop before the first. Two possible solutions are:

"Lock" the node when a token enters it, preventing other tokens entering the loop until the first has left.

Tag each item that enters the loop, buffer results, and emit the items in order. This is like the use of a reorder buffer in an out-of-order CPU to order the committing of instructions.

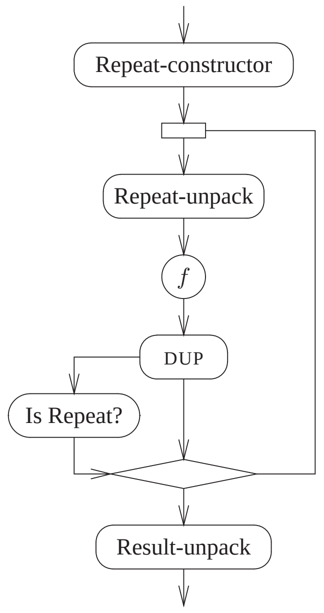


Figure 4.19: Encapsulating a transformed program in a trampoline

4.3.3 Node Implementation

Schematics representing possible implementations of multiplexer and iteration nodes are shown in Figure 4.21 and and Figures 4.22 and 4.23 respectively. The designs work as follows:

> The multiplexer node works by taking a request from any of the incoming edges, and sending it to the subgraph, along with its data. The matching acknowledgement is sent back to the appropriate input. The state of the input request line is forwarded to the output request line (forwarding on the request) when the the subgraph has completed (that is, when the status of its input and output request lines match). The output edges' acknowledgements are routed back to the subgraph. As with the conditional graph, this implementation allows for only a single request to be processed at a time.
>
> The iteration node implementation has been divided into two schematics. Figure 4.22, the inner part, wraps up the subgraph so that it has a single request edge and single acknowledgement edge. The output request edge now also acts as an input acknowledgement, and the input request acts as an output acknowledgement. The schematic is effectively an "depipelining" wrapper, in that only a single request may be passed through at a time.
>
> The output part, shown in Figure 4.23, collects data, either from a new input request, or the result of a previous iteration, and sends it to the subgraph. Depending on the output of that iteration, the result is either sent out of the graph, or is passed around for another iteration.

4.4 Extracting stream buses

For a hardware implementation of the graphs, some representation is needed of the connections between the hardware that requests items from streams, and the hardware that services the requests. *Stream buses* are shared resources used by stream matching nodes to request and receive stream items, and by CONS nodes to detect requests and service them. Each stream bus may at any one time have at most one reader and one writer. SASL has been designed so that each stream value in the program can be statically

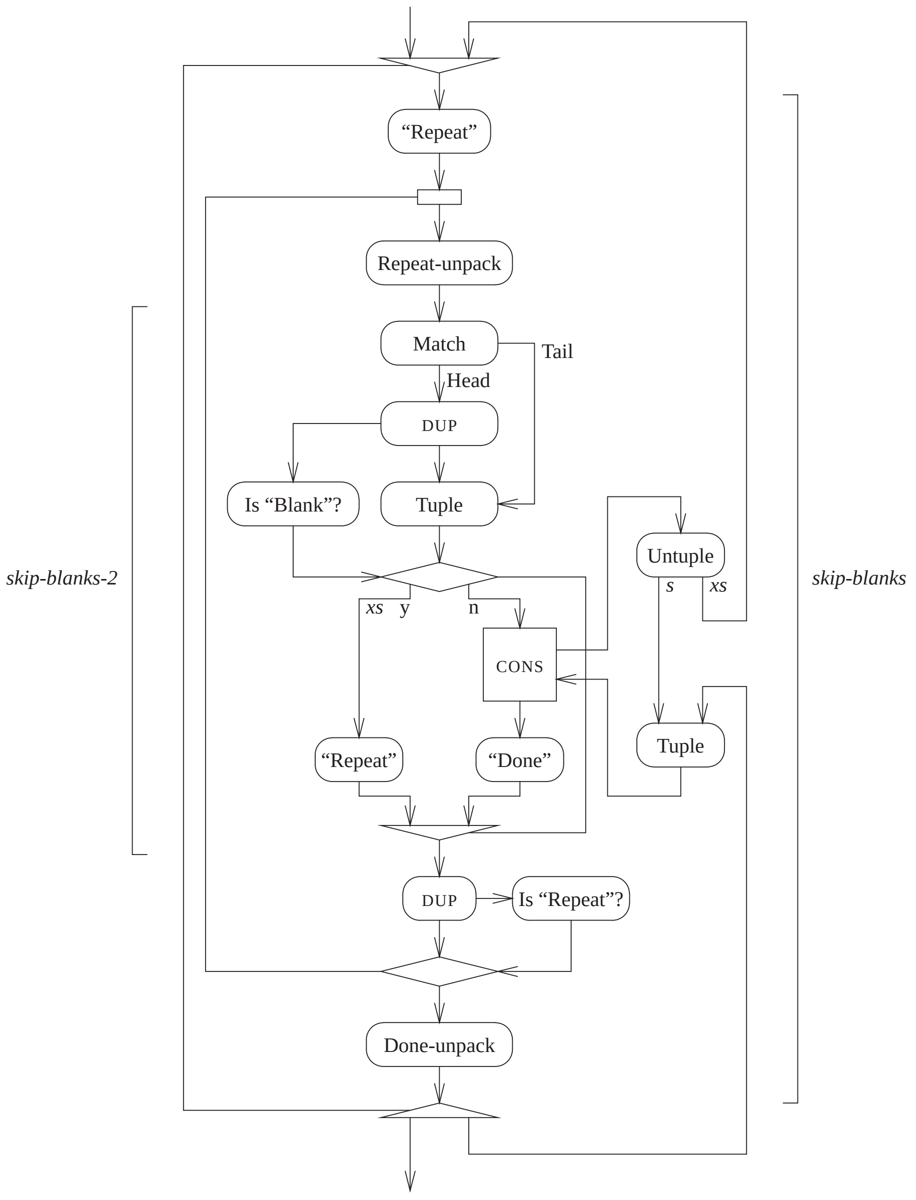


Figure 4.20: The function *skip-blanks* with recursive calls eliminated

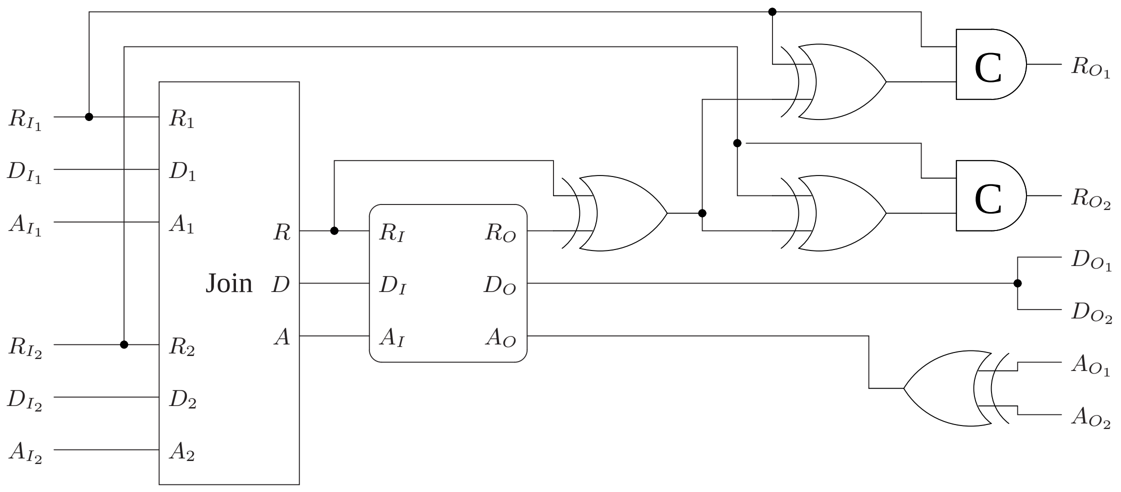


Figure 4.21: Schematics for the multiplexer node

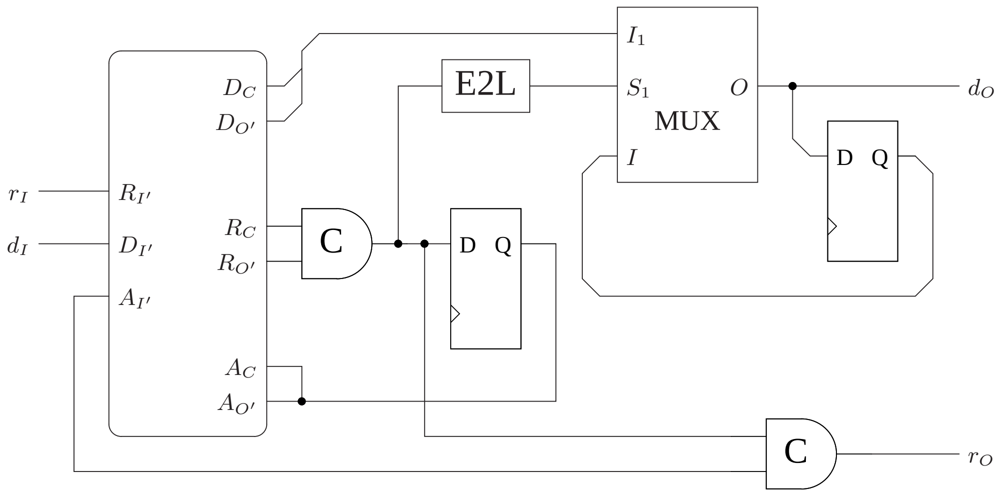


Figure 4.22: Inner part of the iteration node schematic

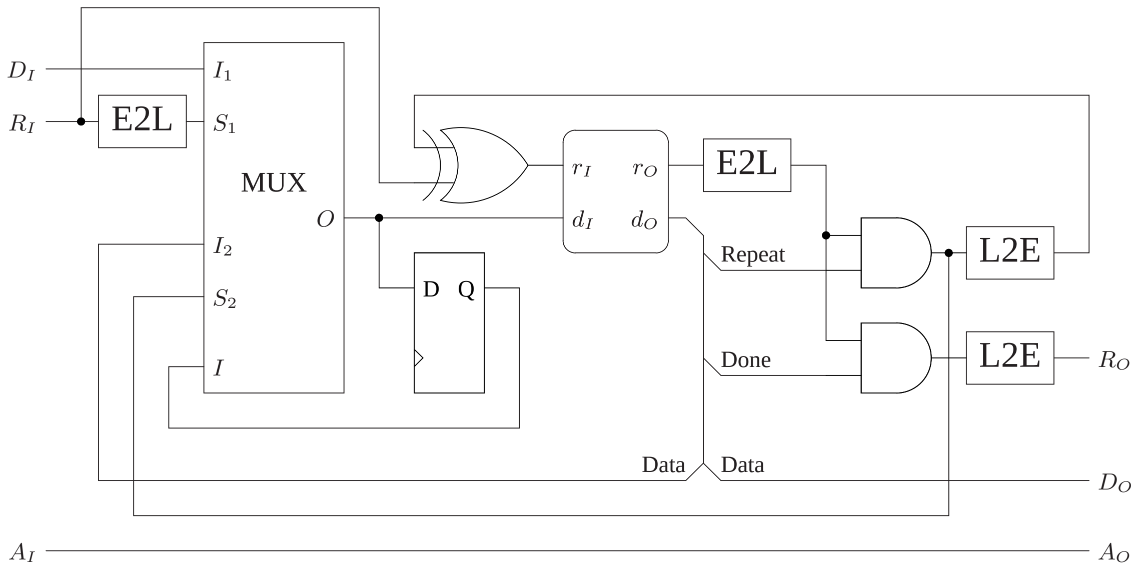


Figure 4.23: Outer part of the iteration node schematic

associated with a stream bus; each stream match node (read) and CONS node (write) is associated with a particular stream bus.

To extract the stream buses required by a graph, and where they are used, all values of stream type are annotated with stream buses. The following sections introduce the use of stream buses, explain the type system extension and then give some examples.

4.4.1 Stream Buses

The stream buses of graph synthesis are fundamentally the same as CSP stream buses, except that the rules for when it is necessary to introduce new stream buses in variable access expressions are made more accurate. This allows better synthesis, with fewer stream forwarders required.

The model of Section 3.3.3 is that an expression which returns a stream bus should not listen for requests on that bus until the expression has been evaluated. By using this model combined with lazy evaluation of CONS, it should be impossible to have a stream bus where two or more CONS nodes simultaneously listen for requests on the same stream bus.

This section introduces a more accurate model. The only types of expression where it is possible to cause more than one CONS node to listen on a stream bus are:

CONS expressions which write to a stream bus that already has a CONS node listening on it.

Variable access expressions which create stream bus forwarders which forward onto a stream bus that already has a listener.

Function call expressions that return values on a stream bus which already has a listener when the function call occurs.

Constraints are introduced to the type system of the following section to prevent these situations from occurring. By introducing fresh stream buses in variable access expressions, it is always possible to meet the above constraints, as was done for CSP synthesis.

For example, in the expression **let** $x = f()$ **in** *True*::x, the stream returned by $f()$ will be listened on before the CONS expression is evaluated, so the variable occurrence x must create a new stream bus and forward to it, so that the CONS expression will be CONSing onto a stream bus that is not being listened on at the time it starts evaluating.

4.4.2 Stream Bus Typing

For this chapter's stream bus typing we replace the stream identifiers with stream bus names, so that a stream will have the type τ *stream*$_i$, where i represents a stream bus. No information is lost, as each parameter stream is given a unique stream bus instead of a stream identifier. Other stream buses are introduced by being returned from functions, or by the stream bus substitutions performed by variable access expressions. These stream buses correspond to a "$\star$" stream identifier. The stream buses used in different functions should be distinct.

The typing rules for stream buses are shown in Figure 4.24.[2] The rules use the function $ASB(e)$, which returns the set of *Active Stream Buses*: these are the stream buses which may be being listened upon by CONS nodes or stream forwarders at the start of the given expression. The function can use any appropriate conservative approximation. A few of the typing rules may require some clarification:

> The APPLY rule matches up stream buses between the caller and callee. It uses θ, a substitution on the stream buses occurring in the type of f (this substitution is similar to the one used in Section 2.4.1). This substitution maps the stream buses present in the formal parameter type to those corresponding ones in the actual parameters. All other stream buses which are only present in the function's return type are mapped to distinct stream buses that do not appear in $ASB(f\ e)$. In this way, it operates very similarly to polymorphic typing.
>
> The CONS-INTRO rule has the requirement $i\ /\ ASB(e)$. This is to prevent the CONS from listening to a stream bus that already has a listener on it.
>
> The CONSTR-ELIM rule requires that all stream buses returned by the conditional expressions are the same. Whichever conditional path is taken, the results must appear on the same stream bus.
>
> The CONS-ELIM rule reads from a stream, and the same stream bus is used for both the stream being read from and the stream representing the tail. After the match has occurred, the remainder of the stream will be accessed through the same bus. The limitation of one active stream reader per stream bus is enforced by ordering the stream matches using dependencies in the graph.
>
> The VAR rule allows streamed items to be moved to a new stream bus (which may be required to use the CONS-INTRO and CONSTR-ELIM rules). The substitution θ is used to replace stream buses with other stream buses. The target stream buses must be distinct and must not be actively listened upon at the start of the expression (that is, they are not in $ASB(x)$). In hardware, this is implemented by a module that is activated when the expression is evaluated, forwarding stream requests from the new stream bus to the original stream bus, and then forwarding results back.

4.4.3 Typing Implementation

The typing rules can be implemented using a unification-based approach. All values of stream types are assigned separate stream bus variables, and the variables are unified in order to meet the constraints of the type system.

The implementation initially assumes that no stream bus substitutions are used in variable access expressions, and whenever a constraint that a particular stream bus must not be an active stream bus in an expression is broken (for example, in the CONS-INTRO rule), the problem is traced back and a stream bus substitution introduced. Using this technique, a minimal number of stream bus substitutions are made.

Under this implementation, if a stream bus that is returned by an expression e is active before e starts evaluating, there must be a live variable containing a stream associated with the stream bus (otherwise,

[2] This thesis relies on a number of type-like rule systems, which could alternatively be implemented using, for example, abstract interpretation [44].

$$(\textsc{apply})\frac{A \quad e : \sigma_1 \quad f : \sigma_2 \quad \sigma\ _3}{A \quad f\,e : \theta(\sigma_3)}\ \theta(\sigma_2) = \sigma_1$$

$$(\textsc{constr-intro})\frac{A \quad e_1 : \tau_1 \qquad\qquad A \quad e_k : \tau_k}{A \quad c(e_1, \ldots, e_k) : \tau}\ c : \tau_1 \ldots \tau_k \quad \tau$$

$$(\textsc{tuple-intro})\frac{A \quad e_1 : \sigma_1 \qquad\qquad A \quad e_k : \sigma_k}{A \quad (e_1, \ldots, e_k) : \sigma_1 \quad \ldots \quad \sigma\ _k}$$

$$(\textsc{cons-intro})\frac{A \quad e_1 : \tau \quad A \quad e_2 : \tau\ \mathit{stream}_i}{A \quad e_1 :: e_2 : \tau\ \mathit{stream}_i}\ i \ / \ ASB(e_1 :: e_2)$$

$$(\textsc{constr-elim})\frac{A \quad e : \tau \quad \begin{cases} A, x_1^1 : \tau_1^1, \ldots, x_{k_1}^1 : \tau_{k_1}^1 \quad e_1 : \sigma \\ \ldots \\ A, x_1^n : \tau_1^n, \ldots, x_{k_n}^n : \tau_{k_n}^n \quad e_n : \sigma \end{cases}}{\begin{array}{l} A \quad \textbf{case}\ e\ \textbf{of}\ c_1(x_1^1, \ldots, x_{k_1}^1) \quad e\ _1 \\ \quad\ldots \\ \quad c_n(x_1^n, \ldots, x_{k_n}^n) \quad e\ _n : \sigma \end{array}}\ \begin{cases} c_1 : \tau_1^1 \ldots \tau_{k_1}^1 \quad \tau \\ \ldots \\ c_n : \tau_1^n \ldots \tau_{k_n}^n \quad \tau \end{cases}$$

$$(\textsc{tuple-elim})\frac{A \quad e_1 : \sigma_1 \quad \ldots \quad \sigma\ _k \quad A, x_1 : \sigma_1, \ldots, x_k : \sigma_k \quad e_2 : \sigma}{A \quad \textbf{case}\ e_1\ \textbf{of}\ (x_1, \ldots, x_k) \quad e\ _2 : \sigma}$$

$$(\textsc{cons-elim})\frac{A \quad e_1 : \tau\ \mathit{stream}_i \quad A, x_1 : \tau, x_2 : \tau\ \mathit{stream}_i \quad e_2 : \sigma}{A \quad \textbf{case}\ e_1\ \textbf{of}\ x_1 :: x_2 \quad e\ _2 : \sigma}$$

$$(\textsc{let})\frac{A \quad e_1 : \sigma_2 \quad A, x : \sigma_2 \quad e_2 : \sigma_1}{A \quad \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \sigma_1}$$

$$(\textsc{var})\frac{}{A, x : \sigma \quad x : \theta(\sigma)}\ \theta \text{ is a stream bus substitution}$$

$$(\textsc{dup})\frac{}{A, x : \sigma \quad \textsc{dup}\ x : \sigma \quad \sigma}$$

$$(\textsc{kill})\frac{A \quad e : \sigma}{A \quad \textsc{kill}(x, e) : \sigma}$$

Figure 4.24: Typing rules (ASB and the type substitutions are explained in Section 4.4.2)

```
(* (a) The function select, with stream bus annotations. *)
fun select($(sel_1, a_2, b_3)$) =
    case sel$_1$ of $x$::$xs_1$
        KILL(xs$_1$, if test($x$) then KILL$(b_3, a_4)_4$ else KILL$(a_2, b_4)_4)_4$

(* (b) The function skip-blanks, with stream bus annotations. *)
fun skip-blanks stream$_1$ =
    (case stream$_1$ of $x$::xs$_1$
    case $x$ of
            Blank     (skip-blanks xs $_1)_2$
        Symbol $s$    ($s$::(skip-blanks xs $_1)_2)_2)_2$
```

Figure 4.25: The linear functions *select* and *skip-blanks*, annotated with stream buses

the stream bus would have been reset). Conversely, all stream buses which are referenced in the environment must be active stream buses. Hence, for this implementation, we can define $ASB(e)$ to be the set of all stream buses in e's environment.

4.4.4 Typing Examples

select: As a simple example, the function *select* from Figure 4.6(b) has been typed, and stream buses produced. The stream buses are represented by integers subscripts in Figure 4.25(a).

The streams in the parameters are given stream buses identified as 1–3. The occurrence of the variable *sel* does not require a substitution on the stream bus in order to type. The variable matching the tail, xs, keeps the same stream bus. Both sub-expressions of the conditional must have the same type, so a new stream bus is required to meet the type constraints (if an old stream bus were used, one of the variable access expressions would be substituting in a stream bus that is already being listened on). The associated variable access expressions have stream bus substitutions, but the other variable occurrences do not require substitutions to type.

skip-blanks: For the next example, we examine the typing of *skip-blanks* from Figure 4.25(b). The parameter stream is given the new identifier 1. The variable occurrence of *stream* does not require a stream bus substitution in order to type correctly, so *xs* also types with bus 1. This is necessary to meet the stability constraint. In order to type correctly, the function must return a new stream bus, which matches up with the stream bus returned by all recursive function calls.

It is necessary for the same stream bus to be returned by both recursive calls and the function itself, in order for a simple implementation of recursive calls in hardware. The property is guaranteed by the fact that all recursive calls occur in tail positions, and that all expressions in tail call position have the same type.

The typing of recursive calls can be implemented by creating stream bus variables for the streams in the return type, and then unifying these stream variables with others as required (as was discussed in Section 4.4.3).

4.4.5 Representing Stream Buses

An extended form of CDFG is used to show the stream buses extracted from the type system annotation. Each stream bus is displayed graphically using a pair of edges—a thin arrow to request a stream item in one direction, and a thick arrow for the stream result in the opposite direction. All nodes that use the same stream bus are connected to the same bus in the graph. The buses can be thought of as tri-state, connecting up CONS nodes to stream matchers, with only one reader and one writer active at any time (although a real-world implementation would probably rely on multiplexers).

The earlier graph representation is modified slightly. Stream matchers and CONS nodes are now displayed in the graph using the symbols from Figure 4.26, explicitly showing connections to stream buses. Stream bus substitutions in variable access expressions add *forwarder* nodes to the edge representing the variable access expression in the dataflow graph. When variables containing streams are killed the edge containing the stream bus value going into the associated KILL normal node has a *stream kill* node inserted on it.

The match node no longer has a separate output edge for the tail of the stream. A tail edge is unnecessary, since all it provides is synchronisation information which is being provided by the head value anyway. Since the stream buses carry the actual stream data, the tokens representing streams are simply unit values, and may be omitted if other edges provide the required ordering between stream requests.

The forwarders representing stream bus substitutions need not be implemented as primitives. A graph implementing the forwarder is shown in Figure 4.27. Using a symbol to represent the commonly repeated graph simplifies the intermediate graphs, allows a "hand-crafted" low-level implementation to be used, and may improve the possibilities for later graph optimisations.

Examples: The graph for the function *select*, with stream buses shown explicitly, is given in Figure 4.28. The stream match becomes a match node, which reads an item from the *sel* stream. The *sel* stream is then killed, as it is not used further. Depending on the item read from the stream, either stream bus 2 or 3 (representing a and b, respectively) is killed, and the other forwarded to stream bus 4 (representing the returned stream), before a token is returned, signifying the graph has finished processing, and that stream bus 4 can now be read from. A further example, the CDFG representing the function *skip-blanks*, with stream buses shown explicitly, is given in Figure 4.29 (the dotted box concerns pipelining, and will be explained in Section 4.4.6.1).

This figure is the lowest-level graph-based representation of the circuit. It can be converted to hardware by instantiating the nodes and connecting them together. The implementation of the node types used in this graph are described throughout this chapter. The nodes and constructs it uses are as follows:

Multiplexer Shown in Figure 4.21.

Iteration Shown in Figures 4.22 and 4.23.

Conditional Shown in Figure 4.14.

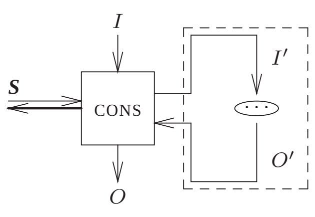
Match Shown in Figure 4.31.

Cons Shown in Figure 4.30.

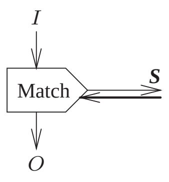
Normal nodes Shown in Figure 4.13. The combinatorial part of the functions depend on the specific nodes: "Repeat" and "Done" nodes simply tag the incoming bus with an extra bit representing whether the value is a "Repeat" or "Done", "Is "Repeat"?" and "Is "Done"?" and the unpack nodes simply test the tag or extract the bus respectively. The "DUP" nodes simply return two copies of the incoming bus, while the "Tuple" and "Untuple" nodes concatenate and separate the parts of buses.

Mutual exclusion boxes Given in Figure 4.33.

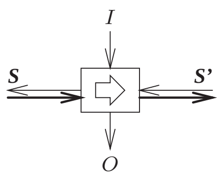
Reset boxes are described in Section 4.4.6. In a basic implementation, the output stream's reset line is the reset line for all the registers in the implementation of the graph.



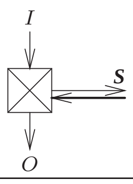
CONS nodes are now attached to the stream S which they listen on and send results to. When a token arrives on I the value is stored, and a unit token representing the stream being ready is passed out on O. When a read on the stream occurs, a request will arrive on the stream bus S, triggering the CONS node to send its stored value over I', and wait for a result on O'. When this arrives the CONS node returns the result over S to the waiting stream match node. The dashed box is used for reset processing (see Section 4.4.6).



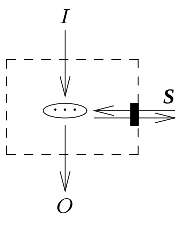
Match nodes are attached to the stream bus they read from, using the stream bus information produced during typing. When the unit token representing the stream arrives on edge I, the request is sent out over the stream bus S, and the node waits for a result. When the result arrives, it is written to the edge O, both producing the result, and signalling that the stream bus is ready for the next request.



Forwarder nodes are used to implement stream bus substitutions. When a token is received on I, representing a request to produce the stream, a forwarding process is started, and the token sent out on O, to show the forwarder is ready. When a stream request is received on S', it is forwarded to S, and the corresponding result forwarded back. A possible implementation is shown in Figure 4.27.



Stream Kill nodes inform the listener on the stream bus that no more stream items will be requested. This causes a reset to be sent to listening CONS nodes; see Section 4.4.6 for details.



Stream Bus Mutual Exclusion nodes prevent more than one stream being produced on a stream bus at a time. A single token is allowed in on the edge I, and sent to the subgraph as normal. No further tokens are allowed in until the stream bus S is reset, at which point another single token is allowed in on I.

Key: (⋯) = a subgraph

Figure 4.26: Stream bus processing nodes

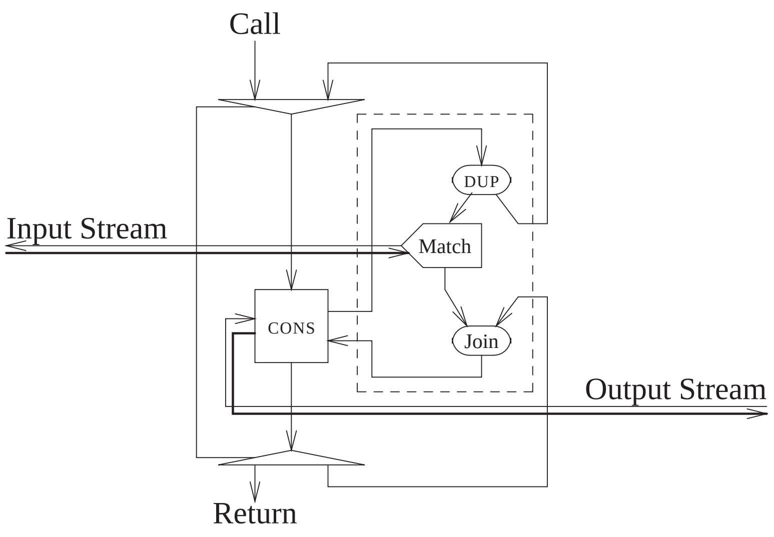


Figure 4.27: A stream-forwarder implementation

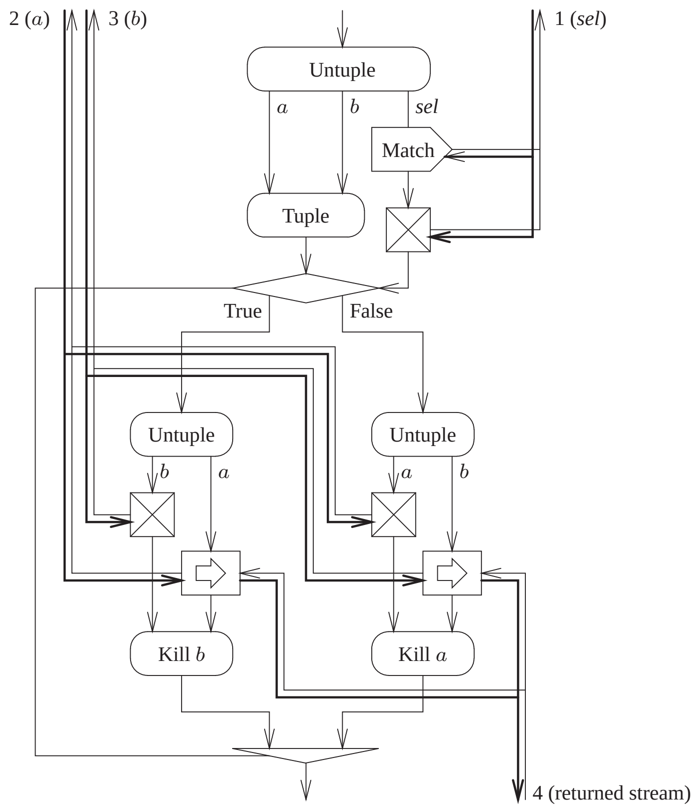


Figure 4.28: The CDFG for the *select* function, with explicit stream buses

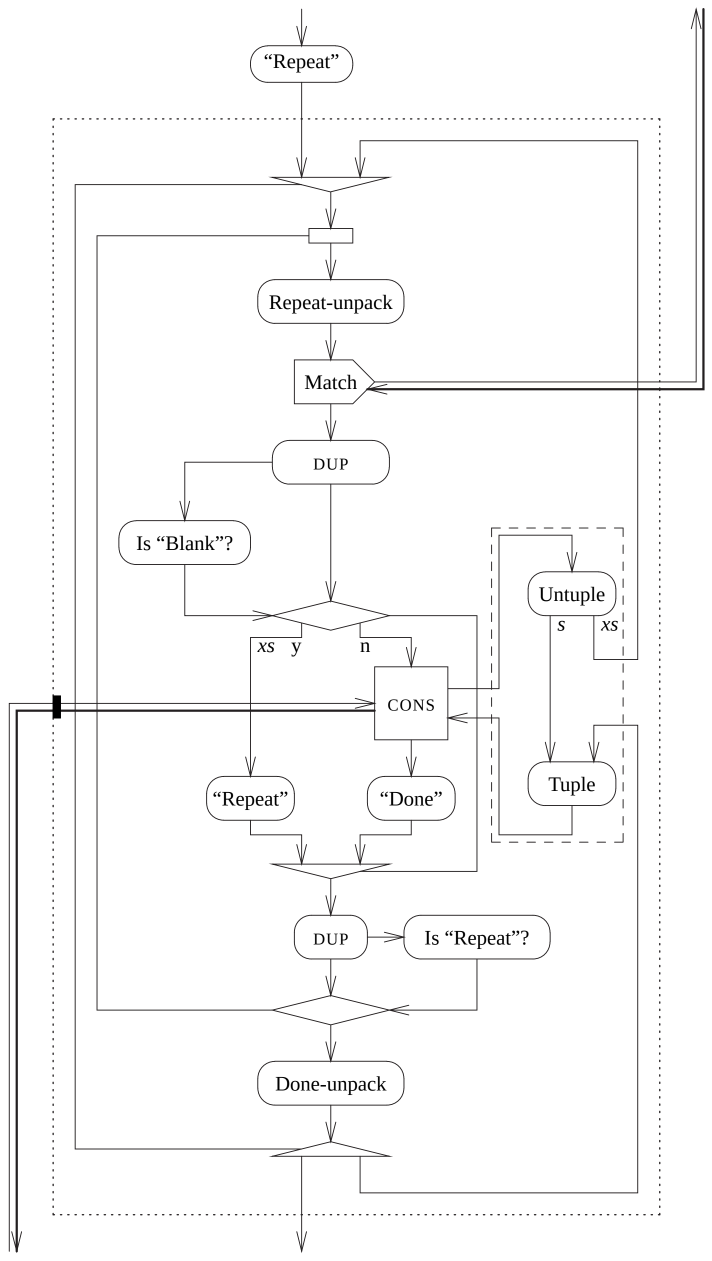


Figure 4.29: The function *skip-blanks* with explicit stream buses

4.4.6 Managing Stream Buses

In earlier sections of this chapter, streams have been identified using tokens, so that it is quite possible to have two streams active on the same stream bus, and distinguish between them for reads by using the appropriate stream token. In an actual implementation, only a single stream may be bound to a stream bus at a time. We must ensure that the previous stream is destroyed before a new stream is created on the same bus, as we are not otherwise able to tell which stream a particular read is intended for.

This section deals with controlling access to stream buses on two levels. Initially, requests are not pipelined, and it is only necessary to ensure that old streams are deleted before the new streams are introduced on the same bus. The second level deals with locking access to stream buses when there might be pipelined requests to use the stream bus.

Resetting Streams In order to prevent multiple streams from being active simultaneously on a single stream bus, it is necessary to reset the stream bus, clearing the original stream data, before processing the new stream. If requests are not pipelined, it is sufficient to reset stream buses when the streams they hold go out of scope.

To show this, we can use the fact that the dataflow graphs are structured so that all the CONS nodes that produce items on a single bus are either nested inside each other, or in alternative branches of a conditional. Therefore, the only way for multiple streams on the same stream bus to occur would be through looping constructs:

> Multiplexer nodes are used to perform recursive calls from within CONS expressions. The original CONS node will be deactivated before the next CONS node is activated.
>
> Iteration nodes could lead to multiple CONS nodes active on the same bus if a stream were kept from previous iterations. However, each iteration represents a tail call, and the stability constraint prevents any streams except the original parameters from being passed between iterations. All newly generated streams would be killed, clearing any active CONS nodes.

Hence, if requests are not pipelined, each stream bus will have at most a single stream active on it at any time. We now discuss the mechanism by which killing a stream leads to the appropriate stream buses being reset.

The CSP synthesis technique relied on explicit reset signals being sent to the hardware block representing a function before it is called again. In graph synthesis, we use the fact that a stream variable will be killed when it goes out of scope in Linear SASL. Each stream bus has a reset line associated with it, written to by stream kill nodes, and read from by CONS nodes. When a variable containing a stream is killed in Linear SASL, a stream kill node is generated in the graph. When the node is activated, the stream is going out of scope, and the CONS node listening for requests on the stream bus is reset. The reset causes the CONS node to not respond to any stream requests until it is activated again by receiving a new token on its input edge.

Resetting is slightly complicated by forwarders and state-holding CONS nodes. When a forwarder receives a reset on the stream it outputs to, that stream is becoming inaccessible, and so the stream that is being forwarded is also becoming inaccessible, and the reset must be cascaded. The CONS nodes may store values representing streams while waiting for stream requests. If such a CONS node receives a reset, these values are lost, and the stream buses associated with them should be reset, as the streams have become inaccessible. The stream buses associated with tokens held by the CONS node will be those that are generated externally to the dashed box around the CONS node's body graph, and read from inside. When the CONS node receives a reset on its stream bus, it should send resets on those stream buses, causing a cascade of resets. The stream dependencies are acyclic, so the cascades will terminate. Race conditions are prevented by mutual exclusion boxes, as described in the next section.

4.4.6.1 Pipelined Requests

Naïvely pipelining requests into a graph that generates a stream could lead to multiple streams being bound to the same bus, since pipelined tokens may enter the same CONS node (or different nodes on the same bus, if the head of the stream is produced conditionally). Mutual exclusion is required on access to stream buses, so that only a single stream may use it at a time. To do this, we introduce a new node type, the stream mutual exclusion box (shown in Figure 4.26). Like CONS nodes, forwarders and multiplexer nodes, each mutual exclusion box is associated with the productions of a particular stream bus.

During synthesis, a mutual exclusion box is generated for each stream bus. The box surrounds the smallest subgraph that generates its stream. To find this, mark each CONS node, forwarder and multiplexer that writes to a particular stream bus. The exclusion box for that stream bus is then placed to enclose the smallest subgraph that contains all of these nodes (while respecting the graph's hierarchical structure).

The box only allows a single request to enter its subgraph at a time. Once the associated stream bus is reset, a new set of tokens may enter. In order to prevent race conditions, the enclosed graph must be fully reset before new tokens are allowed to enter.

It is useful to show that the introduction of mutual exclusion boxes does not lead to deadlock. This can be shown as follows:

1. Graphs without loops will not deadlock because of mutual exclusion. Earlier calls do not depend on later calls, and so later requests can be blocked by the mutual exclusion blocks until the earlier calls have completed.

2. Multiplexer loops are not a problem, since mutual exclusion boxes are placed *around* the associated multiplexer. If the recursive function generates an intermediate stream, the multiplexer may enclose another mutual exclusion box for *that* stream, but that mutual exclusion box will not cause deadlock, as it will be cleared before the next iteration, since intermediate streams must become non-live before recursive calls can occur.

3. Iteration nodes are the last possibility for deadlock. Again, exclusion boxes around intermediate streams do not lead to deadlock. If multiple requests are allowed into an iteration node N, and N returns a stream bus S, deadlock could occur if a request enters the mutual exclusion box for S out of order. The hardware for N would try to produce results in order, and would wait for the first stream s to be returned, but that stream cannot be produced until the stream s' currently under (out-of-order) production is reset. For s' to be reset, all previous streams, including s (which is blocked) must be reset first. This problem is solved by specifying that if an iteration node returns a stream, the node's implementation must only allow a single token into its subgraph at a time.

Under lazy evaluation, pipelined requests cannot occur within a mutual exclusion box, and so any extra hardware to support pipelining can be removed from the implementation of that part of the graph. If lenient evaluation (described in Section 5.2) is used, multiple requests may be pipelined into the head expressions of CONSes within multiplexer nodes, but otherwise the requests will not be pipelined, and so the hardware may be appropriately simplified. Note that the use of mutual exclusion boxes prevents the pipelined production of multiple *streams* on the same stream bus. Lenient evaluation still allows the pipelining of item production within any particular stream.

4.4.7 Node Implementation

The stream forwarder node can be constructed from other node types. The implementations of the other nodes from Figure 4.26 are described below:

The CONS node stores whether it has been activated using the register shown at the top left of the schematic. It is activated if the node has received a request on its input edge, but has not yet received

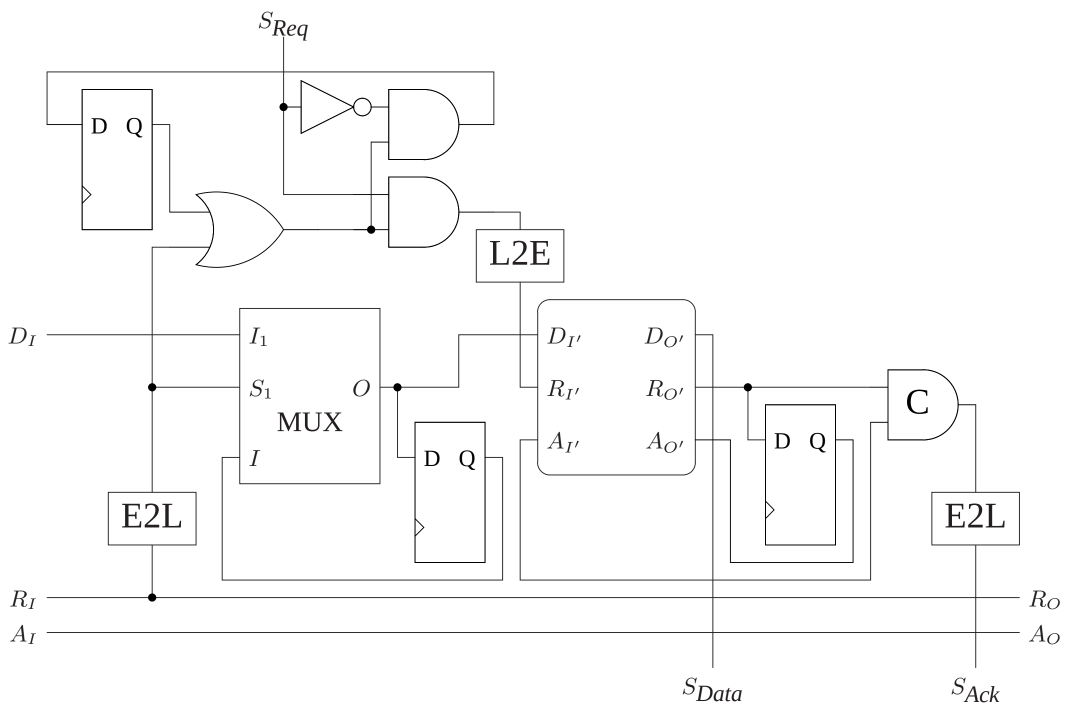


Figure 4.30: Schematics for the CONS node

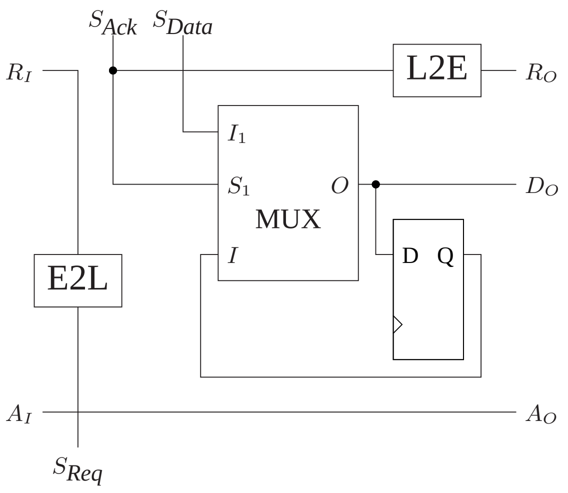


Figure 4.31: Schematics for the Match node

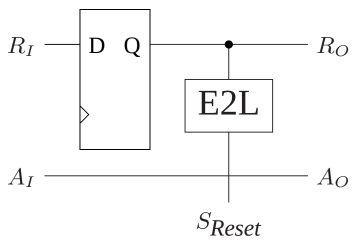


Figure 4.32: Schematics for the Stream Kill node

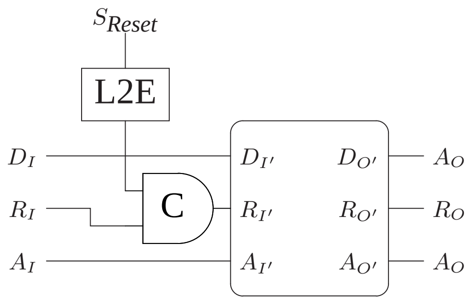


Figure 4.33: Schematics for the Mutual Exclusion node

a stream request, and is otherwise inactive. When the request arrives on the input line, the input data is also latched. The stream bus request and result lines are level sensitive, the first of which triggers the subgraph, the second of which being triggered when the subgraph has completed its calculations.

