Statically-allocated languages for hardware stream processing (extended abstract)

Simon Frankau*
(sgf22@cl.cam.ac.uk)

Alan Mycroft
(am@cl.cam.ac.uk)

Simon Moore
(swm11@cl.cam.ac.uk)

1 Introduction and motivation

Current HDLs present a very low level of abstraction, often requiring the designer to work on details that could be hidden, and leaving little scope for synthesis tools to optimise performance. High-level hardware descriptions thus have a useful rôle in EDA, especially in areas such as rapid development, and reconfigurable computing using devices such as FPGAs. One approach to implementing these high-level description languages is to use software-like languages to describe hardware, with synthesis tools that will convert these "programs" to netlists. Such an approach can allow non-experts to design hardware, increase the productivity of more seasoned users, and increase the flexibility of the implementation by tying it to fewer low-level details. The synthesis tools can apply a wider range of optimisations, at a higher level of abstraction.

This paper covers such a language, called SASL. It is based on the SAFL [9, 8, 12] language, which is a simple functional programming language designed for implementation in hardware. The aim of SASL is to extend the software-like language features available to hardware designers. By introducing features such as algebraic datatypes and lazy lists, it should be possible to raise the level of abstraction at which designs are produced. It is hoped that such designs could then be synthesised to efficient pipelined hardware, by mapping lazy lists to sequences of data items, for example.

To require no external memory, the language is statically-allocated. Recursive data structures are not allowed, and the only form of recursion allowed is tail-recursion, to prevent the need for a dynamically allocated stack. While it would be possible to synthesise circuits that access memories, to allow for more general purpose computation, this could lead to the introduction of von Neumann bottlenecks. The synthesis aims are to improve performance over a software implementation, not only by providing more parallelism, but by holding data closer to the hardware which performs operations on it.

SASL is a pure functional language. Functional programming languages provide a relatively abstract way of describing algorithms, and may provide less of a bias towards an implementation based on sequential computation than many other programming language paradigms. Functional languages also provide a wealth of program transformations and optimisations which seem suited to the creation of useful design trade-offs at the synthesis stage.

As a simple first-order pure functional language, SAFL's I/O capabilities are very limited. The language SAFL+[12] has extended the language to include channels, similar to those used in Occam [11] and Handel-C [4]. With channels, the programmer must be aware of the parallelism, and expressions may have side effects. This paper investigates the use of *streams*, which we believe are a simpler and more natural way of representing I/O for hardware communication.

Streams are *linear lazy lists*. They provide an ordered, possibly infinite, list of data items, each of which can only be read once. The items of data are generated on demand. This form

of communication is suitable for a number of applications, such as audio processing, or the processing of data retrieved using a regular pattern of memory accesses. Stream processing can also capture some of the compositional aspect of hardware, allowing stream-processing functions to be composed in series and so on.

2 Related work

The approach taken here is distinct from languages such as HML [7], Lava [3], muFP [13], Ruby [6] and Hawk [5], which embed a structural hardware language within a functional framework. In such languages, a program is generally executed to generate a structural netlist, whereas SAFL-like languages are "behavioural" descriptions that could be directly interpreted as a standard functional language, as well as compiled to hardware.

Lucid [1] is a language intended for use as a formal system, which takes a rather different approach to streams. Streams are described with the `first` and `next` primitives, and loops are generated by extracting elements from streams using the `as soon as` primitive. This sort of idea is used in synchronous dataflow/signal languages such as Lustre [2] and Hydra [10], where the basic datatypes are streams. The elements of the streams are defined in terms of non-recursive functions of elements of other streams, and earlier stream elements. Iteration is built on top of the streams, with iterations of a loop being mapped to elements of a stream, for example. In SASL the opposite approach is taken, providing tail recursion, and defining streams in terms of the tail-recursive functions that generate them.

3 The language

The abstract grammar[1] for the language is shown in Figure 1. The language is first-order, and without lexical scoping. In order to make the language statically allocated, only tail recursion is allowed. Tail calls must take place in a tail context. Tail contexts are those expressions marked in the grammar with tr, when the enclosing expression is also in a tail context.

$p :=$	$d_1 \ \ldots \ d_n$	program definition
$d :=$	**fun** $f\ x = e^{tr}$	function definition
$e :=$	$f\ e$	function application
$\mid$	$c(e_1, \ldots, e_k)$	constructor
$\mid$	$(e_1, \ldots, e_k)$	tupling
$\mid$	e_1**::**e_2^{tr}	CONS
$\mid$	**case** e **of** $m_1 \mid \ldots \mid m_n$	constructor case
$\mid$	**case** e_1 **of** $(x_1, \ldots, x_k) \Rightarrow e_2^{tr}$	tupling case
$\mid$	**case** e_1 **of** x_1**::**$x_2 \Rightarrow e_2^{tr}$	CONS case
$\mid$	**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

The language is eagerly evaluated, except for CONS, where both the head and tail expressions are lazily evaluated. Upon case matching of a CONS, both the head and tail expressions are evaluated, and the body of the case expression is only evaluated when both the head and tail have finished evaluating (the tail normally evaluating to another lazily evaluated CONS expression). Streams are infinite in length, which does not cause a problem, as they are evaluated lazily. The effect of finite streams can be achieved through streams that just produce the value `Nil` after a finite number of elements.

