Statically-allocated languages for hardware stream processing

Simon Frankau, Alan Mycroft & Simon Moore

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Motivation

We wish to investigate *high-level* hardware description languages. The aim is to produce a language which resembles a conventional software language, but can also be synthesised to efficient hardware.

Such a languages could be useful for:

- Rapid development
- Use by non-specialists and software programmers
- Programming of reconfigurable hardware systems

The language should:

- Abstract away the low-level details of timing and signalling.
- Minimise the amount of explicit parallelism shown to the user, to give the appearance of a conventional language.
- Impose a minimal amount of ordering constraints, to allow synthesis tools to create pipelined circuits with a high degree of parallelism.
- Provide I/O capabilities suited to hardware implementations.

We have chosen to use a functional language with linear lazy lists, as this seems to fulfill the above requirements.

Existing Languages

SAFL is a *statically-allocated* pure functional language:

- The language is limited to non-recursive datatypes and tail-recursion.
- I/O is very limited, consisting of a call/return mechanism matching that used in function calls.
- As a pure functional language, no state is held between calls.

LUSTRE is a synchronous stream-processing language which uses a data-flow model:

- All variables are streams of values that vary over time.
- Values are produced with a synchronous clock.
- Streams can generate values on a subset of the global clock signals—a clock calculus is used to ensure synchronisation.
- Each stream item is generated by combinatorial expression using other stream items, and stream items from previous clocks.

SASL

SASL is similar to **SAFL**—it is a statically-allocated eager functional language. I/O is added in the form of *lazily evaluated, linear list*.

- Each stream item is used only once—streams cannot be "rewound".
- The evaluation of stream items is demand-driven, "back pressure" is provided, and the synchronisation between independent streams is handled automatically. The timing of streams is implicit, in contrast to LUSTRE's model.

To the programmer, streams are like conventional lists in a language like **ML**, using the same "::" CONS operator. At the hardware level, the streams are buses with request/acknowledge signalling. CONS becomes a write, and CONS-matching becomes a read.

```
fun add_blocks (head::rest, acc) =
  case head of Item i -> add_blocks (rest, acc+i)
             | Marker -> acc :: add_blocks(rest, 0)
```

Language Constraints

Without restrictions, programs could be created that require unbounded buffering:

```
fun desynchronise (stream) = zip(stream, filter(stream))
```

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

We use similar constraints to **SAFL**, adding a hierarchical type system to prevent streams of streams being created. On top of this we add *linearity* and *stability* constraints:

Linearity prevents a stream from being rewound, or otherwise used more than once. As each stream can be used only once, generating one new stream, any pair of usable streams can be processed together without synchronisation problems.

Stability stops recursive calls from building up streams, so that the state associated with a stream is statically-allocated.

Further Work

- Researching efficient synthesis techniques for the language.
- Implementing a synthesis tool for the language, compiling to a conventional HDL such as Verilog.
- Extending the language (e.g. adding non-deterministic operators).

simon.frankau@cl.cam.ac.uk