A Statically Allocated Parallel Functional Language

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. We describe SAFL, a call-by-value first-order functional language which is syntactically restricted so that storage may be statically allocated to fixed locations. Evaluation of independent sub-expressions happens in parallel—we use locking techniques to protect shared-use function definitions (i.e. to prevent unrestricted parallel accesses to their storage locations for argument and return values). SAFL programs have a well defined notion of total (program and data) size which we refer to as 'area'; similarly we can talk about execution 'time'. Fold/unfold transformations on SAFL provide mappings between different points on the area-time spectrum. The space of functions expressible in SAFL is incomparable with the space of primitive recursive functions, in particular interpreters are expressible. The motivation behind SAFL is hardware description and synthesis—we have built an optimising compiler for translating SAFL to silicon.

1 Introduction

This paper addresses the idea of a functional language, SAFL, which

- can be statically allocated—all variables are allocated to fixed storage locations at compile time—there is no stack or heap; and
- has independent sub-expressions evaluated concurrently.

While this concept might seem rather odd in terms of the capabilities of modern processor instruction sets, our view is that it neatly abstracts the primitives available to a hardware designer. Our desire for static allocation is motivated by the observation that dynamically-allocated storage does not map well onto silicon: an addressable global store leads to a von Neumann bottleneck which inhibits the natural parallelism of a circuit. SAFL has a call-by-value semantics since strict evaluation naturally facilitates parallel execution which is well suited to hardware implementation.

To emphasise the hardware connection we define the *area* of a SAFL program to be the total space required for its execution. Due to static allocation we see that *area* is $O(\textit{length of program})$; similarly we can talk about execution *time*. Fold/unfold transformations [1] at the SAFL level correspond directly to area-time tradeoffs at the hardware level.

In this paper we are concerned with the properties of the SAFL language itself rather than the details of its translation to hardware. In the light of this and for the sake of clarity, we present an implementation of SAFL by means of a translation to an abstract machine code which we claim mirrors the primitives available in hardware. The design of an optimising compiler which translates SAFL into hardware is presented in a companion paper [11]. A more practical use of SAFL for hardware/software co-design is given in [9].

The body of this paper is structured as follows. Section 2 describes the SAFL language and Section 3 describes an implementation on a parallel abstract machine. In Sections 4 and 5 we argue that SAFL is well suited for hardware description and synthesis. Section 6 shows how fold/unfold transformations can represent SAFL area-time tradeoffs. Finally, Sections 7 and 8 discuss more theoretical issues: how SAFL relates to Primitive Recursive functions and problems concerning higher-order extensions. Section 9 concludes and outlines some future directions.

Comparison with Other Work

The motivation for static allocation is not new. Gomard and Sestoft [2] describe *globalization* which detects when stack or heap allocation of function parameters can be implemented more efficiently with global variables. However, whereas globalization is an optimisation which may in some circumstances improve performance, in our work static allocation is a fundamental property of SAFL enforced by the syntactic restrictions described in Section 2.

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 muFP [12] and Lava [13] 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. We believe our association of function definitions with hardware resources (see Section 4) to be novel.

Various authors have described silicon compilers (e.g. for C [4] and Occam [10]). Although rather beyond the scope of this paper, we argue that the flexibility of functional languages provides much more scope for analysis and optimisation.

Hofmann [6] describes a type system which allows space pre-allocated for argument data-structures to be re-used by in-place update. Boundedness there means that no new heap space is allocated although stack space may be unbounded. As such our notion of static allocatability is rather stronger.

2 Formalism

We use a first-order language of recursion equations (higher order features are briefly discussed in Section 8). Let c range over a set of constants, x over variables (occurring in `let` declarations or as formal parameters), a over primitive functions (such as addition) and f over user-defined functions. For typographical convenience we abbreviate formal parameter lists $(x_1, \ldots, x_k)$ and actual parameter lists $(e_1, \ldots, e_k)$ to $\vec{x}$ and $\vec{e}$ respectively; the same abbreviations are used in `let` definitions. SAFL has syntax of:

- terms e given by:

$$\begin{aligned} e ::= {} & c \mid x \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \mid \texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0 \mid \\ & a(e_1, \ldots, e_{arity(a)}) \mid f(e_1, \ldots, e_{arity(f)}) \end{aligned}$$

