Stream Processing Hardware from Functional Language Specifications

Simon Frankau and Alan Mycroft
Computer Laboratory, University of Cambridge
William Gates Building, J J Thomson Avenue, Cambridge CB3 0FD, UK
{sgf22,am}@cl.cam.ac.uk

Abstract

We describe work-in-progress which aims at compiling suitably restricted functional programs with lazy lists directly into hardware. This extends previous work on SAFL which demonstrated that functional languages are effective at describing "fire-and-wait-for-result" programs, but for which stream-like I/O was awkward or impossible. Other frameworks (e.g. Lava) have used functional languages as a form of macro-language, so that executing a program generates a net-list (structural) description, while our language is compiled directly ("behaviourally") to hardware. In particular our stream operations (represented as creation and pattern matching of lazy CONS *cells) are mapped into channel writes and reads. We introduce our language, SASL, compare our approach to that of synchronous stream/signal languages, and give example functional descriptions which can be compiled to hardware.*

1. Introduction

Statically-allocated languages have an application in the form of hardware description. A statically-allocated program may be able to store all the data it requires within the circuit, reducing the problem of von Neumann bottlenecks caused by dynamically-allocated structures such as heaps and stacks. Such systems are well-suited to processing streams of data, and are guaranteed not to have runtime errors due to running out of memory.

Modern general-purpose processors devote a large amount of resources to hiding memory latency and coping with complex control-flow. The amount of parallelism extracted is very limited. For some applications, with little in the way of control-flow limitations or complex memory access patterns, reconfigurable arrays such as FPGAs can bring extremely large amounts of parallelism to bear on tasks often associated with general-purpose CPUs (examples include DES cracking [11], gene processing [7] and ray-tracing [22]). Such tasks do not require the same support for memory access and control-flow prediction, and can instead spend the resources on actual computation. Predictable memory-access patterns allow for more efficient access to memory as requests can be pipelined and scheduled statically.

Such systems need to be provided with a hardware description rather than a conventional program, and this may be a major obstacle to the acceptance of such systems. Conventional programming languages that can be reasonably efficiently compiled to hardware could be extremely useful. Functional languages provide a relatively abstract way of describing algorithms that may be better suited to hardware compilation than many alternatives. By specifying the program in a functional style we may sidestep the implicit orderings of imperative programs, allowing more parallelism. Program transformation techniques may be applied to produce optimised hardware from a software-like description. This is the approach used by SAFL-based [15, 16, 20] languages.

Although SAFL+ [20] provides channels for communication, it does not provide support for processing streamed data in a pure functional form. The approach we have taken here is to support streams of data as a new data type, extending the language. In comparison to some other stream-processing languages (such as Lucid [1] and Lustre [6]), our language, SASL, retains recursion for iteration, and has both stream and simple values. Streams are provided via a CONS primitive, so that the language is relatively accessible to programmers used to conventional functional languages.

Streams are distinct from lazy lists in that streams cannot be "rewound". The problem with lazy lists is that a reference to an earlier part of a lazy list can be kept, and an unbounded part of the list can be processed repeatedly. Streams are like *linear* lazy lists—once an item in the list is read, it cannot be read again. This allows the input data to be queued in order, with a known access pattern, and may simplify the internal pipelining of hardware.

The underlying computational power of a statically-allocated stream-processing language like SASL is that of a finite state machine or regular language, since the internal

state is finite and the input cannot be "rewound". Pragmatically, this is not a significant problem, if the state space is large enough, since computers in general use are just state machines with very large amounts of state. As long as the statically-allocated hardware can be scaled to process all the data sets we are interested in, this is not a serious limitation.

An issue more relevant to practical implementations is that of avoiding von Neumann bottlenecks. Random-access to large memories can be expensive and difficult to implement efficiently, leading to much of the complexity of modern out-of-order superscalar processors. This is the real reason for looking at statically-allocated stream-processing languages, as they provide a set of programs which are suitable for efficient hardware implementation, with the data being held close to the hardware that will process it.

