The FLaSH Project
Resource-Aware Synthesis of Declarative Specifications

Alan Mycroft[1,2] and Richard Sharp[2]

[1] Computer Laboratory, Cambridge University
New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK

[2] AT&T Laboratories Cambridge
24a Trumpington Street, Cambridge CB2 1QA, UK

am@cl.cam.ac.uk
rws@uk.research.att.com

Abstract. The FLaSH project concerns the development of a hardware synthesis system based around the idea of mapping a high-level functional specification language, SAFL, into hardware using sophisticated compiler technology.
The system has two phases: first we transform SAFL programs using meaning-preserving transformations to choose the area-time position (e.g. by resource duplication/sharing, specialisation, pipelining) while remaining a high-level specification. After this the FLaSH compiler maps the resultant SAFL program into hardware in a *resource-aware* manner, that is we map separate functions into separate functional units; functions which are called twice now become shared functional units—accessed by multiplexers and possibly arbiters. The current compiler outputs hierarchical RTL Verilog.
The first phase is user-guided. The second is completely automatic—it uses optimising compiler technology to insert arbiters for shared functional units and to insert intermediate registers (both on an 'only when needed' basis). We justify SAFL as both amenable to transformation and facilitating an efficient translation to hardware.
The current compiler has been used to implement a small commercial processor; we achieve similar gatecounts to two previous RTL and netlist specifications but with around one tenth the source code.

1 Introduction

Recent interest in hardware/software co-design has sparked an interest in silicon compilers which translate high-level languages such as C [3], Occam [11] and Java [6] into hardware. Although these systems succeed in producing correct circuits from high-level specifications they do not make full use of the high-level structure inherent in their input programs. For example, such compilers typically inline all function calls as a pre-processing stage, flattening program structure and risking an exponential increase in the size of both the source code and the final implementation.

The FLaSH (Functional Languages for Synthesising Hardware) project grew from the motivation for a high-level hardware description language which:

– allows a designer to describe hardware at a high-level whilst still retaining control over circuit structure;
– separates specification from implementation (i.e. a single specification can be compiled to a wide range of hardware styles);
– has high-level properties which aid user-guided program transformation; and
– facilitates program analysis leading to automatic generation of efficient circuits.

In an attempt to achieve these aims we have designed (a) a functional language, SAFL [9], which has a number of properties which are desirable for hardware description; and (b) an optimising compiler [13] which translates SAFL specifications into hardware.

One contribution of this research is the novel way we compile source-level functions (resource awareness). Our approach can be illustrated by considering the compilation of the following SAFL code:

```
fun mult(x, y, acc) =
  if (x=0 | y=0) then acc
     else mult(x<<1, y>>1, if y.bit0 then acc+x else acc)

fun cube(x) = mult(mult(x, x, 0), x, 0)
```

From this specification, the FLaSH compiler generates two hardware resources: a circuit, $H_{\texttt{mult}}$, corresponding to `mult`[1] and a circuit, $H_{\texttt{cube}}$, corresponding to `cube`. The two calls to `mult` are *not* inlined: at the hardware level there is only one shared resource, $H_{\texttt{mult}}$, which is invoked twice by $H_{\texttt{cube}}$. We say that our compiler is *resource-aware* because of the way it translates separate SAFL function definitions into separate hardware resources.

Source-level program transformation is exploited to allow multiple SAFL programs (each representing a different possible implementation when compiled) to be derived from an initial SAFL specification. These transformations are user-guided and applied as a pre-compilation phase. For the above example we can use SAFL-level transformation to produce a variant with two (unshared) shift-add multipliers—this is performed by duplicating the definition of `mult` as `mult'` and replacing `cube`'s definition with:

```
fun cube(x) = mult'(mult(x, x, 0), x, 0)
```

In general, resource-awareness allows us to use fold/unfold transformations [2] at the SAFL-level to express time-area tradeoffs at the hardware level. Section 5 presents a variety of useful transformations which naturally describe time-area tradeoffs.