The Match node schematic simply triggers the stream bus when a request is made on its input line, and upon receiving a stream acknowledgement latches the stream data on its output, and sends out an output request.

The Stream Kill node simply triggers the associated stream kill line. This line resets all latches within the circuit associated with the CONS nodes when the stream is reset.

The Mutual Exclusion node prevents further requests passing through until there has been a reset on the associated stream bus.

4.5 Summary

The last chapter presented a simple synthesis based on CSP, utilising broadcast access to variables, and disallowing pipelining. The synthesis of this chapter relies on a specialist intermediate graph representation in order to allow a more efficient implementation. Unicast variable access simplifies pipelining. Mutual exclusion boxes prevent multiple streams from being active on the same stream bus simultaneously. The use of stream bus typing eliminates many unnecessary stream bus forwarders that would be produced under the CSP synthesis.

The synthesis of this chapter is rather more complex than the one presented in the previous chapter, but allows correspondingly better synthesis results. The synthesis techniques of this chapter form the basis of the optimisations described in the following chapter.

CHAPTER 5

Optimisation

The basic translation of SASL to a graph form, as outlined in Chapter 4, in conjunction with a Verilog implementation of the nodes, provides a relatively simple way to produce hardware. However, the hardware produced would be highly suboptimal. The main problems with the basic synthesis are:

- Much use is made of dynamic scheduling. Request and acknowledge lines are used to transfer all data, even if a statically scheduled pipeline could be created.
- Parallelism is not exploited to a great extent. Stream items are produced in a *demand-driven fashion*, so that the hardware that would produce an item remains idle until the item is requested, when it could be produced ahead of time. The use of pipelining is very limited.
- Program optimisations are not applied. The use of a high-level language allows a variety of program transformations to be performed, but none have been discussed.

These three issues are discussed in the following sections. Section 5.1 covers the use of static scheduling in simplifying the design. Section 5.2 introduces lenient evaluation, which provides a way of looking at ahead-of-time stream production, and allows increased pipelining. Section 5.3 briefly examines program transformations and, more specifically, transformations of the dataflow graphs. Although this is not a thesis on program optimisation, this should give a flavour of the transformations that may be applied. Section 5.4 provides a brief summary of the chapter.

5.1 Static Scheduling

The CDFGs of the previous chapter assume universal dynamic scheduling. For example, with the graph in Figure 5.1, the dynamic schedule would have node 1 trigger as soon as data arrives on edge A, and node 2 when data arrives on edge B. Node 3 then triggers when when both nodes 1 and 2 have completed, and an item has arrived on edge C. The intermediate edges have independent request and acknowledge lines.

In general, this is overkill. If each primitive node takes a single cycle to complete[1], a simple static schedule can be created where all three nodes may be executed in lock-step (much like a classical hardware pipeline). By statically scheduling subgraphs, we can reduce the internal synchronisation overhead, replacing the subgraph with a large node with dynamically scheduled input and output edges. This circuit will have lower area requirements, and may run faster if signalling circuitry was on the critical path.

[1]As we will assume in this section, for simplicity. Fixed multi-cycle nodes can be represented as a sequence of nodes, while nodes with variable timing cannot be statically scheduled.

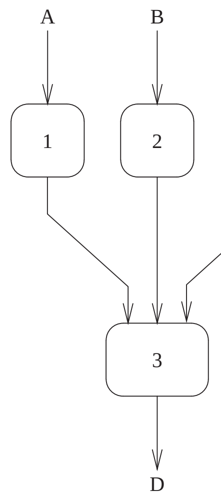


Figure 5.1: A simple example CDFG

5.1.1 The Problem

The idea of static scheduling is to remove redundant dynamic synchronisation. It is not possible to remove all dynamic synchronisation, since there are constructs that create data-dependent delays. The static schedule should generally not be slower than the dynamic schedule, for example by forcing a delay on the critical path that could be shorter under a data-dependent dynamic schedule. In some cases it may be more efficient to introduce a slower static schedule, if it produces suitable area and synchronisation overhead savings.

The use of static scheduling relies on some form of timing to schedule the stages of processing. For synchronous circuits, this timing will be in the form of a global clock, and the timing is relatively simple. Static schedules may still be useful for asynchronous circuits, though. Bundled data systems use delay elements to time combinatorial blocks. If the circuit is designed to balance all the delay elements, the result is rather like a synchronous system with a distributed clock [19]. Static scheduling can be used as a way of attempting to balance the delay elements. Even for asynchronous circuits using completion detection, such as dual-rail encoding, some way of choosing where to insert latches is needed. Static scheduling using approximate timings provides a way to do this and produce relatively balanced asynchronous circuits.

The problem is to identify sets of nodes that may be synchronised to always execute together, so that the synchronisation overhead may be reduced. Once two nodes are synchronised, their outputs will be produced simultaneously, so that nodes that only depend on these outputs can be scheduled for the next cycle, for example.

The scheduling bears many similarities to, but is not the same as, the scheduling used in many other high-level synthesis systems [102]. These rely on *scheduling*, *allocating* and *binding* elements of a dataflow graph. The scheduling phase in such systems chooses which cycle number of the static schedule each node is placed in, in order to minimise latency, while allowing hardware resources to be shared, assuming a fixed arrival time for the inputs. In a SASL graph, the aim of generating a schedule is to the reduce synchronisation overhead, and we cannot assume fixed arrival times for the tokens, but similar techniques may be used as a starting point for SASL's scheduling.

5.1.2 ASAP and ALAP Scheduling

The aim of static scheduling is to find sets of nodes that may be statically scheduled to run together, leading to a decrease in the amount of signalling required, without leading to lower performance than a

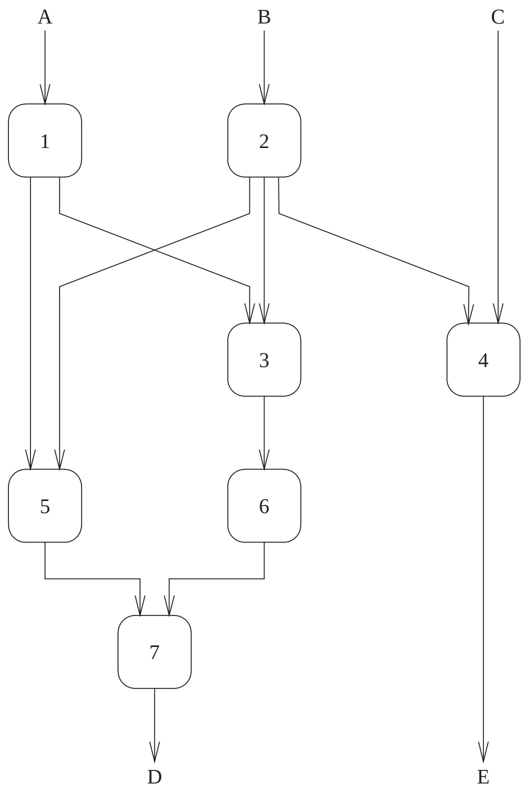


Figure 5.2: A graph to schedule

full dynamic schedule. In general, scheduling problems are often found to be hard, and as scheduling is only tangentially related to the aims of this thesis, only simple ASAP and ALAP scheduling techniques are discussed.

Before trying to apply scheduling, it is necessary to define which graphs we can apply the scheduling to. Static scheduling is performed hierarchically, with nodes that contain subgraphs treated as single nodes for the purpose of scheduling. A particular subgraph may be statically scheduled if all the nodes in it are statically-schedulable. If a graph contains a node that does not take a fixed time to process, the graph may be split into a pair of subgraphs, before and after that node, each of which may be statically scheduled.

Once the statically-schedulable subgraphs have been identified, a scheduling algorithm can be applied. This thesis only covers the simple ASAP (As Soon As Possible) and ALAP (As Late As Possible) scheduling schemes. Figure 5.2 shows a simple graph that will be used as our example. Each node is assumed to take a single cycle to complete, and the inputs A, B and C, and the outputs D and E are all assumed to be coming from and going to different static scheduling domains, so that there is no static synchronisation between them.

As Soon As Possible Scheduling ASAP scheduling places operations in the earliest cycle where they may occur. For example, if we could assume that tokens arrived at A, B and C at the same time, we would combine nodes to produce the schedule in Figure 5.3. Each dotted box effectively becomes a single node, and the set of edges between each pair of dotted boxes becomes a single edge for the purpose of synchronisation. The output E is available 2 cycles before D.

The schedule can simply be generated by assuming the inputs arrive at time 0, and that each node is scheduled to run at $\max(t_i) + 1$, where the t_i are the timesteps at which the node's parameters are produced. Nodes with the same timestamp are scheduled together.

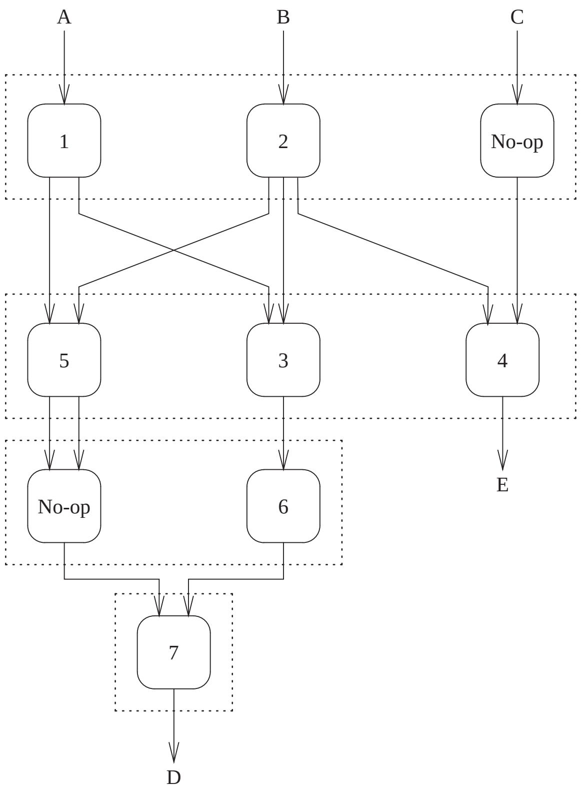


Figure 5.3: Naïve ASAP scheduling

If we cannot assume that the inputs arrive at the same time, scheduling becomes more difficult. If bad assumptions are made, one of the outputs may be produced later than necessary. To work around this, rather than using a plain numerical timestamp, an algebraic one can be used, assuming A, B and C arrive at times a, b and c. For example, node 4 would be scheduled at time $\max(b+1, c)$. Once all the nodes have been scheduled, those that have the same timestamp expression are be scheduled to run together.

Figure 5.4 shows the example scheduled algebraically. Nodes 3 and 5 may be scheduled together, with node 6 scheduled for the next cycle, and 7 the cycle after that (in the absence of back pressure preventing progress). Since a new scheduling domain must be created when different input values are merged together, ASAP scheduling generally works best when a small number of inputs are used to produce a larger number of output values.

As Late As Possible Scheduling An ALAP schedule that assumes that D and E would be read simultaneously is shown in Figure 5.5, with the operations scheduled as late as possible accordingly. The no-operation and node 5 have switched around, compared to Figure 5.3, and value C can arrive 3 cycles later than A or B without affecting timing, but if A is late the production of E (which does not depend upon A) is delayed.

Algebraic ALAP scheduling can be achieved by assuming results must be produced at times d and e, and scheduling everything as late as possible to meet these deadlines. The ALAP schedule is shown in Figure 5.6. ALAP scheduling is simply the dual of ASAP scheduling. Since dependencies still flow forwards at runtime, each statically-scheduled set of nodes still triggers as soon as all data has arrived. In contrast to ASAP, ALAP works best when a small number of results must be produced that depend upon a large number of inputs.

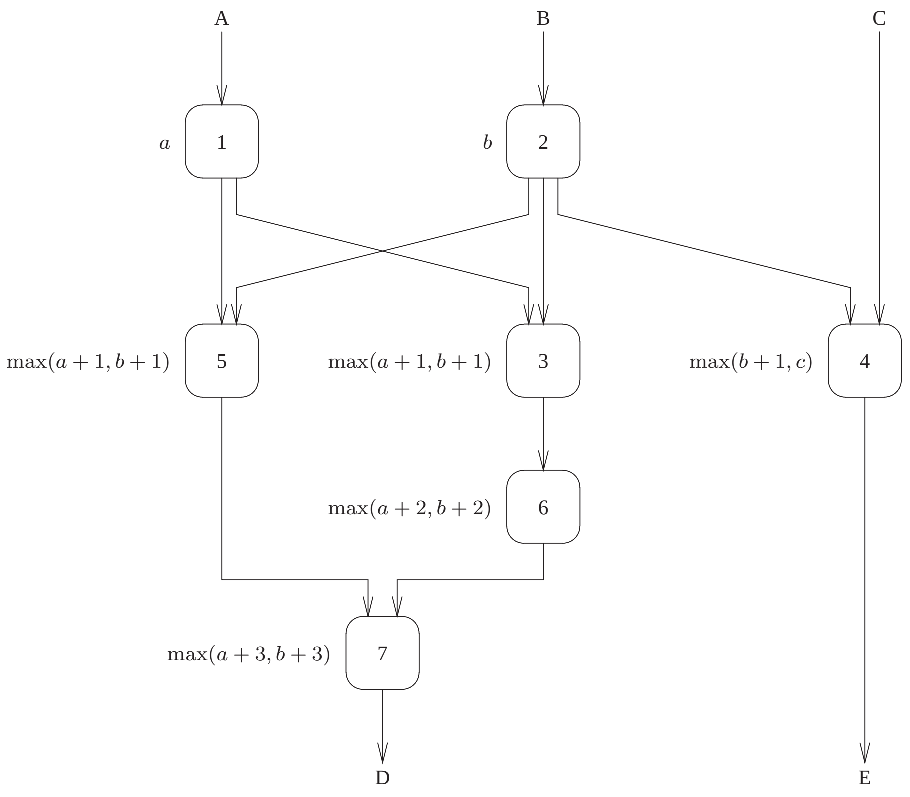


Figure 5.4: Algebraic ASAP scheduling

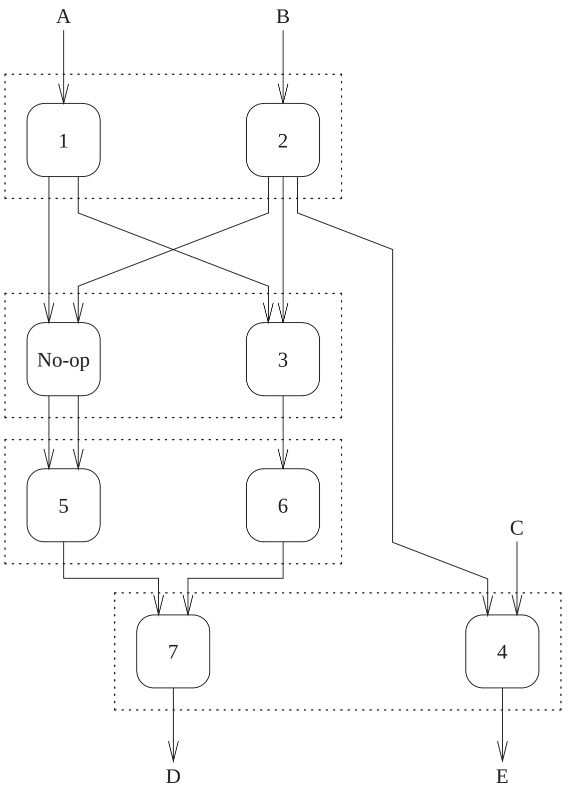


Figure 5.5: Naïve ALAP scheduling

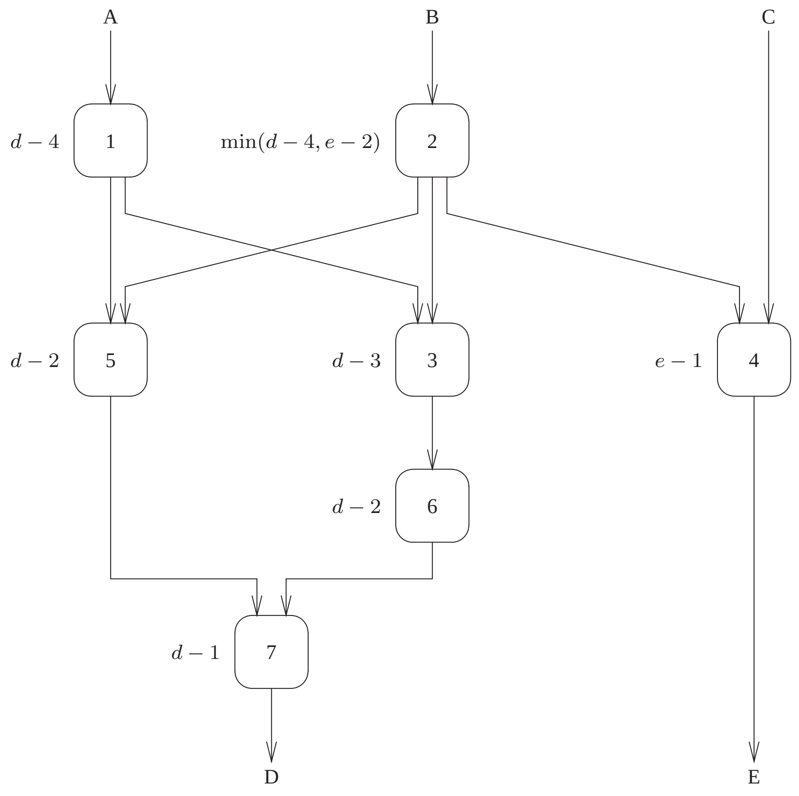


Figure 5.6: Algebraic ALAP scheduling

Real-World Scheduling A real-world scheduling system can generate any schedule between the ASAP and ALAP schedule. As long as the relative timing of the nodes fits between these extremes, the schedule will be as fast as a dynamic schedule. By carefully choosing the relative timings, the nodes may fit into a small number of groups of synchronised nodes, cutting down the number of synchronisation domains. Alternatively, a simpler system may just choose between an ASAP and ALAP schedule, depending on which provides the smaller synchronisation overhead.

The scheduling so far has ignored back-pressure. Real-world static scheduling must take this into account. Back-pressure prevents a node from taking in more data until its output has been accepted by the next stage. If a set of nodes are statically scheduled together, a single node waiting for its output to be acknowledged may now block a large number of other nodes that it is synchronised with. The use of buffers can prevent this situation.

One of the simplest and most common such situations is dealt with using a fixed buffer. In Figure 5.4, nodes 3 and 5 are scheduled together, with 6 the next cycle, then 7. There is an edge leading from 5 to 7, and if this is not buffered then 5 and 3 cannot read in new data until 7 takes in the new data. If a one-item buffer is inserted between nodes 5 and 7, each token will pass through the same number of edges, and so the synchronisation overhead can be reduced, and back pressure no longer caused internally. In general, if two nodes are scheduled a fixed number of cycles apart, a fixed sized buffer can be used.

5.2 Lenient Evaluation

One of the aims of implementing algorithms in hardware is to achieve high performance by extracting parallelism and using as much as possible of the available silicon simultaneously. However, the lazy evaluation of CONS nodes works against this goal, since stream items are produced on demand, serialising request, calculation and use of stream items.[2] In the case of a pipeline of stream mapping functions, a cascade of requests will travel from the output to the input, and the result must be passed back before any further processing of stream items occurs. This is poor utilisation of the hardware, since only a small part of the circuit is active at any time; the lazy evaluation of CONS expressions, although simple from a theoretical point of view, unnecessarily prevents parallelism.

However, it is possible to produce stream items in parallel with the main execution, so that if an item is available when a stream match occurs it can be used immediately. If CONS expressions are evaluated ahead of time, mapping operations may be pipelined, since multiple stages can now be active in parallel, making much better use of the hardware resources.

This evaluation of CONS nodes ahead of time is neither eager nor lazy evaluation, but *lenient* evaluation [144]. The differences in evaluation strategies can be illustrated with the following expression:

$$\textit{let } x = e_1 \textbf{ in } e_2$$

Eager evaluation evaluates e_1 first, and upon completion evaluates e_2 with the new binding. In a pure functional language we can evaluate e_1 in parallel with those parts of e_2 that do not require x, but the overall expression does not finish evaluation until both e_1 and e_2 have completed.

Lazy evaluation does not immediately evaluate e_1, but will evaluate e_2. The first time x is used in e_2, e_1 is evaluated, and the result saved for whenever x is required again. Evaluation of the full expression completes when e_2 finishes, and if x is never used e_1 is never evaluated.

Lenient evaluation computes e_1 and e_2 in parallel. If x is required in e_2, we wait until the result of e_1 is available. Evaluation completes when e_2 finishes. If x was required, the computation of e_1 has finished too. If x was not needed, we do not wait for e_1, and cancel any on-going computation. In effect, the value of x is calculated speculatively.

Lenient evaluation terminates in the same situations that lazy evaluation terminates, but decreases the latency required for a computation by increasing parallelism. It does not affect the pipelining of requests, since it only activates hardware that would otherwise be idle.

The difficulties with lenient evaluation mirror the difficulties of implementing dataflow processors (see Section 4.2.5), since lenient evaluation is the natural evaluation model for dataflow graphs. The problems there included garbage collecting unused values:

> The garbage collection of basic type values that are not used can still be simply implemented by waiting for the results to be produced and then using a kill node, since eager evaluation is still relied upon to produce basic values (the lenient evaluation of basic values is discussed in Section 6.3).
>
> To garbage collect streams, we must now be able to clear a circuit that is *actively evaluating*, if it turns out that further values from that graph are not needed.

5.2.1 Signalling on Lenient Streams: The “Push” Model

Under lazy evaluation, the stream buses use a demand-driven, or “pull”, model. Stream items are requested, which initiates computation, and the result is supplied when it is calculated (which also acknowledges the request). This model may also be applied to lenient evaluation.

However, an alternative “push” model is available. Since stream items are to be produced ahead of time, it is unnecessary to request the items. Instead, an item can be placed on the bus as soon as it is ready, along with a “data ready” signal. To read a stream item, we wait until the bus contains a stream item, and then acknowledge it.

[2]It has even been said of general lazy languages that “Lazy evaluation is more sequential than most imperative languages”, as the next reduction expression is unique under this model.

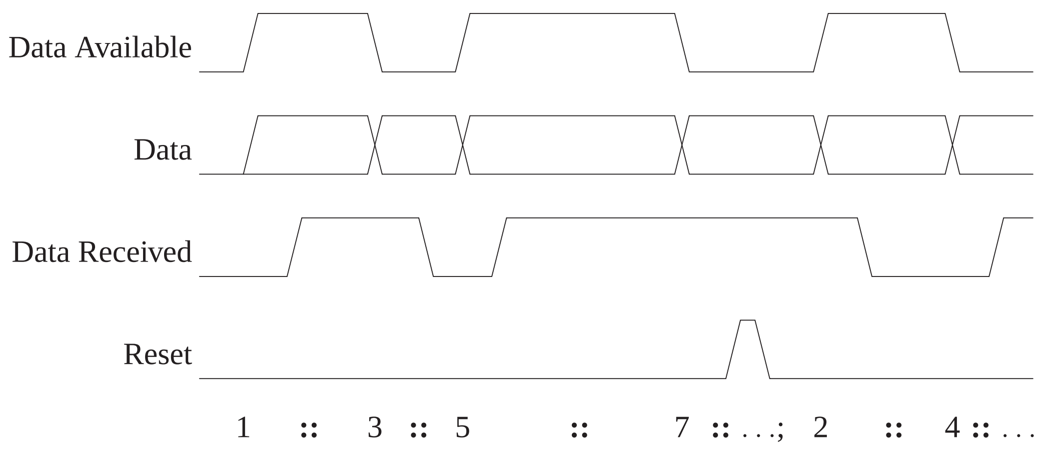


Figure 5.7: Example bus encoding of a lenient stream.

This difference in signalling may seem minor, but it reflects the underlying evaluation model, and will help simplify the underlying implementation. For example, it greatly reduces the complexity of implementing non-deterministic stream reads (as will be introduced in Section 7.1). It is also no longer necessary to ensure that the production of stream items starts before the reading of the stream, as reads now wait for data to become available. For the rest of this section we assume a push model for the stream buses.

A possible encoding of a *lenient* stream bus on physical wires is shown in Figure 5.7 (in contrast to Figure 4.2). In this diagram, the values are now produced before being consumed, although the resets marking a new stream are still signalled by the consumer. Note that the value “7”, although produced, is never consumed, since a reset is triggered first. It is not necessary to wait for a new value to be produced before triggering a reset.

5.2.2 Cancelling Lenient Evaluation

Under lazy evaluation, when a stream is no longer needed, it will be in an inactive state, and the stored values used to generate its next item are simply dropped. In comparison, under lenient evaluation some active computation must be halted. The CSP synthesis of Chapter 3 cannot be used with this model, as CSP provides no way to halt a running computation that is not performing I/O.

In the CDFG synthesis of Chapter 4, each stream bus has a “kill” line associated with it to inform the listening CONS node that no more items will be read, and that the hardware should be reset. The change is that now the hardware associated with the CONS node may be actively computing when the request arrives, and that all current computation associated with the stream should be cleared, rather than just clearing the stored values that would be used in producing the next item. Furthermore, newly introduced race conditions must be avoided.

Under basic graph synthesis, a CONS node is reset if its stream bus receives a reset, and the CONS node is currently active. An alternative approach is to reset a CONS node if it has been activated since the last reset on the stream, and it is a *head* CONS node. A head CONS node is one that is not enclosed by another CONS node that writes to the same stream, and so will represent the very head of a generated stream. When the head CONS node is reset, it should clear the state from all the CONS nodes it contains, too.

For any active CONS node, its enclosing head CONS node will have been activated since the last reset, since reaching the middle of a stream requires reading its head, and so any time the CONS node would reset the enclosing head CONS node will reset, clearing any computation the contained CONS node is performing. At the same time, since the head CONS node’s graph is solely used to generate the stream, resetting it will not clear any processing used for anything other than generating the stream. Furthermore, any streams used by the CONS node will either be generated within the enclosing head node, and therefore be reset when the head is, or are passed into the head node from outside, so that the head node will trigger the cascading of reset signals.

This new reset model is convenient for lenient evaluation, where a “has been activated” signal is simpler to generate than a “is currently activated” one, and reduces the possibilities for race conditions. These possibilities still exists when edges leave the subgraph of a head CONS node, in order to pass into a multiplexer—the reset signal may arrive while tokens are outside of the CONS node’s reset domain. To avoid this problem, we require multiplexer nodes that enclose head CONS nodes to perform the reset instead of the head CONS node.

In order to eliminate unnecessary reset circuitry, and share the same reset system between multiplexer nodes and CONS nodes, we separate the CONS node and the “reset box” which surrounds its subgraph. The reset box is now a separate graph node type, similar to the stream mutual exclusion boxes, and a reset box must surround any multiplexer or CONS node that is not enclosed by another multiplexer or CONS node that is also generating the same stream. The reset box node is activated when a token enters, and deactivated after it has been triggered by an arriving reset signal. Inserting these reset boxes is sufficient to correctly reset the system.

The reset box performs both the computational reset and reset cascading:

> For the computational reset, it is sufficient to reset the reset box’s subgraph to its initial state. All tokens and intermediate results are lost. Any stream item being pushed onto the stream bus should be dropped.
>
> To reset the streams used by the reset box, a reset signal is sent out on all the stream buses generated outside and used within the reset box, as linearity prevents these streams from being used elsewhere.
>
> Since evaluation may have been underway when the reset signal arrived, the stream buses being reset may already have been reset as part of computation. The enclosing reset box must therefore track which streams have already been reset, and only reset the remaining active streams. Otherwise the same stream bus may be reset multiple times, causing successive streams on the same stream bus to be killed. Using push-based streams simplifies the operation, as resets will not occur during stream requests.

As long as the circuitry for the reset boxes is well-designed, there should be no race conditions. When a reset box is reset *all* internal state should be cleared simultaneously, so that there are no internal race conditions. The forwarding of reset signals to stream buses that are used by the stream being reset may seem a possible source of problems. However, as long as stream item acknowledgements and stream resets are atomic, there should be no problem, and as stream dependencies form an acyclic graph, infinite loops of resets are not possible.

5.2.3 Basic Lenient Evaluation

Lenient evaluation cannot be achieved by simply starting the evaluation of CONS nodes as soon as they are reached. The ordering of stream items depends on lazy evaluation to ensure that only a single CONS node will reply to a stream request at a time (or send an item out, if the “push” model is used). If a CONS node activates the next CONS node before a stream request arrives, that next CONS node may try to provide the next stream item, too.

A simple solution is to only allow one stream item to be generated on a stream bus at a time, so that the items are produced in order. The next CONS node will start evaluation only when the preceding item is read. To implement this, each stream bus is provided with an activity line. Only a single CONS node is allowed to assert the line at a time, holding the line high from the moment it starts producing a stream item until the item is acknowledged. The next CONS node to be activated (which will produce the tail of the stream) waits until the line is clear before asserting the line itself and starting computation of the next

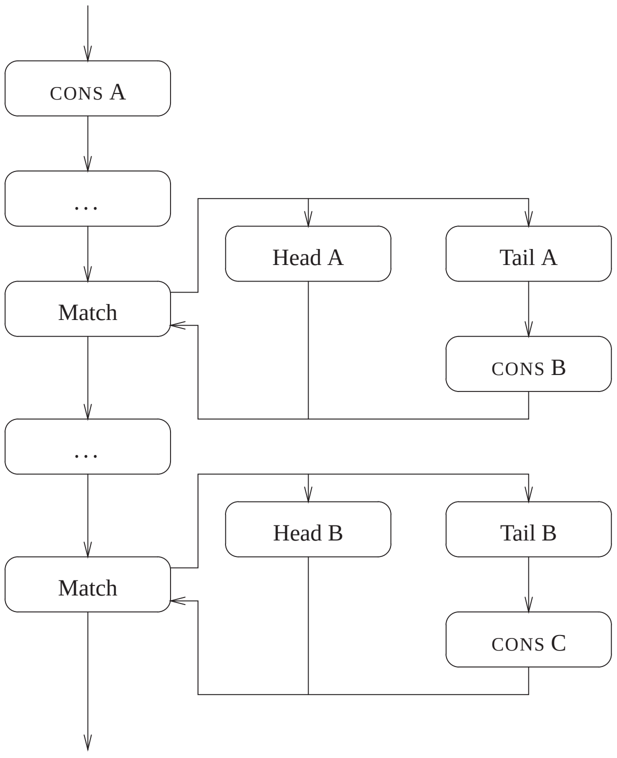


Figure 5.8: Node dependencies for lazy stream reading

stream item. The activity line provides mutual exclusion on the production of stream items. Note that it may be the same CONS node that is both producing the current stream item, and waiting for that item to be read before starting on the next one, having been called through a multiplexer.

The dynamic dependencies between stream processing nodes can be shown in a graph, where edges represent dependencies. For comparison, the dependencies between nodes for a simple stream program under lazy evaluation are shown in Figure 5.8. A stream is generated that is represented by the CONS node A, and then matched. Only when the match is evaluated are the head and tail expressions of A evaluated (the tail expression triggering CONS node B), and the head value returned. Another match is then performed, which triggers the head and tail expressions of CONS node B.

In comparison, the dependencies under basic lenient evaluation are shown in Figure 5.9. Activating the CONS node A triggers evaluation of its head and tail expression, including activating CONS node B, but evaluation of this node does not start yet. When the match occurs, it depends on both the head and tail expressions of CONS node A completing, at which point it returns a value, and triggers the evaluation of CONS node B's head and tail expressions.

While this model provides some basic lenient evaluation, and increases parallelism, only a single stream item may be produced ahead of time. Therefore, it is not possible to pipeline the production of items on a single stream. In a pipeline made by composing mapping functions (so that there are multiple intermediate streams), each pipeline stage produces a stream that can buffer one item produced ahead of time, as shown in Figure 5.10(a). However, if the same function is implemented as a single mapping of a complicated function, only a single item is buffered, and the processing is not pipelined, as shown in Figure 5.10(b). This limitation is addressed by the following sections.

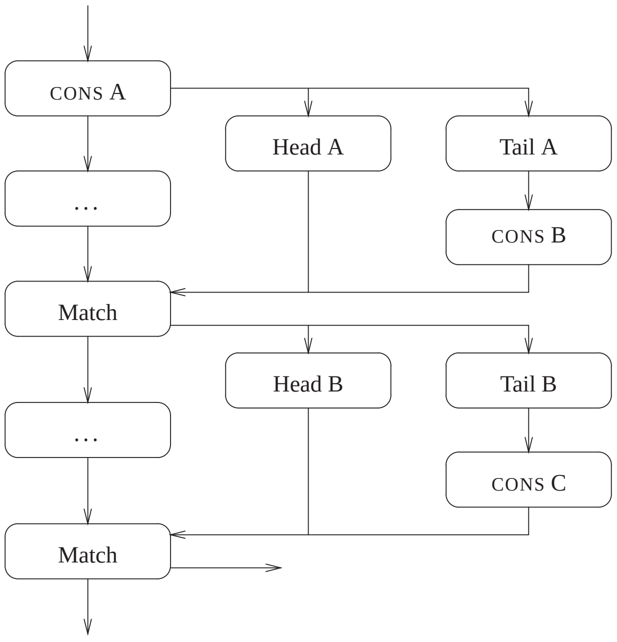


Figure 5.9: Node dependencies for basic lenient stream evaluation

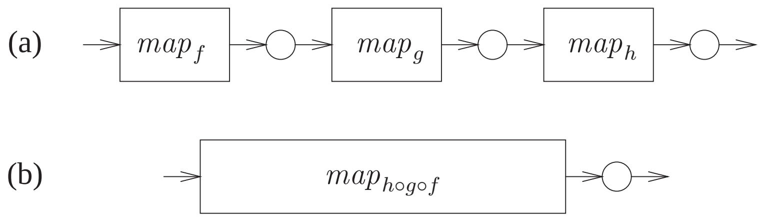


Figure 5.10: Performing mapping operations with a single item buffer

5.2.4 Lenient Evaluation with a Stream Bus Controller

In some programs, execution time depends on reading items from a stream as quickly as possible. In order to speed up execution, it is useful to pipeline production of stream elements. While this can be achieved with the basic lenient evaluation, by arranging the operations as, for example, a pipeline of maps, it would be useful to be able to pipeline a single map operation. This section explains how this may be achieved.

Ideally, every time a CONS node activates another CONS node, the new node would be able to immediately begin execution. However, this may produce stream items out of order. If a CONS node A triggers CONS node B but B produces its stream value before A completes, the values would be produced in an incorrect order. Our previous approach prevented this by only allowing a single CONS node on a stream bus to run at once.

A solution that allows more parallelism is to fit each stream bus with a *Stream Bus Controller* that collects stream bus data items in order, rather like a reorder buffer [66]. Requests to activate CONS nodes go through this stream bus controller, which has a FIFO listing which CONS nodes were activated in which order. If there is no space in the FIFO, back pressure is applied and the next CONS node is not allowed to start evaluating. Otherwise, the new CONS node is added to the FIFO's list and activated.

When an item is available from the CONS node on the front of the FIFO, it is output. When that item is acknowledged, the acknowledgement is forwarded to the CONS node, and that element is removed from the FIFO. If an item is available from the next CONS node in the FIFO, that is now output. In this way, stream items are returned in the order that the CONS nodes were activated. When a reset occurs on a stream the active CONS nodes are reset, and the buffer cleared.

If a CONS node reactivates itself, through a tail recursive call, that CONS node's subgraph will be processing pipelined requests to produce multiple stream items simultaneously. The CONS nodes and multiplexer nodes will need to be designed to cope with this. For example, the multiplexer may need to deal with multiple outstanding requests.

The dependencies between nodes under this model are shown in Figure 5.11. In comparison to Figure 5.9, there are no dependencies between one stream match occurring and the production of the next stream item, so the next item may be produced ahead of time. The stream bus controller is used to match up the Match nodes with their associated CONS nodes, in order.

If a stream is produced by a single CONS node, or static scheduling ensures that the CONS nodes will produce stream items in order, the stream bus controller can be eliminated. For the rest of this section we will assume that a stream bus controller or some other technique is used to ensure that stream items are collected in order, so that the in-order collection of stream items does not need to be addressed in the dataflow graphs.

5.2.5 Changing the Evaluation Model: Lazy Tail Matching

The basic evaluation model assumes lazy evaluation of CONS. Matching on a stream causes evaluation of the head and tail expression, and the variables are only bound when *both* expressions complete evaluation. The evaluation of the tail expression completes when it reaches the next CONS expression. The contents of that CONS node are only evaluated when the next stream read occurs. Under lazy evaluation, the binding of values coincides with the point at which stream evaluation stops.

However, for lenient evaluation, CONS nodes are evaluated before they are read from; the evaluation of the tail expression does not stop at the next CONS expression, but carries on into it. The original reason for delaying the return of the stream's head value until evaluation reaches a CONS in the tail expression has gone. In other words, the edges connecting the CONS nodes to the Match nodes in Figure 5.11 have no real use under lenient evaluation.

This unnecessary edge delays the returning of stream values until the tail execution path has reached a certain point in its evaluation, which may slow down computation, and also increases the complexity of

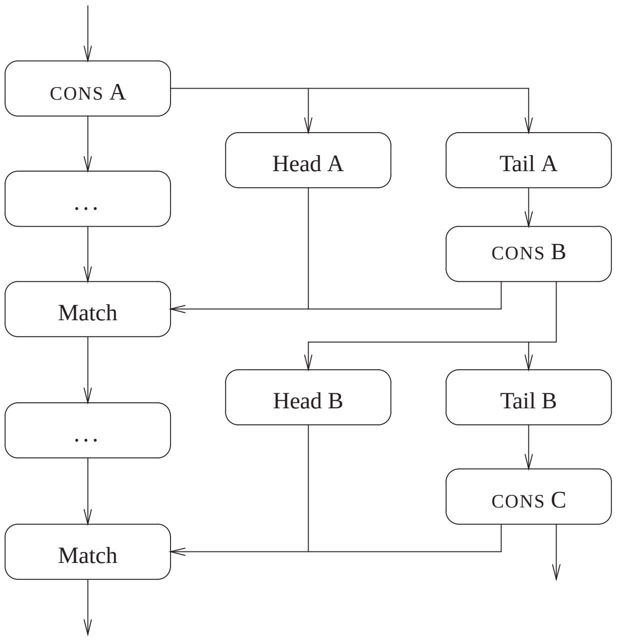


Figure 5.11: Node dependencies for lenient evaluation with a stream bus controller

the implementation.

An alternative lazy evaluation model, which we call *lazy tail matching*, treats the head and tail of the stream as separate lazy values. When a stream match occurs, only the head of the stream is evaluated, the tail remaining unevaluated. When the next stream read occurs, that tail expression is evaluated, to produce a CONS node containing a head expression and a tail expression. The head expression is evaluated to produce the next stream item, and the tail expression is again left unevaluated.

This approach represents a slightly different evaluation model. Using the symbol   to represent a computation that does not terminate, the original SASL semantics treat the values $(x::\ )$ and $(\ ::xs)$ as indistinguishable from $(\ ::\ )$. When the stream is matched upon, the matching only completes when both the head value has been produced, and the tail evaluation has reached the next CONS expression. Under lazy evaluation, this represents a point at which evaluation is suspended until the next stream request.

Under lazy tail matching, a stream matching completes if and only if the evaluation of the head part of the CONS expression representing the stream completes. Lazy tail matching can distinguish the value $(x::\ )$ from $(\ ::xs)$, and allows the reading of the head value x. Lazy tail matching increases the set of expressions that terminate.

The lenient evaluation model we use for the rest of this chapter is a lenient version of lazy tail matching. As in lazy tail matching, stream head values can be used as soon as they have been produced, but now evaluation of the stream's tail expression continues on in the background. A graph showing the new dependencies is given in Figure 5.12. This graph is similar to the one using a stream bus controller (Figure 5.11), except the matches now only depend on the head expression.

We use the function *toggle* to demonstrate how lazy tail matching affects the produced dataflow graphs:

$$\textbf{fun}\ \mathit{toggle}() = \mathit{True}::\mathit{False}::\mathit{toggle}()$$

The dataflow graph for basic lenient evaluation of CONS is shown in Figure 5.13. The dataflow graph is similar to that for lazy evaluation, except the reset box has been separated off and is now enclosed inside the mutual exclusion box. The graph for lenient evaluation using a stream bus controller is identical to

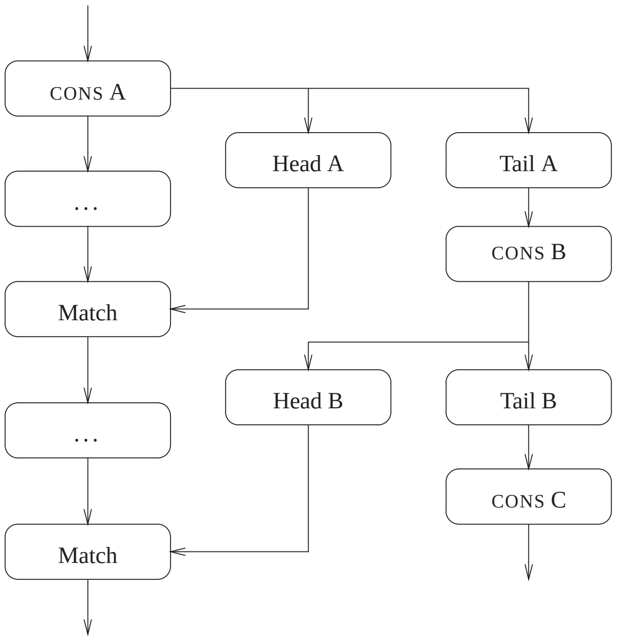


Figure 5.12: Node dependencies for lenient evaluation with lazy tail matching

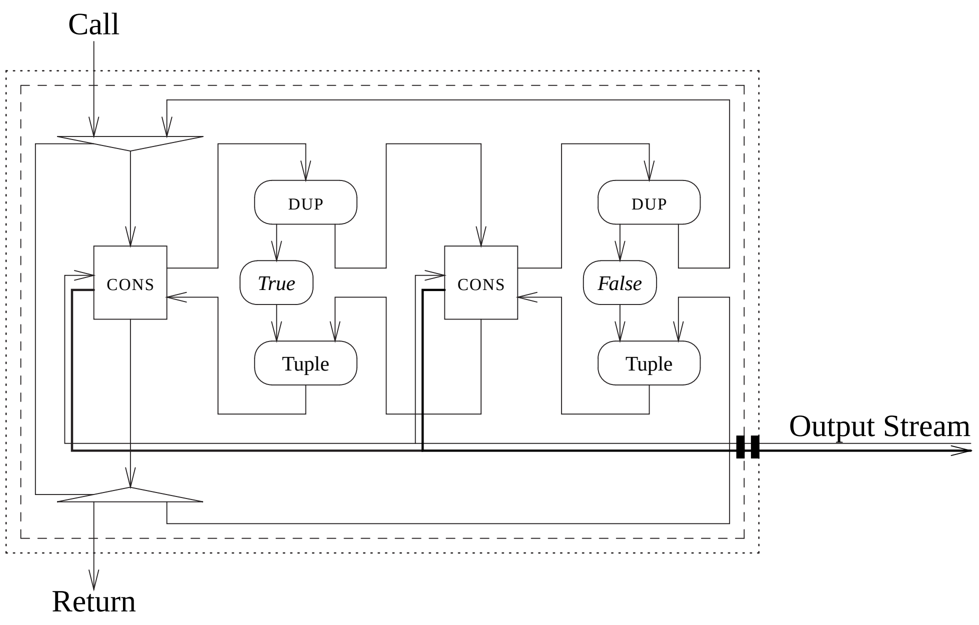
this, since the changes are only in the internal implementation of nodes and stream buses. Figure 5.14 shows the dataflow graph of the same program implemented with lazy tail matching. This simpler graph is no longer valid, as a number of dependencies on the tails of the stream are no longer needed—the output edge of CONS nodes are often not used, and this allows the graph to be rearranged, as explained in the next section.

