Geometry of Synthesis IV

Compiling Affine Recursion into Static Hardware

Dan R. Ghica
University of Birmingham
d.r.ghica@cs.bham.ac.uk

Alex Smith
University of Birmingham
ais523@cs.bham.ac.uk

Satnam Singh
Microsoft Research
satnams@microsoft.com

Abstract

Abramsky's Geometry of Interaction interpretation (GoI) is a logical-directed way to reconcile the process and functional views of computation, and can lead to a dataflow-style semantics of programming languages that is both operational (i.e. effective) and denotational (i.e. inductive on the language syntax). The key idea of Ghica's Geometry of Synthesis (GoS) approach is that for certain programming languages (namely Reynolds's affine Syntactic Control of Interference–SCI) the GoI processes-like interpretation of the language can be given a finitary representation, for both internal state and tokens. A physical realisation of this representation becomes a semantics-directed compiler for SCI into hardware. In this paper we examine the issue of compiling affine recursive programs into hardware using the GoS method. We give syntax and compilation techniques for unfolding recursive computation in space or in time and we illustrate it with simple benchmark-style examples. We examine the performance of the benchmarks against conventional CPU-based execution models.

Categories and Subject Descriptors B.5.2 [*Hardware*]: Design Aids—Automatic Synthesis

General Terms Languages, Theory, Design

1. Introduction

Why compile recursive functions into FPGA hardware? A few years ago the case for compiling recursive functions into FPGA circuits was less compelling because these chips had limited computational resources and no specialized on-chip memory, so a stack would have to be made using precious flip-flops. Modern FPGAs can implement million-gate circuits and can contain thousands of independent dual-port memory blocks (e.g. block-RAMs on Xilinx FPGAs which are around 36 Kbits in size) which are ideal for implementing stacks and other local computational state. Since there are many independent memories it is possible to have many circuit sub-components executing in parallel without one single stack acting as a memory bottle-neck.

So now that we have effective elements for implementing recursive computations on FPGAs are there any applications that actually need recursive descriptions? Certainly. As FPGAs have grown in capacity there has been an increasing desire to map computations that were previously implemented in software for execution on a regular processor onto FPGAs instead, in order to improve performance or reduce energy consumption. Recently FPGAs have gained the capability to access large amounts of external memory using multiple memory controllers and this allows us to manipulate dynamic data-structures like trees with FPGA gates and exploit the thousands of on-chip independent memories as caches. With dynamic data-structures one naturally has the desire to express computations with recursive algorithms (e.g. balancing trees and inserting and removing nodes). Example applications include line-speed network packet processing which may require building up auxiliary dynamic data structures (e.g. trees) which are used to help classify and route packets, such as Netezza's data warehousing project[1] or IBM's DataPower accelerated XML parser[2].

Recursive procedures, functions and methods have been used to describe the behaviour of digital circuits in a variety of programming languages ranging from mainstream hardware description languages like VHDL to experimental embedded domain specific languages like Lava [BCSS98] for many years. Most systems typically implement recursion by inlining it at compile time to produce descriptions that are unfolded in space. These descriptions assume the depth of the recursion to be determinable at compile time. A specific example of such a recursive function is one that adds the elements of an array by using a recursive divide and conquer strategy yielding an adder-tree implementation. At compile time the size of the array has to be known but not the value of the array elements.

Our goal is to allow recursive descriptions of circuits which unfold in time and space where the depth of recursion is not known at compile time. This in turn allows us to describe recursive functions that can operate over dynamic data structures. We aim to research and develop techniques for translating a sub-set of recursive functions into circuits where the depth of recursion is dependent on the dynamic values.

Domain specific hardware description langauges sometimes offer limited support for recursion. For example, Contessa [TL07] is used for finance applications but only supports tail recursion. There has also been considerable interest in describing hardware using *embedded* domain specific langauges i.e. by designing a library that captures important aspects of circuit design. For example, the JHDL system [BH98] provided a system coded in Java for performing structural hardware design. The Lava system provides a combinator library that makes it easier to descibe circuit layout and behaviour [HD08]. All of these systems allow recursive descriptions but they all require the descriptions to be in-lined. Even Bluespec [Nik04] which is a modern hardware description language based on functional programming concepts allows the specification

ICFP'11 September 19–21, 2011, Tokyo, Japan.