We argue that resource-aware silicon compilation of SAFL specifications is a powerful technique for the following reasons:

- It enables a designer to write code at a high-level whilst still having control over the low-level structure of the generated circuit.
- Semantic preserving program transformation becomes a very powerful technique allowing a number of different implementations to be explored from a single specification (see Section 5).
- The functional properties of SAFL allow the FLaSH compiler to produce highly parallel circuits.
- SAFL is a high-level language and is hence implementation independent. This means that SAFL serves equally well to specify, for example, synchronous or asynchronous hardware.
- Issues surrounding resource-awareness, such as placement of arbiters to protect shared resources (see Section 3.1) and insertion of *permanising registers* (see Section 3.2), are handled automatically by high-level program analysis performed by the FLaSH compiler. This frees designers from low-level implementation details, allowing them to concentrate on high-level algorithmic issues.

The purpose of this paper is to give a general overview of the FLaSH project, introducing resource-aware hardware description and synthesis. Due to space constraints we cannot give detailed presentations of the algorithms used in our compiler. These are presented fully in [13]. Similarly, there is not room to describe the language design issues relating to SAFL (see [9]).

The remainder of this paper is structured as follows: Section 2 gives an informal overview of the SAFL language and Section 3 discusses the compilation of SAFL to hardware, presenting two issues that arise as a result of the resource-aware compilation strategy. Sections 4 and 5 attempt to justify some of our claims about resource-aware compilation of SAFL by providing concrete examples of high-level program analysis and SAFL-level program transformation respectively. Finally, Section 6 concludes and outlines future work.

[1] The tail-recursive call is synthesised into a feedback loop at the circuit level.

1.1 Comparison with other work

We are not the first to observe that the mathematical properties of functional languages are desirable for hardware description and synthesis. A number of synchronous dataflow languages, the most notable being LUSTRE [4], have been used to synthesise hardware from declarative specifications. However, whereas LUSTRE is designed to specify reactive systems SAFL describes interactive systems (this taxonomy is introduced in [1]). Furthermore LUSTRE is inherently synchronous: specifications rely on the explicit definition of clock signals. This is in contrast to SAFL which could, for example, be compiled into either synchronous or asynchronous circuits.

The ELLA HDL is often described as functional. However, although constructs exist to define and use functions the language semantics forbid a resource-aware compilation strategy. This is illustrated by the following extract from the ELLA manual [8]:

> Once you have created a named function, you can use instances of it as required in other functions ... [each] instance of a function represents a distinct copy of the block of circuitry.

ELLA contains low-level constructs such as `DELAY` to create feedback loops, restricting high-level analysis. SAFL uses tail-recursion to represent loops; this strategy makes high-level analysis a more powerful technique.

Previous work on compiling declarative specifications to hardware has centred on how functional languages themselves can be used as tools to aid the design of circuits. Sheeran's et al. muFP [14] and Lava [15] systems use functional programming techniques (such as higher order functions) to express concisely the repeating structures that often appear in hardware circuits. In this framework, using different interpretations of primitive functions corresponds to various operations including behavioural simulation and netlist generation. Our approach takes SAFL constructs (rather than gates) as primitive. Although this restricts the class of circuits we can describe to those which satisfy certain high-level properties, it permits high-level analysis and optimisation yielding efficient hardware.

Conventional HDLs such as Verilog and VHDL allow a user to specify a design at various levels of abstraction. We argue that we provide a higher-level of abstraction in that our system provides explicit support for resource management (e.g. automatic insertion of arbiters and temporary registers). Verilog and VHDL leave this to the programmer.

2 An informal overview of the SAFL language

As SAFL is being used for research purposes we deliberately kept the language simple. It is designed to be powerful enough to exhibit the behaviour we wish to study without the clutter of unnecessary features. However, despite its simplicity SAFL is still powerful enough to express a wide spectrum of designs. To demonstrate this we have used SAFL to specify a commercial processor[2].

SAFL is a language of first order recurrence equations with an ML-like syntax [7]; a user program consists of a number of function definitions (declared using the `fun` keyword) and a single *initialising expression* declared with the `do` keyword. The initialising expression is invoked as soon as the program is executed and is thus analogous to C's `main()` function. The abstract syntax of SAFL expressions, e, is as follows (we abbreviate tuples $(e_1, \ldots, e_k)$ as $\vec{e}$):

- variables: x; constants: c;
- user function calls: $f(\vec{e})$;
- primitive function calls: $a(\vec{e})$—where a ranges over primitive operators (e.g. `+`, `-`, `<=`, `&&` etc.);
- conditionals: `if` e_1 `then` e_2 `else` e_3; and
- let bindings: `let` $\vec{x} = \vec{e}$ `in` e_0 `end`

[2] We implemented the XAP processor designed by Cambridge Consultants: `http://www.camcon.co.uk`; we did not implement the `SIF` instruction.