The language is aimed to be used for the core implementation of stream-processing algorithms, with pipelining and parallelism limited only by the dataflow of the given program. It could be used to simply describe algorithms that would otherwise be implemented in VHDL or Verilog, where scheduling, signalling and control logic would have to be designed explicitly by the user. Interface circuitry may be generated automatically by suitable tools, or "hand-crafted" in a low-level HDL.

The following section covers related work in this area. Section 3 introduces the core language and the problems raised when trying to statically allocate a stream-based language. Section 4 introduces SASL, with its type-like restrictions for static allocation. Section 5 covers some example programs, and then Section 6 performs some comparisons with other high-level HDLs. Section 7 briefly outlines language synthesis, and Section 8 finishes the paper with conclusions and possible directions for further work.

2. Related work

The work here extends the work carried out on SAFL [15, 16, 20], a statically-allocated functional language used as part of the FLaSH [14] project. The language we use, SASL, is very similar to SAFL extended to process streams. This contrasts with projects such as HML [12] and Lava [3] (which are based on Standard ML [19] and Haskell [10], respectively). These languages embed hardware description languages within a functional setting, rather than compiling an actual functional description. The functional parts of these languages are often used for macro generation of the hardware netlists. Similar approaches are seen in other languages like muFP [21], Ruby [5] and Hawk [4].

Lucid [1] is a language that takes a stream-like approach, intended for use as a formal system. The primitives "`first`" and "`next`" describe streams, and loops are described using streams from which a particular value can be extracted using the "`as soon as`" primitive.

This concept is used by synchronous stream languages intended for hardware synthesis, such as Lustre [2]. In these languages, 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. The streams are defined in terms of functions on time-delayed versions of streams. The signal approach is also used to specify hardware in a functional language in Hydra [17], to create structural netlists. Section 6 contains a comparison between SASL and signal-based languages.

The main difference between the synchronous stream approach and using a functional language with streams is that the functional language can have non-stream variables, and specify recursion in terms of a tail-recursive program, while the synchronous stream approach creates the answer through a recursive stream definition. We define streams in terms of iteration, by generating zero or more elements of a list in one recursive tail call, and the rest in following iterations. In contrast, synchronous stream languages define iteration in terms of streams, with successive iterations of a loop being represented by successive stream items. Thus programs in SASL use a more familiar functional style.

Another approach to take is that of Kahn-MacQueen networks [13], 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 can still have 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 [9] provides a theoretical viewpoint on the expressiveness of languages with restrictions similar to static allocation. Whereas SASL allows the lazy creation of streams, he considers languages without the ability to create unbounded structures. His "read-only tail recursive" functions are statically allocated, but allow non-linear access to the input, so that it can be rewound. As such, the language has the computational power of LOGSPACE, rather than just that of a finite state machine. The extra power appears because these programs do not provide static allocation with a fixed, finite amount of state—the back references into the input list can represent an unbounded number of values.

Due to issues like these, Jones' model is of limited use for our purposes, but it does raise a point of some relevance. In some restricted languages adding higher-order functions

increases the power of the language, as data can be stored in nested closures. There will be some higher-order programs that cannot be converted to a similarly restricted first-order form. For example, closures can be used to convert programs to continuation-passing form, so that non-tail recursion is eliminated, but the program still cannot be statically allocated. Rather than attempt to limit a higher-order language to programs that are statically allocable, we use a first-order language. Streams can be viewed as lazy lists where the closure representing a CONS node is guaranteed to be statically allocable.

Linear typing [24] 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 [23] allows programs with intermediate lists to be converted to a form where the intermediate lists are never fully generated[1]. In software, this can convert some programs to a form that does not need more than a statically-allocated amount of storage. Both forms of program, when written in SASL, can be translated to the same hardware which, at a low level, serially processes stream values in a pipeline.

Lars Pareto's PhD thesis [18] includes "Synchronous Haskell", a language that uses sized types [8] to guarantee that well-typed programs are free from busy loops and deadlocks. However, this language has a complex typing scheme, that provides other information beyond static-allocability, 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 (also known as *hiatons* in Lucid), which are used to replace elements which would have been filtered out, to maintain the original stream size.

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.

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 0 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. Polymorphic functions can be *specialized* to concrete types during synthesis.