[1] `http://www.netezza.com/data-warehouse-appliance-products/`

[2] `http://www-01.ibm.com/software/integration/datapower/`

of recursive functions but does not support the synthesis of dynamic recursion.

Sklyarov describes a mechanism for compiling recursive C++ descriptions into FPGA circuits [Skl04] using hierarchical finite state machines which have been applied to the design of Huffman encoder circuits and sorting networks. Sklyarov's technique has the advantage of dealing with general recursion but it has the drawback of requring centralized control and it produces rather slow circuits running at only 25 MHz. Our approach is more modular, does not require centralized control and produces faster circuits. The knapsack and Kinght's Tour problems were described as recursive hardware algorithms by Maruyama, Takagi and Hoshino [MTH99] although this technique is limited to loop unrolling and pipeling and the use of a stack to hold state information.

Ferizis and Gindy show how recursive functions can be implemented by using dynamic reconfiguration [FG06] to unroll recursive function calls at run-time which avoids the need for a stack. However, this approach is very limited because it replicates the body of the recursive function, which is likely to quickly exhuast the available FPGA resources, and it also relies on immature dynamic reconfiguration technology.

Contribution. This paper gives the most general method to date for compiling recursive programs into static hardware. The main language-level restriction we impose, the use of an affine type system, is unavoidable because of computational reasons—non-affine functions do not have a finite state model. We describe a compiler to sequential circuits and we illustrate it with a typical benchmark for recursive programs, computing Fibonacci numbers.

2. Geometry of Synthesis

2.1 Objectives, methodology, motivation

Higher-level synthesis (HLS) is the production of gate-level description of circuits from behavioural descriptions given in higher-level programming languages.

Current HLS tools and techniques have significant limitations. One specific limitation is the weak treatment of function calls, which are typically implemented by inlining. This restriction is limiting the more general application of high level synthesis technology to a wider class of algorithms than benefit from dynamic procedure calls and recursion.

By inlining function calls at source-code level, which in hardware corresponds to replication of circuitry, current tools generate code which is often inefficient. Also, ordinary programming languages interface via function calls with libraries and with run-time services. Lack of support for proper function calls makes such interoperability impossible, which means that the entire system must be developed in one single programming language. Separate compilation, foreign function interfaces and application binary interfaces, standard facilities in modern compilers, cannot be supported without a proper function mechanism.

The Geometry of Synthesis (GoS) approach [Ghi07, GS10, GS11] enables full and proper implementation of ordinary programming languages, with an emphasis on correct and effective handling of function calls. Through static analysis the compiler can decide whether the circuit for a particular function must be re-instantiated or can be reused, depending on the desired trade-offs between area/power and latency/throughput. More importantly, code can be compiled without requiring all definitions of all functions to be available at compile time. Appropriate hardware interfaces are generated for the missing functions so that a complete functional circuit can be synthesised later, in a linking stage. This means that pre-compiled libraries can be created, distributed and commercialised with better protection for intellectual property. Finally, code written in one language can interact with code written in other languages so long as certain interface protocols are respected by the components. Moreover, these protocols can be enforced via the type system at the level of the programming language. Designs developed directly in conventional hardware description languages, subject to respecting the interface protocols, can also be used from the programming language via function calls. This allows the reuse of a vast existing portfolio of specialised, high-performance designs. Run-time services: certain circuits, which manage physical resources such as memory, network interface, video or audio interface, etc., cannot be meaningfully replicated as the resource itself cannot be replicated. Such services can also be used from ordinary programming languages via function calls.