5.2.6 Rearranging Graphs for Lazy Tail Evaluation

Under lazy tail evaluation, the only CONS nodes whose outputs are used are head CONS nodes (that is, those that are not enclosed as the subgraph of another CONS node writing to the same stream). Under basic lenient evaluation, if the CONS node is enclosed in the subgraph of another CONS node, its output edge would be used to signal to the enclosing CONS that the tail expression has finished evaluating. However, with lazy tail matching, this information is no longer used. Those CONS nodes that are directly enclosed in multiplexers have their output edge forwarded out of the multiplexer. If a multiplexer is enclosed in another CONS node on the same stream, that output edge is not used, either.

The edge returned by a CONS node or multiplexer is only used if it is not enclosed in another CONS node on the same bus. A multiplexer that is not enclosed by a CONS nodes on the same bus simply returns the value produced in its subgraph to the outside. If the value were not needed outside of the subgraph, its creation inside the subgraph could be eliminated too. Each multiplexer and CONS node used to produce a stream is enclosed in reset box (see Section 5.2.2). By modifying the reset boxes so that they do not expect an edge to be output from their subgraphs, the output edges of all CONS and multiplexer nodes are eliminated. Instead, reset boxes return a token as soon as they have been initialised after receiving a token.

This is a safe transformation. Under the “pull” stream model, the output edge from a CONS node signalled that the CONS node has been activated, so that it would be safe to read from or reset the stream. Under the “push” stream model, it is not necessary to wait for the first CONS node to become activated before reading (as long as the previous stream has been reset), and so the output edge of the reset box only needs to signify that the stream is ready to be reset. The reset boxes can provide this signal, even if



Key: Dashed box is reset, dotted box is mutual exclusion.

Figure 5.13: Basic graph implementation of *toggle*

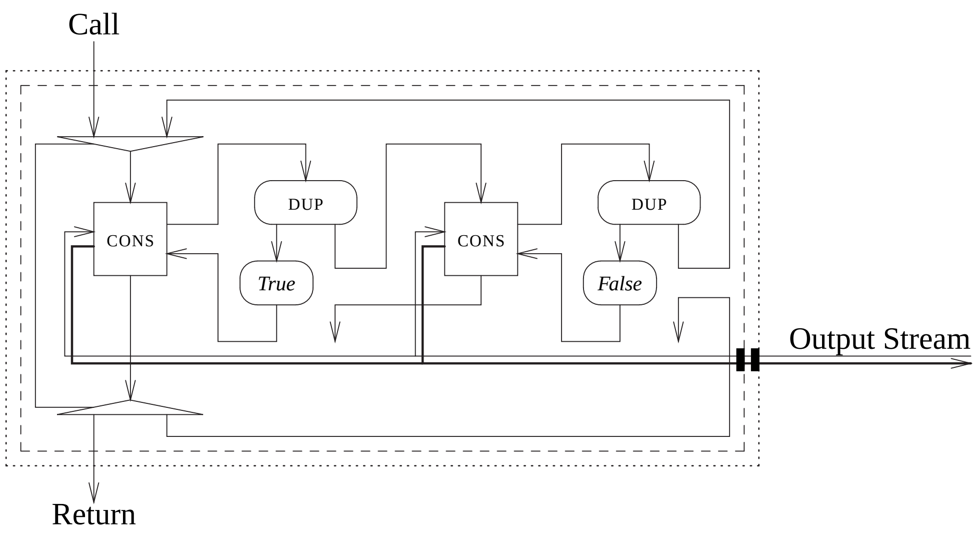


Figure 5.14: Lazy-tail-evaluating graph implementation of *toggle*

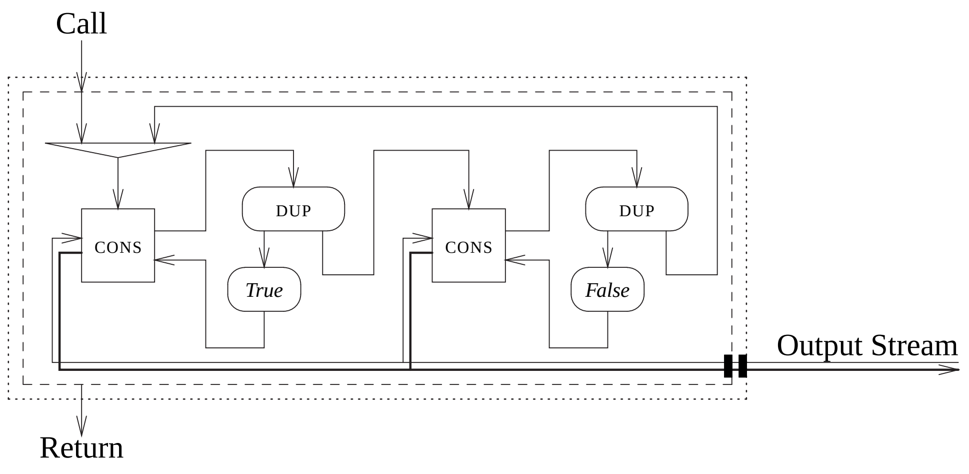


Figure 5.15: The function *toggle* implemented using the new nodes

the enclosed CONS nodes and multiplexers have not yet been activated.

The transformation subtly increases the amount of lenience in the language, since the reset box may now signal readiness before execution reaches a CONS node. In effect, all functions that generate a stream through CONS-enclosed tail calls (and thus get implemented using a multiplexer) are leniently evaluated. The output edges of multiplexers and CONS nodes may also have been used as the synchronisation edges for kill nodes. This can be eliminated by moving the dependence to the edge going *into* the node instead.

Since the output of multiplexer nodes are no longer used, the second half of the node, which collects a token and forwards it to the appropriate destination, can be removed. The multiplexer now only needs to multiplex tokens in, and not demultiplex them out. The new graph for *toggle* is shown in Figure 5.15.

The other node types that have subgraphs (and therefore may need redesigning) are conditional nodes and iteration nodes. Different forms of these nodes must be used with subgraphs that generate a stream, but do not return results:

The return-less conditional node is the top half of the existing conditional node. A conditional token selects which output the input token should be passed to. The subgraphs return no values (although they may send a token back to an enclosing multiplexer, if there is a tail call). This is used to replace the original conditional node when used in a stream-generating graph.

The iteration node type is used to implement tail calls that are not enclosed in a CONS expression. For functions that return a stream, under lazy tail evaluation, a direct tail call is now treated identically to a tail call through a CONS—the edge is connected back to the enclosing multiplexer.

To summarise the changes to the dataflow graphs, the old multiplexer node has been replaced with a top-half-only version, CONS nodes no longer produce an output token, the “reset box” has been taken out of the CONS node and become a separate node, and a new top-half-only conditional node has been introduced.

Example As an example, the *skip-blanks* function of Figure 4.25, whose original dataflow graph is shown in Figure 4.29 is redrawn in Figure 5.16 to use lazy tail evaluation. The graph is now much simplified, removing many of the edges that caused unnecessary dependencies. The conditional expression is implemented using the top-half-only conditional node, both the direct and CONS-based tail recursion go through a top-half-only multiplexer, and the whole graph is enclosed in a reset box.

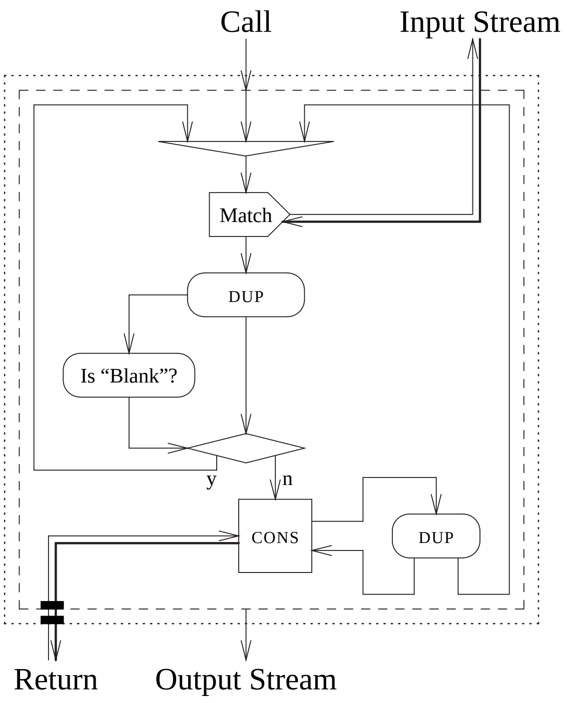


Figure 5.16: The graph for *skip-blanks* under lazy tail evaluation

5.3 Program Transformation

A wide range of functional-style optimisations (such as those in [15]) may be applied to SASL source, although their effectiveness may be somewhat different when applied to programs that will be synthesised to hardware. The folding and unfolding of functions [78], for example, allows an area/parallelism trade-off.

This section discusses a number of optimisations that can be performed on SASL dataflow graphs. The basic intermediate dataflow graph is well suited to optimisation, providing more of the final structure than the initial syntax tree gives, and having more flexibility than the low-level CDFG (since the construction of the recursive calls and stream buses are not specified at that stage). Only graph-level optimisations are covered. Performance improvements that depend on the low-level implementation, such as lenient evaluation and static scheduling, were discussed earlier in this chapter.

5.3.1 Enabling Graph Optimisations

In order to effectively optimise a dataflow graph, it is necessary for the graph to accurately represent the computation. The basic dataflow graph format has a number of deficiencies, in that it can suggest dependencies where none exist. Difficulties include:

Kill nodes may suggest a dependence between the value being killed and the value used to synchronise the elimination.

Tuples (and algebraic datatypes) may suggest a dependence between the elements of the tuple, where none exists in practice.

A single operation may be distributed among a number of nodes, hiding the true nature of the operation.

The rest of this section discusses transformations that may be applied in order to reduce these difficulties. The notation for the graph transformations in this chapter uses dotted boxes labelled with letters to

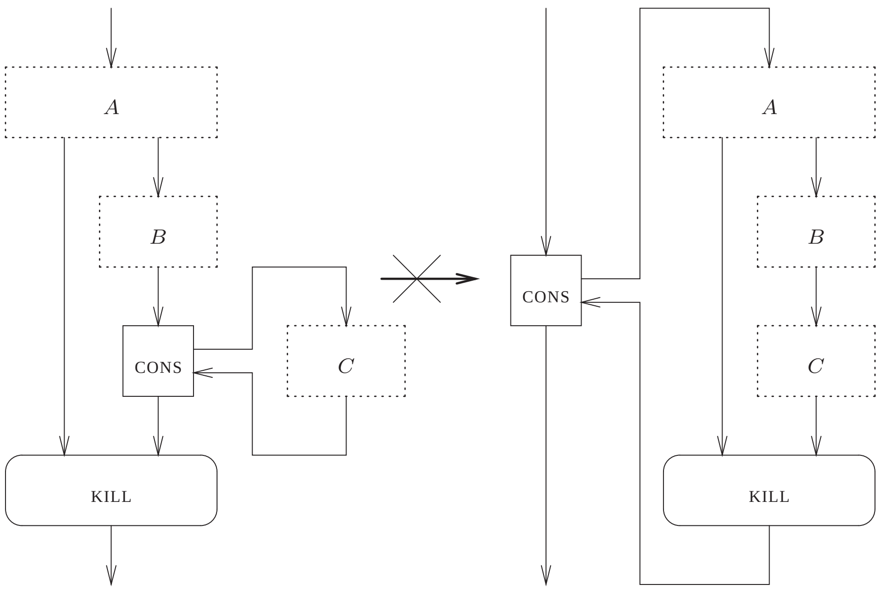


Figure 5.17: A failed graph transformation due to a KILL expression

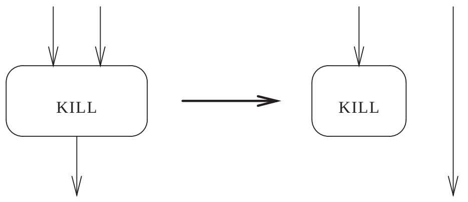


Figure 5.18: A transformation to make KILL nodes more amenable to graph transformations

represent subgraphs. Since the optimisations rely on the dependencies between subgraphs, the notation uses single edges between nodes to represent what could be a set of edges between subgraphs in an actual dataflow graph.

KILL Expression Elimination The use of KILL nodes may introduce an artificial dependency where a value depends on a result that is being killed, but is not otherwise used in the computation. For example, Figure 5.17 shows a valid transformation that increases lenience, but that cannot be performed using a simple transformation. The values produced inside A and B are only used by the CONS node, and so they could be moved inside the node, except the kill node makes it appear that the value being killed also depends on the output of the CONS node. Subgraph B may be moved inside the CONS node, but A is blocked unnecessarily.

One solution is to recognise these "fake dependencies" produced by kill nodes, and to ignore them. A more general approach is to remove the "synchronisation edges" used by kill nodes, as shown in Figure 5.18. The nodes may then be freely moved about, and if the same termination semantics are required the node may then be reconnected to an appropriate edge of the graph, such as an output edge of the subgraph it is in, effectively performing Figure 5.18 in reverse.

Alternatively, the edge can be left unsynchronised. This means that in expressions like **let** $x = e_1$ **in** e_2, the expression may finish evaluating before e_1 does, and after a value is returned the hardware may continue to perform computation. If the program terminates, however, the results will not be affected, and the transformation may allow more parallelism. A similar transformation may be applied to stream reset nodes.

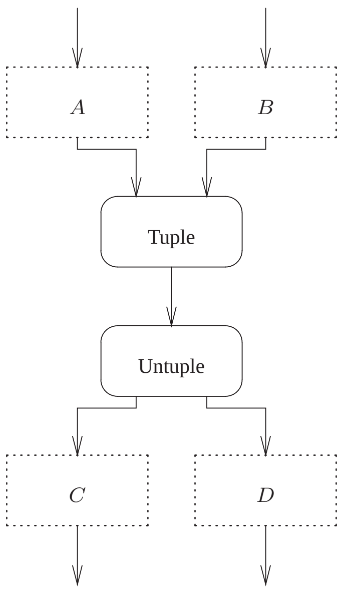


Figure 5.19: Tupling nodes may introduce unnecessary dependencies

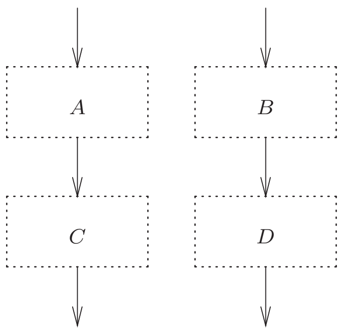


Figure 5.20: Unnecessary dependencies can be removed by eliminating tuple nodes

Removing Tupling Expressions Tupling values together in the dataflow graphs removes their independence, and creates possibly unnecessary synchronisation between the items in the tuple. For example, in the graph shown in Figure 5.19, subgraph D appears to depend on A, and C on B, unnecessarily. An improved version is shown in Figure 5.20.

For tupling nodes followed by untupling nodes, the transformation is simple, with the tupled value edge being replaced by a set of edges representing the components. Normal nodes can generally be rearranged to deal with a set of input edges, rather than a single tupled edge, and the node could be split into a set of normal nodes which work on different parts of the tupled value, increasing parallelism.

The transformation becomes more complex when the tupled value is used by function calls, conditionals or CONS nodes. Such nodes expect and return a fixed number of edges, and use tupling and untupling nodes to take a number of data items and transfer them to the subgraph. These tupling and untupling nodes can be made implicit, so that, for example, conditional nodes take a set of edges in, and each conditional subgraph gets a matching set of edges supplied to it. This allows graph transformations to be simplified by removing the need to deal with tupling nodes, as well as removing unnecessary dependencies. Once the optimisations are complete, tupling and untupling nodes can be reintroduced around function calls, conditional nodes and CONS nodes, but otherwise they can be eliminated, allowing greater flexibility in the scheduling of operations.

Similar transformations can be applied to the components of algebraic datatypes, so that the tag information does not need to wait for the components of the datatype, and the parts can be transfered independently. This transformation may have a cost associated with it, in that many more edges are introduced, leading to more dynamic scheduling, which may reduce the maximum speed and increase area requirements. Static scheduling (from Section 5.1) should be able to remove most of these edges, greatly reducing the overhead.

Associative Node Normalisation Many graph transformations can be applied by performing pattern matching on the graphs (although optimal graph covering is NP-complete, it is still useful to heuristically apply graph transformations in this way).

There are many ways in which n-adic associative operators can be expressed by chaining together diadic operators, which all produce the same result. Common examples of these nodes include addition and various logical operators, as well as the DUP nodes. For some of the more complex pattern matching substitutions to be effective, they must recognise these patterns.

In order to simplify these optimisations, sets of associative operators that work together to form a single operation can be merged into a single compound node. This node can then be split up as necessary when matching for graph transformations. After the graph transformations have been applied, the nodes will have been rearranged, and new sets of associative nodes may be merged. When synthesising, the node can be split up in such a way as to optimise for latency, based on static scheduling information.

An example graph is shown in Figure 5.21(a). The chained-together addition nodes are merged together to give Figure 5.21(b). Pattern-matching for common sub-expression elimination (see Section 5.3.2) can break down the triadic addition as shown in Figure 5.21(c). The final optimised graph is shown in Figure 5.21(d).

5.3.2 Peep-hole Optimisation

Once the graph is converted to the form described in the previous sections, a number of basic software-like optimisations become simple. These include:

Dead Code Elimination Subgraphs whose results are solely used by a KILL node produce no useful value, and if the function returns a value, it will not be affected by the computation done by the subgraph. Eliminating these subgraphs will reduce resource requirements, and may reduce computation time, but the optimisation also affects termination, under eager semantics (under lazy semantics termination is the same, as the value is never needed, and so never computed). We assume that useful programs are productive, and so this is not a problem. The graph transformation is shown in Figure 5.22. The graph can then be tidied further by applying the transformation shown in Figure 5.23.

Common Subexpression Elimination For a subexpression to be repeatedly calculated, the values it is calculated from must be duplicated, and the same operation applied to each copy. The transformation to remove the calculation of common subexpressions is shown in Figure 5.24. This does not cover common subexpressions where one of the expressions is recalculated inside a conditional or loop. For this, the expression must first be pulled out, using a transformation such as one of the ones described below.

Strength Reduction Some combinations of nodes may be replaced by other combinations of nodes that produce the same result, but take less resources, or have a smaller latency. Simple graph substitution rules can be used to perform this strength reduction, such as the example shown in Figure 5.25.

As well as these simple examples, more complex transformations, some specific to hardware synthesis, can be performed. The rest of this chapter discusses a few of these.

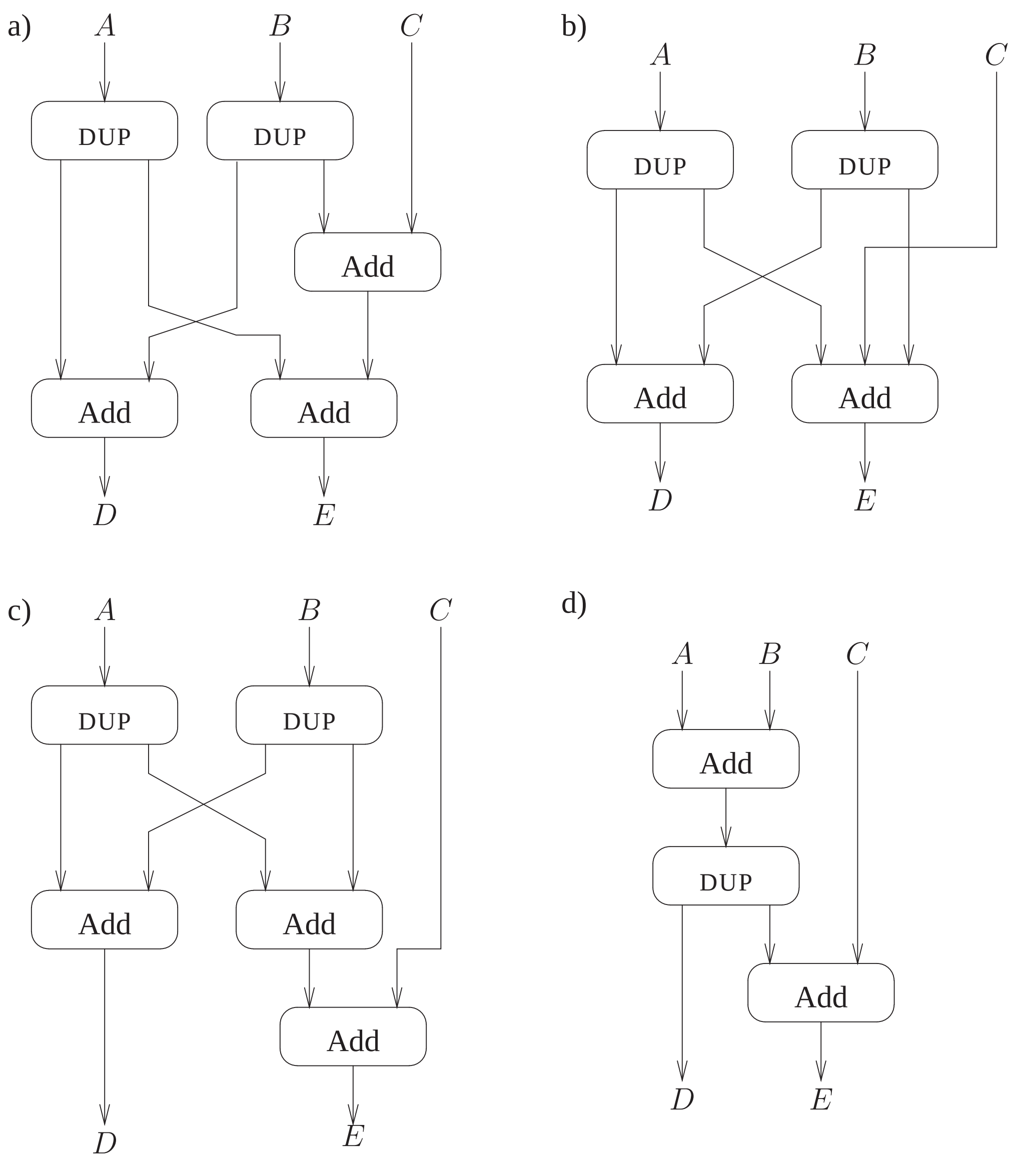


Figure 5.21: Graphs representing the use of merging associative nodes for optimisation

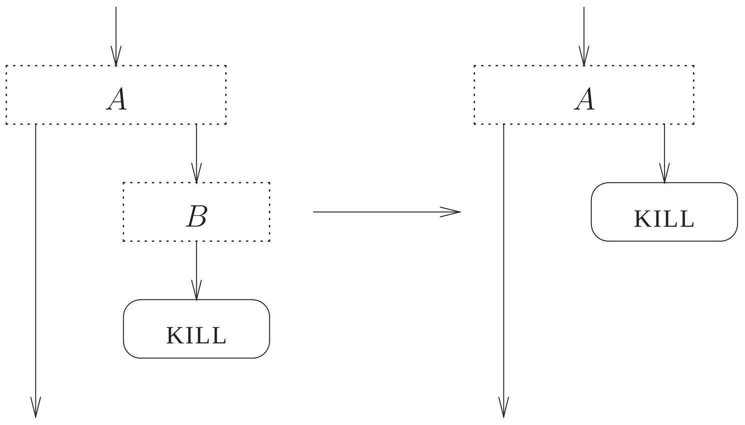


Figure 5.22: Dead code elimination transformation

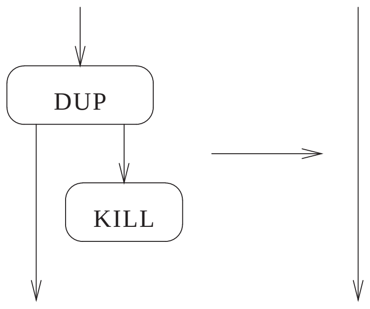


Figure 5.23: Elimination of DUP/KILL pairs

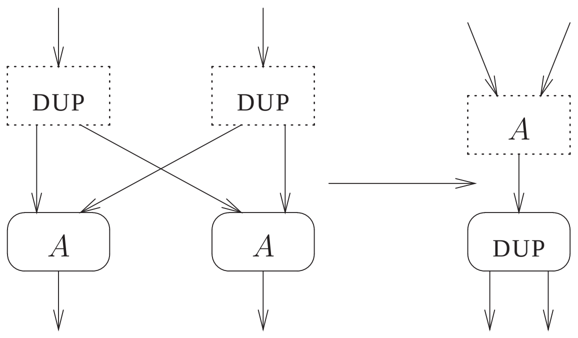


Figure 5.24: Common sub-expression elimination transformation

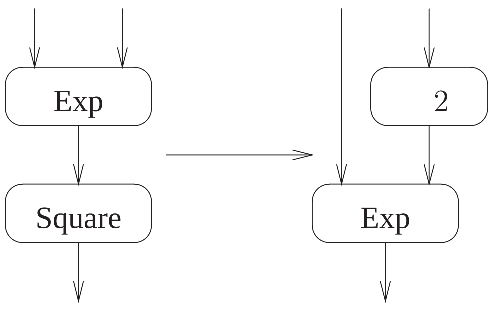


Figure 5.25: An example of a strength reduction transformation

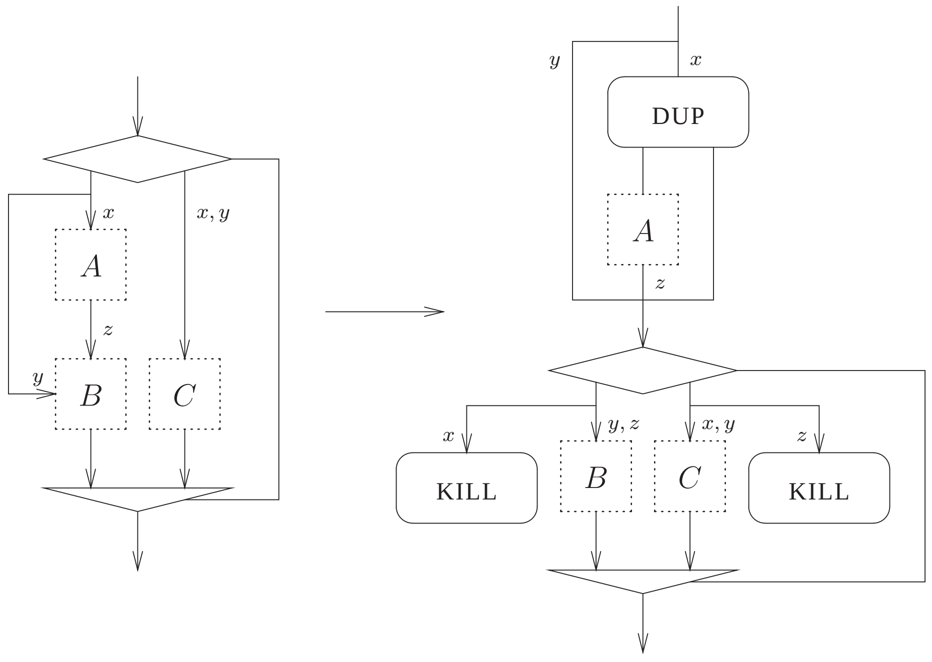


Figure 5.26: Deconditionalisation of a subgraph

5.3.3 Flattening Conditionals

The basic conditional node waits for all inputs to be available before triggering the appropriate conditional subgraph. If the parameters to the subgraphs are available long before the condition value which selects which subgraph to use, computation may be being held up unnecessarily. By moving some parts of the conditional subgraphs out of the conditional node, those parts may be executed ahead of time, which could reduce the overall latency.

This is equivalent to rewriting the expression

$$\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3$$

as the expression

$$\textbf{let } x_2 = e_2 \textbf{ in let } x_3 = e_3 \textbf{ in if } e_1 \textbf{ then } x_2 \textbf{ else } x_3$$

This transformation is also used in software, where it is known as *if conversion*. It allows the elimination of expensive branches, by the use of *conditional move* instructions [66]. All paths are executed, but only the results of the required path are used.

An example of the graph transformation is shown in Figure 5.26, which de-conditionalises the subgraph A. The introduction of the DUP and KILL nodes makes the transformation look rather more complicated than it actually is, and a number of these introduced nodes may be optimised away in turn.

After this transformation, the conditional node will need to wait for the output of subgraph A before activating a subgraph, so this transformation is best used if moving the subgraph A out of the conditional node does not extend the critical path. Some static scheduling is required if this is to be checked. If the same subgraph is removed from multiple conditional cases, common subexpression elimination can be performed.

However, not all node types may be moved out of the conditional node in this way. Nodes that may fail to terminate, such as recursive call nodes and stream match nodes, must not be pulled out. Also, nodes that take in linear values cannot be pulled out, because the deconditionalising transformation introduces a DUP node that would try to duplicate them. Subgraph A in Figure 5.26 is therefore constrained to not

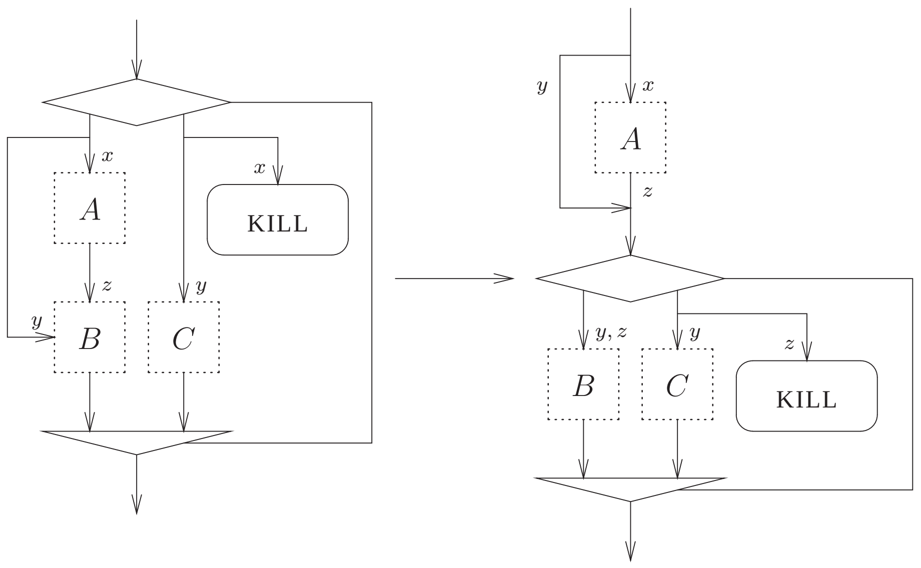


Figure 5.27: Deconditionalisation of a linear-variable-using subgraph

contain unbounded-time or linear-value-reading nodes.

Figure 5.27 shows a graph transformation that permits the deconditionalisation of a subgraph A that reads linear values, as long as it does not take unbounded time. If there are more than 2 conditional arms, all paths but the one containing A must not use the linear value, and are transformed as the right-hand arm is. Figure 5.28 is simply common subexpression elimination across a conditional, and works even if the subgraph A includes unbounded computations or reads from linear values (the graphs A must be identical, but the B_i may all differ).

If a form of lazy evaluation is used (see Section 6.3), it is possible to deconditionalise all the conditional computation, including unbounded loops and access to linear variables. After deconditionalisation, the linear values may pass through DUP nodes, but dynamically the value will not need be used more than once, as only the closure representing the conditional path that is taken will be evaluated. Some form of guards must be present if the lazy values are evaluated leniently, to prevent the linear values from being used multiple times during the lenient evaluation.

This transformation is part of a more general set of techniques where subgraphs are moved through the enclosing node. For example, Figure 5.17 shows an attempted transformation where nodes are pushed inside of a CONS construct.

5.3.4 Removing Conditional Nodes

As mentioned in the previous section, lazy evaluation allows the computational parts of a conditional arm to be made non-conditional. The same can be done with eager evaluation, provided the graph takes bounded time and does not access linear values. Once all the conditional arms of the node have been reduced to selecting an input and returning it, killing the unused values, the conditional node can simply be replaced by a normal node which reads an item from each of the inputs inputs and (depending upon the conditional value) returns one of them—that is, a multiplexer. This is equivalent to generating a *conditional move* instruction when performing if conversion.

This transformation makes it possible to implement low-level logical operations efficiently without resorting to providing logic primitives. Functions representing logical “and”, “or”, “not” and so on can be defined in terms of **case** expressions where the expression to be matched upon returns a boolean value. The conditional node that is synthesised can then be removed, replacing it with a 2-input multiplexer. Strength reduction can then reduce subgraphs containing these multiplexers to nodes representing logic

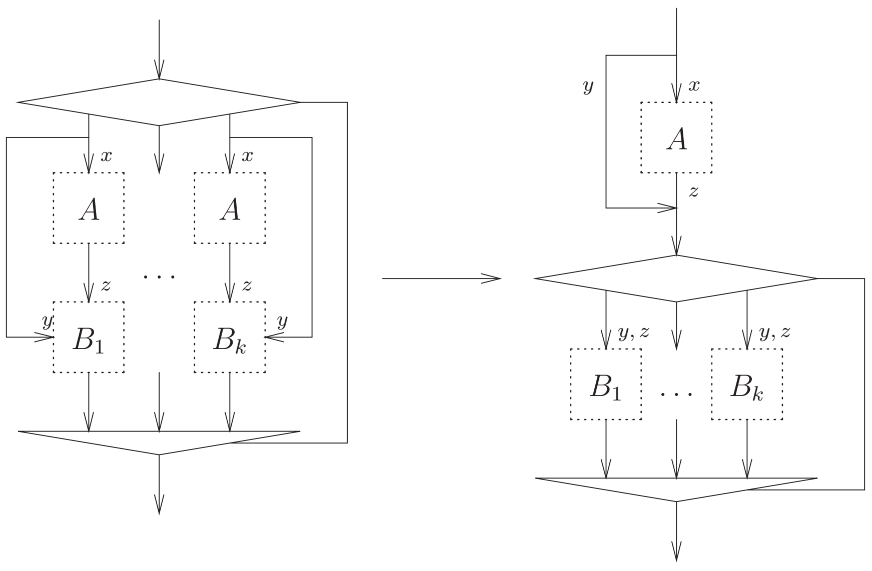


Figure 5.28: Deconditionalisation of a common subexpression

gates.

5.3.5 Unrolling Loops

In software, loops may be unrolled. This is often done to reduce the overhead of the conditional expression on the loop, so that more time is spent in computation than is in testing the loop variable. It also allows more flexibility for instruction scheduling so that, for example, loads may be moved further away from the point at which the value is used. In hardware, unrolling the loops has a different aim. Although static scheduling may be improved in an unrolled loop, the main differences are that the synthesised circuit should provide more parallelism, but will also use a larger area.

Simple implementations of looping hardware will only allow a single set of tokens to go around a loop at a time. More complex implementations may allow multiple sets of tokens running in the loop at a time, so that use of the loop is effectively pipelined. If the ordering of items is unimportant (see Section 7.3), it is relatively simple to allow multiple sets of data in the loop simultaneously. If multiple sets of data can be processed by the loop in a pipelined manner, the increased number of pipeline stages allow more parallelism. The loop may be unrolled by creating multiple copies of the subgraph, connected sequentially. Conditional expressions may be needed to skip fractional iterations of the unrolled loop.

5.4 Summary

This chapter has focused on optimising the performance of the graphs created by the previous chapter, both by optimising the graphs themselves, and the way in which they are synthesised to low-level hardware. More specifically:

Basic static scheduling has been added, as a way of removing a certain amount of synchronisation overhead.

Lenient evaluation increases the amount of parallelism available by letting the lazily-evaluated parts of a program calculate before they are required, without adversely affecting the termination characteristics.

Graph-based program transformations use the flexibility of the graph's format to minimise the work performed, and reschedule when nodes are evaluated in order to improve performance.

The static scheduling and program transformations can be treated as independent of the rest of this thesis. The lenient evaluation model, however, provides the groundwork for the next chapter's *closures* and *promises*.

CHAPTER 6

Closures and Statically-Allocated Laziness

Basic SASL does not support closures, higher-order functions or any form of laziness beyond the explicit lazy CONS expressions. The language was originally intended to act as a simple statically-allocated language in which to experiment with stream-based I/O. Closures and laziness could lead to unbounded storage requirements, and so were disallowed.

Lazy lists can also lead to unbounded storage requirements. SASL manages to restrict them to bounded storage, and we can attempt to apply the same techniques to closures. Higher-order functions are one of the more important features of functional programming languages; however they're often just used as a "macro" mechanism. Section 6.1 discusses this approach, gradually extending it to a more general concept of statically-allocated closures. Section 6.2 takes another tack, based on allowing the closure to evaluate leniently. Section 6.3 then uses the work on closures as a starting point to implement lazy evaluation. The final section of this chapter summarises its contents.

6.1 Higher-order Functions as Macros

A common use of higher-order functions is to simply provide a function template, into which a particular operation can be inserted. For example, the common *map*, *filter* and *fold* list operations are generally defined as higher-order functions which take a function that describes what is to be done to each element. The same result could be achieved by textually substituting the argument function into the original body. A software implementation may or may not do this inlining: the performance improvement must be weighed against the code bloat of producing all the *specialised* functions. However, this issue does not arise with unshared hardware implementations, since all non-recursive call sites are unfolded anyway.

A simple implementation of higher-order macros can be achieved by:

Adding a function type, $(\sigma_1 \quad \sigma_2)_F$, where F is a function identifier representing the particular function. Types with different function identifiers are not equal. The function identifiers in a parameter are treated like polymorphic type variables. Function types are not allowed at the top level. In this way every function value in the unfolded program is associated with a single function. Functions may not be stored in streams.

Creating function access expressions. Top-level function names can be used to create function values. By not supporting nested functions, we do not need to cope with scoping issues (cf combinators).

```
(* Higher-order functions. *)
fun map(f, x::xs) = f(x)::map(f, xs)
fun genmap(f, g) = map(f, g())
fun toggle() = True::False::toggle()
fun invert(x) = case x of True    False    False    True
fun main() = genmap(invert, toggle)

(* Expanded functions. *)
fun map-invert(x::xs) = invert(x)::map-invert(xs)
fun genmap-invert-toggle() = map-invert(toggle())
fun main-expanded() = genmap-invert-toggle()
```

Figure 6.1: Some simple higher-order functions and their macro-style expansions

Changing the function application expression to take a function value and a parameter, rather than just a specific function and parameter.

Constraining recursive calls so that the same functions values are always fed back as the same parameters. This is the closure stability constraint, similar to the stream stability constraint.

Requiring call graph structure constraints to be maintained. Checks must be added to prevent a function f from generating a function value for a function that (possibly indirectly) calls f. The only exception is in creating a function value corresponding to itself for the purpose of performing a tail call to it in a tail position.

These limitations make it easy to implement macro-style higher-order functions, which can be simply expanded out at compile time. By expanding out the functions at the syntax tree or basic dataflow graph stage it is possible to leave the rest of the compilation process unaltered.

Examples Some simple higher-order functions and their macro-style expansions are shown in Figure 6.1.

6.1.1 Nested Function Definitions

Plain SASL does not include lambda abstractions, or other ways of nesting functions. Without values that hold functions, nested function values are of little use. Now that (limited) closures have been introduced, nested functions provide little challenge. Lambda lifting [76] can convert nested function definitions that use static scoping into a set of top-level function definitions.

The idea is to convert variables that are used by a function, but defined outside of its body, into extra parameters. For a full higher-order language, these extra parameters would be curried, and the function partially applied. This can be simulated by making the value representing the function also hold the values that would be used as the extra parameters. Nested functions are generated with the syntax **lambda** $x \quad e$, where x is a variable and e an expression. The result is a function value.

Now that the functions have an environment associated with them, it is possible to store function values within function values. To ensure bounded data structures, we must prevent arbitrarily deeply nested closures. The only way these structures may be created are through recursive calls. The same problem occurs with streams, which lead to the introduction of the stability requirement of Section 2.4.3. The *closure stability constraint* requires that, for recursive calls, the closures supplied to recursive calls must be the same ones that were supplied to the function in the original call. This ensures the closures remain statically allocated.

```
(* Higher-order function. *)
fun check-range(lower, upper, stream) =
    map((lambda x    lower    x and x    upper), stream)

(* Expanded function. *)
fun map-check-range((lower, upper), x::xs) =
    (lower    x and x    upper)::map-check-range((lower, upper), xs)
```

Figure 6.2: Closures that use their environment, and their expansions

Once environments are introduced, it is possible to store a stream inside a closure. A simple solution is to disallow stream values within the environment of closures. A more complex solution allows such values, but then treats such closures as if they were tuples containing streams: they must be used linearly, and cannot be enclosed within streams or algebraic datatypes. Nesting of such closures is allowed, as nesting of tuples is allowed. When one of these closures is killed (that is, when the value becomes non-live, and is explicitly destroyed in Linear SASL), the enclosed streams must be reset.

Example An example function, and its expansion are shown in Figure 6.2.