- programs p given by:

$$\begin{aligned} p ::= {} & \texttt{fun } f^{11}(\vec{x}) = e_{11} \texttt{ and} \ldots \texttt{and } f^{1r_1}(\vec{x}) = e_{1r_1} \\ & \ldots \\ & \texttt{fun } f^{n1}(\vec{x}) = e_{n1} \texttt{ and} \ldots \texttt{and } f^{nr_n}(\vec{x}) = e_{nr_n}. \end{aligned}$$

We refer to a phrase of the form

$$\texttt{fun } f^{i1}(\vec{x}) = e_{i1} \texttt{ and} \ldots \texttt{and } f^{ir_i}(\vec{x}) = e_{ir_i}$$

as a (mutually recursive) function *group*. The notation f^{ij} just means the jth function of group i. Programs have a distinguished function `main` (normally f^{nr_n}) which represents an external world interface—at the hardware level it accepts values on an input port and may later produce a value on an output port.

To simplify semantic descriptions we will further assume that all function and variable names are distinct; this is particularly useful for static allocation since we can use the name of the variable for the storage location to which it is allocated.

We impose additional stratification restrictions[1] on the e_{ij} occurring as bodies of the f^{ij}; arbitrary calls to previous definitions are allowed, but recursion (possibly mutual) is restricted to tail recursion to enforce static allocatability. This is formalised as a well-formedness check. Define the *tailcall contexts*, $\mathcal{TC}$ by

$$\begin{aligned} \mathcal{TC} ::= {} & [\,] \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } \mathcal{TC} \mid \texttt{if } e_1 \texttt{ then } \mathcal{TC} \texttt{ else } e_3 \\ & \mid \texttt{let } \vec{x} = \vec{e} \texttt{ in } \mathcal{TC} \end{aligned}$$

[1] Compare this with *stratified negation* in the deductive database world.

The well-formedness condition is then that, for every user-function application $f^{ij}(\vec{e})$ within function definition $f^{gk}(\vec{x}) = e_{gk}$ in group g, we have that:

$$i < g \vee (i = g \wedge \exists (C \in \mathcal{TC}).e_{gk} = C[f^{ij}(\vec{e})])$$

The first part ($i < g$) is merely static scoping (definitions are in scope if previously declared) while the second part says that a call to a function in the same group (i) as its definition (g) is only valid if the call is actually in tailcall context.

3 Implementing SAFL

We give a translation $[\![\cdot]\!]$ of SAFL programs into an abstract machine code which mirrors primitives available in hardware (its correctness relies on SAFL restrictions). Each function definition corresponds to one block of code. In order to have temporaries available we will assume that each expression and sub-expression is labelled by a unique label number (or 'occurrence') from which a storage location can be generated. Label names are assumed to be distinct from variables so we can use the notation M_x and M_ℓ to mean the storage location associated with variable x or label ℓ respectively. We use the notation $\ell : e$ to indicate expression e has label ℓ. The expression

```
   if x=1 then y else f(x,y-1)
```

might then be more fully written as (temporarily using the notation e^ℓ instead of $\ell : e$ used elsewhere):

$$\texttt{(if (x}^{\ell 21}\texttt{=1}^{\ell 22}\texttt{)}^{\ell 2}\texttt{ then y}^{\ell 3}\texttt{ else (f(x}^{\ell 41}\texttt{,(y}^{\ell 421}\texttt{-1}^{\ell 422}\texttt{)}^{\ell 42}\texttt{)}^{\ell 4}\texttt{)}^{\ell 1}$$

We write $f\texttt{.formals}$ to stand for $(M_{x_1}, \ldots, M_{x_k})$ where $\vec{x}$ is the tuple of formal parameters of f (which are already assumed to be globally distinct from all other variable names). Similarly, we will assume all functions f^{ij} in group i leave their result in the storage location $M_{f^{i*}\texttt{.result}}$—this this is necessary to ensure tailcalls to other functions in group i behave as intended.[2] (The notation $i*$ in general refers to a common resource shared by the members of function group i.)

In addition to the storage location as above, we need two other forms of storage: for each function group i we have a location L_{i*} to store the return link (accessed by *JSR* and *RET*); and a semaphore S_{i*} to protect its (statically allocated) arguments, temporaries and the like from calls in competing *PAR* threads by enforcing mutual exclusion.

The abstract instructions for the machine are as follows:

[2] A type system would require that the result type of all functions in a group are identical—because they return each others' values—so $M_{f^{i*}\texttt{.result}}$ has a well-defined size.