Note the technology fully supports so-called higher-order functions, i.e. functions which operate on functions as argument or result. Such functions play an important role in the design of highly efficient parallel algorithms, such as Map-Reduce. These methodological considerations are discussed at length elsewhere [Ghi11].

2.2 Theoretical background

Abramsky's Geometry of Interaction (GoI) interpretation is a logical-directed way to reconcile the process and and functional views of computation, and can lead to a dataflow-style semantics of programming languages that is both operational (i.e. effective) and denotational (i.e. inductive on the language syntax) [AJ92]. These ideas have already been exploited in devising optimal compilation strategies for the lambda calculus, a technique called *Geometry of Implementation* [Mac94, Mac95].

The key idea of the *Geometry of Synthesis* (GoS) approach is that for certain programming languages, of which Reynolds's affine version of Idealized Algol (called *Syntactic Control of Interference* [Rey78, Rey89]) is a typical representative, the process-like interpretation given by the GoI can be represented in a finite way. This finitary representation applies both to the internal state of the token machine and the tokens themselves.

Subsequent developments in *game semantics* (see [Ghi09] for a survey) made it possible to give interpretations of a large variety of computational features (state, control, names, etc.) in an interactive way which is compatible with the GoI framework. Such computational features often have themselves a finite-state representation, as noted in [GM00].

A physical realisation of the finite-state representation of a GoI-style semantics then becomes a semantics-directed compiler for suitable languages, such as SCI, directly into hardware [Ghi07]. The hardware itself can be synchronous or asynchronous [GS10]. Subsequent work also showed how programs of more expressive type systems, such as full Idealized Algol, can be systematically represented into SCI as an intermediary language via type inference and *serialization* [GS11].

As a final observation, the somewhat surprising way in which higher order abstraction and application are compiled into circuits is best understood via the connection between GoI and monoidal categories, especially compact-closed categories [KL80], since such categories are an ideal formal setting for diagrammatic representation [Sel09].

3. Syntactic Control of Interference

Reynolds's Idealized Algol (IA) is a compact language which combines the fundamental features of imperative languages with a full higher-order procedural mechanism [Rey81]. This combination makes the language very expressive, for example allowing the encoding of classes and objects. Because of its expressiveness and elegance, IA has attracted a great deal of attention from theoreticians [OT81].

The typing system *(Basic) Syntactic Control of Interference (SCI)* is an affine version of IA in which contraction is disal-

lowed over function application and parallel execution. SCI was initially proposed by Reynolds as a programming language which would facilitate Hoare-style correctness reasoning because covert interference between terms is disallowed [Rey78, Rey89]. SCI turned out to be semantically interesting and it was studied extensively [Red96, OPTT99, McC07, McC10]. The restriction on contraction in SCI makes it particularly well suited for hardware compilation because any term in the language has a finite-state model and can therefore be compiled as a static circuit [Ghi07, GS10].

The primitive types of the language are commands, memory cells, and bounded-integer expressions $\sigma ::= \mathsf{com} \mid \mathsf{var} \mid \mathsf{exp}$. The type constructors are product and function: $\theta ::= \theta \times \theta \mid \theta \to \theta \mid \sigma$. Terms are described by typing judgements of the form $x_1 : \theta_1, \ldots, x_k : \theta_k \vdash M : \theta$, where we denote the list of identifier type assignments on the left by Γ. By convention, if we write Γ, Γ' it assumes that the two type assignments have disjoint sets of identifiers.

The term formation rules of the language are those of the affine lambda calculus:

$$\frac{}{x : \theta \vdash x : \theta}\ \text{Identity}$$

$$\frac{\Gamma \vdash M : \theta'}{x : \theta, \Gamma \vdash M : \theta'}\ \text{Weakening}$$

$$\frac{\Gamma, x : \theta, x' : \theta', \Gamma' \vdash M : \theta''}{\Gamma, x' : \theta', x : \theta, \Gamma' \vdash M : \theta''}\ \text{Commutativity}$$

$$\frac{\Gamma, x : \theta' \vdash M : \theta}{\Gamma \vdash \lambda x.M : \theta' \to \theta}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash M : \theta \to \theta' \qquad \Delta \vdash N : \theta}{\Gamma, \Delta \vdash MN : \theta'}\ \text{Application}$$

$$\frac{\Gamma \vdash M_i : \theta_i}{\Gamma \vdash \langle M_1, M_2 \rangle : \theta_1 \times \theta_2}\ \text{Product}$$

Importantly, contraction (identifier reuse) is allowed in product formation but not in function application.