Curried Functions and Mutual Recursion The introduction of lambda abstractions allows us to create curried functions. A simple example is the curried version of *map*:

$$\textbf{fun}\ \textit{map}\ f\ (x\mathbin{::}xs) = f(x)\mathbin{::}\textit{map}\ f\ xs$$

This is syntactic sugar for:

$$\textbf{fun}\ \textit{map}(f) = \textbf{lambda}\ s \quad (\textbf{case}\ s\ \textbf{of}\ x\mathbin{::}xs \quad f(x)\mathbin{::}\textit{map}\ f\ xs)$$

Converting to top-level functions gives the following (with the square brackets as intermediate form notation for environment values):

$$\textbf{fun}\ \textit{map}(f) = \textit{map-2}[f]$$
$$\textbf{fun}\ \textit{map-2}[f](s) = \textbf{case}\ s\ \textbf{of}\ x\mathbin{::}xs \quad f(x)\mathbin{::}\textit{map}\ f\ xs$$

These functions look like they may be mutually recursive. However, when the program is unfolded the calls to function values become calls to fixed functions. It is no longer necessary to pass around the function values to identify the function to call, but the environment associated with each function value still needs to be passed around, so the function values are replaced with environments. For example, given a function g, *map* becomes specialised as follows:

$$\textbf{fun}\ \textit{map}_g(e) = e$$
$$\textbf{fun}\ \textit{map-2}_g(e, x\mathbin{::}xs) = g(e, x)\mathbin{::}\textit{map-2}_g(\textit{map}_g(e), xs)$$

The parameter e supplied to $\textit{map}_g$ represents the environment of f in the original function. The value $\textit{map}_g$ returns is the environment of $\textit{map-2}_g$. The function $\textit{map-2}_g$ takes an environment as well as its parameters. Only the final application in the tail position causes the actual tail call itself, which is directly recursive, so mutual recursion is not required.

We may be tempted to extend the language to allow "**let rec**" style bindings, allowing nested recursive functions. If we prevent enclosed functions from calling enclosing functions, the conversion to a top-level only form is not complicated. However, if we allow the enclosed functions to tail-call enclosing functions, mutually recursive functions may be introduced.

Although the synthesis of mutually recursive functions is not discussed in depth in this thesis, it is quite possible to implement in SASL. One approach would be to move to a more continuation-based approach, so that each function is a state in a state machine, and tail calls become state transitions. Trampolines (as explained in Section 4.3.2) can be used for similar effect. An alternative is to unfold the functions into a set of nested loops.

6.1.2 Lazily-Evaluated Closures

The "macro-like" closures of the previous section have many features of traditional closures. Most limitations are to prevent the construction of unbounded data structures, just as recursion and algebraic datatypes are limited.

However, the introduction of (singleton) function identifiers was solely to simplify synthesis, and is unnecessary from a static allocation point of view: it is unnecessary to prevent different function values being joined together at the end of conditionals. Rather than mark each function value with a single function it can contain, function identifiers can become sets.

A singleton set is compiled as before. If it is not a singleton, the function value can be compiled down to an algebraic datatype, with a different constructor for each of the possible functions being called. The data associated with each constructor is that function's environment. At a call site, all the possible functions are instantiated, and the correct function is selected at run-time, using a conditional expression.

This kind of analysis has been studied in the past, as control-flow analysis (CFA) [132, 64, 9] and closure analysis [137]. The analysis we use is *polyvariant*; the set of functions a function value may take are calculated based on the call site used to reach that expression (or rather the non-recursive chain of calls), instead of conflating all calls to the function. This is a result of the non-sharing synthesis used. The analysis converges as the set of functions is finite.

The removal of higher-order functions has also been studied under a variety of names, including elimination of higher-order functions [125], closure conversion [143, 38], defunctionalization [14, 13] and higher-order removal [40]. The analysis for SASL is greatly simplified by the closure stability constraint, which means that it is not necessary to perform any iteration or find a fixed point.

6.2 Leniently-evaluated Expressions

The earlier sections of this chapter have only looked at using closures as a way of simplifying the expression of a program, rather than as a way of suspending the evaluation of values that may not be needed. For example, the Scheme language [82] uses `delay` and `force` expressions to generate and consume *promises* for this purpose. A promise (or *suspension*) is only executed once, independently of how many times it is forced, so it is more like an explicit version of lazy evaluation than a closure (which may be repeatedly evaluated).

In software, delaying evaluation until required prevents processing resources from being used until they are required. With hardware, this provides no advantage, since the hardware to produce the values will be otherwise be idle.[1] More useful in hardware is the ability to speculatively evaluate an expression, and cancel it should it not be required.

Linear SASL already kills *values* that are not used, but this is quite different from killing *expressions*. If a value is killed, the expression must complete; non-terminating expressions cannot have their resulting value killed, but the expression can be killed. Similarly, killing an expression can be almost instantaneous, while killing a value may require synchronising and waiting for the value to be produced before killing it.

[1] In fact, the hardware may be producing other pipelined values, but we are assuming that sufficient parallelism is available that the "speculative execution" of values that may not be needed is effectively free.

Promises are introduced to SASL using a similar syntax to Scheme, with expressions of the form **promise** e and **force** e. The new type σ *promise* is also introduced, as a stream data type. A promise can return any SASL data type, but cannot be enclosed in a stream or algebraic datatype. A promised data value cannot be accessed except by forcing it to obtain the contained value.

Promised expressions may not contain tail calls. To contain a tail call, a promise p would have to be in a tail context, and the function would return p. If p performs a tail call, it has to return a value of the same type as the function being called—that is, a promise. All that would be produced is a promise that, when forced, returns a promise of the same type.

As with closures and streams, a *promise stability constraint* is introduced to prevent unbounded recursive structures. In recursive calls, promise values must match up between the formal parameters and the arguments of the recursive call.

Using Promises Promises allow expressions that should be evaluated conditionally to be taken out of conditional expressions. For example, a conditional arm can be evaluated in parallel with the conditional value which selects whether the body should be executed. This allows the critical path length of the code block to be shortened to improve performance.

It also allows code duplication between conditional expressions to be eliminated. If two out of three conditional paths required a value, that value can be produced as a promise, and forced in those conditional paths. The hardware that produces the value is implemented only once (as opposed to if each conditional expression had its own function call), and the third path will not be delayed if the value turns out not to be required.

The main advantage of the lazy evaluation of Section 6.3 is that it gives the programmer these optimisations automatically, without having to explicitly create and force promises.

Implementing Promises In order to simplify the implementation of promises, they are initially constrained to be linear, and not return streams or other promises. Such promises can simply be implemented using leniently-evaluated streams in a source-to-source translation. The promised value is placed at the head of a lazy list, which is otherwise populated with "do not care" values:

> **promise** e becomes e**::**$cs_e()$, where cs_e is a function which returns a stream of constant values of the same type as e.
>
> **force** e becomes **case** e **of** x**::**$xs \quad x$.

The promise stability constraint simply maps to the stream stability constraint. If a promise is not used, it is killed, just as an unused stream is killed.

Using streams to implement promises is overkill. The infrastructure to deal with the tail is unnecessary, and the stream system can be specialised to deal with promises. "Promise buses" are like stream buses (and are typed similarly), except that only a single item will ever be transferred over a promise bus before a reset occurs, and can thus be optimised for this. Promise buses are reset in the same situations that stream buses are reset.

Promises are created with promise nodes that are identical to lenient CONS nodes, except that they write to promise buses. Like lenient CONS, they are enclosed in reset boxes (see Section 5.2.2), except that the reset boxes trigger on the promise bus's reset line, rather than the stream bus's. Unlike CONS nodes, the promise node's subgraph does not trigger another node to write to the same bus after it. The actual implementation of a promise node is identical to a lenient CONS node—the only difference being in how the subgraph is constructed, and how the promise bus can be optimised.

A promised value is forced by performing a read on the promise bus. A node like a stream read performs the read from the bus, but then automatically triggers a reset, as linearity ensures that once the promised value has been read it becomes non-live. As forcing is like a stream read, a forcing can be

(* (a) Redundant copy of f. *)
case x **of** $A\ \ f()$
$\qquad\qquad B\ \ f()$
$\qquad\qquad C\ \ g()$

(* (b) Redundancy removed. *)
let $y =$ **promise**$(f())$ **in case** x **of** $A\ \ y$
$\qquad\qquad\qquad\qquad\qquad\qquad\quad B\ \ y$
$\qquad\qquad\qquad\qquad\qquad\qquad\quad C\ \ g()$

Figure 6.3: An expression with and without redundant hardware

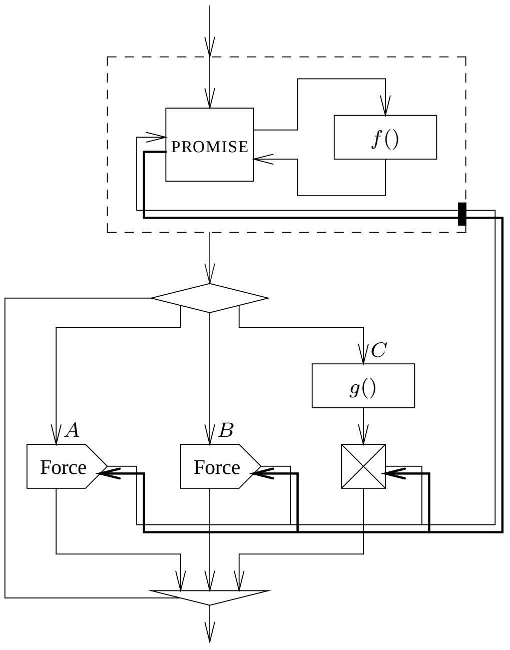


Figure 6.4: A graph using promises

leniently evaluated within another promise. If a promise is killed, all the promises and streams it depends on must be reset too. Promises are killed using a node very similar to a stream kill node.

Example The expression in Figure 6.3(a) can be rewritten as shown in Figure 6.3(b) in order to eliminate a redundant copy of f. The graph for the expression is shown in Figure 6.4. In this graph, the promised value is leniently evaluated, while the conditional expression is started. In cases A and B the read nodes wait for the promised value before resetting the promise bus. In case C the function $g()$ is evaluated instead, and the promise is killed without reading its value.

Returning Streams and Promises If a promise is to return streams or further promises, extra stream and promise buses must be created to transfer these values. These buses can be treated as normal promise and stream buses, except that they must not be read from until the promise enclosing them has been forced.

The type system requires that not only should the enclosing promise buses match up in typing, but also the promise and stream buses of any enclosed promises and streams. To make the buses match up, variable access expressions allow bus substitutions on the enclosed buses, as well as the outermost promise bus. In hardware, these become stream forwarders. Since the enclosed streams and promises depend on the enclosing promise p, if p is killed without being read from, all the enclosed buses must be reset too.

Replicating Promises It may be useful to have more than one expression access the result of a promise (as long as the promise does not return any linear values). The value may be duplicated after forcing the promise, but if it turns out that neither copy is used, the evaluation will have been in vain, possibly slowing execution.

An alternative is to introduce a new explicit promise-replication node, just as the DUP node was introduced to Linear SASL. Such a node copies the token representing the promise's availability, but also handles duplication of the values sent on the promise bus. Since the evaluation only occurs once, the promise replication node must cache the produced value. If any consumer forces a replicated promise, the node will force the original and cache the result. Any further forcings will receive this *memoized* value. Once all consumers have either forced or killed the promise, the node will clear. If all consumers kill the promise, the reset is forwarded to the original producer, and the node resets.

The replication of promises so that the values may be evaluated once and used multiple times, but not evaluated unnecessarily, forms the basis of our implementation of lazy evaluation in Section 6.3. Note that a replication of a promise is not the same as the original promise for the purposes of the promise stability constraint on recursive calls—replication performs substitutions on promise buses.

Pipelining Promises The production of different streams on the same stream bus is not pipelined. Attempting to interleave the production of items from different streams on the same stream bus causes many difficulties, and is likely to be of limited use, since it is expected that the number of items per stream is large, and the time spent switching between streams on a stream bus would be small compared to the time producing the stream items. The actual production of stream items on a particular stream can be pipelined. Mutual exclusion boxes are used around stream-producing graphs, and the same approach can be used for simple implementations of promises.

Although they can be implemented as single-item streams, promises have a very different behaviour. A single item is used from each of these promise streams, and so it does not make sense to worry about the interleaving of streams. If a promise is forced, it acts similarly to a normal SASL expression, where we expect that requests may be pipelined into the graph, and results received in the correct order. Promises may be pipelined, and if no promise is ever killed it can be treated like a normal dataflow graph.

When a pipelined promise is killed, it is not possible to reset the promise-producing hardware completely, since other items in the pipeline must not be cancelled. Instead, the hardware must now cancel only the leading set of tokens:

For straight-line code, this simply requires dropping the set of tokens nearest the graph's exit.

For iteration and conditional nodes, there will be hardware to collect the tokens in pipeline order, and this can be used to identify and eliminate the appropriate set of tokens.

For subgraphs that produce streams, if one of the tokens being killed has passed through a mutual exclusion box, that mutual exclusion box and its contents are cleared.

Pipelining promise production, with selective reset, gives a large overhead compared to the opposite approach of mutual exclusion and a complete reset, and so its use must be weighed against improved performance from pipelining.

6.3 Statically-Allocated Laziness

The bulk of SASL is eagerly evaluated, with lazy (or in practical terms, lenient) CONS expressions. In earlier sections, lazy evaluation was dismissed as inefficient for hardware (by preventing parallelism) and not statically allocatable (lazy evaluation may create what are effectively unbounded nested promises as arguments of tail-recursive calls). Lenient evaluation may be used to ensure sufficient parallelism is achieved. Unrestricted streams and closures may require unbounded memory requirements, but appropriate rules can restrict them to bounded forms, making them practical and useful for statically allocated systems. In the same vein, limited lazy evaluation may be statically-allocatable, useful and practical.

There are several reasons why lazy evaluation may be useful. Moving to lazy evaluation causes CONS nodes to no longer be a special case. Explicit *promise* and *force* expressions become redundant. Lazy evaluation is a common model in software functional languages, and it may be useful to explore the similarities; the blow-up of storage requirements is the bugbear of lazy evaluation in software systems. Static-allocation will limit the language's laziness, so there will be some trade-off.

A key feature of lazy evaluation is that values are not re-evaluated each time they are required. SASL's lazy evaluation should also leniently evaluate values ahead of time. Both of these features are provided by the promises of the last section. The basic approach is to take an eager control/dataflow graph and make it lazy by enclosing data processing elements so that the values returned are promises, and all parameter promises are forced before being used. The DUP nodes are replaced with promise replication nodes.

Dealing with Iteration The promise stability constraint requires a promise that is passed into a function be passed unevaluated in any recursive call. In graph form, this translates to disallowing promises as parameters to iteration and multiplexer nodes, unless the promises are not forced within the loop. This gives us our practical limitation on laziness in hardware: *all values may be evaluated lazily, except those that are passed into recursive calls*. This is as might be expected, since it is the recursive calls that allow the build-up of unbounded nested promises that can make lazy evaluation so memory-hungry.

Handling Streams Streams may not contain promises. As such, values must be forced before being sent over streams. However, streams are leniently evaluated, so this forcing is done leniently too (by performing the forcing within a CONS node) and the forcing may be cancelled by resetting the output stream, just as the leniently evaluated forcing of a promise may be cancelled when the promise value is killed. The other stream-related nodes can be simply treated as normal nodes, in terms of the insertion of forcings and promises.

Synthesis Details Under the simplest version of the transformation, almost every node is encapsulated to force inputs and promise outputs. The specific transformation for each graph construct is described below. The graph form used is the one associated with lenient evaluation, with the nodes given in Section 5.2.6. Figures show a number of the transformations, with the promise buses omitted (as stream buses were in basic dataflow graphs, to simplify the diagrams). Nodes that represent subgraphs, (marked, for example, "A") are transformed to lazy subgraphs, (such as "A'"), when used in the lazy version of the graph.

> Normal nodes, and other nodes without subgraphs (match nodes, forwarder nodes and stream kill nodes) have all input edges preceded by a forcing node, and the new graph is enclosed in a promise box, as shown in Figure 6.5. Normal nodes with more then one output edge are split into a set of nodes with one output each.
>
> Mutual exclusion boxes and reset boxes are not modified (although new reset boxes will be introduced with the new promise nodes).

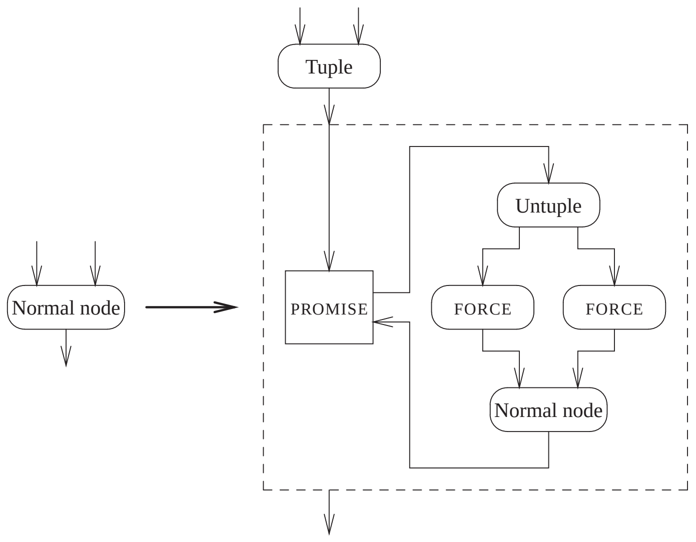


Figure 6.5: Converting normal nodes to lazy form

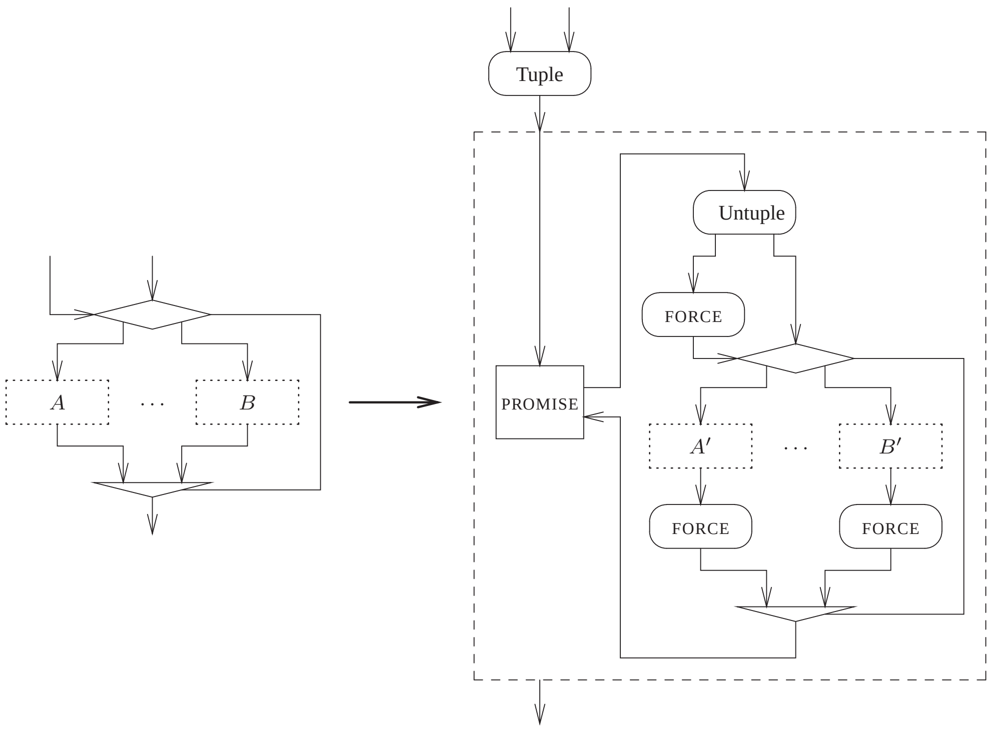


Figure 6.6: Converting conditional constructs to lazy form

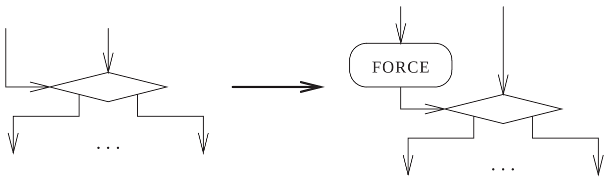


Figure 6.7: Converting top-half-only conditionals to lazy form

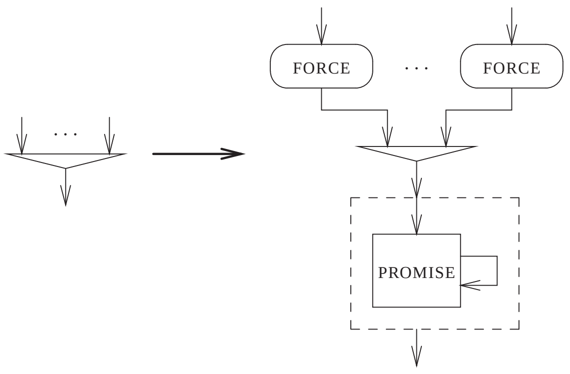


Figure 6.8: Converting multiplexers to lazy form

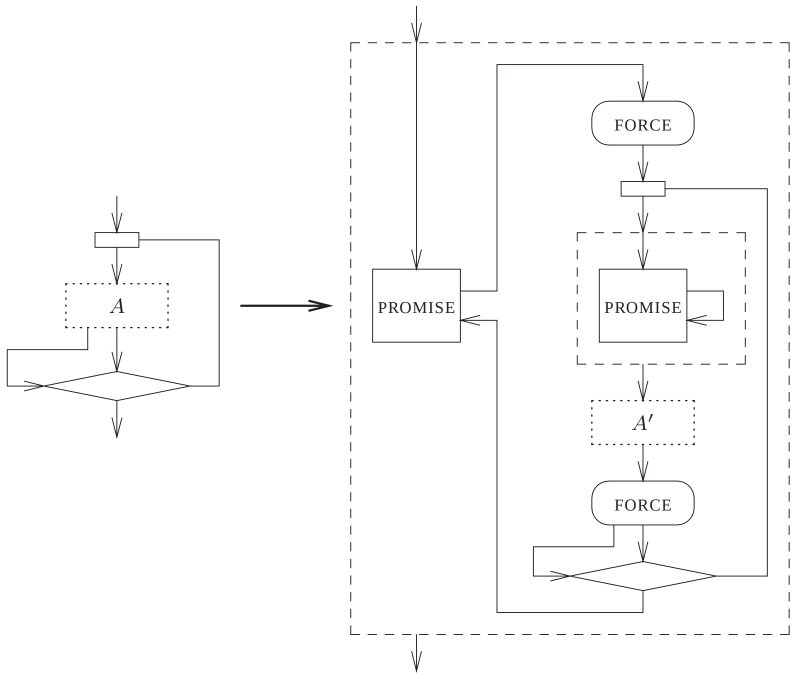


Figure 6.9: Converting iteration constructs to lazy form

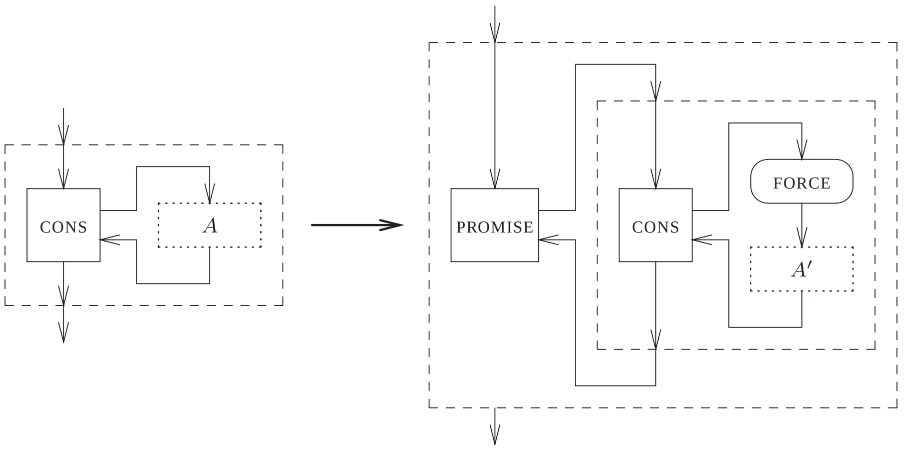


Figure 6.10: Converting CONS nodes to lazy form

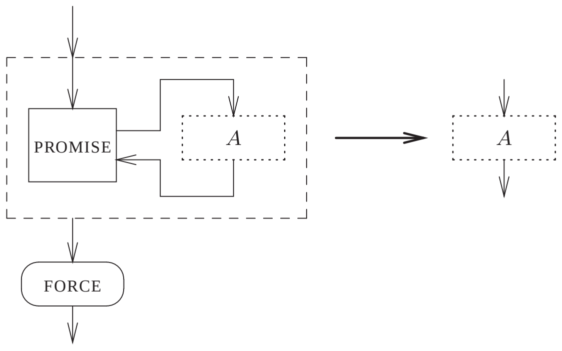


Figure 6.11: Removing unnecessary promises

Conditional expressions force the value used for the condition, and force the result of each conditional arm, and the whole graph produced is enclosed in a promise box. This is shown in Figure 6.6.

Top-half-only conditionals, used in the tail of lazy tail matching loops (see Section 5.2.5) only force the condition value, as illustrated in Figure 6.7.

Multiplexers force all inputs (both external and from the subgraph), and then convert the parameter value to a promise at the start of the subgraph. Figure 6.8 shows this transformation.

Iteration nodes enclose their subgraph so that the parameter value is made into a promise, and the returned value is forced. The iteration node's input is then forced, and the whole graph enclosed in a promise box, as shown in Figure 6.9.

CONS nodes force the edge that supplies the item to be written to the stream, and the node is then enclosed in a promise box. This transformation is illustrated by Figure 6.10.

To keep the same external interface as was used previously, the top-level function is enclosed in a wrapper which converts parameters to promises, and forces the returned value.

Grouping Data Processing Elements The previous section creates far more promises than necessary. Just as the basic synthesis creates a large number of separate nodes which are dynamically scheduled, but which can be statically scheduled in groups (see Section 5.1), we can eliminate a large number of these promises, producing a graph which performs the same set of computations, but with a much lower overhead. This is a form of *strictness analysis*. Note that the optimisations below may be applied to promises in general, as well as lazy evaluation.

The basic transformation is shown in Figure 6.11. A promise that is immediately forced can be simply reduced to an eagerly evaluated graph. If a promise is replicated this optimisation cannot be applied directly. Figure 6.12 shows how a replication node can be moved through force nodes to eliminate replication code and perhaps allow the elimination of the force and its associated promise.

In order to allow further elimination of promises, we need to be able to move promises inside other promises, so that they can be matched up with the appropriate forcing node. This transformation is shown in Figure 6.13. In a non-lazy system this transformation may increase laziness, as an eager value is now evaluated lazily. Along with other transformations that allow blocks of nodes to be moved in and out of subgraphs (see Section 5.3), these transformations allow the majority of promises to be eliminated, without changing the termination characteristics or degrading performance.

Identifying Lazy Values So far we have assumed that there is a one-to-one mapping between lazy values and SASL expressions. When a tuple is created, it forms a promise, and either none of the tuple is evaluated, or all of it is. Finer granularity may be wanted, for example allowing the independent

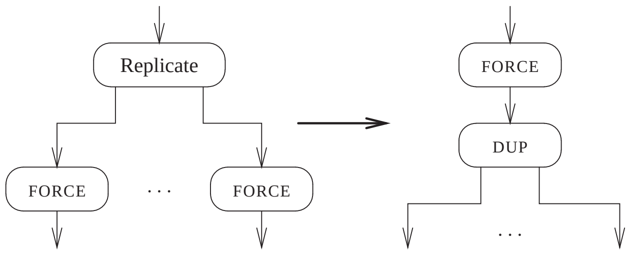


Figure 6.12: Pushing a Replicate node through Force nodes

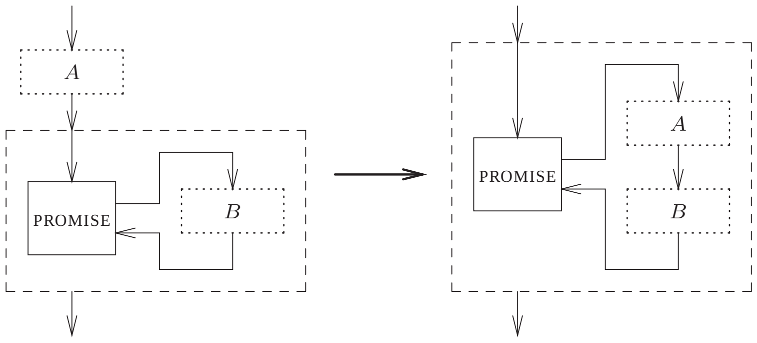


Figure 6.13: Transforming an eager value to a lenient one

evaluation of elements of a tuple, or even the fields within an algebraic datatype constructor. This can be achieved by not forcing the parameters to a tupling or datatype constructor node, and removing unnecessary promises around the deconstruction node. The transformation is slightly complicated by streams, as values that are passed through streams must have all nested promises forced.

The Execution Model It may be useful to try to visualise how statically-allocated laziness is evaluated in parallel. Under lazy execution, evaluation now causes an initial backwards pass through the program, as requesting the value from the output triggers a cascade of requests backwards for values that are definitely needed, effectively running the dataflow graph in reverse. Loops skip their bodies, requiring that their inputs be available for computing their bodies, and similarly conditionals require the conditional value be calculated first. Once the values are available, parts of the forward phase of execution can begin, which behaves similarly to eager evaluation, except that values that are not necessarily required are not evaluated at this stage. Loop and conditional bodies that produce values which are required are evaluated. When unevaluated values turn out to be needed, extra smaller backwards passes are generated to trigger the production of the value. The general structure is akin to full laziness, with eager evaluation for loops. This evaluation model closely matches the pull model of Johnson’s Value State Dependence Graph [72].

Under lenient evaluation, all values will be produced speculatively, so that there will still be backwards requests for the results of promises, but they may be fulfilled immediately if the results were calculated leniently and are available immediately. Alternatively, the lenient evaluation can be viewed as eager evaluation, but with the addition of a backwards dataflow of cancellation signals that are produced as items are discovered to be not needed.

6.4 Summary

This chapter has gradually built up the infrastructure for statically-allocated laziness. Limited support for higher-order functions was introduced through macro expansion, and the support was then extended to more general statically-allocated cases.

Leniently-evaluated closures were then introduced as a way of increasing parallelism, by providing the interface of a partially-evaluated function, while internally computing results ahead of time. These *promises* can be implemented using either stream buses, or more specialised promise buses.

Finally, these closures were used to implement a statically-allocated form of lazy evaluation, where data is produced lazily, except that which is required by loops.

This chapter marks the end of the work on evaluation models. The next chapter extends the basic SASL language in a different direction by investigating non-deterministic stream operators.

CHAPTER 7

Multi-Set Processing and Non-Determinism

The inclusion of non-deterministic operations can improve language expressiveness and allow new compiler optimisations. By non-deterministically reading from a set of streams, taking the first item to be produced, a more flexible I/O model is introduced. When the order of items in a stream does not matter, the items can be processed in any order, leading to a number of optimisations.

Non-deterministic streams in functional programming languages have been studied before, in the context of functional operating systems [138, 79]. In those systems, the streams are for communication between processes connected by mutually recursive definitions. Generalised communication is allowed, introducing the possibility of deadlock. Although SASL's approach is much more limited, eliminating deadlock, some points still apply. Stoye [138] notes that non-deterministic operators are not referentially transparent. For example, given a function *merge* that interleaves two streams non-deterministically, the expressions **let** $x = merge(e_1, e_2)$ **in**(x, x) and $(merge(e_1, e_2), merge(e_1, e_2))$ are not the same. In that paper, the approach taken is to only allow the *merge*s at the top level of a program. SASL does not restrict the use of the non-deterministic operators in this way; if SASL code is duplicated during optimisation, care should be taken that the program's semantics are not changed. An alternative approach that eliminates this problem is discussed in Section 7.5.

Section 7.1 covers the use of non-deterministic stream read primitives, which read an element from one of a given set of streams. Section 7.2 introduces an analysis to identify values "tainted" by non-determinism. Section 7.3 then discusses the use of *multi-sets* or *bags*, to represent streams where the ordering of elements is unimportant. This enables a number of low-level optimisations that increase parallelism. Section 7.4 contains an analysis to identify lists that may be treated as bags without affecting the results generated. Section 7.5 discusses how the language can be restored to referential transparency. A summary is provided in the final section of this chapter.

7.1 Non-Deterministic Stream Reading

There may be situations where multiple input streams are constructed at different rates, and we do not care which stream we read from next. Moreover, it may be useful to read from the first stream which has available data. For example, the streams could represent events from a set of input devices, where we wish to act on the first event that occurs, regardless of the device that it occurs on. This has some similarities to the *alternative command* in CSP, which also introduces non-determinism, and the Unix *select* system call. As differences in program timing may produce different arrival orders for data, such matching on streams using a "first-come, first-served" system leads to non-determinism.

This non-determinism does not fit naturally with SASL's lazy stream model, but matches SASL's lenient streams better:

Lazy evaluation requires a request to be sent before stream items are produced. A non-deterministic read must send a request to each stream, and wait for the first result to arrive, at which point the other requests must somehow either be cancelled or buffered for future reads.

Lenient evaluation produces data as soon as possible. A non-deterministic read will accept data from the first stream that has an item available.

The language needs to provide *productivity* guarantees: if there is a non-deterministic read from a pair of streams, one of which never produces an item, the other of which is productive (that is, it will eventually produce an item), data from the productive stream should always be returned—the non-productive stream should not block execution. Furthermore, if a pair of productive streams are non-deterministically read from, it should be guaranteed that items from both streams will eventually be read. This is a *fairness* guarantee. A suitable model is that of *weak fairness*, or *justice* [93]. This states that every time a stream has data available, it will eventually be read. In temporal logic, if s is a stream, req a predicate for whether data is available on that stream, and ack a predicate for whether the data is being read, it is written as $\Box\Diamond(\quad req(s) \quad ack(s))$ [1]. Note that the time steps for the fairness guarantees are executions of the matching expression: the guarantees only hold as long as the match expression under scrutiny is evaluated an unbounded number of times (which is in no way guaranteed in SASL).

The following sections discuss how non-deterministic stream reads can be introduced to the language's syntax, how they may be implemented in hardware, and give an analysis to mark which values produced by a program may be "tainted" by non-determinism.

7.1.1 Language Considerations

In CSP, reading from one of a set of channels can be achieved through an alternative command, which consists of a list of pairs of input commands and body commands to be executed. If data is available on a channel, that input command is executed, reading data from the channel to the associated variable, before executing the body command. When the body command finishes execution, the alternative command also completes (as with conditional expressions, where only a single body expression is executed). A near-direct translation of this to SASL would be to create an *alternative expression*, containing a list of stream matching expressions, of which one is executed:

$$\begin{array}{l}(\textbf{case}\ e_1^1\ \textbf{of}\ x_1{::}xs_1 \quad e_1^2 \\ [\!]\ldots \\ [\!]\textbf{case}\ e_k^1\ \textbf{of}\ x_k{::}xs_k \quad e_k^2)\end{array}$$

This approach, while possible, has two major drawbacks; it does not use the pattern-matching style ML-based languages typically use, and has an awkward syntax. Furthermore, each of the e_i^1 are not accessible from the other e_j^2, so the e_i^1 will often just be variable access expressions for variables bound outside the expression, creating more awkward programs.

An improved syntax takes a tuple of streams as the value to be matched upon, with the match cases each reading from a single stream, leaving the other streams unread:

$$\begin{array}{lllllll}\textbf{case}\ e\ \textbf{of} & (x_1{::}xs_1^1, & xs_2^1, & , & xs_k^1 &) & e_1 \\ & (\ xs_1^2, & x_2{::}xs_2^2, & , & xs_k^2 &) & e_2 \\ & (\ \vdots, & \vdots, & \ddots, & \vdots &) & \vdots \\ & (\ xs_1^k, & xs_2^k, & , & x_k{::}xs_k^k &) & e_k\end{array}$$

[1] In words, at all points in time there will be a future time at which either there is no request, or there is an acknowledgement.

$$\boxed{\textbf{case}\ e\ \textbf{of}\ x{::}(xs_1,\ldots,xs_k)\quad e'}$$

$$\boxed{\begin{array}{l}\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ (x{::}xs_1,\qquad,xs_k)\quad (x,xs_1,\ldots,xs_k)\\ \qquad\qquad\qquad\vdots\qquad\qquad\qquad\vdots\\ \qquad\qquad\qquad (xs_1,\qquad,x{::}xs_k)\quad (x,xs_1,\ldots,xs_k))\\ \textbf{of}\ (x,xs_1,\ldots,xs_k)\quad e'\end{array}}$$

$$\boxed{\begin{array}{l}\textbf{case}\ e\ \textbf{of}\ (x{::}xs_1,\qquad,xs_k)\quad e_1\\ \qquad\qquad\qquad\vdots\qquad\qquad\vdots\\ \qquad\qquad (xs_1,\qquad,x{::}xs_k)\quad e_k\end{array}}$$

$$\boxed{\begin{array}{l}\textbf{case}\ \mathit{wrap}(e)\ \textbf{of}\ x{::}(xs_1,\ldots,xs_k)\\ \textbf{case}\ \mathit{unwrap}(xs_1,\ldots,xs_k)\ \textbf{of}\ (xs_1,\ldots,xs_k)\\ \textbf{case}\ x\ \textbf{of}\ \mathit{Constr}_1(x)\quad e_1\\ \qquad\qquad\ldots\\ \qquad\qquad \mathit{Constr}_k(x)\quad e_k\end{array}}$$

where

$\quad$ **fun** $\mathit{wrap}(xs_1,\ldots,xs_k) = (\mathit{wrap}_1(xs_1),\ldots,\mathit{wrap}_k(xs_k))$
$\quad$ **fun** $\mathit{wrap}_i(x{::}xs) = \mathit{Constr}_i(x){::}\mathit{wrap}_i(xs)$
$\quad$ **fun** $\mathit{unwrap}(xs_1,\ldots,xs_k) = (\mathit{unwrap}_1(xs_1),\ldots,\mathit{unwrap}_k(xs_k))$
$\quad$ **fun** $\mathit{unwrap}_i(x{::}xs) = ($**case** x **of** $\mathit{Constr}_i(x)\quad x){::}\mathit{unwrap}_i(xs)$

Figure 7.1: Equivalence between the forms of non-deterministic stream matching

In this syntax (which we will call Syntax A), the unmatched streams are still available through other variable names, while the matched stream is split into a head and tail part. As with constructor matches, only a single case is executed, although unlike constructor matches, the match is chosen non-deterministically rather than based on the value of e.

An alternative approach (Syntax B) is based on a different view of how the streams are being processed: given a set of streams, we wish to read an item from one of them, but do not care which stream. This can be performed with an expression of the form

$$\textbf{case}\ e\ \textbf{of}\ x{::}(xs_1,\ldots,xs_k)\quad e'$$

where e provides a tuple of streams of the same type, x is the head of one of the streams, and the xs_i match the tail of the the read stream or the other streams, as appropriate. Syntax B is closely related to the non-deterministic stream-interleaving function *merge*, and is included to provide a contrast with Syntax A. Syntax B separates the non-deterministic read itself from the conditional expression predicated on which stream was read from.

Streams of different types can be non-deterministically read together, and the stream that x came from can be identified, by mapping the elements of each input stream to a different constructor in an algebraic datatype (care must be taken to meet the linearity constraints). This technique is used to show equivalence between Syntax A and B in Figure 7.1.

SASL's streams are lazily-evaluated values, and pattern matching in lazy languages has traditionally been a problem; patterns can be set up such that it is impossible to know in advance which arguments need to be evaluated to select the correct match with minimal evaluation, such as:

$$\begin{array}{rl}\textbf{match}\ e\ \textbf{with}\ (\_,0,1) & e_1\\ (1,\_,0) & e_2\\ (0,1,\_) & e_3\\ (1,1,1) & e_4\\ (0,0,0) & e_5\end{array}$$

This is not a problem, as although the non-deterministic matches are performed on a set of lazy variables,

```
fun merge1(as, bs) = case (as, bs) of (a::as, bs)   a::merge1(as, bs)
                                      (as, b::bs)   b::merge1(as, bs)

fun merge2(as, bs) = case (as, bs) of x::(as, bs)   x::merge2(as, bs)
```

Figure 7.2: Non-deterministic merge functions for Syntaxes A and B

SASL performs these reads in an eager manner. All the streams are matched upon simultaneously, and the case where the first stream match completes is used. The non-deterministic reads rely on the use of lenient evaluation for streams.

Perhaps the simplest real-world function that uses non-determinism is the *merge* function, described above. Implementations of the *merge* function are shown in Figure 7.2. It is not possible to implement general non-deterministic reads in SASL using the *merge* function, since once the streams are merged there is no way to obtain both individual streams again, and linearity prevents access to the original streams. The non-deterministic read is therefore a more powerful non-deterministic operator than the *merge* function, and so is the primitive provided. In other functional languages with non-deterministic *merge*s (such as that described by Stoye [138]), *merge* can be used to implement a SASL-style non-deterministic read, since the streams are not linear.

7.1.2 Hardware Implementation

For the hardware implementation, we assume lenient evaluation, using the "push" signalling described in Section 5.2.1. With this, a non-deterministic stream read is not much more complicated than a plain stream read. Instead of acknowledging and returning an item from the single stream being read when it is available, the system monitors a number of streams, and returns and acknowledges the data from the first one that makes an item available. The match can be implemented with a node which reads from a set of streams, returning the stream item and a source tag identifying the stream from which the the item came. If Syntax A is used, the item is fed into a conditional node, with the condition value coming from the source tag. Syntax B can simply be implemented by dropping the source tag and returning the matched item.

The arbiter used to select the stream to read from must achieve weak fairness. However, since the arbiter is implemented as a finite piece of logic, we can obtain a stronger guarantee, of *finitary fairness* [5]. Instead of requiring that no input stream waits forever, this requires that there is some bound k such that a stream with data available will never have to wait more than k steps (i.e., executions of the stream read expression) before its data is read. It is simple to ensure finitary fairness, for example with an arbiter that "round robins" if multiple items are already available, and otherwise returns the first item to appear.

7.2 Identifying Non-Deterministic Values

With the introduction of non-deterministic operations, we now have values generated that may vary between runs of a program, even if the same data is supplied. These non-deterministic values can make debugging difficult, as bugs become increasingly difficult to reproduce. As such, it would be useful to minimise the parts of the program containing non-deterministic values, and be able to analyse the non-deterministic parts of the program, perhaps using formal methods on those parts where simple testing is no longer sufficient.

In order to minimise the non-deterministic elements of a program, or analyse them, we need to be able to identify them. This section covers the identification of non-deterministic values. Non-determinism "taints" values—a value that depends on a non-deterministic value will itself be non-deterministic, so that non-determinism can spread throughout a program.

This analysis has many similarities to *escape analysis* [56]. Escape analysis identifies values that may be part of a data structure returned by a function, so that the value cannot be allocated in storage that is released when the function returns (such as a stack frame). Escape analysis will tag values with the other values they may reference, so that if that variable is returned, a set of escaping variables can be found. To identify non-deterministic values, a new pseudo-variable representing non-determinism is introduced. The variables used in non-deterministic read matches are made to depend on this pseudo-variable. Any values which may be affected by non-determinism will depend on the pseudo-variable.[2]

The identification of non-deterministic values can be performed using an abstract interpretation or type-like system. The approach used here is based on a type system. This analysis can be viewed as a precursor to the analysis of Section 7.4, which is used to analyse which streams may have their items reordered without affecting the results. The identification of non-deterministic values is undecidable,[3] and so we use a conservative analysis. It will either identify values as definitely deterministic, or possibly non-deterministic.

