Stream Processing Hardware from Functional Language Specifications

Simon Frankau Alan Mycroft
Computer Laboratory, University of Cambridge
`{sgf22,am}@cl.cam.ac.uk`

Motivation

- Aiming towards *higher-level* hardware description languages
  - *cf.* high-level software languages
  - Abstract away timing, signalling and wires
  - Improves productivity, makes life easier for non-specialists, reduces technology dependence
- Removing explicit timing—software-like description:
  - Tool does pipelining and scheduling, etc.
  - Avoid restricting parallelism
- Optimising synthesis tool needed (not covered here)
- Uses:
  - Streamed media
  - Reconfigurable systems
  - Emphasise throughput, not timing

Related Languages—Synthesis

	Imperative	Functional
Structural	Low-level VHDL/Verilog . . .	HML, Lava, muFP, Ruby, Hawk . . .
Behavioural	High-level VHDL/Verilog, Handel-C . . .	**SAFL, SAFL+, SASL**

- **SAFL** is a pure functional language; poor I/O (call/return only)

e.g.

```
fun mult(x, y, acc) =
    if (x=0 or y=0) then acc
    else mult(x<<1, y>>1, if y[0:0] then acc+x else acc)
```

- **SAFL+** adds channels; not pure functional, can deadlock, explicit parallelism
- **SASL** combines best of both

Related Languages—Streamed I/O

- *Lazy lists*
  - many functional languages

    e.g.
    ```
    mapinc [] = []
    mapinc (x::xs) = (x+1) :: (mapinc xs)
    ```
- *Synchronous dataflow*
  - Software language **Lucid** builds streams with `first` and `next`
  - Loops use streams
  - **Lustre** is HDL version
  - Clocked streams, compile-time consistency check
  - (www.esterel-technologies.com)

    e.g.
    ```
    toggle = true → not(pre(toggle))
    ```

A Brief Overview of SASL

A statically-allocated	Maps well to hardware. No recursion except directly recursive tail-calls. No unlimited recursive types.
strongly typed	Prevents run-time errors, simplifies synthesis.
eager	Evaluates expressions as soon as possible. Lazy streams, otherwise eager. Bounds storage requirements.
pure functional	No side-effects or modifiable variables. Good properties for optimisation/analysis. Less implied ordering than imperative
language with	
streams	Linear lazy lists. Generate items on demand, only read once.

SASL's Abstract Syntax

p :=	$d_1 \ \dots \ d_n$	Program definition
d :=	**fun** $f \ x$ **=** e^{tr}	Function definition
e :=	$f \ e$	Function application
$\mid$	$c(e_1, \dots, e_k)$	Constructor
$\mid$	$(e_1, \dots, e_k)$	Tupling
$\mid$	$e_1{::}e_2^{tr}$	Cons expression
$\mid$	**case** e **of** $m_1 \mid \dots \mid m_n$	Constructor case matching
$\mid$	**case** e_1 **of** $(x_1, \dots, x_k) \Rightarrow e_2^{tr}$	Untupling
$\mid$	**case** e_1 **of** $x_1{::}x_2 \Rightarrow e_2^{tr}$	Stream match and evaluation
$\mid$	**let** x **=** e_1 **in** e_2^{tr}	Let expression
$\mid$	x	Variable access
m :=	$c(x_1, \dots, x_k) \Rightarrow e^{tr}$	match

$tr =$ tail recursive context, if the enclosing expression is in a tail-recursive context.

Stream Semantics

- $e_1 :: e_2$
  - Return immediately, giving tuple (A, e_1, e_2), where A is environment
- **case** e_3 **of** $x_1 :: x_2 \Rightarrow e_4$
  - Evaluate e_3, giving (A, e_1, e_2)
  - Evaluate e_1 and e_2 in A, binding results to x_1 and x_2
  - Evaluate e_4

e.g.

```
fun toggle x = x :: toggle(not(x))
fun second x =
    case x of y :: ys ⇒
        case ys of z :: zs ⇒ z
fun example x = second(toggle(x))
```

- “Infinite” streams used—finite streams emulated with terminal symbols
- In hardware, CONS becomes a write, CONS-matching a read
- Demand-driven data production
  - Automatic back pressure
  - Simplifies stream merging

The Need for Restrictions

- Start with **SAFL**'s restrictions
- Without further restrictions, not statically allocatable

e.g.

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

✗

```
fun build-up (stream, item) =
    . . . build-up(item :: stream, item) . . .
```

✗

- Stream descriptions must fit in a fixed amount of storage:
    - Input streams must not be rewound—would need buffers
    - Streams must not be recursively built up
- Want simple rules to meet these criteria—*typing*, *linearity* and *stability*

The Type System

- **Basic Types:** non-recursive, non-stream algebraic datatypes
- **Value Types:** Basic Types, streams of Basic Types, and tuples of Value Types
- Prevents streams of streams, streams inside algebraic datatypes, etc.
- Each stream has a stream identifier:
  - Each stream formal parameter is given a fresh identifier
  - Expressions representing the same stream, or a tail of it, have same identifier
  - Other streams given the identifier “⋆”
  - Used for stability rule

e.g.

```
fun stream-mux (test, stream1, stream2) =
    if test
    then case stream1 of x :: xs ⇒ xs
    else stream2
```