$p :=$	$d_1 \ \ldots \ d_n$	program definition
$d :=$	**fun** $f\ x$ **=** e^{tr}	function definition
$e :=$	$f\ e$	function application
$\mid$	$c(e_1, \ldots, e_k)$	constructor
$\mid$	**case** e **of** $m_1 \mid \ldots \mid m_n$	case expression
$\mid$	**let** x **=** e_1 **in** e_2^{tr}	let expression
$\mid$	x	variable access
$m :=$	$c(x_1, \ldots, x_k) \Rightarrow e^{tr}$	match

Figure 1. The language's abstract grammar

An intermediate abstract grammar[2] for the language is shown in Figure 1. 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. Tail calls may only occur in tail contexts.

Only direct tail recursion is allowed, and non-recursive datatypes, although the language could be extended to allow mutual tail-recursion and sized recursive datatypes [8] with upper size limits, without changing the language power. Similarly, lexical scoping could be introduced, with lambda lifting used to reduce the program to a top-level-only form.

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

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. 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, 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.

[1] Deforestation extends listlessness to work on *tree-like* data structures, but the resulting program may require unbounded storage.

[2] Example code may have some extensions beyond this basic grammar in order to aid readability, but may be easily converted to this form.

$e :=$	$e_1 :: e_2^{tr}$	cons expression
$\mid$	**case** e_1 **of** $x_1 :: x_2 \Rightarrow e_2^{tr}$	stream matching

Figure 2. Grammar extensions for stream processing

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

$$\textbf{datatype}\ \alpha\ \texttt{stream} = \\ \texttt{cons}\ \textbf{of}\ \texttt{unit} \rightarrow (\alpha \times (\alpha\ \texttt{stream}))$$

Translation can then be performed at a syntactic level: $e_1 :: e_2$ becomes $\texttt{cons}(\textbf{fn}() \Rightarrow (e_1, e_2))$ and **case** e_1 **of** $x_1 :: x_2$ becomes **case** e_1 **of** $\texttt{cons}(f) \Rightarrow$ **let** $(x_1, x_2) = f()$ **in** e_2 **end**, where f is a new temporary variables, distinct from all others.

We have decided to only implement infinite streams. 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, infinite streams can be simulated by streams that can end, by making sure that streams that end never crop up in practice, and adding never-executed pieces of code to match on end-of-stream cases. As the streams are lazily evaluated, all streams can be assumed to be infinite without losing functionality.

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. The data from a stream must only be read once, and a linear type system can be used to enforce this. To prevent stream "rewinding", it is sufficient to make all streams and types that contain streams linear, and use the type system of [24].

Beyond basic linearity constraints, if care is not taken, data can accumulate within the system as an unbounded amount of information is required to represent a stream. Example code illustrating these problems is shown in Figure 3. The problems include:

- Streams that produce data at different rates are somehow merged, requiring unbounded buffering (case 1).
- A stream may be recursively built up by repeated CONS operations in non-tail contexts, or more 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.

4. A statically-allocated stream language

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 on each recursive call to represent it.

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.

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. Values of a basic type τ have bounded storage requirements that are a function of τ.

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

$$\sigma := \tau \mid \tau\ stream_i \mid \sigma_1 \times \ldots \times \sigma_n$$

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—$\sigma \times \ldots \times \sigma$. Each stream is given an identifier $i \in S\star$ that is used to identify the stream during stability analysis. The identifier is either a symbol from an infinite alphabet S, representing a stream provided as a parameter to the function, or "$\star$" representing a newly created stream. The set $S \cup \{\star\}$ is represented by $S\star$. The tuple type constructor allows the creation of tuples of value types (as opposed to basic types, which can be tupled using constructors). A new tupling operator and its associated **case** expression are also added to the language. The added forms of expression are shown in Figure 4. The function SI is defined to return the set of stream identifiers (including $\star$) used in a typing.