SAFL has a call-by-value semantics as strict evaluation naturally facilitates fine-grained parallel execution which is well-suited to hardware implementation.

We enforce the restrictions that: (i) a function can only call previously defined functions; and (ii) all recursive calls must be in tail-context. Restriction (i) prohibits mutual recursion[3] which, if treated naïvely can create cycles in our call graph, leading to deadlock in our hardware implementations where functions represent shared resources. Restriction (ii) ensures that the amount of storage (e.g. number of registers) needed for a program's execution can be calculated at compile time, a property that we call *static allocatability.*

3 Resource-Aware Compilation of SAFL

We introduced the concept of resource-awareness by means of an example in Section 1. More generally, our compiler translates each SAFL function definition, $f(\vec{x}) = e$, into a single hardware resource, H_f, consisting of:

- a fixed (and statically computable) amount of storage;
- an output port, P_f; and
- a circuit to compute e to P_f

Hence, multiple calls to the function f at the source level always correspond to sharing resource H_f at the hardware level. Although at first sight this policy may seem restrictive, the reality is quite the opposite. It is because of resource awareness that SAFL program transformation naturally captures resource sharing/duplication tradeoffs (see Section 5.1). This makes our framework more powerful than other silicon compilers which *always* inline function calls—we view inlining as an implementation choice expressed as a source-level transformation.

The FLaSH compiler uses the functional properties of SAFL to produce highly parallel hardware; more precisely, all actual parameters and `let`-declarations are evaluated in parallel. The combination of parallelism and resource-awareness leads to some interesting issues that the FLaSH compiler needs to deal with. Two such issues are presented below.

3.1 Parallel Sharing Conflicts

Circuits generated by the FLaSH compiler consist of multiple threads accessing a shared set of resources. Consider the following SAFL code fragment taken from the specification of a processor:

```
...
fun add(x,y) = x+y
...
fun ALU(op, arg1, arg2, ...) =
     if op=1 then add(arg1,arg2)
             else ...
...
fun calculate_new_PC(current_PC,offset,condition) =
     if condition then add(current_PC,offset)
                  else current_PC
...
```

Since both the `ALU` and `calculate_new_PC` functions call `add`, FLaSH will synthesise a circuit containing a single `add`-unit shared between the `ALU` and `calculate_new_PC` units. One interesting question is, although the `add` circuit is shared, is it subject to multiple concurrent accesses—i.e. is there a scenario where both the `calculate_new_PC` and `ALU` functions may try to call the `add`

[3] In fact the formal semantics of SAFL presented in [9] permits a form of mutual recursion by stratifying function definitions into (potentially mutually recursive) *groups*.

circuit simultaneously? If so we say that the calls to `add` have a *parallel sharing conflict* and automatically synthesise an arbiter to protect $H_{\texttt{add}}$ from multiple concurrent accesses[4].

The FLaSH compiler performs *parallel conflict analysis* to infer which hardware resources are subject to sharing conflicts. This enables us to synthesise arbiters only where necessary—even though a functional unit is shared our compiler is often able to infer from the program structure that an arbiter is not required. Parallel conflict analysis is described in detail in Section 4.

3.2 Register Placement

Consider the following SAFL expression:

```
let x = f(4)
 in let y = f(5)
     in x + y
    end
end
```

In this example `x` is bound to the result of computing `f(4)` whilst `y` is bound to the result of computing `f(5)`. However, since `f` represents a shared resource, H_f, with a single output we see that, if translated naïvely, the second call to `f` will invalidate the first (since both `x` and `y` are bound to H_f's shared output). This is an instance of a *sequential sharing conflict*, the result of which is that we must synthesise a register to latch the value of `f(4)` before it is corrupted. We call these latches *permanising registers* since they make the result of computing an expression permanent, decoupling the caller from the callee.