Linearity

- Use Wadler's *Linear Typing* to prevent streams being rewound
- Variable containing streams (including tuples) may only be used once
  - Can be reused in different conditional branches
- Stream variables effectively represent pointers into streams
  - Once read, the same stream item cannot be read again
  - Rest of stream read through stream variable matched against tail

Stability

- Generation of unbounded stream descriptions requires unbounded iteration
- To statically bound storage requirements, require streams passed recursively to not require more space
- *Stability* requires stream identifiers in formal and actual parameters of recursive tail calls to match

✓

```
fun find-first (x :: xs) =
        if test(x)
        then x
        else find-first(xs)
```

✗

```
fun broken (x :: xs) =
        if test1(x)
        then x
        else if test2(x) then broken(f(x) :: xs) else broken(xs)
```

Language Comparison

- More flexible than **SAFL**
- Cleaner than **SAFL+**
- Same computational power
- **Lustre** ⇒ **SASL** translation is relatively simple:

```
node SR(set, reset: bool)
    returns (value:bool);
let
    value = set →
        if set then true else
        if reset then false else
        pre(value);
tel.
```

becomes

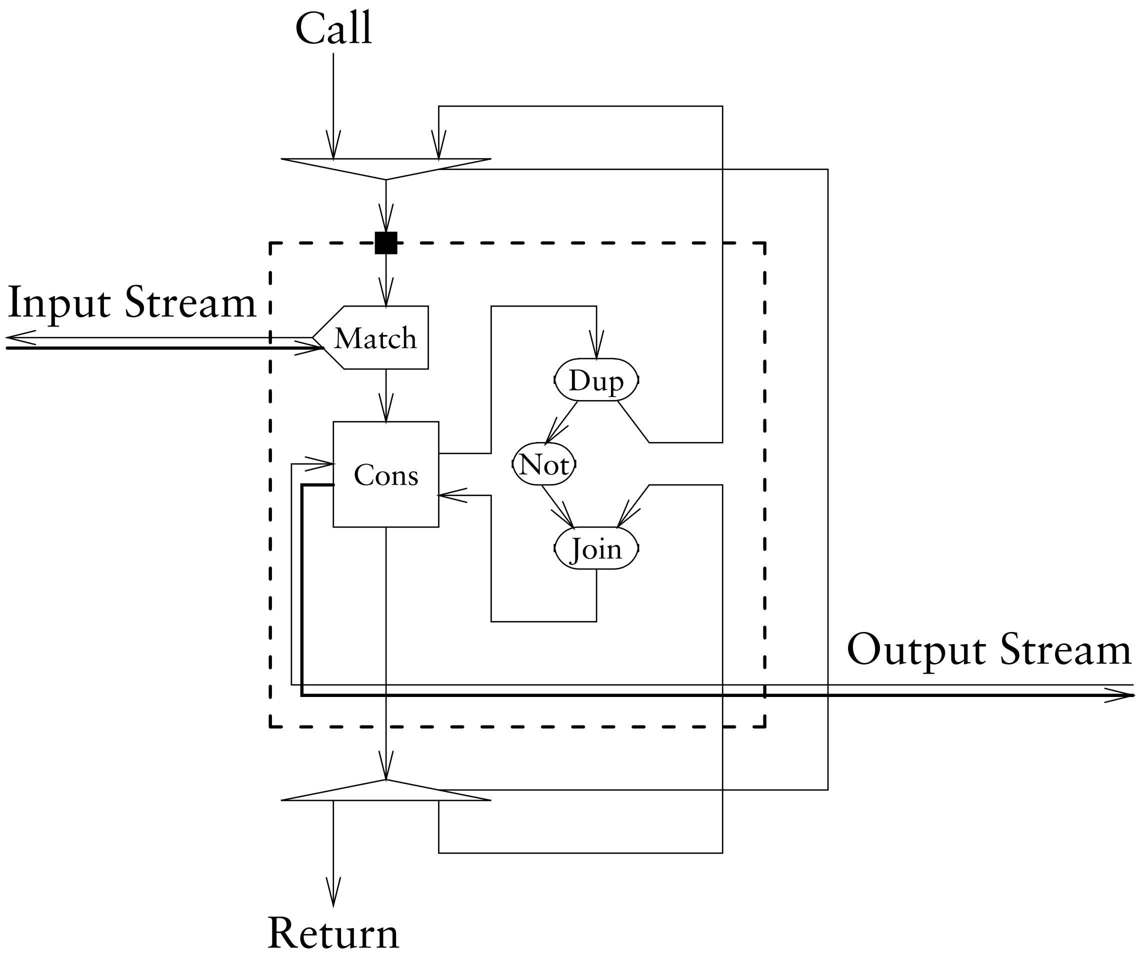
```
fun SR((set, reset)::rest, last) =
    let val = if set then true else
        if reset then false else
        last
    in val :: SR(rest, val)
```

- **SASL** ⇒ **Lustre** rather more complex—e.g. introduction of explicit back-pressure, removal of scalars
- Difference of approaches shown with:

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

✗

Example Synthesis



fun map-not(x :: xs) = not(x) :: map-not(xs)

Conclusions and Future Work

- Pure functional language modelled on conventional software languages
  - Statically-allocated, suitable for implementation in hardware
  - Streamed I/O model for complex reactive I/O, based on a demand-driven, non-synchronous execution model (no clock calculus)
- Future work:
  - Optimising synthesis techniques and implementation
  - Language extensions (e.g. non-determinism)
  - Other (more flexible) language restrictions