Values of stream type can produce an infinite stream of values, but the amount of state required at any point to represent the rest of the stream is finite[3], and so value types can

[3]Externally provided input streams could provide an oracle supplying data that could not be generated internally, but the only state needed internally for that stream is which input stream should be read from.

```
(* 1. Streams that may require unlimited buffering. *)
fun odds (hd1::hd2::tl) = hd1::odds(tl)

fun merge (hd1::tl1, hd2::tl2) = (hd1, hd2)::merge(tl1,tl2)

fun needs_buffer stream = merge((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 (hd::tl) = f(hd)::map_f(tl)

fun map_iter_f (stream) = let stream' = map_f(stream)
                          in case stream' of hd::tl => hd::map_iter_f(tl)
```

Figure 3. Programs that cause problems for static allocation

$e :=$	$(e_1, \ldots, e_k)$	tupling
$\mid$	**case** e_1 **of** $(x_1, \ldots, x_k) \Rightarrow e_2^{tr}$	untupling

Figure 4. Grammar extensions for handling tuples

be stored in a bounded amount of space.

The top layer extends the type system to cover the types of functions and constructors, by adding arrow types. Functions have the type $\sigma_1 \rightarrow \sigma_2$, while constructors have the type $\tau_1 \ \ldots \ \tau_n \rightarrow \tau \ (n \geq 0)$. These typings only apply to functions and constructors, and do not appear in the types of expressions or the typing environment, which only contains value types.

The rules for typing the language are shown in Figure 5. As the typing environment, A, contains only variables, the types of functions and constructors are treated as side-conditions. The typing of functions is very similar to that used for polymorphic typing. For example, if an expression has the typing

$$x_1 : \tau_1 \ \mathit{stream}_\alpha, x_2 : \tau_2 \ \mathit{stream}_\beta, \vdash e \ : \ \tau_1 \ \mathit{stream}_\alpha \times \tau_2 \ \mathit{stream}_\star$$

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

$$\forall \alpha, \beta \quad \tau_1 \ \mathit{stream}_\alpha \times \tau_2 \ \mathit{stream}_\beta \rightarrow \tau_1 \ \mathit{stream}_\alpha \times \tau_2 \ \mathit{stream}_\star$$

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

In general, the type of a function **fun** $f \ x$=e is $f : \sigma_1 \rightarrow \sigma_2$, where σ_1 and σ_2 are given by the typing rules, using $x : \sigma_1 \vdash e : \sigma_2$. New, unused stream identifiers are created for streams in σ_1; all the stream identifiers in σ_1 must be distinct, and $SI(\sigma_1) \subset S$. As this is the only place where new stream identifiers are introduced, $SI(\sigma_2) \subseteq SI(\sigma_1) \cup \{\star\}$. Due to linearity, each non-$\star$ identifier will 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, and provide the same substitution in the return type[4]. The substitution is similar to those done in calls to polymorphic functions. The domain of the substitution is S, its range is $S\star$. The substitution is injective on all stream identifiers except $\star$, due to linearity.

The rule (VAR) relies on a *$\star$-substitution*. This is a substitution that replaces zero or more non-$\star$ elements of $SI(\sigma)$ with "$\star$", but is otherwise the identity substitution. This substitution allows different streams to be unified in (CONSTR-ELIM), by giving all the streams involved a "$\star$" identifier.

4.2. Linearity

The linearity constraint prevents a reference into a stream being reused, so that once an item is read, it cannot be read from the stream again. It does so 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 is also permitted.

[4] As only tail-recursion is allowed, 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.

$$(\text{APPLY})\frac{A \vdash e : \sigma_1 \quad f : \sigma_2 \to \sigma_3}{A \vdash fe : \theta(\sigma_3) \quad \theta(\sigma_2) = \sigma_1}$$

$$(\text{CONSTR-INTRO})\frac{A \vdash e_1 : \tau_1 \quad \cdots \quad A \vdash e_k : \tau_k}{A \vdash c(e_1, \ldots, e_k) : \tau} c : \tau_1 \ldots \tau_k \to \tau$$

$$(\text{TUPLE-INTRO})\frac{A \vdash e_1 : \sigma_1 \quad \cdots \quad A \vdash e_k : \sigma_k}{A \vdash (e_1, \ldots, e_k) : \sigma_1 \times \ldots \times \sigma_k}$$