The constants of the language are described below:

$\mathsf{n} : \mathsf{exp}$ are the integer constants;

$\mathsf{skip} : \mathsf{com}$ is the only command constant ("no-op");

$\mathsf{asg} : \mathsf{var} \times \mathsf{exp} \to \mathsf{com}$ is assignment to memory cell, denoted by ":=" when used in infix notation;

$\mathsf{der} : \mathsf{var} \to \mathsf{exp}$ is dereferencing of memory cell, also denoted by "!";

$\mathsf{seq} : \mathsf{com} \times \sigma \to \sigma$ is command sequencing, denoted by ";" when used in infix notation – if $\sigma \neq \mathsf{com}$ then the resulting expression is said to have side-effects;

$\mathsf{op} : \mathsf{exp} \times \mathsf{exp} \to \mathsf{exp}$ stands for arithmetic and logic operators;

$\mathsf{if} : \mathsf{exp} \times \mathsf{com} \times \mathsf{com} \to \mathsf{com}$ is branching;

$\mathsf{while} : \mathsf{exp} \times \mathsf{com} \to \mathsf{com}$ is iteration;

$\mathsf{newvar} : (\mathsf{var} \to \sigma) \to \sigma$ is local variable declaration in block command or block expression.

Local variable binding is presented with a quantifier-style type in order to avoid introducing new variable binders in the language. Local variable declaration can be sugared into a more familiar syntax as $\mathsf{newvar}(\lambda x.M) \equiv \mathsf{newvar}\ x\ \mathsf{in}\ M$.

We can reuse identifiers in products, so conventional imperative program terms such as x:=!x+1 or c;c can be written as $\mathsf{asg}\langle x, \mathsf{add}\langle \mathsf{der}(x), 1\rangle\rangle$ and $\mathsf{seq}\langle c, c\rangle$ respectively. One immediate consequence of the affine-type restriction is that nested application is no longer possible, i.e. terms such as $f : \mathsf{com} \to \mathsf{com} \vdash f(f(\mathsf{skip}))$ are illegal, so the usual operational unfolding of recursion no longer preserves typing. Therefore, the rec_θ operator in its most general formulation must be also eliminated. An appropriately adapted recursion operator will be presented in the next section.

Despite its restrictions SCI is still expressive enough to allow many interesting programs. Its finite state model makes it perfectly suited for hardware compilation [Ghi07, GS10]. The operational semantics of SCI is essentially the same as that of IA, which is standard. In this paper we are interested in compiling SCI to *sequential hardware*; the GoS method relies essentially on the existence of a game-semantic model for the programming language. The existing game [Wal04] and game-like [McC10] models of SCI are *asynchronous*, and using them to create *synchronous* hardware raises some technical challenges, which are addressed in previous work on *round abstraction* [GM10, GM11].

The compiler is defined inductively on the syntax of the language. Each type corresponds to a circuit interface, defined as a list of ports, each defined by data bit-width and a polarity. Every port has a default one-bit control component. For example we write an interface with n-bit input and n-bit output as $I = (+n, -n)$. More complex interfaces can be defined from simpler ones using concatenation $I_1 @ I_2$ and polarity reversal $I^- = \mathsf{map}\,(\lambda x.-x)I$. If a port has only control and no data we write it as $+0$ or -0, depending of polarity. Note that obviously $+0 \neq -0$ in this notation!