$m := m'$	Copy m' to m.
$(m_1, \ldots, m_k) := (m'_1, \ldots m'_k)$	Copy the m'_i to the m_i in any order.
$m := \mathit{PRIMOP}_a(m_1, \ldots m_k)$	Perform the operation corresponding to built-in primitive a.
$\mathit{LOCK}(S)$	Lock semaphore S.
$\mathit{UNLOCK}(S)$	Release semaphore S.
$\mathit{JMP}(ep)$	Branch to function entrypoint ep—used for tailcall.
$\mathit{JSR}(m, ep)$	Store some representation of the point of call in location m and branch to function entrypoint ep.
$\mathit{RET}(m)$	Branch to the instruction following the point of call specified by m.
$\mathit{COND}(m, seq_1, seq_2)$	If location m holds a 'true' value then execute opcode sequence seq_1 otherwise seq_2.
$\mathit{PAR}(seq_1, \ldots, seq_k)$	Execute opcode sequences $seq_1, \ldots, seq_k$ in parallel, waiting for all to complete before terminating.

Instructions are executed sequentially, except that *JSR*, *JMP* and *RET* alter the execution sequence. The *PAR* construct represents fork-join parallelism (each of the operand sequences are executed) and *COND* the usual conditional (one of the two operand sequences is executed).

Assuming e is a sub-expression of a function body of group g the compilation function $[\![e]\!]^g m$ gives an opcode sequence which evaluates e to storage location m (we omit g for readability in the following—it is only used to identify tailcalls):

$$
\begin{array}{l}
[\![c]\!]m = m := c \\
[\![x]\!]m = m := M_x \\
[\![\texttt{if } (\ell : e_1) \texttt{ then } e_2 \texttt{ else } e_3]\!]m = \\
\qquad [\![e_1]\!]M_\ell; \\
\qquad \mathit{COND}(M_\ell, [\![e_2]\!]m, [\![e_3]\!]m) \\
[\![\texttt{let } (x_1, \ldots, x_k) = (e_1, \ldots, e_k) \texttt{ in } e_0]\!]m = \\
\qquad \mathit{PAR}([\![e_1]\!]M_{x_1}, \ldots, [\![e_k]\!]M_{x_k}); \\
\qquad [\![e_0]\!]m \\
[\![a(\ell_1 : e_1, \ldots, \ell_k : e_k)]\!]m = \\
\qquad \mathit{PAR}([\![e_1]\!]M_{\ell_1}, \ldots, [\![e_k]\!]M_{\ell_k}); \\
\qquad m := \mathit{PRIMOP}_a(M_{\ell_1}, \ldots, M_{\ell_k}) \\
[\![f^{ij}(\ell_1 : e_1, \ldots, \ell_k : e_k)]\!]m = \\
\qquad \left\{ \begin{array}{l}
\mathit{PAR}([\![e_1]\!]M_{\ell_1}, \ldots, [\![e_k]\!]M_{\ell_k}); \\
\mathit{LOCK}(S_{i*}); \\
M_{f^{ij}.\texttt{formals}} := (M_{\ell_1}, \ldots, M_{\ell_k}); \\
\mathit{JSR}(L_{i*}, \mathit{EntryPt}_{ij}); \\
m := M_{f^{i*}.\texttt{result}}; \\
\mathit{UNLOCK}(S_{i*})
\end{array} \right\} \text{ if } i < g \\
[\![f^{ij}(\ell_1 : e_1, \ldots, \ell_k : e_k)]\!]m = \\
\qquad \left\{ \begin{array}{l}
\mathit{PAR}([\![e_1]\!]M_{\ell_1}, \ldots, [\![e_k]\!]M_{\ell_k}); \\
M_{f^{ij}.\texttt{formals}} := (M_{\ell_1}, \ldots, M_{\ell_k}); \\
\mathit{JMP}(\mathit{EntryPt}_{ij})
\end{array} \right\} \text{ if } i = g
\end{array}
$$

Finally we need to compile each function definition to a sequence of instructions labelled with the function name:

$$[\![\texttt{fun } f^{ij}(x_1,\ldots,x_k) = e_{ij}]\!] = \left\{ \begin{array}{l} \mathit{EntryPt}_{ij} : \\ \quad [\![e_{ij}]\!]^i M_{f^{i*}.\texttt{result}}; \\ \quad \mathit{RET}(L_{i*}) \end{array} \right\}$$

The above translation is naïve (often semaphores and temporary storage locations are used unnecessarily) but explains various aspects; an optimising compiler for hardware purposes has been written [11].

Proposition 1. *The translation* $[\![e]\!]$ *correctly implements SAFL programs in that executing the abstract machine code coincides with standard eager evaluation of* e.