[1] Example code may use other constructs that can be reduced to this grammar, to aid readability.

The type system uses Hindley-Milner style type inference, but with a hierarchical type system to prevent streams of streams, or streams being held inside algebraic datatypes. Although type definitions aren't included in the abstract grammar, we assume that non-recursive algebraic datatypes can be constructed, which are used through constructor and constructor case expressions. Constructors take zero or more arguments, returning the appropriate algebraic datatype. *Basic* types are built from constructors. For example, boolean values can be represented with zero-parameter constructors `True` and `False`. Tuples of basic types can be created using constructors. For example, n-bit binary numbers can be represented with constructors that take n boolean arguments. A *value* type represents the type of an expression, and may be a basic type, a stream of basic type items, or a tuple of value types. Functions are typed as taking a value type and returning a value type, while constructors take a list of basic types, and return a basic type.

4 Possible pitfalls

A stream-less version of the language does not require any special constraints to allow static allocation of programs, but the introduction of streams creates new problems. It must not be possible to create programs that require unbounded buffers of stream values, or otherwise build up the processing required by a stream in an unbounded way. A stream should not be able to be "rewound"—after reading an unbounded number of items from a stream, it should not be possible to go back an unbounded number of items. *Linearity* prevents this problem; once a stream is matched to read an item from it, it cannot be used again, and the program may only match on the tail of the stream, to read the next item.

It should not be possible to build up streams recursively, with functions such as

```
fun build(item, stream) = build(item, (item::stream))
```

Similarly, it should not be possible to build up the computation required along a stream. For example, the following program should not be allowed (as the number of times f is applied to an item increases unbounded):

```
fun map_f(stream) = case stream of hd::tl => f(hd)::map_f(tl)

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

5 Constraints for static allocability

Two constraints are used to make programs statically allocated—*linearity* and *stability*. Linearity ensures that each stream variable, or variable containing a stream, is only used at most once. The same item cannot be read out of a stream repeatedly, and the same stream cannot be passed to two different functions, with the same items being read in both. A combination of linearity and the tail recursion constraints prevent the creation of multiple copies of a stream, and other operations that may require unbounded buffering between streams. We use an affine linearity constraint on all variables that contain a stream type.

Stability prevents a stream from requiring more and more storage space over repeated tail calls to a function. The stability constraint we use is that in recursive calls the streams passed as actual parameters must be *substreams* of the matching formal parameter, where a substream is the stream produced by taking the tail of a stream zero or more times. We implement the stability constraint by using *stream identifiers*, attached to the typings of streams. The stability constraint disallows the functions `build` and `map_iter_f` from the previous section, while still allowing statically allocable functions such as `map_f`.

6 Examples

Common list-processing operations, such as `map`, `fold` and `zip` can be applied to the streams. As the language is first-order, fully general functions cannot be created, but appropriate functions can be made by renaming:

```
fun map_f stream =
    case stream of hd::tl => f(hd)::map_f(tl)

fun fold_g (acc, stream) = case stream of hd::tl =>
    if test(hd)
    then let acc' = g(acc, hd) in fold_g(acc', tl)
    else acc

fun zip (stream1, stream2) =
    case stream1 of hd1::tl1 =>
         case stream2 of hd2::tl2 =>
              (hd1, hd2)::zip(tl1, tl2)
```

In terms of real-world examples, we provide a function that searches a stream for a two character header, returning upon a match, and a very simple, abstract version of an audio tone generator and mixer:

```
(* Header matcher. *)
fun stream_sync_internal(stream, last_char) =
    case stream of hd :: tl =>
         if last_char = header_1 and hd = header_2
         then tl
         else stream_sync_internal(tl, hd)

fun stream_sync(stream) = stream_sync_internal(stream, header_1 + 1)

(* Tone generator and mixer. *)
fun sine_gen(t, delta_t, omega) =
    sin(t * omega) :: sine_gen(t + delta_t, delta_t, omega)

fun mix(stream_1, volume_1, stream_2, volume_2) =
    case stream_1 of hd_1::tl_1 =>
         case stream_2 of hd_2::tl2 =>
              hd_1*volume_1 + hd_2*volume_2 ::
                  mix(tl_1, volume_1, tl_2, volume_2)
```

7 Conclusions and further work

The typing rules and static allocation constraints are being formalised, and hardware synthesis techniques investigated. We are currently working on a synthesis tool, and while some extensions to the language are being investigated, the main aim of the research is towards optimising synthesis of the language, by pipelining the stream processing, for example. The aim is to provide the programmer with a powerful and flexible software-like environment which can be used to synthesise efficient hardware.

Acknowledgements

The first author was sponsored by Altera Corporation; this work was supported by (UK) EPSRC grant GR/N64256: “A Resource-Aware Functional Language for Hardware Synthesis”.

References

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

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

[3] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.

[4] Celoxica. Handel-C language reference manual. Product manual.

[5] Byron Cook, John Launchbury, and John Matthews. Specifying superscalar microprocessors in Hawk. In *In Workshop on Formal Techniques for Hardware*, 1998.

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

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

[8] Alan Mycroft and Richard 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.

[9] Alan Mycroft and Richard 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.

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

[11] SGS-THOMSON Microelectronics Limited. *occam 2.1 reference manual*. Prentice Hall International (UK) Ltd, 1988.

[12] Richard Sharp and Alan 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.

[13] Mary 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.