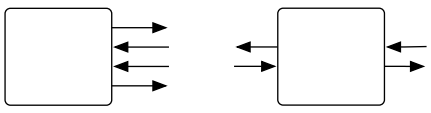
An interface for type θ is written as $[\![\theta]\!]$, defined as follows:

$$[\![\mathsf{com}]\!] = (+0, -0) \qquad [\![\mathsf{exp}]\!] = (+0, -n)$$
$$[\![\mathsf{var}]\!] = (+n, -0, +0, -n)$$
$$[\![\theta \times \theta']\!] = [\![\theta]\!] @ [\![\theta']\!] \qquad [\![\theta \to \theta']\!] = [\![\theta]\!]^- @ [\![\theta']\!].$$

The interface for com has two control ports, an input for starting execution and an output for reporting termination. The interface for exp has an input control for starting evaluation and data output for reporting the value. Variables var have data input for a write request and control output for acknowledgment, and control input for a read request along with data output for the value.

Tensor and arrow types are given interpretations which should be quite intuitive to the reader familiar with compact-closed categories [KL80]. The tensor is a disjoint sum of the ports on the two interfaces while the arrow is the tensor along with a polarity-reversal of the ports in the contravariant position. Reversal of polarities gives the dual object in the compact closed category.

Diagrammatically, a list will correspond to ports read from left-to-right and from top-to-bottom. We indicate ports of zero width (only the control bit) by a thin line and ports of width n by an additional thicker line (the data part). For example a circuit of interface $[\![\mathsf{com} \to \mathsf{com}]\!] = (-0, +0, +0, -0)$ can be written in any of these two ways:



The unit-width ports are used to transmit *events*, represented as the value of the port being held high for one clock cycle. The n-width ports correspond to data lines. The data on the line is only considered to be meaningful while there is an event on the control line.

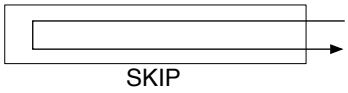
Below, for each language constant we will give the asynchronous game-semantic interpretation and its low-latency synchronous representation [GM11] and the (obvious) circuit implementation. For the purpose of giving the game-semantics and its representation, the circuit interface will correspond to the *set of moves*; the semantics is a set of traces over the possible moves. We denote a move/event on port k of interface $I = (p_1, \ldots, p_m)$ by

n_k, where n is the data value; if the data-width is 0 we write $*_k$. We use $\langle m, m\rangle$ to indicate the simultaneity of move/event m and m'. We use $m \cdot m'$ to indicate concatenation, i.e. move/event m' happens *after* m, but not necessarily in the very next clock cycle. We define $m \bullet m' = \{\langle m, m'\rangle, m \cdot m'\}$. We define the game semantic interpretation by $[\![-]\!]_g$ and the synchronous representation by $[\![-]\!]_s$. The notation $pc-$ applied to a set means closure under prefix-taking.

Skip

$$\mathsf{skip} : \mathsf{com}, [\![\mathsf{com}]\!] = (+0, -0)$$
$$[\![\mathsf{skip}]\!]_g = pc\{*_1 \cdot *_2\}$$
$$[\![\mathsf{skip}]\!]_s = pc\{\langle *_1, *_2\rangle\}.$$

The circuit representation is:

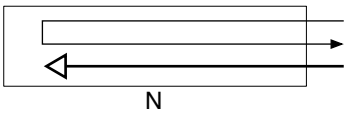


Intuitively the input port of a command is a request to run the command and the output port is the acknowledgment of successful completion. In the case of skip the acknowledgment is immediate.

Integer constant

$$\mathsf{k} : \mathsf{exp}, [\![\mathsf{exp}]\!] = (+1, -n)$$
$$[\![\mathsf{k}]\!]_g = pc\{*_1 \cdot k_2\}$$
$$[\![\mathsf{k}]\!]_s = pc\{\langle *_1, k_2\rangle\}.$$

The circuit representation is:

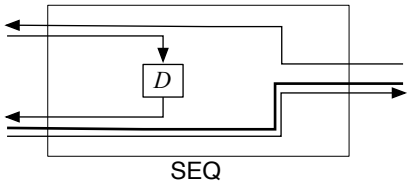


Intuitively the input port of an expression is a request to evaluate the expression and the output port is the data result and a control signal indicating successful evaluation. In the case of a constant n the acknowledgment is immediate and the data is connected to a fixed bit pattern.