$$(\text{CONS-INTRO})\frac{A \vdash e_1 : \tau \quad A \vdash e_2 : \tau\ stream_\star}{A \vdash e_1 :: e_2 : \tau\ stream_\star}$$

$$(\text{CONSTR-ELIM})\frac{A \vdash e : \tau \quad \begin{cases} A, x_1^1 : \tau_1^1, \ldots, x_{k_1}^1 : \tau_{k_1}^1 \vdash e_1 : \sigma \\ \cdots \\ A, x_1^n : \tau_1^n, \ldots, x_{k_n}^n : \tau_{k_n}^n \vdash e_n : \sigma \end{cases} \quad \begin{cases} c_1 : \tau_1^1 \ldots \tau_{k_1}^1 \to \tau \\ \cdots \\ c_n : \tau_1^n \ldots \tau_{k_n}^n \to \tau \end{cases}}{\begin{array}{l} A \vdash \textbf{case}\ e\ \textbf{of}\ c_1(x_1^1, \ldots, x_{k_1}^1) \Rightarrow e_1 \\ \quad \mid \ldots \\ \quad \mid c_n(x_1^n, \ldots, x_{k_n}^n) \Rightarrow e_n : \sigma \end{array}}$$

$$(\text{TUPLE-ELIM})\frac{A \vdash e_1 : \sigma_1 \times \ldots \times \sigma_k \quad A, x_1 : \sigma_1, \ldots, x_k : \sigma_k \vdash e_2 : \sigma}{A \vdash \textbf{case}\ e_1\ \textbf{of}\ (x_1, \ldots, x_k) \Rightarrow e_2 : \sigma}$$

$$(\text{CONS-ELIM})\frac{A \vdash e_1 : \tau\ stream_i \quad A, x_1 : \tau, x_2 : \tau\ stream_i \vdash e_2 : \sigma}{A \vdash \textbf{case}\ e_1\ \textbf{of}\ x_1 :: x_2 \Rightarrow e_2 : \sigma}$$

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

$$(\text{VAR})\frac{}{A, x : \sigma \vdash x : \theta(\sigma)} \theta \text{ is a } \star\text{-substitution}$$

Figure 5. Typing rules

$$\begin{array}{rcl}
lin(f\ e_1) & = & lin(e_1) \\
lin(c(e_1, \ldots, e_k)) & = & lin(e_1) \uplus \ldots \uplus lin(e_k) \\
lin((e_1, \ldots, e_k)) & = & lin(e_1) \uplus \ldots \uplus lin(e_k) \\
lin(e_1 :: e_2) & = & lin(e_1) \uplus lin(e_2) \\
lin(\textbf{case}\ e\ \textbf{of}\ m_1 | \ldots | m_n) & = & lin(e) \uplus (lin_m(m_1) \cup \ldots \cup lin_m(m_n)) \\
lin(\textbf{case}\ e_1\ \textbf{of}\ (x_1, \ldots, x_k) \Rightarrow e_2) & = & lin(e_1) \uplus (lin(e_2) \setminus \{x_1, \ldots, x_k\}) \\
lin(\textbf{case}\ e_1\ \textbf{of}\ x_1 :: x_2 \Rightarrow e_2) & = & lin(e_1) \uplus (lin(e_2) \setminus \{x_1, x_2\}) \\
lin(\textbf{let}\ x = e_1\ \textbf{in}\ e_2) & = & lin(e_1) \uplus (lin(e_2) \setminus \{x\}) \\
lin(x) & = & \begin{cases} \{x\} & : \text{ If the type of } x \text{ contains a stream} \\ \emptyset & : \text{ otherwise} \end{cases} \\
lin_m(c(x_1, \ldots, x_k) \Rightarrow e) & = & lin(e) \setminus \{x_1, \ldots, x_k\}
\end{array}$$

Figure 6. Linearity rules

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, of using an accumulator argument to a function, is disallowed by the stability constraint). 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 streams can only return one stream, each stream can only have one other stream that depends on it for a statically unbounded number of elements. It is not possible to generate multiple distinct streams that depend on the same original stream. Linearity similarly prevents a stream being passed to a function along with a stream it depends upon, since the original stream is "used up". Linearity thus prevents the synchronisation problems mentioned in Section 3.3, as well as preventing "rewinding". For example, linearity prevents the creation of a function that returns two copies of a stream provided as a parameter (such as the function `dup_stream` in Section 5).