As with other analyses, we need to choose how closely the analysis should model the program. At one end of the spectrum, each expression can either be flagged as either returning a deterministic result, or returning a result which may possibly contain some non-deterministic element. At the other end, values may be broken down to show which particular concrete values may be produced non-deterministically by an expression (thus allowing more accurate analysis of which execution paths may be triggered non-deterministically). It is a trade-off between speed and accuracy.

Our approach does not track elements within lists, so that if a non-deterministic element is CONS'd onto a deterministic list and then the tail taken, that list will be marked as non-deterministic. The elements of tuples are tracked independently. The implementation relies on an annotated type system, with each stream type and basic type marked as either being deterministic or possibly non-deterministic. We elide the basic and stream types in this analysis, focusing only on the determinism values. These determinism types σ are therefore similar to the types of Section 2.4.1 except that each basic and stream type is replaced with a boolean expression E representing whether the value is deterministic or not. The boolean value F is used to represent a deterministic type, and T a possibly non-deterministic type, so that, for example, a value that is a pair of a deterministically generated basic value and a non-deterministically generated stream would be typed as $F \quad T$.

The boolean expressions in the type are simply disjunctions of T, F and *determinism variables*. Only disjunctions are needed since non-determinism is "contagious", so a value will be non-deterministic if any of the values it depends upon is. Determinism variables (represented by Greek letters) represent the determinism types of parts of the function's parameter, so that the determinism type returned by a function can depend on the type of its parameter (and if a function is called from multiple sites the analysis need not be repeated).

The syntax for a determinism type σ is:

$$\begin{array}{ll} \sigma & (\sigma_1 \quad \ldots \quad \sigma_k) \quad E \\ E & B_1 \quad \ldots \quad B_n \\ B & T \quad F \quad \alpha \quad \beta \quad \gamma \ldots \end{array}$$

We generate determinism types using the typing rules shown in Figure 7.3, with semiunification, or polymorphic recursion [107]. The type for the function **fun** $f(x) = e$ is given by $\sigma_1 \quad \sigma_2$, where $x : \sigma_1 \quad e : \sigma_2$. The disjunctions on tuples are applied component-wise:

$$(\sigma_1^1 \quad \ldots \quad \sigma_k^1) \quad (\sigma_1^2 \quad \ldots \quad \sigma_k^2) = (\sigma_1^1 \quad \sigma_1^2) \quad \ldots \quad (\sigma_k^1 \quad \sigma_k^2)$$

[2]The "decision variable" of Section 7.5 effectively makes this pseudo-variable into a real variable.

[3]Or rather, undecidable for a Turing-powerful system, and only impractical for a statically allocated system.

The DIST operator distributes an expression over possibly-nested tuples:

$$\begin{aligned} E \text{ DIST } (\sigma_1 \quad \ldots \quad \sigma \;\; _k) &= (E \text{ DIST } \sigma_1) \quad \ldots \quad (E \text{ DIST } \sigma \;\; _k) \\ E_1 \text{ DIST } E_2 &= E_1 \quad E \, _2 \end{aligned}$$

The typing implementation is complicated by recursive calls. For non-recursive calls, a substitution is used on the determinism variables, similar to those used in SASL's original typing system. For recursive calls, the determinism type of the subexpression containing the recursive call will depend on the overall determinism type of the whole function, which is not yet known.

For an example, we will use the function f:

$$\textbf{fun } f(p, q, r) = \textbf{if } p \textbf{ then } q \textbf{ else } f(\textit{true}, r, q)$$

If the type of this function is given as $(\alpha \;\; \beta \;\; \gamma) \quad \delta$, the typing rules give $\delta = \alpha \;\; \beta \;\; \delta[F/\alpha, \gamma/\beta]$. The least fixed point of this is $\delta = \alpha \quad \beta \quad \gamma$. The fixed point can be reached either by repeatedly substituting δ into itself and simplifying until a fixed point is reached, or by repeatedly typing the function, initially assuming that recursive calls produce deterministic results, and repeating with better approximations until a fixed point is reached.

This typing is similar to the Mycroft-Milner type system [107], where *polymorphic recursion* is used. The Mycroft-Milner type system has been shown to be undecidable [65], but that is not a problem in this language, since the lack of recursive datatypes means that the types in a function form a finite-height lattice, so the function's type converges.

Higher-Order Functions Although basic SASL does not deal with closures and higher-order functions, they were introduced as an extension in Chapter 6. Identifying the non-deterministic values in a higher-order program is beyond the scope of this thesis, but the analysis may be implemented by using higher-order escape analysis [56].

Analysing Within Algebraic Datatypes The analysis described in this section does not keep track of non-determinism within algebraic datatypes. For example, if a "pair" *datatype* is created, a deterministic value paired with a non-deterministic one, and the deterministic value read out of the pair, the analysis will mark the value as possibly non-deterministic. Although treating the algebraic datatypes differently to tuples simplifies the analysis, treating them so differently is not necessary. General recursive datatypes would pose a problem to accurate analysis (for example, escape analysis on lists [122] only covers the "spines" of the list as a whole, rather than individual elements), but SASL is limited to non-recursive data structures, so that more accurate analysis is possible.

Extending the analysis to algebraic datatypes should improve the accuracy of the analysis, at relatively little computational cost. A number of approaches are possible, but a relatively simple one consists of treating algebraic datatypes as tuples. An initial item represents the datatype "tag", with a further place in the tuple for each constructor in the datatype. When typing a constructor expression, the places in the tuple for the tag and other cases of the algebraic datatype are left as deterministic (F), and when performing a pattern match on the datatype, the matched variables for each case come from the appropriate part of the tuple. The new rules for CONSTR-INTRO and CONSTR-ELIM are shown in Figure 7.4.

Non-deterministic Termination The analysis of this section has assumed that a value is only non-deterministic if some value required to produce it is non-deterministic. However, the eager evaluation model makes it possible to create expressions where termination depends on a non-deterministic value, but the value that is returned is deterministic. An example is shown in Figure 7.5.

$$(\textsc{apply})\frac{A \quad e : \sigma_1}{A \quad f e : \theta(\sigma_3)} \begin{array}{l} f : \sigma_2 \quad \sigma\ _3 \\ \theta(\sigma_2) = \sigma_1 \end{array}$$

$$(\textsc{constr-intro})\frac{A \quad e_1 : \sigma_1 \qquad\qquad A \quad e_k : \sigma_k}{A \quad c(e_1, \dots, e_k) : \sigma_1 \quad \dots \quad \sigma\ _k \text{ (F for 0-place constructors)}}$$

$$(\textsc{tuple-intro})\frac{A \quad e_1 : \sigma_1 \qquad\qquad A \quad e_k : \sigma_k}{A \quad (e_1, \dots, e_k) : \sigma_1 \quad \dots \quad \sigma\ _k}$$

$$(\textsc{cons-intro})\frac{A \quad e_1 : \sigma_1 \quad A \quad e_2 : \sigma_2}{A \quad e_1 \mathbin{\textbf{::}} e_2 : \sigma_1 \quad \sigma\ _2}$$

$$(\textsc{constr-elim})\frac{A \quad e : \sigma \quad \left\{\begin{array}{l} A, x^1_1 : \sigma, \dots, x^1_{k_1} : \sigma \quad e_1 : \sigma_1 \\ \qquad\qquad \dots \\ A, x^n_1 : \sigma, \dots, x^n_{k_n} : \sigma \quad e_n : \sigma_n \end{array}\right.}{\begin{array}{l} A \quad \textbf{case } e \textbf{ of } c_1(x^1_1, \dots, x^1_{k_1}) \quad e\ _1 \\ \qquad \dots \\ \qquad c_n(x^n_1, \dots, x^n_{k_n}) \quad e\ _n : \sigma \text{ DIST } (\sigma_1 \quad \dots \quad \sigma\ _n) \end{array}}$$

$$(\textsc{tuple-elim})\frac{A \quad e_1 : \sigma_1 \quad \dots \quad \sigma\ _k \quad A, x_1 : \sigma_1, \dots, x_k : \sigma_k \quad e_2 : \sigma}{A \quad \textbf{case } e_1 \textbf{ of } (x_1, \dots, x_k) \quad e\ _2 : \sigma}$$

$$(\textsc{cons-elim})\frac{A \quad e_1 : \sigma_1 \quad A, x_1 : \sigma_1, x_2 : \sigma_1 \quad e_2 : \sigma_2}{A \quad \textbf{case } e_1 \textbf{ of } x_1 \mathbin{\textbf{::}} x_2 \quad e\ _2 : \sigma_2}$$

$$(\textsc{let})\frac{A \quad e_1 : \sigma_1 \quad A, x : \sigma_1 \quad e_2 : \sigma_2}{A \quad \textbf{let } x = e_1 \textbf{ in } e_2 : \sigma_2}$$

$$(\textsc{var})\frac{}{A, x : \sigma \quad x : \sigma}$$

$$(\textsc{nondet-elim-1})\frac{A \quad e : \sigma \quad \left\{\begin{array}{l} A, x_1 : T, \dots, xs^1_k : T \quad e_1 : \sigma_1 \\ \qquad\qquad \dots \\ A, xs^k_1 : T, \dots, xs^k_k : T \quad e_n : \sigma_n \end{array}\right.}{\begin{array}{l} A \quad \textbf{case } e \textbf{ of } (x\ _1 \textbf{::} xs^1_1, \dots, xs^1_k) \quad e\ _1 \\ \qquad \dots \\ \qquad (xs^k_1, \dots, x_n \textbf{::} xs^k_k) \quad e\ _n : T \text{ DIST } (\sigma_1 \quad \dots \quad \sigma\ _n) \end{array}}$$

$$(\textsc{nondet-elim-2})\frac{A \quad e_1 : \sigma_1 \quad A, x : T, xs_1 : T, \dots, xs_k : T \quad e_2 : \sigma_2}{A \quad \textbf{case } e_1 \textbf{ of } x \mathbin{\textbf{::}} (xs_1, \dots, xs_k) \quad e\ _2 : \sigma_2}$$

Figure 7.3: Rules to identify possibly non-deterministic values

Our analysis marks the expression as returning a deterministic value. Although a new analysis could be introduced to identify possible non-termination dependent on non-deterministic values, we view the possible non-termination as a bug, and restrict our analyses to direct dependencies.

7.3 Generalising Streams

Basic SASL deals only with ordered streams of data. In an actual implementation, this may unnecessarily restrict execution. Conditional execution nodes must merge together the elements in the order they entered in. When loops are used to process data, the results are reassembled in the order the requests were sent in. These ordering constraints may not be a necessary part of the processing, and including them may limit performance and require extra hardware resources. By identifying streams that do not actually need to be lists, but could be implemented as *bags*, hardware optimisations can be performed.

Bags, also known as *multi-sets*, can be viewed as lists where the order does not matter, or sets where an element may occur multiple times. Rather than having a CONS operation and associated match that put the data in in a known order, the program is allowed to insert items into the bag and remove them in any order. If ordering restrictions are removed, data processing may no longer need to block, waiting for a particular item. The model bears some similarity to out-of-order execution in superscalar processors.

We treat the non-determinism introduced by multi-sets separately to that introduced by non-deterministic stream matching. Section 7.3.1 discusses some background details in relation to dealing with multisets, while Section 7.3.2 introduces possible syntax. Section 7.3.3 deals with some of the details of the hardware implementation. Section 7.4 then provides an analysis which may be used to identify streams that may be treated as bags.

7.3.1 Dealing with Multi-sets

SASL's streams are lazily-evaluated lists, so appropriate extensions may be found by looking at the formalisms for lists. *Monoids* provide an appropriate formalism for finite data structures. A monoid consists of a set S, an associative operator of type S S S, and a left and right identity e S. A *free monoid* in some sense represents the most general monoid. The free monoid over the category of all monoids represents all the finite lists, with as append. The free monoid over commutative monoids represents bags or multi-sets, while the free monoid over commutative and idempotent monoids represents sets.

SASL implements lists, so sets and multi-sets may be viable alternatives. Sets present a problem, in that once an item has been produced, it must not appear again. To implement this, either functions must be limited to injective maps on the sets, or the system must keep track of emitted elements, and remove duplicates. This may require unreasonably large storage requirements. The implementation of unbounded bags is somewhat simpler. They can be simply be treated as lists where elements may be reordered. In the worst case, the bags may be implemented using lists.

Notice that we assume that bags *may* be reordered, but it is never *necessary* to reorder them. For example, if a bag is being mapped through a function f, and an element x of the bag causes $f(x)$ not to terminate, the whole program may not terminate. The implementation is not forced to let other elements "overtake" this non-terminating element. This assumption greatly simplifies the implementation of unbounded bags in limited hardware resources.

As with stream-reading non-determinism, some form of fairness is necessary. Weak fairness is used, so that if an item is added to a bag it will be read from the bag a finite number of bag reads later. The hardware arbiters must be designed to ensure this fairness.

It should be noted that monoids deal with finite data structures. Unbounded data structures have some rather different properties. With lists, unbounded data structures can be used to represent finite data structures, as described in Section 2.3.2. However, this is not possible with multisets, since the end-of-structure delimiter that would be used in a list could be reordered past useful data elements.

Instead, bounded bags need to be treated separately, with bag-matchings that have cases for non-empty and empty bags. This chapter mostly discusses unbounded bags, but notes on performing the bag-identification analysis on bounded bags, and the hardware implementation of bounded bags are included in Sections 7.4 and 7.3.3 respectively.

7.3.2 Syntax and Types for Reorderable Streams

The difference between ordinary streams and reorderable streams is akin to the difference between integer types and floating-point types, or fixed-size integer types and unbounded integer types. There is a great deal of commonality in the operations that may be applied, although the results may differ. Values may be cast between the types, with possible loss of information. An optimising compiler may be able to move casts around in order to reduce execution cost.

SASL's options in dealing with multisets are mirrored in the approaches that other languages take. Standard ML has a special case that overloads built-in arithmetic operators for integer and floating-point types. However, user-defined functions may not be overloaded in this way. Objective CAML uses separate operators for the different types. Haskell provides type classes, which allows a function to be polymorphic over a set of numeric types. All these languages require explicit casting functions to translate between the types. In comparison, implicit conversion is used in many dynamically-typed scripting languages (and even some languages without dynamic typing, such as C). The rest of this section discusses a variety of approaches.

No Overloading Our initial approach is to use separate notation for list CONS and bag CONS. The original notation remains "**::**", but the notation "**++**" is used for dealing with reorderable streams, giving the following new forms of expression:

$$\begin{gathered} e_1\texttt{++}e_2 \\ \textbf{case}\ e_1\ \textbf{of}\ x_1\texttt{++}x_2 \quad e\ \ _2 \end{gathered}$$

The type system is similar to before, except that stream types now consist of basic types, tuples, list-like streams of basic types, and bag-like streams of basic types. Since there are separate ordered and unordered stream operations, the typing is unambiguous. On the down side, separate functions are required for otherwise identical operations on streams and reorderable streams. Casts may be performed with functions such as:

$$\begin{gathered} \textbf{fun}\ \textit{unorder}(s) = \textbf{case}\ s\ \textbf{of}\ x\texttt{::}xs \quad x\texttt{++}\textit{unorder}(xs) \\ \textbf{fun}\ \textit{order}(s) = \textbf{case}\ s\ \textbf{of}\ x\texttt{++}xs \quad x\texttt{::}\textit{order}(xs) \end{gathered}$$

These functions also show us what to expect when casting between ordered and unordered streams. An unordered stream, when cast, produces a stream which when read will contain the items of the unordered stream, in any order, as before, but if further items are CONS'd onto the stream, and the stream read, the newly CONS'd items will be read first. Once an ordered stream is cast to an unordered one, all original ordering information is lost, even if it is then cast back to an ordered stream.

Overloaded Operators With overloaded operators, both ordered and unordered streams are CONS'd upon with expressions using "**::**", and the type is inferred from the type of the stream being CONS'd upon. The advantage of this is that it makes possible polymorphic stream functions that can read both ordered and unordered streams. This approach is more complex than the overloaded arithmetic operators of Standard ML (which are effectively syntactic sugar), matching more closely the use of type classes.

However, the top-level streams must be identified as either bag-like or list-like in order to infer the types of other streams from them. Functions that generate new streams, not based on existing streams, also need to mark whether these streams are list-like or bag-like. Some functions may only make sense on ordered streams. Type annotations may be used to clear up these problems.

For example, subscripts may be provided in the type annotation representing ordered (O) and unordered (U) streams. Variables may be used when inferring the ordering nature of one stream from another. Figure 7.6(a) is a function that returns a stream that is bag-like if and only if the original stream is bag-like. In general, whether a returned stream is bag-like may depend on whether a number of argument streams are bag-like. For example, in the *zip* function of Figure 7.6(b), there will be no ordering information in the returned stream only if both parameter streams are bags, and so the type annotation for the returned value depends on multiple stream type variables.

Since there are no separate operators, the cast functions have to be rewritten to use type annotations:

$$\begin{aligned} \textbf{fun}\ \textit{unorder}(s) &= \textbf{case}\ s\ \textbf{of}\ x{::}xs \quad x{::}\textit{unorder}(xs)\ :\ \alpha\ \textit{stream}\ _U \\ \textbf{fun}\ \textit{order}(s) &= \textbf{case}\ s\ \textbf{of}\ x{::}xs \quad x{::}\textit{order}(xs)\ :\ \alpha\ \textit{stream}\ _O \end{aligned}$$

Every function that generates a new stream using the overloaded operators will need a type annotation for the returned stream. It may be useful to have some form of default type. Ordered streams may be used anywhere a bag may be used, but not vice-versa. Streams may therefore be treated as list-like by default. Bag-like streams may be treated as a subtype of ordered streams, with a “downcast” used whenever the order of elements in a stream does not matter.

Implicit Typing At the end of the previous section, treating streams as lists by default is mentioned. The type system can be arranged so that all the user ever sees is list-like streams. A new SHUFFLE expression type can be introduced, which effectively takes a list-like stream, casts it to a bag-like stream, and then back again, so that all ordering information is lost, but the user only sees list-like streams.

Internally, the synthesis tool may identify streams that may be treated as bag-like without altering the behaviour of the program, using the analysis of Section 7.4. From this point of view, the identification of bags is not something the user need deal with, but is something the compiler can infer. The SHUFFLE expression acts as a cast to a bag-like stream, in those situations where the compiler cannot infer it, but does not require the user to keep track of types. Only the conversion from list to bag needs user annotation, as this is the direction in which information is lost.

Using implicit typing, the language’s original syntax is minimally extended with the SHUFFLE expression, and bag-like streams may be introduced with minimal effort from the language user’s point of view. The implicit typing approach will be taken in the following sections.

7.3.3 Implementing Reorderable Streams

Once the reorderable streams have been identified (using the analysis of Section 7.4), the information can be taken into account in the way data items are written to or read from their associated stream bus. Note that reorderability is not a property of a stream bus, but of the read and write nodes. For example, the same stream bus may be used to first transfer a header item in order, followed by a set of reorderable items. A stream write is reorderable if the stream produced by the original CONS expression is reorderable. A stream read is reorderable if the original expression being matched upon is reorderable.

Stream Reads Given a set of reorderable stream reads on the same stream, the relative order constraints can be relaxed. However, not all constraints can be removed, since it is still necessary to kill the stream when all reads that were to be performed have completed. Sequenced reorderable stream read operations may be converted to run in parallel, as shown in Figure 7.7. The DUP node duplicates the “stream ready” signal, so that it is available to both stream reads at the same time.

This transformation may not help improve performance much, since the standard stream implementation only allows a single item to be read per cycle, and even with the reads being activated in parallel,

$$(\text{CONSTR-INTRO})\frac{A \quad e_1 : \sigma_1 \qquad A \quad e_k : \sigma_k}{A \quad c_i(e_1,\ldots,e_k) : \sigma = (F,\ldots)}$$ where σ has F in all terminal positions except the $(i+1)^{th}$ element, which is $(\sigma_1 \quad \ldots \quad \sigma \;_k)$.

$$(\text{CONSTR-ELIM})\frac{A \quad e : \sigma_0 \quad \ldots \quad \sigma \;_k \qquad \begin{cases} A, \vec{x_1} : \sigma_1 \quad e_1 : \sigma'_1 \\ \ldots \\ A, \vec{x_n} : \sigma_k \quad e_n : \sigma'_n \end{cases}}{\begin{array}{l} A \quad \textbf{case } e \textbf{ of } c_1(\vec{x_1}) \quad e\;_1 \\ \qquad \ldots \\ \qquad c_n(\vec{x_k}) \quad e\;_k : \sigma_0 \text{ DIST } (\sigma'_1 \quad \ldots \quad \sigma\;'_n) \end{array}}$$

Figure 7.4: Analysing non-determinism within algebraic datatypes

```
fun loop() = loop()
fun nondet(s, t) =
    let y = case (s, t) of (x::xs,    t    )   loop()
                           (   s,    x::xs, )  True in
    True
```

Figure 7.5: A non-deterministically terminating function

```
(* a) The map function applies the function f to a list or bag item-wise. *)
fun map_f(s : α stream_χ) =
    case s of x::xs    f(x)::map  _f(xs)  :  α stream_χ

(* b) The zip function pairs together elements of two streams.              *)
(* The result will only be order-less if both parameter streams are unordered. *)
fun zip(s  :  α stream_χ, t  :  β stream_χ') =
    case s of x::xs    case t of y::ys    (x, y)::zip(xs, ys)  :  (α    β) stream    χ∧χ'
```

Figure 7.6: The functions *map* and *zip*, with casts

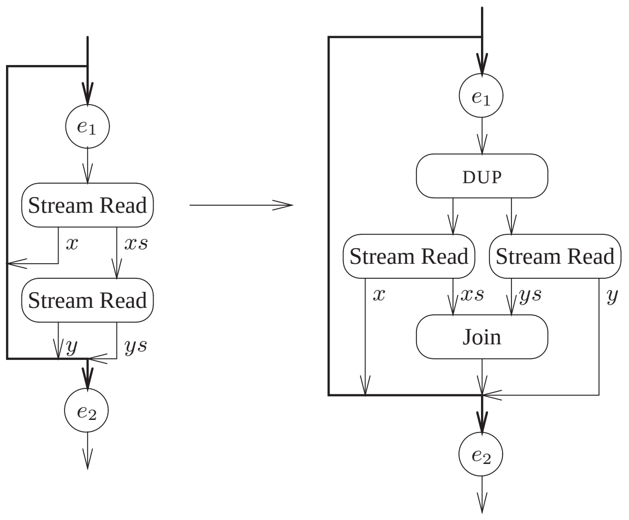


Figure 7.7: Conversion of **case** e_1 **of** x**::**xs **case** xs **of** y**::**ys e_2 to a reorderable form

```
fun iterate(x) = if test(x) then x else iterate(f(x))
fun mapiter(x::xs) = iterate(x)::mapiter(xs)
fun mapiter2(x::xs) = (x, iterate(x))::mapiter2(xs)
```

Figure 7.8: Reordering loop items

the actual stream reads may be serialised as before. There is also the overhead of constructing arbiters, as multiple stream read nodes may now activate at the same time.

An alternative approach is to keep the property of having a single dynamic reader, but create sets of read operations that may be reordered, and choose a static order of reads within the sets to maximise performance under statically scheduling. For example, if there are two reads, the results of which trigger complex operations, the read associated with the longer evaluation path can be triggered first.

Stream Writes Lazily-evaluated stream writes cannot be reordered, since stream items are produced individually, on demand. Using the lenient evaluation of Section 5.2 allows stream items to be produced ahead of time, in advance and in parallel. Lazy tail evaluation, using a stream bus controller, minimises dependencies between the production of items. The stream bus controller is used to deliver the items of a list-like stream in order. Reorderable streams allow the stream bus controller to be eliminated, and replaced with a simpler arbiter.

Reorderable Streams and Pipelining It is not only streams that may increase efficiency by allowing reordering. Tokens passed through pipelined iteration and conditional constructs could be reordered if ordering is unimportant and the path lengths are different for different data. Hardware is normally constructed to ensure data ordering, but if the order is unimportant we can eliminate this overhead.

If an edge carries tokens that *fully represent* (as defined below) items of a reorderable stream, those tokens can be reordered. For example, the function *mapiter* shown in Figure 7.8 maps each item by iteratively applying a function to each item until a test is passed. If the input and output streams are bags, it is not necessary that the calls to *iterate* return in the order they are initiated.

Note, however, that the function *mapiter2* must have its calls to *iterate* complete in order, since the results are paired up with the original requests. This is what is meant by saying the tokens must "fully represent" an item. For an edge to fully represent an item read from a reorderable stream, all possible dataflow paths from the read to a stream write must pass through that edge. For stream writes, all non-constant dependencies of the write must pass through that edge. The possible dataflow paths must include those passing the data through intermediate streams, as well as over plain graph edges. The dependencies are based on the dynamic flow of data. For example, in the body of a conditional expression, an edge will fully represent a stream read if all the stream write dependencies from that read go through that edge *given that conditional branch is taken*.

The only place where tokens may be pipelined when processing a reorderable stream is in the stream's head expression. Under lenient evaluation, each stream only ever has a single activation of a tail expression at a time, which evaluates to find the next head and tail (the dependencies are illustrated in Figure 5.12. The evaluation of the sequence of tail expressions can only be done serially. However, the same head expression may be activated multiple times simultaneously, if the chain of tail expressions reaches the same CONS node before the previous head value has been produced.

If an edge in a head expression graph fully represents a reorderable stream, the tokens that pass through the edge may be reordered. The possibility of token-overtaking generally only occurs with conditional and iteration nodes, where the time a token takes to pass through can be data dependent. Such nodes are analysed, and the stream items may be reordered if the data passed through the node either fully represents the stream item being generated, or depends only on a set of constant (over the stream) values and streams that are fully represented by that edge.

If a conditional or iteration node can be reorderably pipelined, it can be marked as such, and a hardware implementation chosen to make use of this. A reorder buffer or locking mechanism is no longer required to keep the tokens that pass through the node in order, greatly simplifying the design, and allowing higher performance.

Bounded Bags Bounded bags can be implemented by adding an extra “bag empty” line to the stream buses, and creating a “bus active” line to be monitored by the nodes which generate the bag empty signal. These “null nodes” are the translation of an empty-bag expression, and are activated when a token enters them, but do not instantly signal the bag empty line. If there is any activity in the circuitry that generates values on the associated stream bus, the bus active line is held high, and the bag empty signal is not raised, as further items could appear. When the bag *is* finally empty, the only active node associated with the stream will be the null node, the bus active line is dropped, and the null node can then send the bag empty signal. Once this is acknowledged by the stream read, the node clears itself, and the bus is reset (enclosing mutual exclusion nodes may need to be signalled). Bounded bag reads can be represented in SASL by a standard list-matching on the empty and CONS cases, which will then be implemented in the dataflow graph by a combination of stream read and conditional nodes.

7.4 Identifying Reorderable Streams

Using implicit typing, we need to be able to infer which stream values are bag-like and which are list-like. Even if explicit typing is used, it is valuable to be able to identify list-like streams that may be treated as bag-like without altering the results produced. Bag-like streams may require less resources to implement, and can allow increased parallelism.

We first need to identify what is meant by a stream that may be treated in a bag-like manner. A list-like stream s may be treated as bag-like if, for every reordering of the elements of the stream, there is some reordering of other actual bag-like streams t_i that would produce the same result. That is, if we cannot tell if the result produced comes from s being reordered, or s remaining ordered and the t_i being reordered.

As a simple example, given the expression SHUFFLE$($*map*$(s))$, the stream returned from *map* may be treated in a bag-like manner, and so the parameter to the function may be too, since we cannot tell if a particular ordering of the result comes from s being reordered, or the returned stream.

The analysis marks streams as bag-like or list-like at the typing level. SHUFFLE expressions are used to explicitly mark streams as bag-like. From this, the analysis conservatively identifies streams that are bag-like (unlike Section 7.2, which conservatively identifies deterministic values). The analysis does not identify values that may depend on a bag. For this, we can use a dependence analysis like that of Section 7.2 to trace values that depend on the result of SHUFFLE expressions.

Reorderability Typing To identify streams that may be treated as bags, it is necessary to identify the flow of data between streams. For this purpose, we perform a dependency analysis which is similar to the processing done in escape analysis.

The analysis is performed on a per-function basis, ordered so that the types of all functions called non-recursively are known. Annotated types are used, similar to those used in Section 7.2. We elide the basic type information, so that the value types are defined as follows:

$$\sigma := (\sigma \quad \ldots \quad \sigma) \quad B \quad S_R$$

Since reorderability is a property of streams, we now distinguish between basic types, B, and stream types, S. Furthermore, the identification of bag-like streams relies on the bidirectional flow of information, so the expressions E of Section 7.2 are replaced with boolean reorderability variables R, and the type system introduces constraints on the values of the R. True represents a bag-like stream, and false a list-like stream. The typing used is the one that produces the smallest set of reorderable streams that satisfies the constraints.

The basic typing rules are shown in Figure 7.9. At the top level, those parameter and result streams that are reorderable should have their associated R constrained to true. The rest of this section explains the details of the typing rules and their implementation.

Typing Non-recursive Function Calls The (APPLY) rule works like the other typing rules, in that it substitutes identifiers in the called function in order to match the call site's typing. For each call site, a fresh set of reorderability variables and associated constraints are created for the called function, and for each reorderability variable substituted by θ, a constraint is generated making the original and substituted reorderability variables equal.

Typing Recursive Calls Recursive function calls may be typed using polymorphic recursion, much as they were in Section 7.2 when identifying non-deterministic values. Again, constraints are generated to make the values of the reorderability variables match up between the function arguments and parameters.

Stream CONS and Matching The (CONS-INTRO) and (CONS-ELIM) rules only provide constraints in a single direction, since CONSing an element onto a bag may produce something that is not itself a bag, and similarly reading an element from a stream, thereby producing a bag, does not mean that the original stream was a bag.

Constructor Matching The (CONSTR-ELIM) rule must provide the constraints that streams returned from all the conditional arms are reorderable if and only if the stream returned by an expression as a whole is reorderable. Similarly, a stream in the environment is reorderable if and only if all uses of that stream are reorderable. These constraints are expressed as side-conditions on (CONSTR-ELIM). The side conditions on the return type are used to generate constraints as follows:

$$(\sigma^1 \quad \ldots \quad \sigma^k) = (\sigma^1_1 \quad \ldots \quad \sigma^k_1) \quad \ldots \quad (\sigma^1_n \quad \ldots \quad \sigma^k_n) \qquad \sigma^i = \sigma^i_1 \quad \ldots \quad \sigma^i_n$$
$$S_R = S_{R_1} \quad \ldots \quad S_{R_n} \qquad R = R_1 \quad \ldots \quad R_n$$

Using these rules, we can generate the side conditions on the environments, by applying the rules to each element of the environment:

$$A = A_0 \quad \ldots \quad A_n \qquad x \quad dom(A)\ A(x) = A_0(x) \quad \ldots \quad A_n(x)$$

Stream Reordering The (SHUFFLE) rule types SHUFFLE expressions, which conceptually perform an arbitrary reordering of the stream. In practice, they mark the stream as bag-like (assigning its R the value T), but will create no new hardware at the implementation level. It is provided to explicitly mark a stream as reorderable.

Examples Some simple examples are shown in Figure 7.10. The function *id-stream* is typed as S_R S_R. The returned value is reorderable if and only if its parameter is. The function *read-stream* would be of type $S_R \quad B \quad S_{R'}$, with the constraint $R \quad R'$. That is, if the parameter stream is a bag, the returned stream will be a bag (reading an item from a bag produces a bag), but if the returned stream is bag-like, this does not mean the parameter stream is. For example, the parameter stream may represent a bag of data with a single-element header. Similarly, the function *write-stream* has the type $B \quad S_R \quad S_{R'}$, with the constraint $R' \quad R$.

$$(\textsc{apply})\frac{A \quad e : \sigma_1 \quad f : \sigma_2 \quad \sigma\ _3}{A \quad f\,e : \theta(\sigma_3)}\ \theta(\sigma_2) = \sigma_1$$

$$(\textsc{constr-intro})\frac{A \quad e_1 : B \qquad\qquad A \quad e_k : B}{A \quad c(e_1, \ldots, e_k) : B}$$

$$(\textsc{tuple-intro})\frac{A \quad e_1 : \sigma_1 \qquad\qquad A \quad e_k : \sigma_k}{A \quad (e_1, \ldots, e_k) : \sigma_1 \quad \ldots \quad \sigma\ _k}$$

$$(\textsc{cons-intro})\frac{A \quad e_1 : B \quad A \quad e_2 : S_R}{A \quad e_1 :: e_2 : S_{R'}}\ R' \quad R$$

$$(\textsc{constr-elim})\frac{A_0 \quad e : B \quad \begin{cases} A_1, x_1^1 : B, \ldots, x_{k_1}^1 : B \quad e_1 : \sigma_1 \\ \ldots \\ A_n, x_1^n : B, \ldots, x_{k_n}^n : B \quad e_n : \sigma_n \end{cases}}{A \quad \textbf{case}\ e\ \textbf{of}\ c_1(x_1^1, \ldots, x_{k_1}^1) \quad e\ _1 \quad \ldots \quad c_n(x_1^n, \ldots, x_{k_n}^n) \quad e\ _n : \sigma}\ \begin{array}{l} A = A_0 \quad \ldots \quad A\ _n \\ \sigma = \sigma_1 \quad \ldots \quad \sigma\ _n \end{array}$$

$$(\textsc{tuple-elim})\frac{A \quad e_1 : \sigma_1 \quad \ldots \quad \sigma\ _k \quad A, x_1 : \sigma_1, \ldots, x_k : \sigma_k \quad e_2 : \sigma}{A \quad \textbf{case}\ e_1\ \textbf{of}\ (x_1, \ldots, x_k) \quad e\ _2 : \sigma}$$

$$(\textsc{cons-elim})\frac{A \quad e_1 : S_R \quad A, x_1 : B, x_2 : S_{R'} \quad e_2 : \sigma}{A \quad \textbf{case}\ e_1\ \textbf{of}\ x_1 :: x_2 \quad e\ _2 : \sigma}\ R \quad R\ '$$

$$(\textsc{let})\frac{A \quad e_1 : \sigma_2 \quad A, x : \sigma_2 \quad e_2 : \sigma_1}{A \quad \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \sigma_1}$$

$$(\textsc{var})\frac{}{A, x : \sigma \quad x : \sigma}$$

$$(\textsc{shuffle})\frac{A \quad e : S_R}{A \quad \textsc{shuffle}(e) : S_R}\ R = T$$

Figure 7.9: Rules for identifying reorderable streams

fun *id-stream*$(s) = s$
fun *read-stream*$(s) =$ **case** s **of** $x{::}xs \quad (x, xs)$
fun *write-stream*$(x, s) = x{::}s$

Figure 7.10: Simple examples for reorderability of streams

```
**fun** *skip-until*$(a, s_1) =$ **case** $s_1$ **of** $x$::$xs_2$
    (**if** $a = x$
    **then** $xs_3$
    **else** *skip-until*$(a, xs_4)_5)_6$

**fun** *copy-until*$(a, s_1, t_2) =$ **case** $s_1$ **of** $x$::$xs_3$
    (**if** $a = x$
    **then** $t_4$
    **else** $(x$::*copy-until*$(a, xs_3, t_5)_6)_7)_8$
```

Figure 7.11: Further examples for reorderability of streams

More complex examples are given in Figure 7.11, with each stream value annotated with the identifier of its reorderability variable:

> The type of *skip-until* is $B \quad S^{R^1} \quad S^{R^6}$, with constraints $R^1 \quad R^2$, $R^2 = R^3 \quad R^4$ and $R^6 = R^3 \quad R^5$, as well as those generated by the recursive call. The fixed point is found by initially assuming the recursive call always takes and returns bags, and using this to generate an initial approximation to the constraints between parameters and return value. Trying $R^4 = T$ and $R^5 = T$ gives the constraint $R^1 \quad R^6$. The next approximation to the constraints from the recursive call is $R^4 \quad R^5$. This gives the overall constraint $R^1 \quad R^6$, and a fixed-point is reached.
>
> If a bag is supplied, the result is a bag, but if the result is a bag, the parameter is not necessarily a bag.
>
> The type of *copy-until* is $B \quad S_{R^1} \quad S_{R^2} \quad S_{R^8}$, with the constraints $R^1 \quad R^3$, $R^2 = R^4 \quad R^5$, $R^7 \quad R^6$ and $R^8 = R^7 \quad R^4$. The initial approximation that $R^3 = T$, $R^5 = T$ and $R^6 = T$ leads to the constraint $R^8 \quad R^2$. This becomes the constraint $R^6 \quad R^5$ on the recursive call, and the fixed point is reached with the constraint $R^8 \quad R^2$.
>
> If the result is a bag, the parameter t must be a bag, but not necessarily vice versa.

The two functions demonstrate how bag-like streams may be inferred in different directions, depending on whether the streams are being read or constructed. The function *skip-until* repeatedly reads from a stream, so that if the original stream was a bag, the returned stream will be. The function *copy-until* repeatedly CONSes onto the stream t, so that if the returned stream is a bag, the original t will be.

Limitations and Extensions The above analysis is limited, however. Some functions, such as *map* and *filter* read one stream and produce another, and given the above analysis we cannot infer reorderability between these streams. It would be useful for the *map* function to have the type $S^R \quad S^R$, since it works in an item-wise fashion. An analysis that produces such typings is given in Appendix C.

7.5 Restoring Referential Transparency

SASL with non-deterministic extensions lacks referential transparency: identical function calls may lead to different results. However, it is another question as to whether this may lead to unsafe program folding/unfolding transformations. For a non-deterministic operation to be duplicated, the stream being read or shuffled must be duplicated. Linearity prevents stream variables being used repeatedly. However, if the entire stream is internally generated, and does not depend on a parameter stream, the entire piece of stream-generation code may be duplicated. In other words, the following function, valid in a referentially transparent system, shows the problems of nondeterminism:

$$\begin{array}{l}\textbf{fun}\ \textit{toggle}(x) = x\text{::}\textit{toggle}(\textit{not}(x))\\ \textbf{fun}\ \textit{folded}() = \textbf{let}\ x = \textit{hd}(\text{SHUFFLE}(\textit{toggle}(\textit{True})))\ \textbf{in}\ (x, x)\\ \textbf{fun}\ \textit{unfolded}() = (\textit{hd}(\text{SHUFFLE}(\textit{toggle}(\textit{True}))), \textit{hd}(\text{SHUFFLE}(\textit{toggle}(\textit{True}))))\end{array}$$

If the original syntax is to be used, care should be taken to avoid such transformations, either by disallowing non-deterministic operations that do not process any externally-supplied streams, or by never unfolding non-deterministic operations.

An alternative approach is to make the language purely functional again. To do this, we can use Burton's approach [31], where an infinite (lazy) tree of decisions is used to steer non-deterministic operations. The tree is supplied as a parameter to the function, and passed into all non-deterministic operations. When non-deterministic operations are unfolded, they receive the same tree, and take the same decision.

This approach must be modified for SASL. The infinite lazy tree could require infinite storage, if references to old parts of the tree are kept. Instead the tree is made linear, so that each non-deterministic operation takes a decision value and returns a new decision value, along with its result. With this constraint, referential transparency is not restored by forcing the same decisions to be taken in different unfolded branches, but rather by disallowing the unfolding of non-deterministic operations. A split operator would take a decision value and return two decision values, allowing multiple non-deterministic operations to occur in parallel.

As with streams, the non-deterministic decisions, rather than being represented by an object being passed in, could be represented as a monad, at the expense of moving away from the simple functional model.

7.6 Summary

In the other chapters of this thesis, deterministic streams have been assumed. This chapter has examined what happens if this assumption is broken. Non-determinism has been used in this chapter to reorder the reading of items from different streams, and the reading of items within a single stream.

Non-deterministic stream reading allows a wider range of I/O operations to be performed than otherwise, but can make a program's output non-deterministic. An analysis was presented that identifies the non-deterministically-generated values produced by a function.

Using multi-sets (bags) instead of lists gives the synthesis tool greater flexibility in situations where order is not important, and may result in higher-performance output. A number of possible syntaxes were presented, and an analysis given that is able to mark streams as being bag-like, given that the ordering on certain other streams is unimportant.

CHAPTER 8

Conclusions and Further Work

8.1 Conclusions

Recall the initial thesis from Chapter 1:

The thesis of this work is that statically-allocated pure functional languages, extended to use streams (linear lazy lists), are suitable languages for behavioural hardware synthesis of reactive systems. Furthermore, higher-level functional features such as closures and lazy evaluation may be usefully incorporated in a statically-allocated form to produce an optimising synthesis tool with a high level of abstraction.

The language presented in this thesis, SASL, meets the requirements for a "statically-allocated pure functional language", and uses streams. It is a behavioural language and was shown to be suitable for synthesis in Chapters 3 and 4. The stream I/O model was shown to fit closely with the reactive paradigm.

Chapters 6 and 7 show how higher-level features may be integrated into the language, and Chapter 5 demonstrates a number of optimisations may be performed. These features are useful in that they extend the power of the language and allow complex operations to be described simply. Overall, this thesis has provided evidence of the feasibility of this approach, showing that a high level of abstraction can be achieved. The language meets its targets of providing a useful research tool for the implementation of stream-processing algorithms, although the challenge remains to create an optimising version of the compiler which integrates well with a general industrial design-flow.

The rest of this chapter looks at extending SASL, both from the point of view of language features, and in terms of synthesis possibilities. Appendix B provides a simple case study.

8.2 Language Extensions

SASL, as presented in this thesis, is a relatively minimalist language, missing many of the features that might be expected of a modern ML variant. A practical language may be expected to have features such as a module or functor system, allowing the creation of abstract data types. This section discusses various language extensions which interact with SASL's streams or static allocation.

Arrays Memories are a common feature of hardware designs, despite our reluctance to use them in SASL due to possible von Neumann bottlenecks. Arrays are a common representation of memory blocks in hardware, but are typically not well-supported in pure functional languages. To update an array, there is typically a function that takes an array, an index and a new value, and returns a new array. However, there may be references to original array left, so that the entire array may need to be copied to perform the update. In SASL's type system we can use linear types to ensure that array usage is efficient.

Furthermore, the pure functional approach to arrays ensures that there is a fixed ordering on the array accesses, so that all memory accesses occur in a deterministic order. Non-deterministic array access could be achieved by including operators to "fork" and "join" the array value in a controlled manner, providing multiple sub-functions with simultaneous access to the array. Alternatively, arrays could only be split into non-overlapping sub-ranges, to provide deterministic parallelism. This approach is rather like the monadic state-splitting of Brisk [69].

Sized Types and Non-Linear Arrays SASL lacks the ability to natively describe an n-bit bus. Multi-bit values can be represented using tuples of boolean values, or more generally by using streams of boolean values. In an unrestricted language the stream approach would be more flexible, as n-bit operations could simply be described in terms of *map*, *fold* and so on. SASL, however, provides linearity constraints, and only allows for tail calls when processing recursive structures. Moreover, SASL will implement these buses serially, processing a single bit at a time and making poor use of the possible parallelism.