Sequential composition

$$\mathsf{seq} : \mathsf{com} \times \mathsf{exp} \to \mathsf{exp}$$
$$[\![\mathsf{com} \times \mathsf{exp} \to \mathsf{exp}]\!] = (-0, +0, -0, +n, +0, -n)$$
$$[\![\mathsf{seq}]\!]_g = pc\{*_5 \cdot *_1 \cdot *_2 \cdot *_3 \cdot k_4 \cdot k_6 \mid k \in Z\}$$
$$[\![\mathsf{seq}]\!]_s = pc\{\langle *_5, *_1\rangle \bullet *_2 \cdot *_3 \bullet \langle k_4 \cdot k_6\rangle \mid k \in Z\}.$$

The circuit representation is:



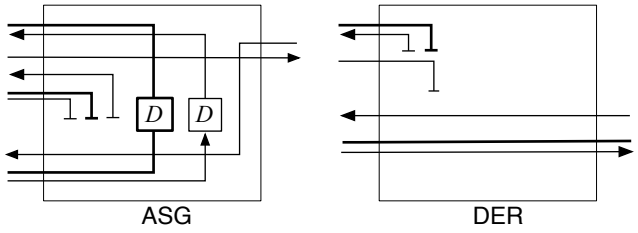
Above, D denotes a one-clock delay (D-flip-flop). A sequencer SEQ propagates the request to evaluate a command in sequence with an expression by first sending an execute request to the command, then to the expression upon receiving the acknowledgment from the command. The result of the expression is propagated to the calling context. Note the unit delay placed between the command acknowledgment and the expression request. Its presence is a necessary artifact of correctly representing asynchronous processes synchronously and cannot be optimised away [GM11].

Assignment and dereferencing

$$\mathsf{asg} : \mathsf{var} \times \mathsf{exp} \to \mathsf{com}$$
$$[\![\mathsf{var} \times \mathsf{exp} \to \mathsf{com}]\!] = (-n, +0, -0, +n, -0, +n, +0, -n)$$
$$[\![\mathsf{asg}]\!]_g = pc\{*_7 \cdot *_5 \cdot k_6 \cdot k_1 \cdot *_2 \cdot *_8 \mid k \in Z\}$$
$$[\![\mathsf{asg}]\!]_s = pc\{\langle *_7, *_5\rangle \bullet k_6 \cdot k_1 \bullet \langle *_2, *_8\rangle \mid k \in Z\}.$$

$$\mathsf{der} : \mathsf{var} \to \mathsf{exp}, [\![\mathsf{var} \to \mathsf{exp} = (-n, +0, -0, +n, +0, -n)]\!]$$
$$[\![\mathsf{der}]\!]_g = pc\{*_5 \cdot *_3 \cdot k_4 \cdot k_6 \mid k \in Z\}$$
$$[\![\mathsf{der}]\!]_s = pc\{\langle *_5, *_3\rangle \bullet \langle k_4, k_6\rangle \mid k \in Z\}$$

The circuit representations are, respectively:

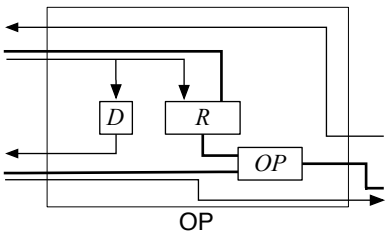


The variable type has four ports: writing data (n bits), acknowledging a write (0 bits), requesting a read (0 bits) and providing data (n bits). Assignment is a sequencing of an evaluation of the integer argument with a write request to the variable; the unused variable ports are grounded. Dereferencing is simply a projection of a variable interface onto an expression interface by propagating the read-part of the interface and blocking the write part.