Note that the translation would fail if we use one semaphore-per-function instead of one-per-group. Consider the program

```
fun f(x) = if x=0 then 1 else g(x-1)
and g(x) = if x=0 then 2 else f(x-1);
fun h(x,y) = x+y;
fun main() = h(f(8),g(9));
```

where there is then the risk that the *PAR* construct for the actual arguments to `h` will simultaneously take locks on the semaphores for `f` and `g` resulting in deadlock.

4 Hardware Synthesis using SAFL

As part of the FLaSH project (Functional Languages for Synthesising Hardware) [11], we have implemented an optimising silicon compiler which translates SAFL specifications into structural Verilog. We have found that SAFL is able to express a wide range of hardware designs; our tools have been used to build a small commercial processor.[3]

The static allocation properties of SAFL allow our compiler to enforce a direct mapping between a function definition:

$$f(\vec{x}) = e$$

and a hardware block, H_f, with output port, P_f, consisting of:

– a fixed amount of storage (registers holding values of the arguments $\vec{x}$) and
– a circuit to compute e to P_f.

[3] We implemented the instruction set of the Cambridge Consultants XAP processor: `http://www.camcon.co.uk`; we did not support the SIF instruction.

Hence, multiple calls to a function f at the source level corresponds directly to sharing the resource H_f at the hardware level. As the FLaSH compiler synthesises multi-threaded hardware, we have to be careful to ensure that multiple accesses to a shared hardware resource will not occur simultaneously. We say that resource, H_f, is subject to a *sharing conflict* if multiple accesses may occur concurrently. Sharing conflicts are dealt with by inserting arbiters (cf. semaphores in our software translation). Program analysis is used to detect potential sharing conflicts—arbiters are only synthesised where necessary.

Our practical experience of using the FLaSH system to design and build real hardware has brought to light many interesting techniques that conventional hardware description languages cannot exploit. These are outlined below.

4.1 Automatic Generation of Parallel Hardware

Hammond [5] observes:

> "It is almost embarrassingly easy to partition a program written in a strict [functional] language [into parallel threads]. Unfortunately, the partition that results often yields a large number of very fine-grained tasks."

He uses the word *unfortunately* because his discussion takes place in the context of software, where fairly course-grained parallelism is required to ensure the overhead of `fork/join` does not outweigh the benefits of parallel evaluation.

In contrast, we consider the existence of "a large number of very fine-grained tasks" to be a very *fortunate* occurrence: in a silicon implementation, very fine-grained parallelism is provided with virtually no overhead! The FLaSH compiler produces hardware where all function arguments and `let`-declarations are evaluated in parallel.

4.2 Source-Level Program Transformation

We have found that source-level program transformation of SAFL specifications is a powerful technique. A designer can explore a wide range of hardware implementations by repeatedly transforming an initial specification.

We have investigated a number of transformations which correspond to concepts in hardware design. Due to space constraints we can only list the transformations here:

Resource Sharing vs Duplication: Since a single user-defined function corresponds to a single hardware block SAFL provides fine-grained control over resource sharing/duplication.

Static vs Dynamic Scheduling: By default, function arguments are evaluated in parallel. Thus compiling `f(4)+f(5)` will generate an arbiter to sequentialise access to the shared resource $H_{\mathtt{f}}$. Alternatively we can use a `let`-declaration to specify an ordering statically. The circuit corresponding to `let x=f(4) in x+f(5)` does not require dynamic arbitration; we have specified a static order of access to $H_{\mathtt{f}}$.

Area-Time Tradeoffs: We observe that fold/unfold transformations correspond directly to area-time tradeoffs at the hardware level. This can be seen as a generalisation of resource sharing/duplication (see Section 6).

Hardware-Software Partitioning: We have demonstrated [9] a source-source transformation which allows us to represent hardware/software partitioning within SAFL.

At the SAFL level it is relatively straightforward to apply these transformations. Investigating the same tradeoffs entirely within RTL Verilog would require time-consuming and error-prone modifications throughout the code.

4.3 Static Analysis and Optimisation

Declarative languages are more susceptible to analysis and transformation than imperative languages. In order to generate efficient hardware, the FLaSH compiler performs the following high-level analysis techniques (documented in [11]):

Parallel Conflict Analysis is performed at the abstract syntax level, returning a set of function calls which require arbitration at the hardware level.

Register Placement is the process of inserting temporary storage registers into a circuit. The FLaSH compiler translates specifications into intermediate code (based on control/data flow graphs) and performs data-flow analysis at this level in order to place registers. (This optimisation is analogous to minimising the profligate use of M_ℓ temporaries seen in the software translation $[\![\cdot]\!]$.)