The reason for these restrictions is that SASL cannot make any assumptions about the size of the datatype. If the maximum size were known, it would no longer be necessary to process the data linearly, and limited non-tail recursion could be performed. A known size bound would allow the synthesis tool to unfold the function, so that the whole structure may be processed in parallel.

Sized types [70, 41] provide a way of reasoning about the size of data structures. Sized types normally provide "at least as large as" and "no bigger than" information, although for our purposes we are only interested in upper bounds on size. There is no reason to restrict ourselves to sized streams—we can use general size-limited restricted datatypes. For example, a restricted binary tree could be defined as follows:

$$\begin{aligned} \textbf{datatype}\ \alpha\ \textit{tree}^{i+1} &= \textit{Node}\ \textbf{of}\ \alpha\ \textit{tree}^{i} \quad \alpha\ \textit{tree}^{\ i} \\ \alpha\ \textit{tree}^{0} &= \textit{Leaf}\ \textbf{of}\ \alpha \end{aligned}$$

The superscripts define the size of the structure, in this case in terms of the number of nodes on paths to the leaves. The size is calculated recursively over the structure. Using such definitions, the storage requirements for any tree can be found from the concrete size values that are used as superscripts (just as the type variables are replaced with concrete types). As the size of the data structure is known, functions that include non-tail recursive calls can be created, as long as such calls can be determined to form a bounded depth call chain, based on the size of the structure. For example, the following function could be statically allocated with sized types:

$$\begin{aligned} \textbf{fun}\ \textit{flip}(\textit{Node}(a, b)) &= \textit{Node}(\textit{flip}(b), \textit{flip}(a)) \\ \textit{flip}(\textit{Leaf}(x)) &= \textit{Leaf}(x) \end{aligned}$$

These sized types effectively provide a type of polymorphism, and as with standard polymorphism, the top-level functions must provide the exact types, and the synthesis process compiles the program down to concrete types. For sized types, this involves unfolding the functions that work on sized types. The above function would be fully unfolded, and optimised down to a rearranging of the bus representing the value.

Another approach is to treat bus-like data structures as small arrays. Unlike the arrays described above, they need not be linear. As well as allowing the update and fetching of individual bits, mapping and folding functions may be provided. However, arrays do not tie in as well with the type system, and may be less flexible.

Exceptions Exceptions provide a useful mechanism to handle error conditions. For example, when processing an incoming stream of data where an error can be detected but not corrected, it may be useful to be able to raise an exception, and handle the error appropriately. However, it may not be obvious when an exception should be raised, for example if the exception is raised while generating a stream. The appropriate solution appears to be to raise the exception when the stream element where the exception occurred is matched. If lenient evaluation is used, the generated exception may be delayed, or even discarded unused.

In effect, exceptions are treated as variant types, arranged so that any eager expression that uses a value containing an exception immediately returns an exception. Streams, closures and promises allow the raising of the exception to be postponed. By implementing exceptions as variants, the synthesis system does not need to provide non-local jumps, so that existing value-killing reset circuitry does not need to be rearranged. On the other hand, it will complicate the normal datapath with tests for exception values. An alternative approach more similar to software exceptions may also be possible.

Exceptions would need restrictions in order to be statically allocatable. Preventing recursive tail calls from occurring within "try" blocks should be sufficient to allow static allocation.

Continuations Continuation-passing style is popular within some areas of the functional programming community, such as in Scheme [145]. A tail call is effectively a "goto", and by eliminating all function calls but tail calls we can view the program as a finite state machine with the functions as states, which may seem attractive for a hardware implementation. Continuations can also be used to construct a variety of complex control-flow structures.

However, the conversion to continuation-passing form does not seem useful. Continuation passing eliminates the stack, but SASL already eliminates the stack, and, as SASL unfolds rather than shares resources, every function statically knows where to return to. The use of continuations can hide ordering constraints normally expressed by expecting function calls to return in order, so that pipelining becomes difficult.

It seems feasible to construct a synthesis system based on continuation-passing, but continuations appear to be difficult to integrate with the synthesis approach presented in this thesis.

Streams in Algebraic Datatypes The basic SASL type system disallows streams from occurring inside algebraic datatypes. This restriction should be able to be removed. Killing an unused value of such an algebraic datatype must reset any streams it contains, which can be achieved by performing a kill on the enclosed value based on the datatype's tag. For the stability constraint, the stream identifiers on any streams passed recursively must match up for streams in the same position, just as with tuple types. A "Top" stream identifier is introduced for streams in datatypes that cannot occur—that is, if a stream cannot be passed back in that position, the constraint is met.

Monads Monads provide an alternative way of performing I/O in a pure functional language (as described in Section 2.1.3). It should be possible to macro-convert monads into stream reads and writes (function definitions that evaluate their arguments cannot be used for this translation, since the expressions used by the monad may need to be lazily evaluated).

Explicit Parallelism Explicit parallelism is a major feature of most HDLs, and in some situations it may be useful to introduce it to SASL, although it may be difficult to do so without losing the pure functional aspects of the language or introducing deadlock. The use of the join calculus [32] is one possible approach.

A more SASL-like approach would be to keep parallelism implicit, but allow the programmer to explicitly destroy the ordering dependencies that prevent parallel execution. SASL uses linear values to enforce ordering of operations, and by forking and joining these *values* (rather than forking and joining the control-flow) it may be possible to create programs in the style of explicit parallelism without resorting to parallel and sequential composition.

Streams of Streams and Linear Trees One of the last typing restrictions that is not directly associated with ensuring static allocation is the prevention of nested streams. From streams of streams, it should be possible to generalise to arbitrary linear recursive data-structures. With the appropriate stability constraints it should be possible to statically allocate programs with such data structures, although it seems that the hardware required would become increasingly complex.

Relaxing the Stability Constraint SASL's basic stability constraint requires that stream identifiers match up across recursive calls. A weaker, yet sufficient, constraint is that none of the streams passed recursively have a "$\star$" stream identifier. The number of streams and stream identifiers passed to the function is finite, so that the number of ways of arranging those stream identifiers in the function's parameter typing is also finite. By repeatedly unfolding the function, each recursive call path will eventually lead to a repeated arrangement of stream identifiers, so that we now have a set of mutually tail recursive functions that meet the original stability constraint. These functions can then be arranged into a single tail recursive function.

This conversion allows the creation of functions such as *interleave*:

$$\textbf{fun } \textit{interleave}(x\texttt{::}xs, ys) = x\texttt{::}\textit{interleave}(ys, xs)$$

However, the unfolding operation may lead to a blow-up in code size, and it is not clear how useful a less restrictive constraint would be in practice, although there may be more effective ways to relax the constraint.

Relaxing the Linearity Constraint Similarly, the linearity constraint could be made more flexible. If a stream has a bounded amount read from it, and then the original stream value is reused, this can be implemented with a fixed-size buffer. Similarly, if a stream is read twice while producing two streams, whose results are merged at the correct rate (in the style of a synchronous stream language), this should be synthesisable. It appears that this approach would require something akin to sized types (for typing a stream that has a finite amount read) or a clock calculus [60] (for deciding if a steam may be duplicated and later the values that depend on it merged).

The Dangers of Over-Extension This section has covered a variety of ways in which SASL could be extended. However, there are dangers in adding too many features. Compilers are expected to be reliable, as people wish to only debug their own code, and not have to deal with faulty code generation. Synthesis tools should be more reliable still (although in practice this appears not to be the case), as debugging hardware issues can be even more painful, and fixing a fielded broken system can be much more difficult and expensive. Increasing the complexity of the language can increase both the possibilities for compiler errors, and also the complexity the programmer must deal with, leading to unwieldy tools and possible user error. Producing sensible error messages becomes increasingly difficult as more complex language constraints are added. For example, the error messages produced by an extended linearity constraint failure could be quite obscure.

8.3 Synthesis Extensions

As well as extending the language itself, we can extend the synthesis tool, either providing incremental improvements, or changing the fundamental assumptions of the translation:

Loop Unfolding and Resource Awareness Although SASL attempts to pipeline execution, the parallelism may be limited by the number of simultaneously available pipeline stages in a loop. In order to improve throughput, it may be useful to either unroll loops, or produce multiple copies. For loops with known bounds, fully unrolling the loop removes the overhead of loop control. These transformations are likely to be most useful when applied to inner loops. This is similar to software loop unrolling [11], but performed for a different reason.

However, such transformations need to be performed within a resource-aware context: unfolding hardware takes up physical resources, which are unlikely to be unbounded. Without hardware constraints, impractical unfoldings may be performed. The unfolding can be constrained by the area usage, requiring it to stay below some limit, and either estimating the area required [157, 101] or performing feedback directed optimisation based on the results of low-level synthesis.

Loop unfolding also allows the program to be partially evaluated, or specialised [78]. The resulting specialised function may not only be faster than the original, but also take up less space, since functionality that is not required may be removed.

Resource Sharing SASL assumes that resource-sharing is not worthwhile, as it limits the ability to pipeline. This is overly simplistic, since performance can depend much more strongly on inner loops than on less frequently-called code. By sharing the resources used by infrequently-called code, more area becomes available to implement inner loops. Sharing resources may increase the amount of dynamic scheduling required, but as this occurs on less time-critical code this should not be a problem for performance.

There is a danger that this *program folding* will not improve area usage, as well as decreasing performance. The folding may prevent specialisation, and will require the use of multiplexers and other control circuitry to manage the sharing. For small functions, folding may introduce long wires from all the call sites to the shared function instance. These can take up space, complicate place-and-route and decrease the performance of the circuit. Hence optimising synthesis tools that can share resources will need to take into account these overheads when selecting functions to perform the transformation on.

Throughput Estimation In order to correctly select expressions for folding and unfolding, it is necessary to know how critical the function is to the program's performance. This could be estimated using programmer-supplied hints, by performing simple estimates, or by simulation on realistic data. Once the data is obtained parts of the program may be folded and unfolded until bottlenecks are removed. Note that production rates of different pipeline stages may be uneven, so that buffers may be needed between stages if good throughput is to be achieved.

Look-up Tables and Memoization Some functions may be very complex, but only work on a small domain, so that they may be efficiently implemented using a look-up table. Other functions could have a much larger domain, but dynamically only use a small fraction of it. Referential transparency allows us to *memoize* a function, that is, to provide a cache of the function's recent results. This makes it possible to return the appropriate value immediately upon receiving a call with the same parameters, without invoking the function's body (depending on whether different call sites to the same function are correlated or not, folding function instances may or may not improve memoization performance). Lenient evaluation even makes it possible to call the function in parallel with a table look-up, and cancel the request if the item is found. The tables can be implemented using similar content addressable memories (CAMs) to those used for processor caches [66].

Hardware/Software Co-Design As mentioned in the introductory chapter, hardware/software co-design is becoming increasingly important, especially in the application areas for which SASL is intended. SASL could be extended to allow co-design with both manual and automatic partitioning of functional programs. SASL's software-like approach may make it simpler to efficiently simulate on a software system than many other HDLs during the early stages of co-design.

Linear Values and "Unpipelining" SASL's synthesis approach relies on passing around data, latching it into sets of registers as it is passed through the pipeline. Values can be easily duplicated. While this is well-suited to processing relatively small data values in a pipelined fashion, this is not appropriate for large values. Normally, such values are stored in a memory, and we do not expect the values to be effortlessly duplicated. The use of arrays can be generalised to general linear data structures, so that the data becomes localised to one set of registers. The functions that process the value then work directly on the stored value. In such a situation, it makes sense to share the function instances working on that value, since each instance would be identical.

In effect, object instances are being created, with a fixed piece of encapsulated state, and a set of methods which perform operations on the object. The object is no longer passed along a pipeline, but individual function call requests will be sent to the "object". This approach can be used not only for large linear values whose implementations are represented within SASL, but can also be used as an interface to non-SASL resources.

This technique can be used to create large data structures with integrated processing operations that make use of parallelism. For example, a sized type could be used to create a heap data structure, and the heapify operation may be unfolded and distributed over the storage so that most operations can be performed in parallel (theoretically the large heaps could be implemented by passing the heap values along a pipeline, but this would require unrealistic amounts of storage). Data structures may be used that rely on the available parallelism for efficiency, such as the tagged up/down sorter [106].

Once large, linear values are being created, it may be useful to extend the type system to simplify the generalised in-place modification of such values without requiring extra memory. For example, a list reversal may be performed in place, but may naïvely use extra buffering. Hofmann's work on in-place updates may be applicable [68].

In a similar vein, analyses based on globalization [58] may be used to find values which can be pulled out of pipelines, so that they may be held constant in a register external to the pipeline, rather than being duplicated across each stage. This should allow resource usage due to pipeline latches to be reduced.

APPENDIX A

Example node implementations

This appendix contains example implementations of a few representative nodes used in the graph-based synthesis. They are designed to always produce partial normality if composed correctly, and be pipelineable. This appendix describes a simple signalling model which is also used in Appendix B, and then discusses the implementation of a few example node types, namely normal nodes, CONS, stream matching and reset boxes. The nodes are very much proof of concept, and are designed with simplicity in mind, rather than optimisation for speed or area. A section at the end discusses the implementation of other node types. Although the nodes here assume a synchronous implementation, there is nothing in principle preventing an asynchronous version.

A.1 Signalling

In the example synthesis, two-phase signalling has been relied upon. For each signalling line, there is a request wire and an acknowledgement wire. Events are signalled by edges of the request line, and acknowledgements by edges of the acknowledgement line, so that when the request has been acknowledged both lines will be in the same state. Only one request may be outstanding at a time.

The usefulness of this signalling mechanism comes from how signals may be combined. Requests can be merged by waiting for all request lines to go high before setting the output high, and waiting for all to go low before setting the output low, while the acknowledgement consists simply of replicating the acknowledgement signal. Forking a request value can be coped with by joining acknowledgement lines in a similar fashion.

Waiting for all inputs to become the same before changing output state can be achieved using Muller-C elements. The implementation given here relies on a module that not only detects completion, but also copes with reset signals and provides a one-cycle pulse when new data arrives to simplify the implementation of nodes. This `sync` module is defined as follows:

```
module sync(clk, reset, newdata, ready, latch, inreqhi, inreqlo, inack, outreq, outackhi, outacklo);

input clk, reset, inreqhi, inreqlo, outackhi, outacklo, ready;
output inack, outreq, newdata, latch;
reg inack, outreq, newdata, got_in, got_out;

assign latch = !got_in & ((inreqhi & ~inack) | (~inreqlo & inack));

always @(posedge clk) begin
    if (reset) begin
        inack <= 0;
        outreq <= 0;
        newdata <= 0;
        got_in <= 0;
        got_out <= 0;
    end else begin
        newdata <= latch;

        if (latch) begin
            inack <= inreqhi;
            got_in <= 1;
        end

        if (ready) begin
            outreq <= ~outreq;
            got_out <= 1;
        end

        if (got_out & ((outackhi & outreq) | (~outacklo & ~outreq))) begin
            got_in <= 0;
            got_out <= 0;
        end
    end
end

endmodule
```

The module's I/O lines are as follows:

> The lines `inreq` and `inack` receive requests from previous stages and acknowledge them, respectively. Similarly, `outreq` and `outack` synchronise with the next stage.
>
> As there may be any number of predecessor and successor nodes, the `inreq` and `outack` lines are actually implemented as a pair of lines (`inreqhi` and `inreqlo`, and `outackhi` and `outacklo`). The `hi` line is fed with the `and`ing of the input lines, and `lo` with the `or`ings. In this way, both the required "all high" and "all low" signals can be detected.
>
> The clock (`clk`) and reset (`reset`) lines provide the global synchronous clock and a reset line. The reset line is not global, but associated with the enclosing reset box, so that parts of the circuit may be selectively cleared.
>
> The `latch` signal informs the data-flow part of the node that new data should be latched, to be processed on the next cycle.
>
> The `newdata` signal informs the data-flow part of the node that the data latched on the previous cycle is ready to be processed. Once processing completes, the `ready` line is asserted for a cycle. Simple combinatorial circuits connect `latch` directly to `newdata`.

Since streams do not have to be synchronised together, a simple level-sensitive scheme can be used. The stream bus consists of a register to hold the "has data" value, plus a register for the data itself. Acknowledgement simply clears the "has data" value. Resets are provided by a stream bus reset line being held high for a single cycle. For basic lenient evaluation (as used in Appendix B), an extra "activity" line can be provided, signalling that data is being produced on the bus, to prevent another CONS node from starting execution. This is the model that will be used in the nodes of this appendix, although more complex systems will use a separate stream bus controller.

A.2 Normal Nodes

The example normal node is a module that takes two inputs, and returns two outputs, consisting of the sum and difference of those inputs:

```
module addsub(clk, reset, req_i1, req_i2, ack_i1, ack_i2, req_o1, req_o2, ack_o1, ack_o2,
    data_i1, data_i2, data_o1, data_o2);

input clk, reset, req_i1, req_i2, ack_o1, ack_o2;
output req_o1, req_o2, ack_i1, ack_i2;
input [7:0] data_i1, data_i2;
output [7:0] data_o1, data_o2;

wire ack, req, newdata, latch;
reg [7:0] d1, d2;

/* Synchronisation. */
sync s(.clk(clk), .reset(reset), .newdata(newdata), .ready(newdata), .latch(latch),
    .inreqhi(req_i1 & req_i2), .inreqlo(req_i1 | req_i2), .inack(ack),
    .outreq(req), .outackhi(ack_o1 & ack_o2), .outacklo(ack_o1 | ack_o2));

assign ack_i1 = ack;
assign ack_i2 = ack;
assign req_o1 = req;
assign req_o2 = req;

/* Data latching. */
always @(posedge clk) begin
    if (latch) begin
        d1 <= data_i1;
        d2 <= data_i2;
    end
end

/* Data processing. */
assign data_o1 = d1 + d2;
assign data_o2 = d1 - d2;

endmodule
```

The main work is done in the instance of the `sync` module. The `ready` line is connected to the `newdata` line so that the results become available on the next cycle, since the data processing is simply a combinatorial function. The inputs are latched as the data is received.

A.3 CONS Nodes

A CONS node may be implemented by the following Verilog-like code (see below for why pure Verilog is not used):

```
module cons_e1_e2(clk, reset, req, ack, data, str_act, str_res, str_data);

input clk, reset, req;
output ack, str_act, str_res;

input [7:0] data;
output [7:0] str_data;

reg [7:0] str_data;
reg str_act, str_res, ack, sub_req_in, sub_ack_out;

wire sub_ack_in1, sub_ack_in2, sub_req_out, ready;
wire [7:0] result;

assign ready = ((sub_req_in & sub_ack_in1 & sub_ack_in2) | !(sub_req_in | sub_ack_in1 | sub_ack_in2)) & !str_act;

always @(posedge clk) begin
    if (reset) begin
        sub_req_in <= 0;
        sub_ack_out <= 0;
        ack <= 0;
    end else begin
        // Input request.
        if (ready) begin
            str_act <= 1;
            sub_req_in <= req;
        end
        // Input acknowledge.
        if ( sub_ack_in1 &  sub_ack_in2) ack <= 1;
        if (!sub_ack_in1 & !sub_ack_in2) ack <= 0;
        // Collect results.
        if (sub_req_out != sub_ack_out) begin
            sub_ack_out <= sub_req_out;
            str_data <= result;
            str_res <= 1;
        end
    end
end

e1 inst1(.clk(clk), .reset(reset), .data_i(data), .data_o(result),
    .req_i(sub_req_in), .ack_i(sub_ack_in1), .req_o(sub_req_out), .ack_o(sub_ack_out));

e2 inst2(.clk(clk), .reset(reset), .data(data),
    .req(sub_req_in), .ack(sub_ack_in2), .str_act(str_act), .str_res(str_res), .str_data(str_data));

endmodule
```

A real Verilog implementation would need to provide extra buses to the instance of `e2` in order to allow connection back to any multiplexer that encloses this node. Furthermore, the `str_act`, `str_res` and `str_data` data lines are treated as registers which may be shared between the different modules. This is disallowed in Verilog, but the same effect can be achieved by either implementing the program in a flattened description (as is used for Appendix B's example), or by providing both input and output nets and some extra logic. This would complicate the explanation of the module, and so has been omitted.

The CONS module works by passing new requests on to both the head and tail subgraph when they are idle. The stream is marked as active (using `str_act`), to prevent other CONS nodes writing to it, as part of basic lenient evaluation (see Section 5.2.3), and the input acknowledged. The node provides no scalar return value, since it only outputs to the stream. Once `e1` produces a result, the value is acknowledged, and written to the stream bus's data bus (`str_data`), and `str_res` is set high to signal available results. The appropriate match node will take this data, and reset the `str_act` and `str_res` lines, allowing the next CONS node to evaluate.

Note that the CONS module does not need to handle stream resets explicitly. These are handled by the enclosing reset box for the stream, which will send an appropriate signal on the `reset` line.

A.4 Match Nodes

A match node can be implemented with the following Verilog-like code (again, the `str_*` buses have been simplified to make the code more readable):

```
module match(clk, reset, req_i, ack_i, req_o, ack_o, data_o, str_act, str_res, str_data);

input clk, reset, req_i, ack_o;
output ack_i, req_o;

output [7:0] data_o;

inout str_act, str_res;
inout [7:0] str_data;

reg str_act, str_res, waiting;
reg [7:0] data_o;

wire newdata, ready, latch;

/* Synchronisation. */
sync s(.clk(clk), .reset(reset), .newdata(newdata), .ready(ready), .latch(latch),
    .inreqhi(req_i), .inreqlo(req_i), .inack(ack_i),
    .outreq(req_o), .outackhi(ack_o), .outacklo(ack_o1));

assign ready = (newdata | waiting) & str_res;

always @(posedge clk) begin
    if (reset) begin
        waiting <= 0;
    end else begin
        if (newdata & !str_res) waiting <= 1;
        if (ready) begin
            data_o <= str_data;
            str_act <= 0;
            str_res <= 0;
            waiting <= 0;
        end
    end
end

endmodule
```

The module works by reading the value from the stream bus and returning it, while clearing the stream bus activity and result flags in order to allow the next item to be produced. If an item is not available when the module is activated, the `waiting` flag is set so that the result can be returned when it does become available.

A.5 Reset Nodes

The example code below encloses a function which takes no scalar parameters and returns no scalar results, but takes a stream input (represented by `str_i_*`), and returns a stream output (`str_o_*`):

```
module reset_e(clk, reset, req_i, ack_i, req_o, ack_o,
    str_i_act, str_i_res, str_i_data, str_i_rst, str_o_act, str_o_res, str_o_data, str_o_rst);

input clk, reset, req_i, ack_o, str_i_act, str_i_res, str_o_rst;
output ack_i, req_o, str_o_act, str_o_res, str_i_rst;

input [7:0] str_i_data;
output [7:0] str_o_data;

wire str_i_rst2, new_reset;
reg str_i_is_reset;

assign new_reset = reset | str_o_rst;
assign str_i_rst = str_i_rst2 | (new_reset & !str_i_is_reset);

e inste(.clk(clk), .reset(new_reset), .req_i(req_i), .ack_i(ack_i), .req_o(req_o), .ack_o(ack_o),
    .str_i_act(str_i_act), .str_i_res(str_i_res), .str_i_data(str_i_data), .str_i_rst(str_i_rst2),
    .str_o_act(str_o_act), .str_o_res(str_o_res), .str_o_data(str_o_data), .str_o_rst(str_o_rst));

always @(posedge clk) begin
    if (new_reset)
        str_i_is_reset <= 0;
    else if (str_i_rst2)
        str_i_is_reset <= 1;
end

endmodule
```

For the computational reset, all that the node does is supply the enclosed node with a reset signal that goes high if the output stream bus's reset (`str_o_rst`) goes high. To forward the stream reset, `str_i_rst` is triggered if `str_o_rst` goes high, *provided that the stream has not already been reset since the last time the node was activated*. This prevents two streams on the same stream bus being killed if the node's input stream is killed before the output stream is killed.

A.6 Other Nodes

The other node types can be implemented as follows:

Conditional nodes can be implemented most simply if only a single item is allowed in at a time, although a pipelined version with in-order collection is quite possible. Requests are forwarded to the appropriate sub-module, and when any results appear they are forwarded to the output, with acknowledgements routed appropriately. Top-half-only conditionals (see Section 5.2.6) are simpler still, as no mutual exclusion or collection of results is required.

Multiplexers are implemented as nodes that may be triggered from one of a number of sites. Arbitration is not required amongst the inputs, as only a single set tokens will be passed around the tail part of a multiplexer's subgraph at any time.

Iteration nodes work similarly, except the produced value may be routed to either cause another iteration, or generate a result. The simplest implementation relies on preventing more than one set of tokens from entering at a time. More complex, pipelined solutions must be prepared to sort results into the correct order before emitting them.

Mutual exclusion boxes are set up like normal nodes, except that a flag is kept of whether the appropriate stream bus has been reset, and new items will not be accepted until that event has occurred.

Stream forwarders can be implemented within the language itself, and synthesised to other basic node types.

Stream kill nodes simply hold the reset line of the appropriate stream bus high for one cycle, when triggered.

APPENDIX B

Case Study

This appendix examines a simple case study, and the performance of the synthesised circuits. The first section introduces the example, Section B.2 examines the CSP synthesis, and Section B.3 examines graph synthesis. Section B.4 finishes off by providing an overview of the performance seen.

B.1 The Example

This appendix provides a small synthesis example. A simple program is translated, as the synthesis tools produces rather voluminous output. The program is a sawtooth-wave generator. It takes a stream of (*duration*, *pitch*) pairs, and produces a stream of 8-bit values representing the wave. Each waveform lasts for *duration* cycles, with the increment in output value changing by *pitch* each time.

The code is shown in Figure B.1. Rather than building up the arithmetic operations up from first principles, the tools use primitives supplied by the underlying low-level HDL, to reduce clutter in the synthesis results. The synthesis tools for CSP and graph-based synthesis are described in the following sections.

Both compilers are proof-of-concept, and rely on simplistic, inefficient code generation techniques. Some inefficiencies, such as the introduction of redundant wires, are easily optimised away by the low-level synthesis tool, while others, such as the creation of unnecessary pipeline stages or inefficient signalling lines, would require much more difficult optimisation, and would best be fixed by improving the SASL compiler's output stage.

Although the example program appears relatively simple, the volume of synthesised code is not totally unreasonable. Rather than being a very simple signal generator, the produced code includes full streams, with back-pressure, resets and the scope for pipelining. While such features may be overkill for a circuit of this size, they become increasingly useful in larger programs.

Both compilers use a flattened output approach, producing a single module containing the entire graph, rather than many individual modules representing individual nodes. As there are few pairs of nodes which are identical, this does not lead to a great amount of redundancy, and reduces the complexity of the synthesised output, as the individual modules do not need to be wired together. This makes both synthesis easier and the output shorter, although a module-structured graph may make life easier for humans working with the generated code.

B.2 CSP Synthesis

The Handel-C [39] language was chosen as the target for CSP synthesis, as the language includes all the necessary CSP features and is intended for synthesis to hardware. The Handel-C tools include a simulator, which was used to test the tool's output.

The synthesis process implemented is very similar to that described in Chapter 3, with only a few minor differences. For example, the stream buses were forwarded in the CONS and constructor-matching expressions, rather than in the variable access expression (effectively forwarding as late as possible rather than earlier), as this fitted more easily into the compilation framework.

```
(* Test program for CSP synthesis. *)
fun signal_internal(remaining, value, step, commands) =
    if isnonzero(remaining)
    then value::signal_internal(decr(remaining), add(value, step), step, commands)
    else case commands of
        (count, step)::commands    signal internal(count, zero(), step, commands)

fun signal commands = signal_internal(zero(), zero(), zero(), commands)

fun main x = signal x
```

Figure B.1: The example SASL program

As can be seen from the code below, the synthesis tool is very inefficient. Each basic operation creates a new process (or set of processes), with its own input, output and reset channels. Static scheduling (see Section 5.1) is necessary to merge operations into a more manageable number of processes. The large number of declarations also stems from the many unread variables which are created as the destinations for the reads from 0-bit signalling channels.

The synthesised code is as follows:

```
void main(void)
{
unsigned 0 var_124; unsigned 0 var_123; unsigned 0 var_122;
unsigned 0 var_121; unsigned 0 var_120; unsigned 0 var_119;
unsigned 0 var_118; unsigned 0 var_117; unsigned 0 var_116;
unsigned 0 var_115; unsigned 0 var_114; unsigned 0 var_113;
unsigned 0 var_112; unsigned 0 var_111; unsigned 0 var_110;
unsigned 8 var_109; unsigned 0 var_108; unsigned 0 var_107;
chan unsigned 0 zero_66_reset; chan unsigned 8 zero_66_out;
chan unsigned 0 zero_66_in; unsigned 0 x_67;
chan unsigned 0 chan_101; chan unsigned 0 chan_100;
chan unsigned 0 chan_99; unsigned 0 var_106;
unsigned 0 var_105; unsigned 0 var_104; unsigned 0 var_103;
unsigned 8 var_102; unsigned 0 var_101; unsigned 0 var_100;
chan unsigned 0 zero_64_reset; chan unsigned 8 zero_64_out;
chan unsigned 0 zero_64_in; unsigned 0 x_65;
chan unsigned 0 chan_98; chan unsigned 0 chan_97;
chan unsigned 0 chan_96; unsigned 0 var_99;
unsigned 0 var_98; unsigned 0 var_97; unsigned 0 var_96;
unsigned 8 var_95; unsigned 0 var_94; unsigned 0 var_93;
chan unsigned 0 zero_62_reset; chan unsigned 8 zero_62_out;
chan unsigned 0 zero_62_in; unsigned 0 x_63;
chan unsigned 0 chan_95; chan unsigned 0 chan_94;
chan unsigned 0 chan_93; unsigned 0 var_92;
unsigned 8 var_91; unsigned 8 var_90; unsigned 8 var_89;
chan unsigned 0 chan_92; chan unsigned 0 chan_91;
chan unsigned 0 chan_90; chan unsigned 0 chan_89;
chan unsigned 0 chan_88; chan unsigned 8 chan_87;
chan unsigned 8 chan_86; chan unsigned 8 chan_85;
chan unsigned 0 chan_84; chan unsigned 0 chan_83;
chan unsigned 0 chan_82; chan unsigned 0 chan_81;
unsigned 0 var_88; unsigned 0 var_87; unsigned 0 var_86;
unsigned 0 var_85; unsigned 0 var_84; unsigned 0 var_83;
unsigned 0 var_82; unsigned 0 var_81; unsigned 0 var_80;
unsigned 0 var_79; unsigned 0 var_78; unsigned 0 var_77;
unsigned 0 var_76; unsigned 0 var_75; unsigned 0 var_74;
unsigned 0 var_73; unsigned 0 var_72; unsigned 0 var_71;
unsigned 0 var_70; unsigned 0 var_69; unsigned 0 var_68;
unsigned 0 var_67; unsigned 0 var_66; unsigned 0 var_65;
unsigned 0 var_64; unsigned 0 var_63; unsigned 0 var_62;
unsigned 8 var_61; unsigned 8 var_60;
chan unsigned 0 chan_80; chan unsigned 0 chan_79;
chan unsigned 8 chan_78; chan unsigned 8 chan_77;
chan unsigned 0 chan_76; chan unsigned 0 chan_75;
unsigned 8 var_59; unsigned 0 var_58; unsigned 0 var_57;
chan unsigned 0 add_76_reset; chan unsigned 8 add_76_out;
chan unsigned 0 add_76_in; unsigned 16 x_77;
chan unsigned 0 chan_74; chan unsigned 16 chan_73;
chan unsigned 0 chan_72; unsigned 0 var_56;
unsigned 0 var_55; unsigned 0 var_54; unsigned 0 var_53;
unsigned 8 var_52; unsigned 0 var_51; unsigned 0 var_50;
chan unsigned 0 decr_74_reset;
chan unsigned 8 decr_74_out; chan unsigned 0 decr_74_in;
unsigned 8 x_75; chan unsigned 0 chan_71;
chan unsigned 8 chan_70; chan unsigned 0 chan_69;
unsigned 0 var_49; unsigned 8 var_48; unsigned 8 var_47;
unsigned 8 var_46;
chan unsigned 0 chan_68; chan unsigned 0 chan_67;
chan unsigned 0 chan_66; chan unsigned 0 chan_65;
chan unsigned 0 chan_64; chan unsigned 8 chan_63;
chan unsigned 8 chan_62; chan unsigned 8 chan_61;
chan unsigned 0 chan_60; chan unsigned 0 chan_59;
chan unsigned 0 chan_58; chan unsigned 0 chan_57;
chan unsigned 0 chan_56; chan unsigned 24 chan_55;
chan unsigned 0 chan_54; unsigned 0 var_45;
unsigned 0 var_44; chanout unsigned 8 Sack_9;
chan unsigned 0 Sreq_9; unsigned 8 var_43;
chan unsigned 0 chan_53; chan unsigned 0 chan_52;
chan unsigned 0 chan_51; chan unsigned 8 chan_50;
chan unsigned 0 chan_49; chan unsigned 0 chan_48;
chan unsigned 0 chan_47; unsigned 0 var_42;
unsigned 0 var_41; unsigned 0 var_40; unsigned 0 var_39;
unsigned 0 var_38; unsigned 0 var_37; unsigned 0 var_36;
unsigned 0 var_35; unsigned 0 var_34; unsigned 0 var_33;
unsigned 0 var_32; unsigned 0 var_31; unsigned 0 var_30;
unsigned 0 var_29; unsigned 0 var_28; unsigned 0 var_27;
unsigned 0 var_26; unsigned 0 var_25; unsigned 8 var_24;
unsigned 0 var_23; unsigned 0 var_22;
chan unsigned 0 zero_82_reset; chan unsigned 8 zero_82_out;
chan unsigned 0 zero_82_in; unsigned 0 x_83;
chan unsigned 0 chan_46; chan unsigned 0 chan_45;
chan unsigned 0 chan_44; unsigned 0 var_21;
unsigned 0 var_20; unsigned 0 var_19; unsigned 8 var_18;
unsigned 8 var_17; unsigned 8 var_16;
chan unsigned 0 chan_43; chan unsigned 0 chan_42;
chan unsigned 0 chan_41; chan unsigned 0 chan_40;
chan unsigned 0 chan_39; chan unsigned 8 chan_38;
chan unsigned 8 chan_37; chan unsigned 8 chan_36;
chan unsigned 0 chan_35; chan unsigned 0 chan_34;
chan unsigned 0 chan_33; chan unsigned 0 chan_32;
chan unsigned 0 chan_31; chan unsigned 24 chan_30;
chan unsigned 0 chan_29; unsigned 0 var_15;
unsigned 0 var_14; unsigned 8 step_81;
unsigned 8 count_80; unsigned 16 var_13;
chan unsigned 0 chan_28; chan unsigned 0 chan_27;
chan unsigned 0 chan_26; chan unsigned 16 chan_25;
chan unsigned 0 chan_24; unsigned 0 var_12;
unsigned 0 var_11; unsigned 0 commands_79;
unsigned 16 TMP_1_78; chan unsigned 16 Sack_7;
chan unsigned 0 Sreq_7; chan unsigned 0 chan_23;
chan unsigned 0 chan_22; chan unsigned 0 chan_21;
chan unsigned 0 chan_20; chan unsigned 0 chan_19;
chan unsigned 0 chan_18; chan unsigned 0 chan_17;
unsigned 1 var_10; unsigned 0 var_9; unsigned 0 var_8;
unsigned 0 var_7; unsigned 0 var_6; unsigned 1 var_5;
unsigned 0 var_4; unsigned 0 var_3;
chan unsigned 0 isnonzero_84_reset;
chan unsigned 1 isnonzero_84_out;
chan unsigned 0 isnonzero_84_in;
unsigned 8 x_85; chan unsigned 0 chan_16;
chan unsigned 8 chan_15; chan unsigned 0 chan_14;
chan unsigned 0 chan_13; chan unsigned 1 chan_12;
chan unsigned 0 chan_11; unsigned 0 var_2;
unsigned 0 var_1; unsigned 0 commands_73;
unsigned 8 step_72; unsigned 8 value_71;
unsigned 8 remaining_70; unsigned 24 var_0;
chan unsigned 0 chan_10; chan unsigned 0 chan_9;
chan unsigned 0 chan_8; chan unsigned 24 chan_7;
chan unsigned 0 chan_6;
chan unsigned 0 signal_internal_68_reset;
chan unsigned 0 signal_internal_68_out;
chan unsigned 0 signal_internal_68_in;
unsigned 24 TMP_2_69; chan unsigned 0 chan_5;
chan unsigned 24 chan_4; chan unsigned 0 chan_3;
chan unsigned 0 signal_60_reset;
chan unsigned 0 signal_60_out;
chan unsigned 0 signal_60_in;
unsigned 0 commands_61; chan unsigned 0 chan_2;
chan unsigned 0 chan_1;
chan unsigned 0 chan_0; chan unsigned 0 main_58_reset;
chan unsigned 0 main_58_out; chan unsigned 0 main_58_in;
unsigned 0 x_59;
{
par {
        /* Test harness. */
        {
                unsigned int 0 result;
                main_58_reset ! 0;
                x_59 = 0;
                main_58_in ! 0;
                main_58_out ? result;
                for(;;) { Sreq_9 ! 0; }
        }
        { Sreq_7 ? x_59; Sack_7 ! 0x1020;
          Sreq_7 ? x_59; Sack_7 ! 0x2020; }
        /* Generated code. */
        for(;;) prialt {
        case main_58_in ? var_124:
                { chan_2 ! 0; chan_1 ? commands_61;
                signal_60_in ! 0; signal_60_out ? var_120;
                main_58_out ! var_120; } break;
        case main_58_reset ? var_123:
                par { chan_0 ! 0; signal_60_reset ! 0; }
                break;
        }
        for(;;) prialt {
        case chan_2 ? var_122:
                chan_1 ! x_59; break;
        case chan_0 ? var_121:
                {} break;
        }
        for(;;) prialt {
        case signal_60_in ? var_119:
                { chan_5 ! 0; chan_4 ? TMP_2_69;
                signal_internal_68_in ! 0;
                signal_internal_68_out ? var_88;
                signal_60_out ! var_88; } break;
        case signal_60_reset ? var_118:
                par { chan_3 ! 0;
                signal_internal_68_reset ! 0; } break;
        }
        for(;;) prialt {
        case chan_5 ? var_117:
                { par { chan_81 ! 0; chan_82 ! 0;
                chan_83 ! 0; chan_84 ! 0; } par {
                chan_85 ? var_89; chan_86 ? var_90;
                chan_87 ? var_91; chan_88 ? var_92; }
                chan_4 ! (var_92) @ (var_91) @ (var_90) @
                (var_89); } break;
        case chan_3 ? var_116:
                par { chan_89 ! 0; chan_90 ! 0; chan_91 ! 0;
                chan_92 ! 0; } break;
        }
        for(;;) prialt {
        case chan_84 ? var_115:
                chan_88 ! commands_61; break;
        case chan_92 ? var_114:
                {} break;
        }
        for(;;) prialt {
        case chan_83 ? var_113:
                { chan_101 ! 0; chan_100 ? x_67;
                zero_66_in ! 0; zero_66_out ? var_109;
                chan_87 ! var_109; } break;
        case chan_91 ? var_112:
                par { chan_99 ! 0; zero_66_reset ! 0; }
                break;
        }
        for(;;) prialt {
        case chan_101 ? var_111:
                chan_100 ! 0; break;
        case chan_99 ? var_110:
                {} break;
        }
        for(;;) prialt {
        case zero_66_in ? var_108:
                { zero_66_out ! 0; } break;
        case zero_66_reset ? var_107:
                {} break;
        }
        for(;;) prialt {
        case chan_82 ? var_106:
                { chan_98 ! 0; chan_97 ? x_65;
                zero_64_in ! 0; zero_64_out ? var_102;
                chan_86 ! var_102; } break;
        case chan_90 ? var_105:
                par { chan_96 ! 0; zero_64_reset ! 0; }
                break;
        }
        for(;;) prialt {
        case chan_98 ? var_104:
                chan_97 ! 0; break;
        case chan_96 ? var_103:
                {} break;
        }
        for(;;) prialt {
        case zero_64_in ? var_101:
                { zero_64_out ! 0; } break;
        case zero_64_reset ? var_100:
                {} break;
        }
        for(;;) prialt {
        case chan_81 ? var_99:
                { chan_95 ! 0; chan_94 ? x_63;
                zero_62_in ! 0; zero_62_out ? var_95;
                chan_85 ! var_95; } break;
        case chan_89 ? var_98:
                par { chan_93 ! 0; zero_62_reset ! 0; }
                break;
        }
        for(;;) prialt {
        case chan_95 ? var_97:
                chan_94 ! 0; break;
        case chan_93 ? var_96:
                {} break;
        }
        for(;;) prialt {
        case zero_62_in ? var_94:
                { zero_62_out ! 0; } break;
        case zero_62_reset ? var_93:
                {} break;
        }
        for(;;) prialt {
        case signal_internal_68_in ? var_87:
                { chan_8 ! 0; chan_7 ? var_0; par {
                remaining_70 = (var_0)[7:0];
                value_71 = (var_0)[15:8];
                step_72 = (var_0)[23:16]; commands_73 = 0; }
                chan_10 ! 0; } break;
        case signal_internal_68_reset ? var_86:
                par { chan_6 ! 0; chan_9 ! 0; } break;
        }
        for(;;) prialt {
        case chan_10 ? var_85:
                { chan_13 ! 0; chan_12 ? var_10;
                switch(var_10[0:0]) { case 0: chan_18 ! 0;
                break; case 1: chan_48 ! 0; break;} }
                break;
        case chan_9 ? var_84:
                par { chan_11 ! 0; chan_17 ! 0;
                chan_47 ! 0; } break;
        }
        for(;;) { chan_50 ? var_43; chan_53 ! 0;
                signal_internal_68_out ? var_83;
                Sack_9 ! var_43; }
        for(;;) prialt {
        case chan_48 ? var_82:
                { signal_internal_68_out ! 0; prialt {
        case Sreq_9 ? var_80:
                chan_51 ! 0; break;
        case chan_47 ? var_79:
                par { chan_49 ! 0; chan_52 ! 0; } break;
        } } break;
        case chan_47 ? var_81:
                par { chan_49 ! 0; chan_52 ! 0; } break;
        }
        for(;;) prialt {
        case chan_53 ? var_78:
                { chan_56 ! 0; chan_55 ? TMP_2_69; par {
                chan_54 ! 0; signal_internal_68_reset ! 0; }
                chan_52 ? var_76;
                signal_internal_68_in ! 0; } break;
        case chan_52 ? var_77:
                chan_54 ! 0; break;
        }
        for(;;) prialt {
        case chan_56 ? var_75:
                { par { chan_57 ! 0; chan_58 ! 0;
                chan_59 ! 0; chan_60 ! 0; } par {
                chan_61 ? var_46; chan_62 ? var_47;
                chan_63 ? var_48; chan_64 ? var_49; }
                chan_55 ! (var_49) @ (var_48) @ (var_47) @
        (var_46); } break;
case chan_54 ? var_74:
        par { chan_65 ! 0; chan_66 ! 0; chan_67 ! 0;
        chan_68 ! 0; } break;
}
for(;;) prialt {
case chan_60 ? var_73:
        chan_64 ! commands_73; break;
case chan_68 ? var_72:
        {} break;
}
for(;;) prialt {
case chan_59 ? var_71:
        chan_63 ! step_72; break;
case chan_67 ? var_70:
        {} break;
}
for(;;) prialt {
case chan_58 ? var_69:
        { chan_74 ! 0; chan_73 ? x_77;
        add_76_in ! 0; add_76_out ? var_59;
        chan_62 ! var_59; } break;
case chan_66 ? var_68:
        par { chan_72 ! 0; add_76_reset ! 0; }
        break;
}
for(;;) prialt {
case chan_74 ? var_67:
        { par { chan_75 ! 0; chan_76 ! 0; } par {
        chan_77 ? var_60; chan_78 ? var_61; }
        chan_73 ! (var_61) @ (var_60); } break;
case chan_72 ? var_66:
        par { chan_79 ! 0; chan_80 ! 0; } break;
}
for(;;) prialt {
case chan_76 ? var_65:
        chan_78 ! step_72; break;
case chan_80 ? var_64:
        {} break;
}
for(;;) prialt {
case chan_75 ? var_63:
        chan_77 ! value_71; break;
case chan_79 ? var_62:
        {} break;
}
for(;;) prialt {
case add_76_in ? var_58:
        { add_76_out ! x_77[7:0] + x_77[15:8]; }
        break;
case add_76_reset ? var_57:
        {} break;
}
for(;;) prialt {
case chan_57 ? var_56:
        { chan_71 ! 0; chan_70 ? x_75;
        decr_74_in ! 0; decr_74_out ? var_52;
        chan_61 ! var_52; } break;
case chan_65 ? var_55:
        par { chan_69 ! 0; decr_74_reset ! 0; }
        break;
}
for(;;) prialt {
case chan_71 ? var_54:
        chan_70 ! remaining_70; break;
case chan_69 ? var_53:
        {} break;
}
for(;;) prialt {
case decr_74_in ? var_51:
        { decr_74_out ! x_75 - 1; } break;
case decr_74_reset ? var_50:
        {} break;
}
for(;;) prialt {
case chan_51 ? var_45:
        chan_50 ! value_71; break;
case chan_49 ? var_44:
        {} break;
}
for(;;) prialt {
case chan_18 ? var_42:
        { chan_21 ! 0; chan_20 ? var_40; Sreq_7 ! 0;
        Sack_7 ? TMP_1_78; commands_79 = 0;
        chan_23 ! 0; } break;
case chan_17 ? var_41:
        par { chan_19 ! 0; chan_22 ! 0; } break;
}
for(;;) prialt {
case chan_23 ? var_39:
        { chan_26 ! 0; chan_25 ? var_13; par {
        count_80 = (var_13)[7:0];
        step_81 = (var_13)[15:8]; } chan_28 ! 0; }
        break;
case chan_22 ? var_38:
        par { chan_24 ! 0; chan_27 ! 0; } break;
}
for(;;) prialt {
case chan_28 ? var_37:
        { chan_31 ! 0; chan_30 ? TMP_2_69; par {
        chan_29 ! 0; signal_internal_68_reset ! 0; }
        chan_27 ? var_35;
        signal_internal_68_in ! 0; } break;
case chan_27 ? var_36:
        chan_29 ! 0; break;
}
for(;;) prialt {
case chan_31 ? var_34:
        { par { chan_32 ! 0; chan_33 ! 0;
        chan_34 ! 0; chan_35 ! 0; } par {
        chan_36 ? var_16; chan_37 ? var_17;
        chan_38 ? var_18; chan_39 ? var_19; }
        chan_30 ! (var_19) @ (var_18) @
        (var_17) @ (var_16); } break;
case chan_29 ? var_33:
        par { chan_40 ! 0; chan_41 ! 0; chan_42 ! 0;
        chan_43 ! 0; } break;
}
for(;;) prialt {
case chan_35 ? var_32:
        chan_39 ! commands_79; break;
case chan_43 ? var_31:
        {} break;
}
for(;;) prialt {
case chan_34 ? var_30:
        chan_38 ! step_81; break;
case chan_42 ? var_29:
        {} break;
}
for(;;) prialt {
case chan_33 ? var_28:
        { chan_46 ! 0; chan_45 ? x_83;
        zero_82_in ! 0; zero_82_out ? var_24;
        chan_37 ! var_24; } break;
case chan_41 ? var_27:
        par { chan_44 ! 0; zero_82_reset ! 0; }
        break;
}
for(;;) prialt {
case chan_46 ? var_26:
        chan_45 ! 0; break;
case chan_44 ? var_25:
        {} break;
}
for(;;) prialt {
case zero_82_in ? var_23:
        { zero_82_out ! 0; } break;
case zero_82_reset ? var_22:
        {} break;
}
for(;;) prialt {
case chan_32 ? var_21:
        chan_36 ! count_80; break;
case chan_40 ? var_20:
        {} break;
}
for(;;) prialt {
case chan_26 ? var_15:
        chan_25 ! TMP_1_78; break;
case chan_24 ? var_14:
        {} break;
}
for(;;) prialt {
case chan_21 ? var_12:
        chan_20 ! commands_73; break;
case chan_19 ? var_11:
        {} break;
}
for(;;) prialt {
case chan_13 ? var_9:
        { chan_16 ! 0; chan_15 ? x_85;
        isnonzero_84_in ! 0;
        isnonzero_84_out ? var_5;
        chan_12 ! var_5; } break;
case chan_11 ? var_8:
        par { chan_14 ! 0;
        isnonzero_84_reset ! 0; } break;
}
for(;;) prialt {
case chan_16 ? var_7:
        chan_15 ! remaining_70; break;
case chan_14 ? var_6:
        {} break;
}
for(;;) prialt {
case isnonzero_84_in ? var_4:
        { isnonzero_84_out ! x_85 != 0; } break;
case isnonzero_84_reset ? var_3:
        {} break;
        }
        for(;;) prialt {
        case chan_8 ? var_2:
                chan_7 ! TMP_2_69; break;
        case chan_6 ? var_1:
                {} break;
        }
}
}
}
```

The test harness resets the function and then calls it with a unit value (as there are no scalar parameters), before reading a unit value out (as there are no scalar return values). The test harness then attempts to read as many items from the output stream as possible.