Operators

$$\mathsf{op} : \mathsf{exp} \times \mathsf{exp} \to \mathsf{exp}$$
$$[\![\mathsf{exp} \times \mathsf{exp} \to \mathsf{exp}]\!] = (-0, +n, -0, +n, +0, -n)$$
$$[\![\mathsf{op}]\!]_g = pc\{*_5 \cdot *_1 \cdot k_2 \cdot *_3 \cdot k'_4 \cdot k''_6 \mid op(k, k') = k'' \in Z\}$$
$$[\![\mathsf{op}]\!]_s = pc\{\langle *_5, *_1\rangle \bullet k_2 \cdot *_3 \bullet \langle k'_4 \cdot k_6\rangle \mid k'' \in Z\}.$$

The circuit representation is:



R above is a register. The input control of port 2 is connected to the *load* pin of the register. The (combinatorial) circuit *OP* implements the operation. Note that the value of the first operator is saved in the register because expressions can change their value in time due to side-effects.

Branching

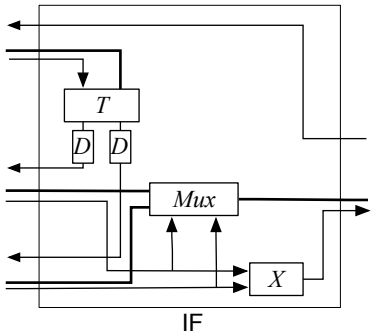
$\mathsf{if} : \mathsf{exp} \times \mathsf{exp} \times \mathsf{exp} \to \mathsf{exp}$

$[\![\mathsf{exp} \times \mathsf{exp} \times \mathsf{exp} \to \mathsf{exp}]\!] = (-0, +n, -0, +n, -0, +n, +0, -n)$

$[\![\mathsf{if}]\!]_g = pc\{*_7 \cdot *_1 \cdot 0_2 \cdot *_5 \cdot k_6 \cdot k_8, *_7 \cdot *_1 \cdot k'_2 \cdot *_3 \cdot k_4 \cdot k_8 \mid k' \neq 0, k \in Z\}$

$[\![\mathsf{if}]\!]_s = pc\{\langle *_7, *_1\rangle \bullet 0_2 \cdot *_5 \bullet \langle k_6, k_8\rangle, \langle *_7, *_1\rangle \bullet k'_2 \cdot *_3 \bullet \langle k_4, k_8\rangle \mid k' \neq 0, k \in Z\}$

The corresponding circuit is:



IF

Above, *Mux* is a (combinatorial) n-bit multiplexer which selects one data path or the other depending on the control signal. X is a merge of two control signals (*or* or *exclusive-or*) and T is a demultiplexer which propagates the input control signal to the first or second output, depending on whether the data value is zero or nonzero. As before, the delay D is necessary for correctness considerations.

Iteration

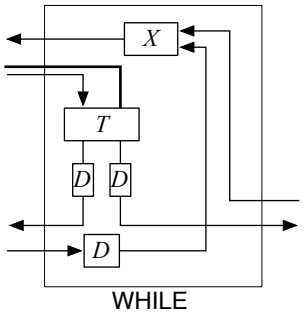
$[\![\mathsf{while} : \mathsf{exp} \times \mathsf{com} \to \mathsf{com}]\!]$

$[\![\mathsf{exp} \times \mathsf{com} \to \mathsf{com}]\!] = (-0, +n, -0, +0, +0, -0)$

$[\![\mathsf{while}]\!]_g = pc\{*_5 \cdot (*_1 \cdot 0_2 \cdot *_3 \cdot *_4)^* \cdot *_1 \cdot k_2 \cdot *_6 \mid k \in Z \setminus \{0\}\}$

$[\![\mathsf{while}]\!]_s = pc\{\langle *_5, *_1\rangle \bullet 0_2 \cdot *_3 \bullet *_4 \cdot (*_1 \bullet 0_2 \cdot *_3 \bullet *_4)^* \cdot *_1 \bullet k_2 \cdot *_6, \langle *_5, *_1\rangle \bullet k_2 \cdot *_6 \mid k \in Z \setminus \{0\}\}$

The circuit is:



WHILE

The iterator will keep executing the second argument as long as the first argument is zero.