Linearity is 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 $\uplus$ operator. If the sets being merged contain a repeated linear variable, an error is yielded. A linear value cannot be held inside a non-linear value, so any type containing a stream is linear.

The operators $\uplus$ and $\cup$ are defined as:

$$s \uplus t = \begin{cases} \textbf{error} & : \text{ if } s = \textbf{error} \vee t = \textbf{error} \vee s \cap t \neq \emptyset \\ s \cup t & : \text{ otherwise} \end{cases}$$

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

The linear variables used by a typed expression can be found using the function lin shown in Figure 6. A program has the required linearity property if none of its function bodies yield **error** when lin is applied.

4.3. Stability

In this language, processing of streams is achieved using recursive functions. To produce non-stream results tail-recursion can be used, while to generate an entire new stream a tail recursive call is done 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[5] of Figure 7. 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 CONSed 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 itself, and those streams reached by repeatedly taking the tail of the stream. By limiting the streams that can be passed recursively in this manner, the space requirements of the streams cannot build up.

In the typing system of Section 4.1, a stream that is guaranteed to be a substructure of one of the streams in the function's parameters will have the same stream identifier. Otherwise the stream will have the identifier "$\star$".

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

$$(\text{APPLY})\frac{A \vdash e : \sigma_1}{A \vdash f\,e : \theta(\sigma_3)} \quad \begin{array}{c} f : \sigma_2 \to \sigma_3 \\ \theta(\sigma_2) = \sigma_1 \end{array}$$

if $f\,e$ is a recursive tail call then θ must be the identity

4.4. Static Allocation

Viewing CONS expressions as closures, with CONS-matching performing the execution, the programs can be statically allocated if the closures are guaranteed to be able to be statically allocated. With non-recursive datatypes, a closure may only take an unbounded amount of space if the environment of a closure may contain other closures, nested arbitrarily deeply. The only way to generate arbitrarily deeply nested closures would be through tail recursive calls, but the stability constraint prevents layers of closures building up, as the only stream parameters passed to the recursive call are substructures of the original parameters—in terms of closures, functions may only evaluate closures, not generate them, before performing tail calls.

5. Example Programs

Common operations to generate, map, filter and fold lists are simple to write in SASL:[6]

```
fun from (i) = i::from(i+1)
```

[5] The displayed parts of the function bodies are assumed to be in tail context of a larger, valid expression.

[6] To avoid using higher-order functions, we provide *f*, *g* and so on as top-level functions rather than parameters.

```
(* Plain tail recursion. *)
fun f1 (x) = ... 1 + f1(x)        (* Disallowed. *)
fun f2 (x) = ... f2(x + 1)        (* Allowed.    *)

(* Stream tail recursion. *)
fun g1 (x::xs) = ... 1 :: g1(xs)  (* Allowed.    *)
fun g2 (xs)    = ... g2(1 :: xs)  (* Disallowed. *)
```

Figure 7. Examples of recursive functions

```
fun map_f (x::xs) = f(x)::map_f(xs)

fun filter (x::xs) =
  if test(x) then x::filter(xs) else filter(xs)

fun fold_g (x::xs, acc) =
  if done(x)
  then accumulator
  else fold_g(xs, g(acc, x))
```