Timing Analysis (with respect to a particular implementation strategy) is performed through an abstract interpretation where functions and operators return the times taken to compute their results.

4.4 Implementation Independence

The high level of specification that SAFL provides means that our hardware descriptions are implementation independent. Although the current FLaSH compiler synthesises hardware in a particular style,[4] there is the potential to develop a variety of back-ends, targeting a wide range of hardware implementations.

In particular, we believe that SAFL would lend itself to asynchronous circuit design as the compositional properties of functions map directly onto the compositional properties of asynchronous hardware modules. We plan to design an asynchronous back-end for FLaSH in order to compare synchronous and asynchronous implementations.

5 A Hardware Example

In order to provide a concrete example of the benefits of designing hardware in SAFL consider the following specification of a shift-add multiplier:

[4] The generated hardware is synchronous with bundled data and ready signals.

```
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)
```

From this specification, the FLaSH compiler generates a hardware resource, $H_{\mathtt{mult}}$, with three data-inputs: (x, y and acc), a data-output (for returning the function result), a control-input to trigger computation and a control-output which signals completion. The multiplier circuit contains some control logic, two 1-place shifters, an adder and three registers which are used to latch data-inputs. We trigger $H_{\mathtt{mult}}$ by placing argument values on the data-inputs and signalling an event on the control-input. The result can be read from the data-output when a completion event is signalled on the control-output.

These 3 lines of SAFL produce over 150 lines of RTL Verilog. Synthesising a 16-bit version of mult, using Mentor Graphics' *Leonardo* tool, yields 1146 2-input equivalent gates.[5] Implementing the same algorithm directly in RTL Verilog took longer to write and yielded an almost identical gate count.

6 Fold/Unfold for Area-Time Tradeoff

In section 4.2 we observed that the fold/unfold transformation [1] can be used to trade area for time. As an example of this consider:

```
fun f x = ...
fun main(x,y) = g(f(x),f(y))
```

The two calls to f are serialised by mutual exclusion before g is called. Now use fold/unfold to duplicate f as f', replacing the second call to f with one to f'. This can be done using an unfold, a definition rule and a fold yielding

```
fun f  x = ...
fun f' x = ...
fun main(x,y) = g(f(x),f'(y))
```

The second program has more area than the original (by the size of f) but runs more quickly because the calls to f(x) and f'(y) execute in parallel.

Although the example given above is trivial, we find fold/unfold to be a useful technique in choosing a hardware implementation of a given specification. Note that fold/unfold allows us to do more than resource/duplication sharing tradeoffs. For example, folding/unfolding recursive function calls before compiling to synchronous hardware corresponds to trading the amount of work done per clock cycle against clock speed—mult can be mechanically transformed into:

```
fun mult(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
```

[5] This figure includes the gates required for the three argument registers.

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

which takes half as many clock cycles.

7 Theoretical Expressibility

Here we consider the expressibility of programs in SAFL. Clearly in one sense, each such program represents a finite state machine as it has a bounded number of states and memory locations, and therefore is very inexpressive (but this argument catches the essence of the problem no more than observing that a personal computer is also a finite state automaton).

Consider the relation to Primitive Recursive (PR) functions. In the PR definition scheme, suppose g and h are already shown to be primitive recursive functions (of k and $k+1$ arguments), then the definition of f

$$\begin{aligned} f(0, x_1, \ldots, x_k) &= g(x_1, \ldots, x_k) \\ f(n+1, x_1, \ldots, x_k) &= h(f(n, x_1, \ldots, x_k), x_1, \ldots, x_k) \end{aligned}$$

is also primitive recursive of $k+1$ arguments.[6] We see that the SAFL restrictions require h to be the identity projection on its first argument, but that our definitional scheme allows recursion more general than descent through a well-founded order ($n+1$ via n eventually to 0 in the integer form above). In particular SAFL functions may be partial.

Thus we conclude that, in practice, our statically allocatable functions represent an incomparable subset of general recursion than that subset specified by primitive recursion. Note that we cannot use translation to continuation-passing form where all calls are tailcalls because SAFL is first order (but see the next section). Jones [7] shows the subtle variance of expressive power on recursion forms, assignability and higher-order types.