The FLaSH compiler translates SAFL code into a control/data-flow intermediate representation and uses dataflow analysis to place registers. The details of this process are documented in [13].

4 High-level Analysis of SAFL for Optimising Compilation

We have stated that SAFL is well suited to high-level analysis and, further, that such analysis allows our compiler to generate efficient circuits. Here we attempt to justify this claim by giving a concrete example of one such analysis: *parallel conflict analysis* (introduced in Section 3.1 and abbreviated to *PCA* for the remainder of this paper). PCA is: (*i*) useful—it allows us to infer which shared resources require arbiters; (*ii*) only practically possible due to SAFL's high-level properties; and (*iii*) implemented efficiently as part of the FLaSH compiler.

We perform PCA at the abstract syntax level. In this presentation we use the notation introduced in Section 2 to represent SAFL expressions. In order to distinguish distinct calls we assume that each abstract-syntax node is labelled with a unique identifier, α, writing $f^\alpha(e_1, \ldots, e_n)$ to indicate a call to function f at abstract-syntax node α. We define a *call set* to be a set of calls. The result of PCA is a *conflict set*: a call set containing the calls which require arbiters. For example, if the resulting conflict set is $\{f^1, f^2, f^5, g^{10}, g^{14}\}$ then we would synthesise two arbiters: one for calls $\{f^1, f^2, f^5\}$ and one for calls $\{g^{10}, g^{14}\}$.

4.1 Equations for Parallel Conflict Analysis

Let e_f represent the body of function f. Let the predicate $\mathrm{RecursiveCall}(f^\alpha)$ hold iff f^α is a recursive call (i.e. occurs within e_f). $\mathcal{C}[\![e]\!]$ returns the set of non-recursive calls which may occur as

[4] Note the similarity between parallel sharing conflicts and structural hazards [5] in pipelined processor design.

a result of evaluating expression e:

$$\begin{aligned}
\mathcal{C}[\![x]\!] &= \emptyset \\
\mathcal{C}[\![c]\!] &= \emptyset \\
\mathcal{C}[\![a(e_1,\ldots,e_k)]\!] &= \bigcup_{1\le i\le k} \mathcal{C}[\![e_i]\!] \\
\mathcal{C}[\![f^\alpha(e_1,\ldots,e_k)]\!] &= (\bigcup_{1\le i\le k} \mathcal{C}[\![e_i]\!]\ ) \cup \begin{cases} \emptyset & \text{if RecursiveCall}(f^\alpha) \\ \{f^\alpha\} \cup \mathcal{C}[\![e_f]\!] & \text{otherwise} \end{cases} \\
\mathcal{C}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] &= \bigcup_{1\le i\le 3} \mathcal{C}[\![e_i]\!] \\
\mathcal{C}[\![\texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0]\!] &= \bigcup_{0\le i\le k} \mathcal{C}[\![e_i]\!]
\end{aligned}$$

$PC(\mathcal{S}_1,\ldots,\mathcal{S}_n)$ takes call sets, $(\mathcal{S}_1,\ldots,\mathcal{S}_n)$, and returns the conflict set resulting from the assumption that calls in each $\mathcal{S}_i$ are evaluated in parallel with calls in each $\mathcal{S}_j$ $(j \neq i)$:

$$PC(\mathcal{S}_1,\ldots,\mathcal{S}_n) = \bigcup_{i\neq j} \{f^\alpha \in \mathcal{S}_i \mid \exists\beta.\ f^\beta \in \mathcal{S}_j\}$$

We are now able to define $\mathcal{A}[\![e]\!]$ which returns the conflict set due to expression e:

$$\begin{aligned}
\mathcal{A}[\![x]\!] &= \emptyset \\
\mathcal{A}[\![c]\!] &= \emptyset \\
\mathcal{A}[\![a(e_1,\ldots,e_k)]\!] &= PC(\mathcal{C}[\![e_1]\!],\ldots,\mathcal{C}[\![e_k]\!]) \cup \bigcup_{1\le i\le k} \mathcal{A}[\![e_i]\!] \\
\mathcal{A}[\![f(e_1,\ldots,e_k)]\!] &= PC(\mathcal{C}[\![e_1]\!],\ldots,\mathcal{C}[\![e_k]\!]) \cup \bigcup_{1\le i\le k} \mathcal{A}[\![e_i]\!] \\
\mathcal{A}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] &= \bigcup_{1\le i\le 3} \mathcal{A}[\![e_i]\!] \\
\mathcal{A}[\![\texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0]\!] &= PC(\mathcal{C}[\![e_1]\!],\ldots,\mathcal{C}[\![e_k]\!]) \cup \bigcup_{0\le i\le k} \mathcal{A}[\![e_i]\!]
\end{aligned}$$