Streams may be merged together (subject to linearity), but streams may not be duplicated, or multiple streams created depending on unbounded sections of the same stream, since the resulting streams may not be able to be merged without unbounded buffering:

```
(* An allowed merge function. *)
fun merge_h (x1::xs1, x2::xs2) =
  h(x1, x2)::merge_h(xs1, xs2)

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

This difference to synchronous stream languages is discussed in the next section.

6. Language Comparison

We compare SASL with the statically-allocated functional language SAFL+, and the synchronous stream language Lustre. In comparison to SAFL+, our language simplifies the processing of streams of data, as the programmer need not have to deal with the explicit parallelism of passing data over channels, and need not worry about the possibility of deadlock. SASL is a pure functional language, unlike SAFL+.

Comparisons with Lustre seem useful, as this is another language that concentrates on stream-processing, although it takes a quite different approach. 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, which may be useful for component-based design, where the programmer need not specify the exact timing of components and signalling between them.

The demand-driven nature of SASL streams makes the `merge_h` function simple to write, while in Lustre it is necessary to organise the clocks so that values occur on both input streams at the same time, perhaps through back-pressure if either input stream may use 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 show how to embed Lustre in SASL).

The use of streams for everything in Lustre may complicate some things, such as loops. While iteration in SASL can be achieved through a tail-recursive function call, Lustre requires a data-flow program that either performs an iteration of the loop, or resets the loop with new values if a request comes 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 loop has finished.

SASL hides the implementation of back-pressure and signalling. 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 aims to provide features similar to those in a conventional software language, presenting a higher-level interface to the programmer. Control over the cycle-based timing of streams is lost, in exchange for more flexible synthesis, freeing the programmer from many details[7]. For example, functions may be composed without worrying about timing or signalling:

```
fun compose1 (i) =
  fold_g(filter(map_f(from(i))), 0)

fun compose2 (i) =
  if predicate(i)
```

[7]It may be possible to extend the language with pragmas or annotations specifying timing requirements the synthesis tool must meet—this is a possible further area of research.

```
  then i
  else compose2(compose1(i))

fun compose3 (x::xs) =
  compose2(x)::compose3(xs)
```

To aid comparison, we will compare implementations of a simple example program from [6] in Lustre and SASL. The Lustre program is as follows:

```
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.
```

The most direct translation to SASL would involve putting all the Lustre parameter streams into one tupled SASL stream, producing a program such as:

```
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 = deadline and is_set in
    (alarm, set, reset, deadline)
      :: WatchDogInt(rest, False, is_set)

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

In this translation, each item in the stream of tuples represents the values held by Lustre streams on that clock. Since all the Lustre streams are synchronised by a clock scheme, they are put in the same SASL stream, as unused SASL streams are independent. Other Lustre streams using different clocks can be added using hiatons, and Lustre `pre` values are generated by passing extra non-stream parameters such as `prev_is_set`. The returned stream passes back all the data passed in, since the stream passed in cannot be reused 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 implemented as a function of the form

```
fun example ((x1,... xn)::tl, state) =
  (f1(...), ..., fm(...))::example(tl, g(...))
```

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, duplication of a stream in Lustre can be represented in SASL by duplicating elements:

```
fun dup_elt (x::xs) = (x, x)::dup_elt(xs)
```

(as compared to the disallowed `dup_stream` function of Section 5).

A better translation would match the task required to the language features of SASL. For example, the function could read tuples of set, reset and deadline values until the "alarm" condition is met, and then return the remaining input at that point, making use of the function call mechanism:

```
fun WatchDogInt (str, prev_is_set) =
  case str of (set, reset, deadline)::rest =>
    let is_set = if set then True
                 else if reset then False
                 else prev_is_set in
    if deadline and is_set
    then rest
    else WatchDogInt(rest, is_set)

fun WatchDog (str) =
  WatchDogInt(str, False)
```

Note that translating programs suited to Lustre into SASL does not show off the natural usage of SASL, or features difficult to describe in Lustre, as discussed above.

7. Hardware synthesis

We plan to cover the details of synthesis in another paper. As a brief outline, streams are converted to a form of synchronous channel, with CONS becoming a channel write and CONS-matching a channel read. Functions become modules that provide call and return buses to take and return non-stream values, and input and output channels represent the supplied and returned streams respectively. An initial "function call" sets up the state for producing channel values, and returns non-stream results. Further reads from output channels return stream items on demand. Each active stream can be mapped to a separate hardware channel.

In terms of optimising synthesis, parallelism can be extracted at the dataflow level, as can be done in SAFL, and also at the stream processing level, by speculatively producing stream items before demand on otherwise idle hardware, and pipelining the processing of stream items.