The test harness supplies `0x1020` (a step size of `0x10` for `0x20` cycles) and `0x2020` (a step size of `0x20` for `0x20` cycles) as two items on the input stream. After that, the input stream becomes unproductive.

When synthesised using Celoxica's tool, the following resource-usage information is produced:

	Gates	Inverters	Latches	Others
Compiled	1	0	340	1629
Optimised	1	0	286	1271
Expanded	2208	452	976	155
Optimised	1313	305	805	155

When simulated, the following output is produced:

```
  95: Output from channel 'Sack_9' = 0
 126: Output from channel 'Sack_9' = 16
 157: Output from channel 'Sack_9' = 32
 188: Output from channel 'Sack_9' = 48
 219: Output from channel 'Sack_9' = 64
 250: Output from channel 'Sack_9' = 80
 281: Output from channel 'Sack_9' = 96
 312: Output from channel 'Sack_9' = 112
 343: Output from channel 'Sack_9' = 128
 374: Output from channel 'Sack_9' = 144
 405: Output from channel 'Sack_9' = 160
 436: Output from channel 'Sack_9' = 176
 467: Output from channel 'Sack_9' = 192
 498: Output from channel 'Sack_9' = 208
 529: Output from channel 'Sack_9' = 224
 560: Output from channel 'Sack_9' = 240
 591: Output from channel 'Sack_9' = 0
 622: Output from channel 'Sack_9' = 16
 653: Output from channel 'Sack_9' = 32
 684: Output from channel 'Sack_9' = 48
 715: Output from channel 'Sack_9' = 64
 746: Output from channel 'Sack_9' = 80
 777: Output from channel 'Sack_9' = 96
 808: Output from channel 'Sack_9' = 112
 839: Output from channel 'Sack_9' = 128
 870: Output from channel 'Sack_9' = 144
 901: Output from channel 'Sack_9' = 160
 932: Output from channel 'Sack_9' = 176
 963: Output from channel 'Sack_9' = 192
 994: Output from channel 'Sack_9' = 208
1025: Output from channel 'Sack_9' = 224
1091: Output from channel 'Sack_9' = 240
1122: Output from channel 'Sack_9' = 0
1153: Output from channel 'Sack_9' = 32
1184: Output from channel 'Sack_9' = 64
1215: Output from channel 'Sack_9' = 96
1246: Output from channel 'Sack_9' = 128
1277: Output from channel 'Sack_9' = 160
1308: Output from channel 'Sack_9' = 192
1339: Output from channel 'Sack_9' = 224
1370: Output from channel 'Sack_9' = 0
1401: Output from channel 'Sack_9' = 32
1432: Output from channel 'Sack_9' = 64
1463: Output from channel 'Sack_9' = 96
1494: Output from channel 'Sack_9' = 128
1525: Output from channel 'Sack_9' = 160
1556: Output from channel 'Sack_9' = 192
1587: Output from channel 'Sack_9' = 224
1618: Output from channel 'Sack_9' = 0
1649: Output from channel 'Sack_9' = 32
1680: Output from channel 'Sack_9' = 64
1711: Output from channel 'Sack_9' = 96
1742: Output from channel 'Sack_9' = 128
1773: Output from channel 'Sack_9' = 160
1804: Output from channel 'Sack_9' = 192
1835: Output from channel 'Sack_9' = 224
1866: Output from channel 'Sack_9' = 0
1897: Output from channel 'Sack_9' = 32
1928: Output from channel 'Sack_9' = 64
1959: Output from channel 'Sack_9' = 96
1990: Output from channel 'Sack_9' = 128
2021: Output from channel 'Sack_9' = 160
2052: Output from channel 'Sack_9' = 192
```

The stream values produced are as expected. The large number of intermediate channels shows itself not only in the size of the Handel-C output, but also in terms of its performance. Each value takes about 30 cycles to produce! This is to be unsurprising with such naïve synthesis. Note also that the 64th item is never produced. For that item to be returned, not only does the head expression of the CONS have to complete, but also the tail expression. The tail expression blocks waiting for the third item from the input stream, and so the value is not produced.

B.3 Graph Synthesis

The same program, when synthesised to Verilog, under graph synthesis, produces the following:

```
// Standard synchronisation block.
'include sync.v

// Synthesised module.
module synth();

reg clk; reg req_in; wire ack_in; wire req_out;
wire ack_out; reg reset;

reg str_act_10; reg str_res_10; wor str_rst_10;
reg str_act_12; reg str_res_12; wor str_rst_12;
reg [15:0] str_dat_10; reg [7:0] str_dat_12;

wire [7:0] w1_1_0; wire [7:0] w1_3_0; wire [7:0] w1_5_0;
wire [23:0] w1_6_0; wire [23:0] w2_g_0; wire [23:0] w3_g_0;
wire [7:0] w4_0_0; wire [7:0] w4_0_1; wire [7:0] w4_0_2;
wire [15:0] w4_1_0; wire [7:0] w4_2_0; wire [7:0] w4_2_1;
wire [0:0] w4_3_0; wire [0:0] w5_g_0; wire [7:0] w5_g_1;
wire [7:0] w5_g_2; wire [7:0] w5_g_3; wire [15:0] w6_0_0;
wire [7:0] w6_1_0; wire [7:0] w6_1_1; wire [7:0] w7_g_0;
wire [7:0] w7_g_1; wire [7:0] w7_g_2; wire [7:0] w7_g_3;
wire [7:0] w9_g_0; wire [7:0] w9_g_1; wire [7:0] w9_g_2;
wire [7:0] w9_g_3; wire [15:0] w10_0_0; wire [7:0] w10_1_0;
wire [7:0] w10_1_1; wire [7:0] w10_2_0; wire [15:0] w10_3_0;
wire [7:0] w10_4_0; wire [23:0] w10_5_0; wire [0:0] w14_g_0;
wire [7:0] w14_g_1; wire [7:0] w14_g_2; wire [7:0] w14_g_3;
wire [7:0] w16_1_0; wire [7:0] w16_1_1; wire [7:0] w17_3_0;
wire [23:0] w18_4_0; reg [23:0] w4_g_0; reg [7:0] w6_g_1;
reg [7:0] w10_g_1; reg [7:0] w10_g_2; reg [7:0] w10_g_3;
reg [7:0] w12_1_0; reg [7:0] w12_g_2; reg [7:0] w12_g_3;
reg [7:0] w12_1_1; reg [15:0] w15_0_0; reg [15:0] w16_0_0;
reg [7:0] w18_1_0; reg [7:0] w18_3_0; reg [7:0] w18_1_1;
reg [0:0] w20_3_0; reg [7:0] w20_0_1; reg [7:0] w20_0_2;
reg [7:0] w20_2_1; reg [23:0] w22_g_0; reg [23:0] w24_6_0;

wire req_0; wire latch_1, data_1, req_1, ack_1; wire req_2;
wire req_3; wire latch_4, data_4, req_4, ack_4; wire req_5;
wire latch_6, data_6, req_6, ack_6; wire req_7; wire ack_8;
wire req_9; wire latch_10, data_10, req_10, ack_10;
wire ack_11;
wire latch_12, ready_12, data_12, req_12, ack_12;
wire ack_13; wire req_14;
wire latch_15, ready_15, data_15, req_15, ack_15;
wire latch_16, data_16, req_16, ack_16;
wire latch_17, data_17, req_17, ack_17;
wire latch_18, data_18, req_18, ack_18;
wire ack_19;
wire latch_20, ready_20, data_20, req_20, ack_20;
wire ack_21;
wire latch_22, ready_22, data_22, req_22, ack_22;
wire ack_23;
wire latch_24, ready_24, data_24, req_24, ack_24;
wire ack_25;

reg wait_10; reg trigger_7; reg wait_12; reg old_req_7;
reg wait_15; reg wait_18; reg trigger_5; reg trigger_14;
reg old_req_12; reg old_req_18; reg trigger_3;
reg trigger_2;

always @(posedge clk) begin
        if (str_rst_10) str_act_10 <= 0;
        if (str_rst_10) str_res_10 <= 0;
        if (str_rst_12) str_act_12 <= 0;
        if (str_rst_12) str_res_12 <= 0;
end

sync s0(.clk(clk), .reset(reset),
        .inreqhi(req_0), .inreqlo(req_0), .inack(ack_1),
        .outreq(req_1), .outackhi(ack_24),
        .outacklo(ack_24),
        .newdata(data_1), .ready(data_1), .latch(latch_1));
sync s1(.clk(clk), .reset(reset_24),
        .inreqhi(req_3), .inreqlo(req_3), .inack(ack_4),
        .outreq(req_4), .outackhi(ack_20),
        .outacklo(ack_20),
        .newdata(data_4), .ready(data_4), .latch(latch_4));
sync s2(.clk(clk), .reset(reset_24),
        .inreqhi(req_5), .inreqlo(req_5), .inack(ack_6),
        .outreq(req_6), .outackhi(ack_12),
        .outacklo(ack_12),
        .newdata(data_6), .ready(data_6), .latch(latch_6));
sync s3(.clk(clk), .reset(reset_24),
        .inreqhi(req_9), .inreqlo(req_9), .inack(ack_10),
        .outreq(req_10), .outackhi(ack_11),
        .outacklo(ack_11), .newdata(data_10),
        .ready(data_10), .latch(latch_10));
sync s4(.clk(clk), .reset(reset_24),
        .inreqhi(req_5 & req_6), .inreqlo(req_5 | req_6),
        .inack(ack_12), .outreq(req_12), .outackhi(ack_13),
        .outacklo(ack_13), .newdata(data_12),
        .ready(ready_12), .latch(latch_12));
sync s5(.clk(clk), .reset(reset_24),
        .inreqhi(req_14), .inreqlo(req_14), .inack(ack_15),
        .outreq(req_15), .outackhi(ack_16 & ack_18),
        .outacklo(ack_16 | ack_18), .newdata(data_15),
        .ready(ready_15), .latch(latch_15));
sync s6(.clk(clk), .reset(reset_24),
        .inreqhi(req_15), .inreqlo(req_15), .inack(ack_16),
        .outreq(req_16), .outackhi(ack_18),
        .outacklo(ack_18), .newdata(data_16),
        .ready(data_16), .latch(latch_16));
sync s7(.clk(clk), .reset(reset_24),
        .inreqhi(req_14), .inreqlo(req_14), .inack(ack_17),
        .outreq(req_17), .outackhi(ack_18),
        .outacklo(ack_18), .newdata(data_17),
        .ready(data_17), .latch(latch_17));
sync s8(.clk(clk), .reset(reset_24),
        .inreqhi(req_15 & req_16 & req_17),
        .inreqlo(req_15 | req_16 | req_17), .inack(ack_18),
        .outreq(req_18), .outackhi(ack_19),
        .outacklo(ack_19), .newdata(data_18),
        .ready(data_18), .latch(latch_18));
sync s9(.clk(clk), .reset(reset_24),
        .inreqhi(req_4), .inreqlo(req_4), .inack(ack_20),
        .outreq(req_20), .outackhi(ack_21),
        .outacklo(ack_21), .newdata(data_20),
        .ready(ready_20), .latch(latch_20));
sync s10(.clk(clk), .reset(reset_24),
        .inreqhi(req_2), .inreqlo(req_2), .inack(ack_22),
        .outreq(req_22), .outackhi(ack_23),
        .outacklo(ack_23), .newdata(data_22),
        .ready(ready_22), .latch(latch_22));
sync s11(.clk(clk), .reset(reset),
        .inreqhi(req_1), .inreqlo(req_1), .inack(ack_24),
        .outreq(req_24), .outackhi(ack_25),
        .outacklo(ack_25), .newdata(data_24),
        .ready(ready_24), .latch(latch_24));

always @(posedge clk) begin
        if (latch_4) w4_g_0 <= w3_g_0;
        if (latch_6) w6_g_1 <= w5_g_1;
        if (latch_10) w10_g_1 <= w9_g_1;
        if (latch_10) w10_g_2 <= w9_g_2;
        if (latch_10) w10_g_3 <= w9_g_3;
        if (latch_12) w12_1_0 <= w6_1_0;
        if (latch_12) w12_g_2 <= w5_g_2;
        if (latch_12) w12_g_3 <= w5_g_3;
        if (latch_12) w12_1_1 <= w6_1_1;
        if (latch_16) w16_0_0 <= w15_0_0;
        if (latch_18) w18_1_0 <= w16_1_0;
        if (latch_18) w18_3_0 <= w17_3_0;
        if (latch_18) w18_1_1 <= w16_1_1;
        if (latch_20) w20_3_0 <= w4_3_0;
        if (latch_20) w20_0_1 <= w4_0_1;
        if (latch_20) w20_0_2 <= w4_0_2;
        if (latch_20) w20_2_1 <= w4_2_1;
        if (latch_22) w22_g_0 <= w2_g_0;
        if (latch_24) w24_6_0 <= w1_6_0;
end

always @(posedge clk) begin
        if (reset_24) begin
                wait_10 <= 0;
                trigger_7 <= 0;
                wait_12 <= 0;
                old_req_7 <= 0;
                wait_15 <= 0;
                wait_18 <= 0;
                trigger_5 <= 0;
                trigger_14 <= 0;
                old_req_12 <= 0;
                old_req_18 <= 0;
                trigger_3 <= 0;
        end else begin
                wait_10 <= data_10 &
                        ˜((req_3 & ack_23) |
                        !(req_3 | ack_23));
                if (data_10 | (wait_10 &
                        ((req_3 & ack_23) |
                        !(req_3 | ack_23)))) wait_10 <= 0;
                if (data_10 |
                        (wait_10 & ((req_3 & ack_23) |
                        !(req_3 | ack_23))))
                        trigger_3 <= ˜trigger_3;
                if (data_10 |
                        (wait_10 & ((req_3 & ack_23) |
                        !(req_3 | ack_23))))
                        w22_g_0 <= w10_5_0;
                if (data_12 & !str_act_12)
                        trigger_7 <= ˜trigger_7;
                if (data_12 & !str_act_12)
                        str_act_12 <= 1;
                if (data_12 & str_act_12)
                        wait_12 <= 1;
                if (wait_12 & !str_act_12)
                        trigger_7 <= ˜trigger_7;
                if (wait_12 & !str_act_12)
                        str_act_12 <= 1;
                if (wait_12 & !str_act_12)
                        wait_12 <= 0;
                old_req_7 <= req_7;
                if (req_7 != old_req_7)
                        str_dat_12 <= w7_g_0;
                if (req_7 != old_req_7)
                        str_res_12 <= 1;
                if (data_15 & ˜str_res_10)
                        wait_15 <= 1;
                if (str_res_10)
                        wait_15 <= 0;
                if ((data_15 | wait_15) & str_res_10)
                        str_act_10 <= 0;
                if ((data_15 | wait_15) & str_res_10)
                        str_res_10 <= 0;
                if ((data_15 | wait_15) & str_res_10)
                        w15_0_0 <= str_dat_10;
                wait_18 <= data_18 & ˜((req_3 & ack_23) |
                        !(req_3 | ack_23));
                if (data_18 | (wait_18 & ((req_3 & ack_23) |
                        !(req_3 | ack_23)))) wait_18 <= 0;
                if (data_18 | (wait_18 & ((req_3 & ack_23) |
                        !(req_3 | ack_23))))
                        trigger_3 <= ˜trigger_3;
                if (data_18 | (wait_18 & ((req_3 & ack_23) |
                        !(req_3 | ack_23))))
                        w22_g_0 <= w18_4_0;
                if (data_20 & (w20_3_0[0:0] == 1))
                        trigger_5 <= ˜trigger_5;
                if (data_20 & (w20_3_0[0:0] == 0))
                        trigger_14 <= ˜trigger_14;
                old_req_12 <= req_12;
                old_req_18 <= req_18;
                if (data_22) trigger_3 <= ˜trigger_3;
                if (data_24) trigger_2 <= ˜trigger_2;
        end
        if (reset) begin
                trigger_2 <= 0;
        end else begin
        end
end

assign w1_1_0 = 7’d0;
assign w1_3_0 = 7’d0;
assign w1_5_0 = 7’d0;
assign w1_6_0 = w1_5_0, w1_3_0, w1_1_0;
assign w2_g_0 = w24_6_0;
assign w3_g_0 = w22_g_0;
assign w4_0_0 = w4_g_0[7:0];
assign w4_0_1 = w4_g_0[15:8];
assign w4_0_2 = w4_g_0[23:16];
assign w4_1_0 = w4_0_0, w4_0_0;
assign w4_2_0 = w4_1_0[7:0];
assign w4_2_1 = w4_1_0[15:8];
assign w4_3_0 = w4_2_0 != 0;
assign w5_g_0 = w20_3_0;
assign w5_g_1 = w20_0_1;
assign w5_g_2 = w20_0_2;
assign w5_g_3 = w20_2_1;
assign w14_g_0 = w20_3_0;
assign w14_g_1 = w20_0_1;
assign w14_g_2 = w20_0_2;
assign w14_g_3 = w20_2_1;
assign w6_0_0 = w6_g_1, w6_g_1;
assign w6_1_0 = w6_0_0[7:0];
assign w6_1_1 = w6_0_0[15:8];
assign w7_g_0 = w12_1_0;
assign w9_g_0 = w12_1_0;
assign w7_g_1 = w12_g_2;
assign w9_g_1 = w12_g_2;
assign w7_g_2 = w12_g_3;
assign w9_g_2 = w12_g_3;
assign w7_g_3 = w12_1_1;
assign w9_g_3 = w12_1_1;
assign w10_0_0 = w10_g_1, w10_g_1;
assign w10_1_0 = w10_0_0[7:0];
assign w10_1_1 = w10_0_0[15:8];
assign w10_2_0 = w10_g_2 - 1;
assign w10_3_0 = w10_1_0, w10_g_3;
assign w10_4_0 = w10_3_0[7:0] + w10_3_0[15:8];
assign w10_5_0 = w10_1_1, w10_4_0, w10_2_0;
assign w16_1_0 = w16_0_0[7:0];
assign w16_1_1 = w16_0_0[15:8];
assign w17_3_0 = 7'd0;
assign w18_4_0 = w18_1_1, w18_3_0, w18_1_0;
assign req_0 = req_in;
assign ack_in = ack_1;
assign req_out = req_24;
assign ack_25 = ack_out;
assign req_7 = trigger_7;
assign req_9 = trigger_7;
assign ready_12 = data_12;
assign ack_8 = req_7;
assign ack_11 = req_10;
assign ready_15 = (data_15 | wait_15) & str_res_10;
assign req_5 = trigger_5;
assign req_14 = trigger_14;
assign ack_13 = old_req_12;
assign ack_19 = old_req_18;
assign ready_20 = (old_req_18 != req_18) |
        (old_req_12 != req_12);
assign ready_22 = data_22;
assign req_3 = trigger_3;
assign ack_21 = req_20;
assign reset_24 = reset | str_rst_12;
assign ready_24 = data_24;
assign req_2 = trigger_2;
assign ack_23 = req_22;
assign str_rst_10 = str_rst_12 /* Wired OR */;

// Test harness:

initial #10000 $finish;

// Clock cycle has length 10.
initial clk = 0;
always #5 clk = ~clk;

// Initially reset main line and returned stream.
initial begin reset = 1; #10 reset = 0; end
assign str_rst_12 = reset;

// Send a request after reset.
initial begin req_in = 0; #10 req_in = 1; end

// Immediately acknowledge stream result.
assign ack_out = req_out;

// Read output stream.
always @(posedge clk) begin
        if (str_res_12) begin
                $display("Time: %4d Value: %3d",
                        $time/10, str_dat_12);
                str_res_12 <= 0;
                str_act_12 <= 0;
        end

end

initial begin
        // Wait for reset
        #50
        // Send first stream item.
        str_res_10 <= 1; str_dat_10 <= 16'h1020;
        // Wait until first item has been read.
        #3000
        // Send second stream item.
        str_res_10 <= 1; str_dat_10 <= 16'h2020;
end

endmodule
```

The synthesis tool implements streams using basic lenient evaluation (see Section 5.2.3) with lazy tail matching (see Section 5.2.5). Signalling between nodes is performed using two-phase signalling, to simplify synchronisation, while the streams, having a single reader and writer at any time, are implemented as level sensitive wires. The details of this scheduling are discussed in Appendix A. The node implementations rely on simple mutual exclusion in conditional and iteration nodes, which should be sufficient for small programs, and require less overhead than a more complex solution.

Nodes are scheduled by placing basic operations into a single cycle, based on an ASAP schedule. The produced code is rather smaller than the Handel-C code, when the fact that the output is at a much lower level is taken into account.

Synthesis to a 0.18 micron process, optimising for speed, gives a design with 667 cells (each cell consisting of at most a few logic gates), of which 291 are latches. When simulated, the following results are produced:

```
Time:  26 Value:   0
Time:  36 Value:  16
Time:  46 Value:  32
Time:  56 Value:  48
Time:  66 Value:  64
Time:  76 Value:  80
Time:  86 Value:  96
Time:  96 Value: 112
Time: 106 Value: 128
Time: 116 Value: 144
Time: 126 Value: 160
Time: 136 Value: 176
Time: 146 Value: 192
Time: 156 Value: 208
Time: 166 Value: 224
Time: 176 Value: 240
Time: 186 Value:   0
Time: 196 Value:  16
Time: 206 Value:  32
Time: 216 Value:  48
Time: 226 Value:  64
Time: 236 Value:  80
Time: 246 Value:  96
Time: 256 Value: 112
Time: 266 Value: 128
Time: 276 Value: 144
Time: 286 Value: 160
Time: 296 Value: 176
Time: 306 Value: 192
Time: 316 Value: 208
Time: 326 Value: 224
Time: 336 Value: 240
Time: 356 Value:   0
Time: 366 Value:  32
Time: 376 Value:  64
Time: 386 Value:  96
Time: 396 Value: 128
```

```
Time: 406 Value: 160
Time: 416 Value: 192
Time: 426 Value: 224
Time: 436 Value:   0
Time: 446 Value:  32
Time: 456 Value:  64
Time: 466 Value:  96
Time: 476 Value: 128
Time: 486 Value: 160
Time: 496 Value: 192
Time: 506 Value: 224
Time: 516 Value:   0
Time: 526 Value:  32
Time: 536 Value:  64
Time: 546 Value:  96
Time: 556 Value: 128
Time: 566 Value: 160
Time: 576 Value: 192
Time: 586 Value: 224
Time: 596 Value:   0
Time: 606 Value:  32
Time: 616 Value:  64
Time: 626 Value:  96
Time: 636 Value: 128
Time: 646 Value: 160
Time: 656 Value: 192
Time: 666 Value: 224
```

New results are produced every ten cycles, a three-fold improvement over CSP synthesis. Moreover, the synthesis performed relied on a simplistic synchronisation scheme which inserts many unnecessary latches in order to simplify the synthesis process. For an optimising compiler much better performance figures would be expected. The graph synthesis not only provides better results than CSP output under basic compilation, but also gives many more opportunities for effective low-level optimisation (as well as the high-level optimisations discussed in the thesis).

Note that, compared to the CSP implementation, an extra value is produced, as expected, since the lazy tail evaluation does not wait for the tail expression blocking on the input stream. This is a benefit as it prevents internal buffering from holding items back unnecessarily, and allows as many results as possible to be produced at the earliest opportunity.

B.4 Performance

Table B.1 shows the results of a number of simulation and synthesis runs. The *signal* function was synthesised to both CSP (or rather, Handel-C) and Verilog, using the synthesis techniques described in Chapters 3 and 4 respectively. In order to test scalability, a series of *map* functions (described below) were also synthesised.

The table gives the number of cycles to execute (both for the first item, and as the number of cycles between items) and the hardware resources required, separated into a logic unit count and register count. Unfortunately, different synthesis systems provide different logic primitives, with the Synopsis tool counting library cells, and Quartus counting FPGA elements (generally look-up tables, or LUTs). The differences between tools make comparisons difficult, so only high-level metrics have been provided, and the logic unit count should be treated as approximate.

B.4.1 Tools

Handel-C version 2.1 and Synopsis vcs 7.0.1 were the simulators used to generate the cycle counts. The low-level synthesis tools used were Handel-C version 2.1, which targets Xilinx FPGAs, Synopsis release dc-2003.12-sp1, targeting a 6 metal layer generic logic 0.18 micron process, and Quartus II version 4.2, targeting the Altera Stratix FPGA family.

B.4.2 The signal program

The signal program was synthesised using both the CSP and graph synthesis paths. Even taking into account the possible differences in what counts as a logic unit under the different synthesis tools, it can be seen that the graph synthesis is vastly superior to naïve CSP synthesis, as might be expected. The difference between the Synopsys and Quartus logic counts can be put down to the primitives used. The differences in register count are rather smaller.

An extra copy of the *signal* program was synthesised, with the static-scheduling optimisation of Section 5.1 disabled. The table shows that there is a large overhead in scheduling, and minimising this is a vital operation if good synthesis is to be achieved, since simple scheduling can halve both the resources required and the execution cycle count.

B.4.3 The map programs

In order to demonstrate how the area and time requirements of the produced hardware scales, the very simple functions shown in Figure B.2 were synthesised, producing the results seen at the bottom of Table B.1.

The synthesis tool does not attempt to merge the maps together (that is, it does not take advantage of the fact that $(map\ f)\ \ (map\ g) = map(f\ \ g)$), so the composition of the functions represents a realistic chaining of SASL elements. By simplistically scal-

Program	**High-level Synthesis**	**Initialisation Cycles**	**Per Item Cycles**	**Low-level Synthesis**	**Logic Count**	**Register Count**
signal	CSP	95	31	Handel-C	1773	805
signal	Graph	26	10	Synopsys	376	291
signal	Graph	26	10	Quartus	92	230
signal	Graph (unscheduled)	46	28	Quartus	211	422
map-1	Graph	13	7	Quartus	34	53
map-2	Graph	19	7	Quartus	67	105
map-3	Graph	25	7	Quartus	102	158

Table B.1: Table comparing performance figures for different synthesis techniques and options

(* The basic map function. *)
fun *map-dec*$($*str*$) =$ **case** *str* **of** x**::**xs *decr*(x)**::***map-dec*(xs)

(* One-, two- and three-stage maps. *)
fun$($*map-1*$)\ x =$ *map-dec*(x)
fun$($*map-2*$)\ x =$ *map-dec*$($*map-dec*$(x))$
fun$($*map-3*$)\ x =$ *map-dec*$($*map-dec*$($*map-dec*$(x)))$

Figure B.2: The *map* programs

ing up a simple mapping function with extra stages, it can be seen that the pipeline length affects the set-up latency, but once the pipeline is full the inter-item time is the same in all cases. The resource utilisation scales up linearly with the pipeline length.

APPENDIX C

Extending the Identification of Reorderable Streams

This appendix extends the analysis of Section 7.4. Where the basic analysis cannot identify whether a newly-generated stream is bag-like, or the streams used to generate it are, the analysis of this section can do so, for common functions such as *map* and *fold*.

C.1 The Type System

The type system presented here is an extension of that given in Section 7.4. In order to track stream items through basic values, the values are now marked with the set of parameter values upon which they depend. To deal with loop dependencies, basic values are marked as being constant or variable over loops. The new value types are defined as follows:

$$\begin{array}{lcl} \sigma & := & (\sigma \quad \ldots \quad \sigma) \quad B\,{}^{X}_{I} \quad S^{R}_{I} \\ X & := & C \quad V \end{array}$$

B stands for a basic type value, and S for a stream type value. Stream type values are marked with a boolean R, which is true if the stream is reorderable, and false otherwise, as before.

Each stream and basic type is now also annotated with a set I, representing the set of parameter values on which the value depends. Each basic and stream type in the parameter is associated with a new parameter tag i. For example, if a function has a parameter of type $Bool \quad Int$ stream, the parameter type for this analysis could be $B^{C}_{\{1\}} \quad S\,{}^{R}_{\{2\}}$. A basic value that depends on both parameters would have type $B^{V}_{\{1,2\}}$.

A basic type value may be marked as either constant (C) or variable (V). A basic type parameter that is a constant value is one that is kept constant over recursive calls. This is similar to the stability constraint on streams, but instead of requiring that recursive calls feed back the same stream in recursive calls, constant parameters must have the same value fed back.

A non-parameter value marked with C is identical to the constant basic type parameter it depends upon. Its I set will contain only the tag associated with that parameter. In recursive calls the formal and actual argument types should be identical for constant parameters. The analysis of constant values is conservative, so that parameters marked as constants are definitely so, but some constant parameters may be marked as variable unnecessarily. If a function contains no recursive calls, all basic type parameter values are marked as being constant.

The typing rules from Section 7.4 must be extended to collect dependence information. The new rules are shown in Figure C.1. The rest of this section explains the details of these typing rules.

Typing Non-Recursive Function Calls The (APPLY) rule depends on the RETURN function and the CONST predicate. The RETURN function generates the return type of the function, given the formal typing and actual arguments, while the CONST function ensures the constant stability constraint is met.

The constant stability constraint requires that in recursive calls constant type values in the formal parameters should match the actual parameters. To test a recursive function call with formal parameters σ_F and actual parameters σ_A, we calculate $\text{CONST}(\sigma_F, \sigma_A)$. CONST is defined as follows:

$$\begin{array}{rcl} \text{CONST}((\sigma_1^1 \quad \dots \quad \sigma\ {}_k^1), (\sigma_1^2 \quad \dots \quad \sigma\ {}_k^2)) & = & \text{CONST}(\sigma_1^1, \sigma_1^2) \quad \dots \quad \text{CONST}(\sigma\ {}_k^1, \sigma_k^2) \\ \text{CONST}(B_I^C, \sigma) & = & (B_I^C = \sigma) \\ \text{CONST}(B_I^V, \sigma) & = & T \\ \text{CONST}(S_I^R, \sigma) & = & T \end{array}$$

The function RETURN is then used to calculate the appropriate return type of a function, given the type of the function and the type of the actual arguments. The generation of the I and X values are covered here, as the generation of the R was covered in Chapter 7.

The formal parameter type is matched up with the actual parameter type, and mappings are generated from the is in the formal parameters to the corresponding elements in the the actual parameters, using the rules shown in Figure C.2. The function *Dep* returns the type associated with i, and *IDeps* returns the I associated with that type. *AllIDeps*(I) generates all the I' from the actual parameters associated with a set of I from the called function.

These functions are used by the pseudocode for RETURN, which is shown in Figure C.3. This function omits the R values for the streams, as their calculation was shown earlier, in Chapter 7. For constant basic values, the type is preserved directly from the actual parameter type, while for other types the dependence set I is generated from the union of all the dependence sets from the actual arguments it depends upon.

Typing recursive calls The function RETURN, described above, is used for non-recursive calls, but recursive calls are typed by finding a fixed point. The I are found by initially making the unsafe approximation that the dependency sets for the values returned from recursive function calls are empty. The function's generated return type is then used as the next approximation of the return type for the recursive calls. This will converge as the set of dependencies will only increase, and is bounded by the case where a value depends on all parameters. This process is similar to that used for finding the sets of constraints on the R.

Other Typing Rules The (CONSTR-ELIM) rule must merge several different types, when joining together the paths from a conditional join. Not only should the returned value depend on all the values which the various conditionally-generated values depend upon, but it should also depend upon the expression's condition. To achieve this, the rule relies on the conditional merge function $(\sigma_1 + \sigma_2)_I$. This function creates a type that is of the same structure as σ_1 and σ_2, and combines the dependencies of the two. Each stream and (non-constant) basic type in the returned type will also depend on those identifiers in I. The function $(\sigma_1 + \sigma_2)_I$ is defined as follows:

$$\begin{array}{rcl} ((\sigma_1^1 \quad \dots \quad \sigma\ {}_k^1) + (\sigma_1^2 \quad \dots \quad \sigma\ {}_k^2))_I & = & ((\sigma_1^1 + \sigma_1^2)_I \quad \dots \quad (\sigma\ {}_k^1 + \sigma_k^2)_I) \\ (B_{I'}^C + B_{I'}^C)_I & = & B_{I'}^C \\ (B_{I'}^C + B_{I''}^C)_I & = & B_{I \cup I' \cup I''}^V \text{ (if } I' \quad = I''\text{)} \\ (B_{I'}^{X'} + B_{I''}^{X''})_I & = & B_{I \cup I' \cup I''}^V \text{ (if } X' = V \quad X'' = V\text{)} \end{array}$$

$$(\textsc{apply})\frac{A \quad e \;:\; \sigma_3 \qquad\qquad f \;:\; \sigma_1 \quad \sigma_{\ 2}}{A \quad f\,e \;:\; \textsc{return}(\sigma_1,\sigma_2,\sigma_3)\ \ \textsc{const}(\sigma_1,\sigma_3) \text{ if call is recursive}}$$

$$(\textsc{constr-intro})\frac{A \quad e_1 \;:\; B^{X_1}_{I_1} \qquad\qquad A \quad e_k \;:\; B^{X_k}_{I_k}}{A \quad c(e_1,\ldots,e_k) \;:\; B^{V}_{I_1\cup\ldots\cup I_k}}$$

$$(\textsc{tuple-intro})\frac{A \quad e_1 \;:\; \sigma_1 \qquad\qquad A \quad e_k \;:\; \sigma_k}{A \quad (e_1,\ldots,e_k) \;:\; \sigma_1 \quad \ldots \quad \sigma_{\ k}}$$

$$(\textsc{cons-intro})\frac{A \quad e_1 \;:\; B^{X}_{I} \quad A \quad e_2 \;:\; S^{R}_{I'}}{A \quad e_1 :: e_2 \;:\; S^{R'}_{I\cup I'}} R' \quad R$$

$$(\textsc{constr-elim})\frac{A_0 \quad e \;:\; B^{X}_{I} \quad \begin{cases} A_1, x^1_1 : B^X_I,\ldots,x^1_{k_1} : B^X_I \quad e_1 \;:\; \sigma_1 \\ \ldots \\ A_n, x^n_1 : B^X_I,\ldots,x^n_{k_n} : B^X_I \quad e_n \;:\; \sigma_n \end{cases}}{\begin{array}{l} A \quad \textbf{case}\ e\ \textbf{of}\ c_1(x^1_1,\ldots,x^1_{k_1}) \quad e_{\ 1} \\ \qquad\qquad \ldots \\ \qquad\qquad c_n(x^n_1,\ldots,x^n_{k_n}) \quad e_{\ n} \;:\; (\sigma_1+\ldots+\sigma_n)_I \end{array}} A = A_0 \quad \ldots \quad A_{\ n}$$

$$(\textsc{tuple-elim})\frac{A \quad e_1 \;:\; \sigma_1 \quad \ldots \quad \sigma_{\ k} \quad A, x_1 : \sigma_1,\ldots,x_k : \sigma_k \quad e_2 \;:\; \sigma}{A \quad \textbf{case}\ e_1\ \textbf{of}\ (x_1,\ldots,x_k) \quad e_{\ 2} \;:\; \sigma}$$

$$(\textsc{cons-elim})\frac{A \quad e_1 \;:\; S^{R}_{I} \quad A, x_1 : B^V_I, x_2 : S^{R'}_I \quad e_2 \;:\; \sigma}{A \quad \textbf{case}\ e_1\ \textbf{of}\ x_1 :: x_2 \quad e_{\ 2} \;:\; \sigma} R \quad R\ '$$

$$(\textsc{let})\frac{A \quad e_1 \;:\; \sigma_2 \quad A, x : \sigma_2 \quad e_2 \;:\; \sigma_1}{A \quad \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \;:\; \sigma_1}$$

$$(\textsc{var})\frac{}{A, x : \sigma \quad x \;:\; \sigma}$$

$$(\textsc{shuffle})\frac{A \quad e \;:\; S^{T}_{I}}{A \quad \textsc{shuffle}(e) \;:\; S^{T}_{I}}$$

Figure C.1: Rules for identifying reorderable streams

$$\begin{array}{rl}
\textsc{gen}((\sigma_1^1 \quad \ldots \quad \sigma\ {}_k^1),(\sigma_1^2 \quad \ldots \quad \sigma\ {}_k^2)) & \quad \textsc{gen}(\sigma\ {}_1^1,\sigma_1^2),\ldots,\textsc{gen}(\sigma_k^1,\sigma_k^2) \\
\textsc{gen}(B_{\{i\}}^X,\sigma) & \quad \mathit{Dep}(i)=\sigma \\
\textsc{gen}(S_{\{i\}},\sigma) & \quad \mathit{Dep}(i)=\sigma \\
 & \\
\mathit{Dep}(i)=B_{\{I\}}^X & \quad \mathit{IDeps}(i)=I \\
\mathit{Dep}(i)=S_{\{I\}} & \quad \mathit{IDeps}(i)=I \\
 & \\
\mathit{AllIDeps}(I) & = \bigcup_{i\in I}\mathit{IDeps}(i)
\end{array}$$

Figure C.2: Definitions for *Dep*, *IDeps* and *SetIDeps*

```
let RETURN(formal-from, formal-to, actual-from) =
    GEN(formal-from, actual-from);
    let rec RETURN′($\sigma_1 \quad \ldots \quad \sigma\ {}_k$) = RETURN′($\sigma_1$)   ...   RETURN ′($\sigma_k$)
            RETURN′($B_{\{i\}}^C) = \mathit{Dep}(i)$
            RETURN′($B_I^V) = B_{AllIDeps(I)}^V$
            RETURN′($S_I) = S_{AllIDeps(I)}$
    in RETURN′(formal-to)
```

Figure C.3: Definition for the function RETURN

```
**fun** *loop*() = *loop*()
**fun** *compare*($s, t$) =
    **case** $s$ **of** $x$::$xs$   **case** $t$ **of** $y$::$ys$
        **let** $z$ = **if** $x = y$ **then** *True* **else** *loop*() **in** *True* :: *compare*(xs, ys)
```

Figure C.4: A possibly non-terminating function

$$(S^{R}_{I'} + S^{R'}_{I''})_I \quad = \quad S^{R \wedge R'}_{I \cup I' \cup I''}$$

C.2 Stream-Generating Functions

The analysis of this section is devoted to the generation of new streams, which the work of Section 7.4 could not deal with. We use the term *stream-generating functions* to describe those functions that return a single stream, with *every* call to the function ending in a recursive call. Every leaf expression in a tail position of the syntax tree must be a recursive call. A stream-generating function may take in a number of streams as parameters that are used in generating the new stream. Common examples of stream-generating functions are *map*, *filter*, *zip* and the constant stream function.