Finally, for a program, p, consisting of:

- a sequence of user-function definitions: `fun` $f_1(\ldots) = e_1$; $\ldots$; `fun` $f_n(\ldots) = e_n$ and
- an initial expression, e_0

$\mathcal{A}[\![p]\!]$ returns the conflict set resulting from program, p:

$$\mathcal{A}[\![p]\!] = \bigcup_{0\le k\le n} \mathcal{A}[\![e_k]\!]$$

(The letter $\mathcal{A}$ is used since $\mathcal{A}[\![p]\!]$ represents the calls which require arbiters.)

5 Transformation of SAFL specifications

Resource-awareness makes source-level program transformation of SAFL specifications a powerful technique. A designer can explore a wide range of hardware implementations by repeatedly transforming an initial specification in ways which preserve semantics but vary implementation.

We have investigated a number of transformations which correspond to concepts in hardware design. Some examples of these transformations are presented below.

5.1 Resource duplication vs sharing

The SAFL code fragment in Section 3.1 specifies a single `add` unit shared between the `ALU` and `calculate_new_PC` circuits. Transforming this into a design which specifies two separate adders is trivial:

```
fun add_1(x,y) = x+y
fun add_2(x,y) = x+y
...
fun ALU(op, arg1, arg2, ...) =
     if op=1 then add_1(arg1,arg2)
             else ...
...
fun calculate_new_PC(current_PC,offset,condition) =
     if condition then add_2(current_PC,offset)
                  else current_PC
```

Hence we observe that straightforward (and obviously correct) SAFL transformations provide fine-grained control over resource sharing/duplication. The reason for the simplicity of such transformations is that the task of generating inter-resource glue-logic (such as arbiters and temporary registers) is the duty of the FLaSH compiler rather than the programmer. Investigating similar tradeoffs in a conventional HDL (such as Verilog) would require time-consuming and error-prone modifications throughout the code—arbiters and permanising registers would have to be placed by hand.

5.2 Static vs dynamic scheduling

Call-by-value evaluation and referential transparency allows the FLaSH compiler to produce circuits where all function call arguments are evaluated in parallel. This means that for the following SAFL expression:

```
g(f(4), f(5))
```

`f(4)` and `f(5)` (the arguments of the call to `g`) will be evaluated concurrently[5]. PCA (Section 4) determines that an arbiter is to be synthesised to enforce mutual exclusion to $H_{\texttt{f}}$, thus dynamically scheduling access to the shared resource.

Now consider transforming the above expression into:

```
let x = f(4)
 in g(x, f(5))
end
```

In this case an arbiter is not required since we have specified a static schedule of access to shared resource $H_{\texttt{f}}$: `f(4)` is evaluated strictly before `f(5)`.

More generally, by using `let` as a sequentialisation operator we see that SAFL is expressive enough to represent: (*i*) static scheduling (order of access to a shared resource specified within the source code) and (*ii*) dynamic scheduling (when no order of access is specified directly an arbiter is generated to perform dynamic scheduling).

[5] This simple example is chosen for expository purposes only; in reality, for this trivial case, the FLaSH compiler automatically performs the above transformation to remove the need for an arbiter. However, in more complex cases (e.g. where the temporal ordering is data-dependent) it is beneficial to synthesise an arbiter.

5.3 Time vs area

We define the *area* of a SAFL program to be the total space required for its execution. Due to static allocation it is easy to show that area is $O(\textit{length of program})$. Similarly, we can talk about execution *time*.