8. Conclusions and Further Work

We have introduced a pure functional language, SASL, that can deal with streamed I/O, and that can be statically allocated. The language is modelled on conventional functional languages, using tail recursion and standard list-processing notation. Synthesis techniques are being investigated, and we hope to produce a SASL-based optimising high-level synthesis tool for software-like descriptions.

There is much scope for further work. As it stands, SASL may be overly restrictive, and it may be useful to investigate variations on the linearity and stability constraints

to give the programmer more flexibility. For example, linearity constraints prevent variables of stream type being "broadcast" to multiple functions, even if this causes no synchronisation problems. Extended constraints could use something like the clock calculus of synchronous stream languages, or just allow limited reuse of streams, as long as related streams are never merged.

Also, the language could be extended, such as adding higher order features or non-deterministic stream processing (useful for merging together data items from different streams as they arrive, for example). Our main area of ongoing research, however, is in language synthesis, with the aim of creating efficient pipelined hardware based around stream processing.

Acknowledgements

The first author gratefully acknowledges a PhD studentship from Altera; this work was partly supported by (UK) EPSRC grant GR/N64256. The anonymous reviewers provided helpful remarks.

References

[1] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[2] G. Berry. The foundations of Esterel. In *Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.

[4] B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in Hawk. In *In Workshop on Formal Techniques for Hardware*, 1998.

[5] S. R. Guo and W. Luk. Compiling Ruby into FPGAs. In *Field-Programmable Logic and Applications*, pages 188–197, 1995.

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[7] D. T. Hoang. Searching genetic databases on Splash 2. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, Napa, CA, Apr. 1993.

[8] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Symposium on Principles of Programming Languages*, pages 410–423, 1996.

[9] N. D. Jones. The expressive power of higher-order types or, life without CONS. *Journal of Functional Programming*, 11(1):55–94, January 2001.

[10] S. P. Jones and J. Hughes. Report on the programming language Haskell 98, a non-strict purely functional language. Technical Report YALEU/DCS/RR-1106, Yale University, February 1999.

[11] J.-P. Kaps and C. Paar. Fast DES implementation for FPGAs and its application to a universal key-search machine. In S. E. Tavares and H. Meijer, editors, *Selected Areas in Cryptography '98, SAC'98, Kingston, Ontario, Canada, August 17-18, 1998, Proceedings*, volume 1556 of *Lecture Notes in Computer Science*, pages 234–247. Springer-Verlag, 1999.

[12] Y. Li and M. Leeser. HML: an innovative hardware description language and its translation to VHDL. In *Proceedings of CHDL'95*, 1995.

[13] D. MacQueen. Models for distributed computing. Technical Report 351, Institut de Recherche d'Informatique et d'Automatique, 1979.

[14] A. Mycroft and R. Sharp. The FLaSH project: Resource-aware synthesis of declarative specications. In *Proceedings of The International Workshop on Logic Synthesis 2000*, 2000.

[15] A. Mycroft and R. Sharp. A statically allocated parallel functional language. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1853 of *Lecture Notes in Computer Science*, 2000.

[16] A. Mycroft and R. Sharp. Hardware synthesis using SAFL and application to processor design. In T. Margaria and T. Melham, editors, *Correct Hardware Design and Verification Methods: 11th IFIP WG10.5 Advanced Research Working Conference, CHARME 2001: Livingston, Scotland, UK, September 4–7 2001: Proceedings*, volume 2144 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[17] J. J. O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Functional Programming Languages in Education*, pages 195–214, 1995.

[18] L. Pareto. *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology, Sweden, 2000.

[19] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[20] R. Sharp and A. Mycroft. A higher-level language for hardware synthesis. In *Proceedings of 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, 2001.

[21] M. Sheeran. Designing regular array architectures using higher order functions. In Jouannaud, editor, *Proceedings of International Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 220–237, 1985.

[22] J. Srinivasan. Hardware accelerated ray tracing. Final year undergraduate project, Cambridge University Computer Laboratory, 2002.

[23] P. Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *ACM Symposium on Lisp and Functional Programming*, pages 45–52, 1984.

[24] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.