As a slightly implausible aside, suppose we consider statically allocatable functional languages where values can range over any natural number. In this case the divergence from primitive recursion becomes even clearer—even if we have an assertion that the statically allocated functional program is total then we cannot in general transform it into primitive recursive form. To see this observe that we can code a register machine interpreter as such a statically allocated program with register machine program being Ackermann's function.

8 Higher Order Extensions to SAFL

Clearly a simple addition of higher-order functions to SAFL would break static allocatability by allowing recursion other than in the program structure. Consider for example the traditional

[6] In practice we widen this definition to allow additional intensional forms without affecting the space of functions definable.

```
let g(n,h) = if n=0 then 1 else n * h(n-1,h)
let f(n) = g(n,g)
```

trick to encode the factorial function in a form which requires $O(n)$ space.[7]

A simple extension is to allow functions to be passed as values, and function valued expressions to be invoked. (No closures can be constructed because of the recursion equation syntax). One implementation of this is as follows: use a control-flow analysis such as 0-CFA to identify possible targets of each *indirect call* (i.e. call to a function-valued expression). This can be done in cubic time. We then fault any indirect calls which are incompatible with the function definition hierarchy in the original program—i.e. those with a function f^{ij} containing a possible indirect call to f^{gk} where $g \geq i$ (unless the indirect call is in tailcall context and $g = i$). Information from 0-CFA also permits a source-to-source transformation to first-order using a case branch over the possible functions callable at that point: we map the call $e'(\vec{e})$ where e' can evaluate to `g`, `h` or `i` into:

```
if e'=g then g(ē) else if e'=h then h(ē) else i(ē)
```

The problem with adding nested function definitions (or λ-expressions) is that it is problematic to statically allocate storage for the resulting closures. Even programs which use only tail-recursion and might at first sight appear harmless, such as

```
fun r(x) = ⟨some function depending on x⟩
fun f(x,g) = if x=0 then g(0)
                    else f(x-1, r(x) ∘ g)
```

require unlimited store. Similarly, the translation to CPS (continuation passing style) transforms any program into an equivalent one using only tailcalls, but at the cost of increasing it to higher-order—again see [7] for more details.

One restriction which allows function closures is the Algol solution: functions can be passed as parameters but not returned as results (or at least not beyond the scope any of their free variables). It is well known that such functions can be stack implemented, which in the SAFL world means their storage is bounded. Of course we still need the 0-CFA check as detailed above.

9 Conclusions and Further Work

This paper introduces the idea of statically allocated functional languages which are interesting in themselves as well as being apposite and powerful for expressing hardware designs. However there remains much to be done to explore their uses as hardware synthesis languages, e.g. optimising hardware compilation, type systems, synchronous versus asynchronous translations, etc.

[7] Of course one could use an accumulator argument to implement this in $O(1)$ space, but we want the statically allocatability rules to be intensional.

Currently programs support a 'start and wait for result' interface. We realise that in real hardware systems we need to interact with other devices having internal state. We are considering transactional models for such interfaces including the use of channels. Forms of functional language input/output explored in Gordon's thesis [3] may be also be useful.

Acknowledgments

We are indebted to Neil Jones, Martin Hofmann, Simon Peyton-Jones and the anonymous referees for comments which have improved this paper.

References

1. Burstall, R.M. and Darlington, J. A Transformation System for Developing Recursive Programs, JACM 24(1), 1977.
2. Gomard, C.K. and Sestoft, P. Globalization and Live Variables. Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 166-177. ACM Press. 1991. SIGPLAN NOTICES, vol. 26, number 9.
3. Gordon, A.D., Functional Programming and Input/Output. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.
4. Greaves, D.J. An Approach Based on Decidability to Compiling C and Similar, Block-Structured, Imperative Languages for Hardware Software Codesign. Unpublished memo, 1999.
5. Hammond, K. Parallel Functional Programming: An Introduction, International Symposium on Parallel Symbolic Computation, World Scientific, 1994.
6. Hofmann, M. A Type System for Bounded Space and Functional In-Place Update. Proc. ESOP'2000 Berlin, Springer-Verlag LNCS, 2000.
7. Jones, N.D. The Expressive Power of Higer-order Types or, Life without CONS. J. Functional Programming, to appear.
8. Milner, R., Tofte, M., Harper, R. and MacQueen, D. The Definition of Standard ML (Revised). MIT Press, 1997.
9. Mycroft, A. and Sharp, R.W. Hardware/Software Co-Design using Functional Languages. Submitted for publication.
10. 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.
11. 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`
12. Sheeran, M. muFP, a Language for VLSI Design. Proc. ACM Symp. on LISP and Functional Programming, 1984.
13. 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.