In this framework, Burstall and Darlington's fold/unfold transformations [2] applied at the SAFL-level reflect area-time tradeoffs at the hardware level. As an example of this, consider applying the *unfold* rule to the recursive call in the definition of `mult` (from Section 1) to yield:

```
fun mult2(x, y, acc) =
  if (x=0 | y=0) then acc
  else let (x',y',acc') = (x<<1, y>>1,
                            if y.bit0 then acc+x else acc) in
    if (x'=0 | y'=0) then acc'
    else mult2(x'<<1, y'>>1, if y'.bit0 then acc'+x' else acc')
```

Folding/unfolding recursive function calls before compiling to synchronous hardware corresponds to trading the amount of work done per clock cycle against clock speed. The FLaSH compiler translates `mult2` into a design which requires half as many clock cycles as `mult`, but more gates.

5.4 Hardware vs software

We have demonstrated [10] how source-level transformation of SAFL allows one to partition a design into hardware and software. As part of the transformation process, a parameterised processor is automatically generated to execute the software instructions.

Although the idea of using source-level transformation to formalise hardware/software co-design is not new [12], we argue that the extra transformational power of functional languages allows more freedom—in particular resource-awareness allows one to generate designs consisting of multiple processors capable of operating concurrently. The details of this transformation are too long for this paper; they are described fully in [10].

6 Future directions and further work

By designing a working compiler and prototype development environment we have demonstrated that resource-aware silicon compilation of declarative specifications is a promising technique. However, more research is needed if the FLaSH system is to be used for large scale industrial applications:

– We are currently investigating a type-system for SAFL with provision for user defined recursive types and statically expanded recursion.
– What are the limits of pure functional languages for hardware synthesis? We have briefly experimented with side-effecting function calls (e.g. `memory_write(address,data)`). We are designing a transaction-based system to manage access to external devices with state in a controlled way.
– We plan to experiment with other compilation strategies. In particular, we intend to produce a back-end for the FLaSH compiler which outputs asynchronous logic.

7 Acknowledgement

This work is part of a collaborative project involving Cambridge University Computer Laboratory (EPSRC grant GR/L86326), AT&T Laboratories Cambridge and Cambridge Consultants Ltd.

References

1. Berry, G. Real-time programming: General purpose or special-purpose languages. In G. Ritter (ed.), Information Processing 1989, pages 11-17. Elsevier Science Publishers B.V. (North Holland), 1989.
2. Burstall, R.M. and Darlington, J. A Transformation System for Developing Recursive Programs, JACM 24(1), 1979.
3. Greaves, D.J. An Approach Based on Decidability to Compiling C and Similar, Block-Structured, Imperative Languages for Hardware Software Codesign, Unpublished memo, 1999.
4. Halbwachs, N., Caspi, P., Raymond, P. and Pilaud D. The Synchronous Dataflow Programming Language LUSTRE. Proc. IEEE, vol. 79(9). September 1991.
5. Hennessy, J.L. and Patterson, D.A. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 1990.
6. The LavaLogic "Forge" software-to-hardware compiler. See `www.lavalogic.com`
7. Milner, R., Tofte, M., Harper, R. and MacQueen, D. The Definition of Standard ML (Revised). MIT Press, 1997.
8. Morison, J.D. and Clarke, A.S. ELLA 2000: A Language for Electronic System Design. Cambridge University Press 1994.
9. Mycroft, A. and Sharp, R.W. A Statically Allocated Parallel Functional Language. To appear: Proc. ICALP 2000, Springer-Verlag LNCS, July 2000.
10. Mycroft, A. and Sharp, R.W. Hardware/Software Co-Design using a Functional Language. Submitted for publication.
11. Page, I. and Luk, W. Compiling Occam into Field-Programmable Gate Arrays. In Moore and Luk (eds.) FPGAs, pages 271-283. Abingdon EE&CS Books, 1991.
12. Page, I. Parameterised Processor Generation. Presented at the September'93 International Workshop on FPGAs at Oxford.
13. Sharp, R.W. and Mycroft, A. The FLaSH Compiler: Efficient Circuits from Functional Specifications. Technical Report tr.2000.3, AT&T Laboratories Cambridge. Available from: `www.uk.research.att.com`
14. Sheeran, M. muFP, a Language for VLSI Design. Proc. ACM Symp. on LISP and Functional Programming, 1984.
15. Bjesse, P., Claessen, K., Sheeran, M. and Singh, S. Lava: Hardware Description in Haskell. Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming, 1998.