A function is *stateless* if none of the parameters are of type B^V_I. This name is used because each recursive call to the function then does not use any state from previous iterations, beyond passing in the unread parts of the streams. This allows each iteration to be considered independently, so that streams may be treated as bags if each element is processed independently and reorderably.

Given a stateless stream-generating function, we can attempt to infer streams in two directions:

Forward analysis: If the returned stream has at most one element generated per iteration, and each generated item does not depend on any ordered streams, the resulting stream is bag-like, as each item is generated independently, and any output ordering could be achieved by reordering the parameter streams.

Backward analysis: Conversely, if a parameter stream is read from at the rate of at most one item per iteration, and that value is not combined with values from any other streams (which could cause an ordering dependence), and the result is used to create a bag-like stream, that stream is bag-like, too. Reordering the elements of the parameter stream would only reorder the elements of the returned bag.

The rest of this chapter is devoted to formalising these analyses.

Changing Termination Properties Note that, as in Section 7.2, we work with the actual dependencies generated, ignoring dependencies which affect termination but not the actual values (expressions that affect termination but not the result are an effect of eager evaluation). An example where the conversion from a list to a bag does not affect the results, but may change termination is given in Figure C.4. Such programs are viewed as a programmer error.

These *hidden dependencies* do not affect the forwards analysis, and the backwards analysis section covers the case where the hidden dependencies are ignored, with a paragraph dealing with how to extend the algorithm to identify hidden dependencies.

C.3 Forwards Analysis

For the forwards analysis, we must know the maximum number of times the stream returned by the tail call is CONS'd onto over any dynamic path through the function. For this, we use the CONS-counting function CC, as shown in Figure C.5. This function returns the maximum number of items CONS'd

$$
\begin{array}{rcl}
\textsc{cc}(f\,e) & = & \begin{cases} 0 & : \text{ Recursive call} \\ & : \text{ Non-recursive call} \end{cases} \\
\textsc{cc}(c(e_1,\ldots,e_k)) & = & \\
\textsc{cc}((e_1,\ldots,e_k)) & = & \\
\textsc{cc}(e_1\mathbf{::}e_2) & = & \textsc{cc}(e_2)+1 \\
\textsc{cc}(\mathbf{case}\ e\ \mathbf{of}\ c_1(x_1^1,\ldots,x_{k_1}^1) \quad e\ \ _1 & & \\
\ldots & = & \max(\textsc{cc}(e_1),\ldots,\textsc{cc}(e_n)) \\
c_n(x_1^n,\ldots,x_{k_n}^n) \quad e\ \ _n) & & \\
\textsc{cc}(\mathbf{case}\ e_1\ \mathbf{of}\ (x_1,\ldots,x_k) \quad e\ \ _2) & = & \textsc{cc}(e_2) \\
\textsc{cc}(\mathbf{case}\ e_1\ \mathbf{of}\ x_1\ \mathbf{::}\ x_2 \quad e\ \ _2) & = & \textsc{cc}(e_2) \\
\textsc{cc}(\mathbf{let}\ x=e_1\ \mathbf{in}\ e_2) & = & \textsc{cc}(e_2) \\
\textsc{cc}(x) & = & \\
\textsc{cc}(\textsc{shuffle}(e)) & = & \textsc{cc}(e)
\end{array}
$$

Figure C.5: Rules for CONS-counting expressions

onto the returned value in an expression. It returns for any expression where one of the tail position subexpressions does not contain a recursive call. For any function **fun** $f\ x = E$, $\textsc{cc}(E) \quad =$ if and only if the function is a stream-generating function.

If the function has type $\sigma \quad S\ {}_I^R$, and $\textsc{cc}(E) \quad 1$, then the stream is generated at the rate of at most one element per iteration. For the returned stream to be a bag, we just require that it only depends on bag-like streams and constant basic types. In other words, if a function **fun** $f\ x = E$ has type $\sigma \quad S\ {}_I^R$ and $\textsc{cc}(E) \quad 1$, then we can add the constraint $\bigwedge_{i\in I} R_i \quad R$, where $R\ \ _i$ is defined as follows:

$$
R_i = \begin{cases} T & : \ B_{\{i\}}^C \text{ is in the parameter type } \sigma \\ F & : \ B_{\{i\}}^V \text{ is in the parameter type } \sigma \\ R' & : \ S_{\{i\}}^{R'} \text{ is in the parameter type } \sigma \end{cases}
$$

Note that for the returned stream to be a bag, f must be a stateless function as far as the generation of the stream goes (the function may have non-constant parameters, but they must not be used in the generation of the stream).

Examples The example functions used both in this section and the next are shown in Figure C.6. Forwards analysis applies to these functions as follows:

The function *map*$_f$ has the type $S_{\{1\}}^R \quad S\ {}_{\{1\}}^{R'}$. We cannot infer this function produces a bag when given a bag using the basic rules. However, the CONS-counting function returns 1, so we can add the constraint $R \quad R\ '$, as required. We cannot infer that $R' \quad R$—for that, we need to use the backwards analysis of the next section.

The *filter*$_p$ function works similarly, as it has a CONS-count of 1, and the returned stream depends only on the parameter stream.

The *zip* function also works with forwards analysis. The function has a CONS-count of 1, so given the type $S_{\{1\}}^R \quad S\ {}_{\{2\}}^{R'} \quad S\ {}_{\{1,2\}}^{R''}$, the function has the constraint $R \quad R' \quad R\ ''$.

(* Perform a function on each element of a stream. *)
fun *map*$_f$ $s =$ **case** s **of** x**::**xs $\quad f(x)$ **::** *map* $_f(xs)$

(* Remove the elements which fail a test from a stream. *)
fun *filter*$_p$ $s =$ **case** s **of** x**::**xs $\quad$ **if** $p(x)$ **then** x**::***filter* $_p(xs)$ **else** *filter*$_p(xs)$

(* Merge two streams by creating pairs of elements. *)
fun *zip*$(s, t) =$ **case** s **of** x**::**xs $\quad$ **case** t **of** y**::**ys $\quad (s, t)$**::***zip*(xs, ys)

(* Create a stream where each item in the parameter stream is repeated. *)
fun *dup*$(s) =$ **case** s **of** x**::**xs $\quad x$**::**x**::***dup*(xs)

Figure C.6: Example functions

Finally, the function *dup* has a CONS-count of 2, so we cannot use the forwards analysis to infer if the returned stream is bag-like. This is as we might expect, since the returned stream may well not be a bag even if the parameter stream is, as the function introduces some correlation between consecutive elements.

C.4 Backwards Analysis

The backwards analysis identifies streams that are used in a bag-like way. This means that the stream is read in a stateless stream-generating function, with at most one element read per iteration. The read items must not be merged with items from other stream reads, or used to control the reading of other streams, as this may make it necessary to read the items from the two streams in the same order, so that each stream cannot be reordered independently. If more than one item is read from a stream per iteration, the items may be treated differently, so the input stream cannot be inferred to be a bag.

For example, the *map*$_f$ function from the previous section allows us to infer the parameter stream is a bag if the returned stream is, since the read item is not merged with any other read items, while we cannot create a similar constraint on the *zip* function, since the items on the output stream are created by merging together items read from different input streams.

To count the number of reads performed on a stream, a read-counting function RC is used, as shown in Figure C.7. The function counts the number of reads associated with a stream identifier (as introduced in Section 2.4.1). Stream identifiers from the original typing system are used, rather than the dependence set identifiers, as we only wish to count reads from the original stream, and not from new streams that somehow depend upon it. We ignore sub-expressions that cannot return a stream which will pass the stability constraint, since these expressions cannot contains reads from streams which will be passed recursively. If a parameter stream is passed to (and returned from) another function the EXTCOUNT function is used, which in this analysis conservatively marks the stream as possibly having been read an unbounded amount. For a stream to be considered for backwards analysis, its read count must be less than or equal to one.

For stream-generating functions, we now add the rule (CONS-ELIM-2), shown in Figure C.8. This rule allows the constraint $U_i \quad R'' \quad R$ to be added if the stream read (on a stream of type $S\ ^{R}_{\{i\}}$) occurs in a stateless stream-generating function that returns a stream of type $S^{R''}_{I'}$, and only one read is performed on that stream per iteration. U_i represents the requirement that the data read from the stream (which will therefore have i in its dependencies) be used in a bag-like manner. This is generated by giving the U_i a default value of true, and constraining it to false if an item CONS'd onto the returned stream somehow depends on both i and any other non-constant value.

$$
\begin{array}{rcl}
\textsc{rc}(\alpha, f\ e) & = & \left\{ \begin{array}{rcl} \textsc{rc}(\alpha, e) & : & \text{Recursive call} \\ \textsc{extcount}(\alpha, e) & : & \text{Non-recursive call} \end{array} \right. \\
\textsc{rc}(\alpha, c(e_1, \ldots, e_k)) & = & 0 \\
\textsc{rc}(\alpha, (e_1, \ldots, e_k)) & = & \textsc{rc}(\alpha, e_1) + \ldots + \textsc{rc}(\alpha, e_k) \\
\textsc{rc}(\alpha, e_1 \mathbf{::} e_2) & = & 0 \\
\textsc{rc}(\alpha, \textbf{case}\ e\ \textbf{of}\ c_1(x_1^1, \ldots, x_{k_1}^1) \quad e\ _1 & & \\
\ldots & = & \max(\textsc{rc}(\alpha, e_1), \ldots, \textsc{rc}(\alpha, e_n))) \\
c_n(x_1^n, \ldots, x_{k_n}^n) \quad e\ _n) & & \\
\textsc{rc}(\alpha, \textbf{case}\ e_1\ \textbf{of}\ (x_1, \ldots, x_k) \quad e\ _2) & = & \textsc{rc}(\alpha, e_1) + (\alpha, e_2) \\
\textsc{rc}(\alpha, \textbf{case}\ e_1\ \textbf{of}\ x_1 \mathbf{::} x_2 \quad e\ _2) & = & \textsc{rc}(\alpha, e_1) + \textsc{rc}(\alpha, e_2) + \textsc{isread}(\alpha, \sigma)\ \text{where}\ e_1 : \sigma \\
\textsc{rc}(\alpha, \textbf{let}\ x = e_1\ \textbf{in}\ e_2) & = & \textsc{rc}(\alpha, e_1) + \textsc{rc}(\alpha, e_2) \\
\textsc{rc}(\alpha, x) & = & 0 \\
\textsc{rc}(\alpha, \textsc{shuffle}(e)) & = & \textsc{rc}(\alpha, e) \\
 & & \\
\textsc{isread}(\alpha, \tau\ \textit{stream}_\beta) & = & \left\{ \begin{array}{rcl} 1 & : & \alpha = \beta \\ 0 & : & \alpha \quad = \beta \end{array} \right. \\
 & & \\
\textsc{extcount}(\alpha, e) & = & \left\{ \begin{array}{rcl} & : & \alpha\ \text{is contained in}\ \sigma\ \text{where}\ e : \sigma \\ 0 & : & \text{otherwise} \end{array} \right.
\end{array}
$$

Figure C.7: Rules for read-counting expressions

$$
(\textsc{cons-elim-2})\dfrac{A \quad e_1 \ :\ S^R_{\{i\}} \quad A, x_1 : B^V_{\{i\}}, x_2 : S^{R'}_{\{i\}} \quad e_2 \ :\ \sigma}{A \quad \textbf{case}\ e_1\ \textbf{of}\ x_1 \mathbf{::} x_2 \quad e\ _2 \ :\ \sigma}
\begin{array}{l} \textsc{rc}(\alpha, E) \quad 1 \\ R \quad R\ ' \\ U_i \quad R'' \quad R \end{array}
$$

where:

α is the stream identifier associated with the parameter annotated $S^R_{\{i\}}$,

the rule is applied in a stateless stream-generating function with body expression E,

the function returns a stream of type $S^{R''}_{I'}$.

Figure C.8: The (CONS-ELIM-2) rule

To find the dependencies between items, it is not sufficient to look at the Is of items CONS'd onto the returning stream, as this loses dependencies caused by control-flow. For example, if an item is CONS'd onto the stream conditionally, depending on the value of another stream item, there will be a dependence. Instead, we generate *dependence sets*, which are sets of values which are all used together when generating stream items. An i is treated in a bag-like way if any dependence set it is in only contains it and constant parameters.

The set of dependence sets is generated by the typing-like "$\triangleright$" rules of Figure C.9. Each dependency set represents either:

All the parameter values depended upon by an item that is CONS'd onto the returned stream. The dependencies include dependencies on conditionals that enclose the CONS expression.

For streams that are conditionally read, a set is generated containing a dependence on that stream, and dependencies on the enclosing conditionals.

The (CONSTR-ELIM) rule is the most complex. It uses the READS function, which generates the set of parameter streams that are read in a particular expression. READS is defined as follows:

$$\textsc{reads}(e) = \quad i \quad \textsc{rc}(\alpha_i, e) \quad 1$$

where α_i is the stream identifier associated with the parameter annotated $S^R_{\{i\}}$. The rule then distributes the conditional dependencies I over all dependency sets in D, using the notation $(D)_I$, defined as:

$$(D)_I = \quad d \quad I \quad d \quad D$$

If any dependency set contains more than one parameter stream, it means the streams involved cannot be inferred to be bag-like. This can be expressed by a formula to generate the U_i constraints:

$$d \quad D.\ i \quad d.(\ j \quad d.i \ = j \quad \textsc{nonconst}(j) \qquad U \quad {}_i)$$

where NONCONST(i) is true if $V^C_{\{i\}}$ isn't in the parameter type. This rules ensures that if a dependence set contains two or more streams, those streams cannot be inferred to be bag-like.

Identifying Hidden Dependencies Hidden dependencies occur when a stream's productivity depends on two streams being synchronised, even though no elements of the returned stream depend on both of these streams (as mentioned in Section C.2). To identify all the hidden dependencies, in order to constrain the associated U_i to false, the dependence sets of values that are bound to variables which are not used must be generated. This must be done not only in the current function, but also in all functions called from it. This extension should not be too complex, but makes the exposition of the analysis somewhat more difficult.

Examples The examples from Figure C.6 are used again:

The *map*$_f$ function is a stateless stream-generating function (as are all the functions in that figure), and the parameter s has a read-count of 1, so it is a candidate for backwards analysis. If the function is given the type $S^R_{\{1\}} \quad S^{\ R'}_{\ \{1\}}$, the set of dependence sets generated by the function is simply $\quad 1 \quad$, so U_1 is true, and $R' \quad R$.

The *filter*$_p$ function adds a conditional expression, but the condition depends solely on the item read from the parameter stream, so it too produces the constraint that if the returned stream is a bag, the parameter stream will be too.

The *zip* function (of type $S^{R}_{\{1\}} \quad S^{R'}_{\{2\}} \quad S^{R''}_{\{1,2\}}$) produces the dependence sets $\quad 1, 2 \quad$, and so both U_1 and U_2 are constrained to false. We cannot infer the parameter streams are bags given that the returned stream is a bag.

Finally, the *dup* function (of type $S^{R}_{\{1\}} \quad S^{R'}_{\{1\}}$) contains two CONS expressions, but each depends on a single read from the parameter stream, so that the generated set of dependence sets is just $\quad 1 \quad$, and we can produce the constraint $R' \quad R$.

Analysing Within Algebraic Datatypes As with the analysis of Section 7.2, the accuracy of the analysis can be increased by looking inside algebraic datatypes, for example by encoding them as tuples.

Summary The analysis of this appendix extends that provided in Section 7.4 to provide a set of constraints that should identify bag-like streams in a wide range of common functions, allowing a number of optimisation that would otherwise not be possible.

$$(\textsc{apply})\frac{}{f\ e \rhd}$$

$$(\textsc{cons-intro})\frac{e_1\ :\ B_I^X \quad e_2 \rhd D}{e_1 \mathbin{::} e_2 \rhd D \quad I}$$

$$(\textsc{constr-elim})\frac{e\ :\ B_I^X \quad e_i \rhd D_i \quad d_i = \textsc{reads}(e_i)}{\begin{array}{l}\textbf{case}\ e\ \textbf{of}\ c_1(x_1^1, \ldots, x_{k_1}^1) \quad e\ {}_1 \\ \quad \ldots \\ \quad c_n(x_1^n, \ldots, x_{k_n}^n) \quad e\ {}_n \rhd (D_1 \quad \ldots \quad D\ {}_n \quad d\ {}_1 \quad \ldots \quad d\ {}_n)_I\end{array}}$$

$$(\textsc{tuple-elim})\frac{e_2 \rhd D}{\textbf{case}\ e_1\ \textbf{of}\ (x_1, \ldots, x_k) \quad e\ {}_2 \rhd D}$$

$$(\textsc{cons-elim})\frac{e_2 \rhd D}{\textbf{case}\ e_1\ \textbf{of}\ x_1 \mathbin{::} x_2 \quad e\ {}_2 \rhd D}$$

$$(\textsc{let})\frac{e_2 \rhd D}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \rhd D}$$

$$(\textsc{shuffle})\frac{e \rhd D}{\textsc{shuffle}(e) \rhd D}$$

Figure C.9: Rules for generating dependence sets

Bibliography

[1] ABDALLAH, A. E. Derivation of parallel algorithms from functional specifications to CSP processes. In *Proceedings of the International Conference on the Mathematics of Program Construction MPC* (1995), vol. 947 of *LNCS*, pp. 67–96.

[2] ABDALLAH, A. E., AND HAWKINS, J. Formal behavioural synthesis of Handel-C parallel hardware implementations from functional specifications. In *Proceedings of the 36th Hawaii International Conference on System Sciences* (2003).

[3] ACHTEN, P., AND PLASMEIJER, M. J. The ins and outs of Clean I/O. *Journal of Functional Programming 5*, 1 (1995), 81–110.

[4] ALTERA. *Nios 3.0 CPU Data Sheet*. Altera Corporation, 2003.

[5] ALUR, R., AND HENZINGER, T. A. Finitary fairness. *ACM Transactions on Programming Languages and Systems 20*, 6 (1998), 1171–1194.

[6] AMAGBEGNON, P., BESNARD, L., AND GUERNIC, P. L. Implementation of the data-flow synchronous language SIGNAL. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (1995), pp. 163–173.

[7] AMERICAN NATIONAL STANDARDS INSTITUTE. *ANSI Fortran X3.9–1978*, 1978. Approved April 3, 1978 (also known as Fortran 77).

[8] ASHCROFT, E. A., AND WADGE, W. W. Lucid, a nonprocedural language with iteration. *Communications of the ACM 20*, 7 (July 1977), 519–526.

[9] ASHLEY, J. M., AND DYBVIG, R. K. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems 20*, 4 (July 1998), 845–868.

[10] BABB, J., RINARD, M., MORITZ, C. A., LEE, W., FRANK, M., BARUA, R., AND AMARASINGHE, S. Parallelizing applications into silicon. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* (1999).

[11] BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. Compiler transformations for high-performance computing. *ACM Computing Surveys 26*, 4 (1994), 345–420.

[12] BAUMGARTE, V., MAY, F., NÜCKEL, A., VORBACH, M., AND WEINHARDT, M. PACT XPP - a self-reconfigurable data processing architecture. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)* (2001).

[13] BELL, J., AND HOOK, J. Defunctionalization of typed programs. Tech. Rep. CSE-94-024, Oregon Graduate Institute, 1994.

[14] BELL, J. M., BELLEGARDE, F., AND HOOK, J. Type-driven defunctionalization. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)* (1997), vol. 32, pp. 25–37.

[15] BELLEGARDE, F. Notes for pipelines of transformations for ML, 1995.

[16] BERRY, G. The foundations of Esterel. In *Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[17] BERRY, G., AND GONTHIER, G. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming 19*, 2 (1992), 87–152.

[18] BJESSE, P., CLAESSEN, K., SHEERAN, M., AND SINGH, S. Lava: Hardware design in Haskell. In *Proceedings of the International Conference on Functional Programming* (1998), pp. 174–184.

[19] BLUNNO, I., CORTADELLA, J., KONDRATYEV, A., LAVAGNO, L., LWIN, K., AND SOTIRIOU, C. Handshake protocols for de-synchronization. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and System* (2004), IEEE Computer Society Press, pp. 149–158.

[20] BOHLMANN, K., LOOGEN, R., AND ORTEGA-MALLÉN, Y. Concurrent functional processes. In *Proceedings of the 5th International Workshop on the Implementation of Functional Languages* (1993).

[21] BOLOTSKI, M., DEHON, A., AND KNIGHT, JR., T. F. Unifying FPGAs and SIMD arrays. In *Proceedings of the 2nd International ACM/SIGDA Workshop on FPGAs* (1994), pp. 1–10.

[22] BREBNER, G. A virtual hardware operating system for the Xilinx XC6200. In *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications* (1996), vol. 1142 of *LNCS*, pp. 327–336.

[23] BREBNER, G. The swappable logic unit: A paradigm for virtual hardware. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* (1997), pp. 77–86.

[24] BREBNER, G. Circlets: Circuits as applets. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* (1998), pp. 300–301.

[25] BREBNER, G. Field-programmable logic: Catalyst for new computing paradigms. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications* (1998), vol. 1482 of *LNCS*, pp. 49–58.

[26] BUCK, J., AND LEE, E. A. The token flow model. In *Advanced Topics in Dataflow Computing and Multithreading*. IEEE, 1993, pp. 267–290.

[27] BUDIU, M., AND GOLDSTEIN, S. C. Fast compilation for pipelined reconfigurable fabrics. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (1999), pp. 195–205.

[28] BUDIU, M., AND GOLDSTEIN, S. C. Pegasus: An efficient intermediate representation. Tech. Rep. CMU-CS-02-107, Carnegie Mellon University, 2002.

[29] BURNS, J., DONLIN, A., HOGG, J., SINGH, S., AND DE WIT, M. A dynamic reconfiguration run-time system. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* (1997), pp. 66–75.

[30] BURSTALL, R. M., MACQUEEN, D. B., AND SANNELLA, D. T. Hope: An Experimental Applicative Language. In *Conference Record of the 1980 LISP Conference* (1980), ACM Press, pp. 136–143.

[31] BURTON, F. W. Nondeterminism with referential transparency in functional programming languages. *Computer Journal 31*, 3 (1988), 243–247.

[32] BUSCEMI, M. G., AND SASSONE, V. High-level petri nets as type theories in the join calculus. *Lecture Notes in Computer Science 2030* (2001).

[33] CADAMBI, S., AND GOLDSTEIN, S. C. Fast and efficient place and route for pipeline reconfigurable architectures. In *Proceedings of the International Conference on Computer Design* (2000).

[34] CADAMBI, S., WEENER, J., GOLDSTEIN, S. C., SCHMIT, H., AND THOMAS, D. E. Managing pipeline-reconfigurable FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (1998), pp. 55–64.

[35] CALLAHAN, T., CHONG, P., DEHON, A., AND WAWRZYNEK, J. Fast module mapping and placement for datapaths in FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (1998), pp. 123–132.

[36] CALLAHAN, T. J., AND WAWRZYNEK, J. Instruction-level parallelism for reconfigurable computing. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications* (1998), vol. 1482 of *LNCS*, pp. 248–257.

[37] CASPI, P., AND POUZET, M. A functional extension to Lustre. In *Proceedings of the 8th International Symposium on Languages for Intensional Programming (ISLIP)* (1995).

[38] CEJTIN, H., JAGANNATHAN, S., AND WEEKS, S. Flow-directed closure conversion for typed languages. In *Proceedings of the European Symposium on Programming* (2000), pp. 56–71.

[39] CELOXICA. Handel-C language reference manual. Available as part of Celoxica's University Program.

[40] CHIN, W.-N., AND DARLINGTON, J. A higher-order removal method. *Lisp and Symbolic Computation 9*, 4 (December 1996), 287–322.

[41] CHIN, W.-N., AND KHOO, S.-C. Calculating sized types. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation* (2000), pp. 62–72.

[42] CHURCH, A. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[43] COOK, B., LAUNCHBURY, J., AND MATTHEWS, J. Specifying superscalar microprocessors in Hawk. In *Proceedings of the Workshop on Formal Techniques for Hardware* (1998).

[44] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of program by construction of approximate fixpoints. In *Conference record of the 4th ACM Symposium on Principles of Programming Languages* (1977).

[45] DAHL, O.-J., DIJKSTRA, E. W., AND HOARE, C. A. R. *Structured Programming*. Academic Press, 1972.

[46] DEHON, A. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines* (1994), pp. 31–39.

[47] DEHON, A. DPGA utilization and application. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (1996), pp. 115–121.

[48] DEHON, A., AND WAWRZYNEK, J. Reconfigurable computing: what, why, and implications for design automation. In *Proceedings of the 36th ACM/IEEE design automation conference* (1999), pp. 610–615.

[49] DONLIN, A. Self modifying circuitry - a platform for tractable virtual circuitry. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications* (1998), vol. 1482 of *LNCS*, pp. 200–208.

[50] FERGUSON, A. B., AND WADLER, P. When will deforestation stop? In *Proceedings of the Glasgow Workshop on Functional Programming* (1988), pp. 39–56.

[51] FRANKAU, S., AND MYCROFT, A. Stream processing hardware from functional language specifications. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS)* (2003).

[52] FRANKAU, S., MYCROFT, A., AND MOORE, S. Statically-allocated languages for hardware stream processing (extended abstract). In *Proceedings of the UK ACM SIGDA Workshop on Electronic Design Automation* (2002).

[53] FRIEDMAN, D. P., AND WISE, D. S. Cons should not evaluate its arguments. In *Proceedings of the International Colloquium on Automata, Languages and Programming* (1976), Edinburgh University Press, pp. 257–284.

[54] GENIN, D., HILFINGER, P., RABAEY, J., SCHEERS, C., AND MAN, H. D. DSP specification using the Silage language. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing* (1990), vol. 2, pp. 1056–1060.

[55] GILL, A., LAUNCHBURY, J., AND JONES, S. L. P. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (1993), pp. 223–232.

[56] GOLDBERG, B., AND PARK, Y. G. Higher order escape analysis. In *Proceedings of the 3rd European Symposium on Programming* (1990), vol. 432 of *LNCS*, pp. 152–160.

[57] GOLDSTEIN, S. C., SCHMIT, H., MOE, M., BUDIU, M., CADAMBI, S., TAYLOR, R. R., AND LAUFER, R. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture* (1999).

[58] GOMARD, C., AND SESTOFT, P. Globalization and live variables. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (1991), vol. 26 of *SIGPLAN Notices*, ACM Press, pp. 166–177.

[59] GUO, S. R., AND LUK, W. Compiling Ruby into FPGAs. In *Proceedings of the International Workshop on Field-Programmable Logic and Applications* (1995), pp. 188–197.

[60] HALBWACHS, N., CASPI, P., RAYMOND, P., AND PILAUD, D. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE 79*, 9 (September 1991), 1305–1320.

[61] HAREL, D., AND PNUELI, A. On the development of reactive systems. In *Logics and models of concurrent systems*. Springer-Verlag New York, Inc., 1985, pp. 477–498.

[62] HARTENSTEIN, R. W., KRESS, R., AND REINIG, H. A new FPGA architecture for word-oriented datapaths. In *Proceedings of the 4th International Workshop on Field-Programmable Logic and Applications* (1994), vol. 849 of *LNCS*, pp. 144–155.

[63] HAYNES, S. D., CHEUNG, P. Y., LUK, W., AND STONE, J. SONIC - a plug-in architecture for video processing. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* (1999), pp. 280–281.

[64] HEINTZE, N., AND MCALLESTER, D. A. Linear-time subtransitive control flow analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (1997), pp. 261–272.

[65] HENGLEIN, F. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems 15*, 2 (April 1993), 253–289.

[66] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, second ed. Morgan Kaufmann, 1996.

[67] HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM 21*, 8 (August 1978), 666–677.

[68] HOFMANN, M. A type system for bounded space and functional in-place update—extended abstract. In *Proceedings of the European Symposium on Programming* (2000), pp. 165–179.

[69] HOLYER, I., AND SPILIOPOULOU, E. Concurrent monadic interfacing. In *Proceedings of the International Workshop on the Implementation of Functional Languages* (1998), vol. 1595 of *LNCS*, pp. 73–89.

[70] HUGHES, J., PARETO, L., AND SABRY, A. Proving the correctness of reactive systems using sized types. In *Proceedings of the Symposium on Principles of Programming Languages* (1996), pp. 410–423.

[71] HUTCHINGS, B. L., AND WIRTHLIN, M. J. Implementation approaches for reconfigurable logic applications. In *Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications* (1995), vol. 975 of *LNCS*, pp. 419–428.

[72] JOHNSON, N., AND MYCROFT, A. Combined register allocation and code motion using the value state dependence graph. In *Proceedings of Compiler Construction* (2003), vol. 2622 of *Lecture Notes in Computer Science*, pp. 1–16.

[73] JOHNSON, S., AND BOSE, B. DDD: A system for mechanized digital design derivation. In *Proceedings of the ACM/SIGDA Workshop on Formal Methods in VLSI Design* (1991). (Unfortunately, the proceedings of the workshop have not been officially published).

[74] JOHNSON, S. C. Code generation for silicon. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1983), pp. 14–19.

[75] JOHNSON, S. D. Daisy, DSI, and LiMP. Tech. Rep. 288, Indiana University Computer Science Department, 1989.

[76] JOHNSSON, T. Lambda lifting: transforming programs to recursive equations. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (1985).

[77] JONES, N. D. The expressive power of higher-order types or, life without CONS. *Journal of Functional Programming 11*, 1 (January 2001), 55–94.

[78] JONES, N. D., GOMARD, C. K., AND SESTOFT, P. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[79] JONES, S. B., AND SINCLAIR, A. F. Functional programming and operating systems. *The Computer Journal 32*, 2 (1989), 162–174.

[80] JONES, S. L. P., AND WADLER, P. Imperative functional programming. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1993), pp. 71–84.

[81] JONES, S. P., AND HUGHES, J. Report on the programming language Haskell 98, a non-strict purely functional language. Tech. Rep. YALEU/DCS/RR-1106, Yale University, February 1999.

[82] KELSEY, R., CLINGER, W., AND REES, J. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33*, 9 (1998), 26–76.

[83] KIM, B.-S., CHOI, Y. H., AND KIM, L.-S. IRAM design for multimedia applications. In *Proceedings of the Workshop on Mixing Logic and DRAM: Chips that Compute and Remember at ISCA97* (1997).

[84] KIM, H.-S., SOMANI, A. K., AND TYAGI, A. A reconfigurable multi-function computing cache architecture. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (2000), pp. 85–94.

[85] LI, Y., AND LEESER, M. HML: an innovative hardware description language and its translation to VHDL. In *Proceedings of the Conference on Computer Hardware Description Languages* (1995).

[86] LUK, W., ANDREOU, P., DERBYSHIRE, A., DUPONT-DE-DINECHIN, F., RICE, J., SHIRAZI, N., AND SIGANOS, D. A reconfigurable engine for real-time video processing. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications* (1998), vol. 1482 of *LNCS*, pp. 169–178.

[87] LUK, W., SHIRAZI, N., AND CHEUNG, P. Y. K. Compilation tools for run-time reconfigurable designs. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* (1997), pp. 56–65.

[88] LUK, W., SHIRAZI, N., GUO, S. R., AND CHEUNG, P. Y. K. Pipeline morphing and virtual pipelines. In *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications* (1997), vol. 1304 of *LNCS*, pp. 111–120.

[89] MACHADO, R. J., AND ET AL. An evolutionary approach to the use of petri net based models: From parallel controllers to HW/SW co-design, 2000.

[90] MACQUEEN, D. Models for distributed computing. Tech. Rep. 351, Institut de Recherche d'Informatique et d'Automatique (INRIA), 1979.

[91] MACVICAR, D., AND SINGH, S. Accelerating DTP with reconfigurable computing engines. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications* (1998), vol. 1482 of *LNCS*, pp. 391–395.

[92] MAI, K., PAASKE, T., JAYASENA, N., HO, R., DALLY, W. J., AND HOROWITZ, M. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (2000).

[93] MANNA, Z., AND PNUELI, A. A hierarchy of temporal properties. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (1990), pp. 377–410.

[94] MARINESCU, M.-C. V., AND RINARD, M. High-level automatic pipelining for sequential circuits. In *Proceedings of the International Symposium on Systems Synthesis* (2001), ACM, pp. 215–220.

[95] MARINESCU, M.-C. V., AND RINARD, M. High-level specification and efficient implementation of pipelined circuits. In *Proceedings of the IEEE Design Automation Conference* (2001), pp. 655–661.

[96] MARINESCU, M.-C. V., AND RINARD, M. C. High-level automatic pipelining for sequential circuits. In *Proceedings of the International Symposium on Systems Synthesis (ISSS)* (2001), pp. 215–220.

[97] MARK OSKIN, FREDERIC T. CHONG, T. S. Active pages: A computation model for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (1998), pp. 192–203.

[98] MARLOW, S., AND WADLER, P. Deforestation for higher order functions. In *Proceedings of the Glasgow Workshop on Functional Programming* (1992).

[99] MCCARTHY, J. LISP - notes on its past and future. In *Conference Record of the 1980 LISP Conference, Stanford University* (1980), ACM.

[100] MCKEEVER, S., LUK, W., AND DERBYSHIRE, A. Towards verifying parametrised hardware libraries with relative placement information. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS)* (2003), IEEE Computer Society Press.

[101] MECHA, H., FERNANDEZ, M., TIRADO, F., SEPTIN, J., MOZOS, D., AND OLCOZ, K. A method for area estimation of data-path in high-level synthesis. *IEEE Transactions on CAD of Integrated Circuits and Systems 15*, 2 (1996), 258–265.

[102] MICHELI, G. D. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[103] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML*. The MIT Press, 1991.

[104] MIRSKY, E., AND DEHON, A. MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *IEEE Symposium on FPGAs for Custom Computing Machines* (1996), pp. 157–166.

[105] MOORE, G. E. Cramming more components onto integrated circuits. *Electronics* (April 1965), 114–117.

[106] MOORE, S. W., AND GRAHAM, B. T. Tagged up/down sorter - a hardware priority queue. *The Computer Journal 38*, 9 (1995), 695–703.

[107] MYCROFT, A. Polymorphic type schemes and recursive definitions. In *Proceedings of the International Symposium on Programming* (1984), vol. 167 of *LNCS*, pp. 217–239.

[108] Mycroft, A., and Sharp, R. The FLaSH project: Resource-aware synthesis of declarative specications. In *Proceedings of the International Workshop on Logic Synthesis 2000* (2000).

[109] Mycroft, A., and Sharp, R. A statically allocated parallel functional language. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP)* (2000), vol. 1853 of *LNCS*.

[110] Mycroft, A., and Sharp, R. Hardware synthesis using SAFL and application to processor design. In *Proceedings of 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)* (2001), vol. 2144 of *LNCS*.

[111] Mycroft, A., and Sharp, R. Higher-level techniques for hardware description and synthesis. *International Journal on Software Tools for Technology Transfer (STTT) 4*, 3 (May 2003).

[112] Oberlander, J. Grice for graphics: pragmatic implicature in network diagrams. *Information Design Journal 8*, 6 (1996), 163–179.

[113] Odersky, M. Functional nets. In *Proceedings of the European Symposium on Programming* (2000), vol. 1782 of *LNCS*, pp. 1–25.

[114] Odersky, M. An overview of functional nets. Lecture Notes, APPSEM Summer School, 2000.

[115] O'Donnell, J. Hardware description with recursion equations. In *Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications* (1987), pp. 363–382.

[116] O'Donnell, J. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming* (1992), Workshops in Computing, Springer-Verlag, pp. 178–194.

[117] O'Donnell, J. J. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Proceedings of the First International Symposium on Functional Programming Languages in Education* (1995), pp. 195–214.

[118] Page, I. Parameterised processor generation. In *More FPGAs*, W. Moore and W. Luk, Eds. Abingdon EE&CS Books, Abingdon, England, 1993, pp. 225–237.

[119] Page, I. Construction of hardware-software systems from a single description. *Jounal of VLSI Signal Processing* (Mar. 1996).

[120] Page, I. Reconfigurable processor architectures. *Microprocessors and Microsystems* (1996).

[121] Pareto, L. *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology, Sweden, 2000.

[122] Park, Y. G., and Goldberg, B. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (1992), vol. 27, pp. 116–127.

[123] Plotkin, G. D. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[124] Razdan, R., and Smith, M. D. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture* (1994), IEEE/ACM, pp. 172–80.

[125] REYNOLDS, J. C. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference* (1972), pp. 717–740.

[126] SCHMIT, H. Incremental reconfiguration for pipelined applications. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* (1997), pp. 47–55.

[127] SGS-THOMSON MICROELECTRONICS LIMITED. *occam 2.1 reference manual*. Prentice Hall International (UK) Ltd, 1988.

[128] SHARP, R. *Higher-Level Hardware Synthesis*. PhD thesis, University of Cambridge, November 2002. Available as LNCS vol. 2963.

[129] SHARP, R., AND MYCROFT, A. The FLaSH compiler: Efficient circuits from functional specifications. Tech. Rep. tr.2000.3, AT&T, 2000.

[130] SHARP, R., AND MYCROFT, A. A higher-level language for hardware synthesis. In *Proceedings of 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)* (2001).

[131] SHEERAN, M. Designing regular array architectures using higher order functions. In *Proceedings of International Conference on Functional Programming and Computer Architecture* (1985), vol. 201 of *LNCS*, pp. 220–237.

[132] SHIVERS, O. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation* (1998), pp. 164–174.

[133] SINGH, S. Architectural descriptions for FPGA circuits. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* (1995), pp. 145–154.

[134] SINGH, S., HOGG, J., AND MCAULEY, D. Expressing dynamic reconfiguration by partial evaluation. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* (1996), pp. 188–194.

[135] SINGH, S., AND SLOUS, R. Accelerating Adobe Photoshop using the XC6200 FPGA. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* (1998), pp. 236–244.

[136] STEELE, JR., G. L., AND GABRIEL, R. P. The evolution of Lisp. *ACM SIGPLAN Notices 28*, 3 (1993), 231–270.

[137] STEENSGAARD, B. A polyvariant closure analysis with dynamic widening, 1994.

[138] STOYE, W. Message-based functional operating systems. *Science of Computer Programming 6*, 3 (1986), 291–311.

[139] SWAN, S. An introduction to system level modeling in SystemC 2.0. Downloadable from Cadence Design Systems, Inc. `http://www.cadence.com/whitepapers/systemc_wp20.pdf`, 2001.

[140] TARDITI, D., LEE, P., AND ACHARYA, A. No assembly required: Compiling standard ML to C. *ACM Letters on Programming Languages and Systems 1*, 2 (June 1992), 161–177.

[141] TAU, E., CHEN, D., ESLICK, I., BROWN, J., AND DEHON, A. A first generation DPGA implementation. In *Proceedings of the Third Canadian Workshop of Field-Programmable Devices* (1995), pp. 138–143.

[142] TENSILICA INCORPORATED. CTOV: C to Verilog compiler. `http://www.cl.cam.ac.uk/users/djg/ctovpage/ctovpage.html`.

[143] TOLMACH, A. Combining closure conversion with closure analysis using algebraic types. In *Proceedings of the ACM SIGPLAN Types in Compilation Workshop* (1997).

[144] TRAUB, K. R. Sequential implementation of lenient programming languages. Tech. Rep. MIT-LCS//MIT/LCS/TR-417, MIT, 1988.

[145] TUNA, M. E., JOHNSON, S. D., AND BURGER, R. G. Continuations in hardware-software codesign. In *Proceedings of the IEEE International Conference on Computer Design* (1994), pp. 264–269.

[146] WADLER, P. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (1984), pp. 45–52.

[147] WADLER, P. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the European Symposium on Programming* (1988), vol. 300 of *LNCS*, pp. 344–358.

[148] WADLER, P. Linear types can change the world! In *Proceedings of the Working Conference on Programming Concepts and Methods* (1990), pp. 347–359.

[149] WAINGOLD, E., TAYLOR, M., SRIKRISHNA, D., SARKAR, V., LEE, W., LEE, V., KIM, J., FRANK, M., FINCH, P., BARUA, R., BABB, J., AMARASINGHE, S. P., AND AGARWAL, A. Baring it all to software: RAW machines. *IEEE Computer* (1997), 86–93.

[150] WAZLOWSKI, M., AGARWAL, L., LEE, T. S., SMITH, A., LAM, E., ATHANAS, P., SILVERMAN, H. F., AND GHOSH, S. PRISM-II compiler and architecture. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (1993), pp. 9–16.

[151] WEINHARDT, M. Compilation and pipeline synthesis for reconfigurable architectures. In *Proceedings of the Reconfigurable Architectures Workshop* (1997).

[152] WEINHARDT, M., AND LUK, W. Memory access optimization and RAM inference for pipeline vectorization. In *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications* (1999), vol. 1673 of *LNCS*, pp. 61–70.

[153] WEINHARDT, M., AND LUK, W. Pipeline vectorization for reconfigurable systems. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* (1999), pp. 52–62.

[154] WIRTHLIN, M. J., AND HUTCHINGS, B. L. DISC: the dynamic instruction set computer. In *Proceedings of Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing* (1995), SPIE – The International Society for Optical Engineering, pp. 92–103.

[155] WIRTHLIN, M. J., AND HUTCHINGS, B. L. Improving functional density through run-time constant propagation. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (1997), pp. 86–92.

[156] WIRTHLIN, M. J., HUTCHINGS, B. L., AND GILSON, K. L. The Nano processor: A low resource reconfigurable processor. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines* (1994), pp. 23–30.

[157] XU, M., AND KURDAHI, F. J. Area and timing estimation for lookup table based FPGAs. In *Proceedings of the European Design and Test Conference* (1996).