State The local variable binder is a higher-order constant.

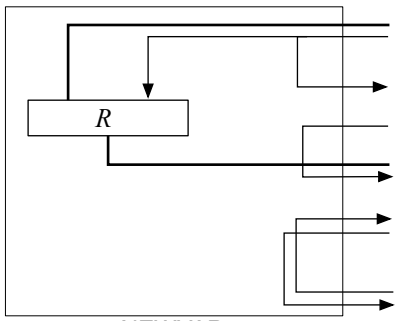
$\mathsf{newvar} : (\mathsf{var} \to \mathsf{com}) \to \mathsf{com}$

$[\![(\mathsf{var} \to \mathsf{com}) \to \mathsf{com}]\!] = (+n, -0, +0, -n, -0, +0, +0, -0)$

$[\![\mathsf{newvar}]\!]_g = pc\{*_7 \cdot *_5 \cdot v \cdot *_6 \cdot *_8 \mid v \in \big(\sum_{k \in Z} k_1 \cdot *_2 \cdot (*_3 \cdot k_4)^*\big)^*\}$

$[\![\mathsf{newvar}]\!]_s = pc\{\langle *_7, *_5\rangle \bullet v \bullet \langle *_6, *_8\rangle \mid v \in \big(\sum_{k \in Z} \langle k_1, *_2\rangle \bullet \langle *_3, k_4\rangle^*\big)^*\}$

The circuit with this behaviour is basically just a register:



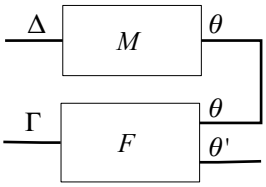
NEWVAR

In addition to the constants of the language we also interpret structural rules (abstraction, application, product formation) as constructions on circuits. In diagrams we represent bunches of wires (*buses*) as thick lines. When we connect two interfaces by a bus we assume that the two interfaces match in number and kind of port perfectly.

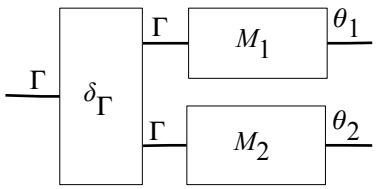
In general a term of signature $x_1 : \theta_1, \ldots, x_k : \theta_k \vdash M : \theta$ will be interpreted as a circuit of interface $[\![\theta_1 \times \cdots \times \theta_k \to \theta]\!]$.

Abstraction Semantically, in both the original game semantics and the synchronous representation the abstraction $\Gamma \vdash \lambda x : \theta.M : \theta'$ is interpreted by the currying isomorphism. Similarly, in circuits the two interfaces for this circuit and $\Gamma, x : \theta \vdash M : \theta'$ are isomorphic.

Application To apply a function of type $\Gamma \vdash F : \theta \to \theta'$ to an argument $\Delta \vdash M : \theta$ we simply connect the ports in $[\![\theta]\!]$ from the two circuits:

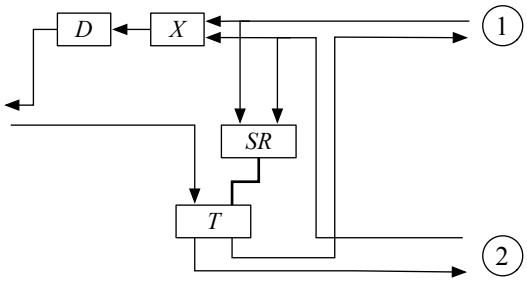


Product formation Unlike application, in product formation we allow the sharing of identifiers. This is realised through special circuitry implementing the *diagonal* function $\lambda x.\langle x, x\rangle$ for any type θ. Diagrammatically, the product of terms $\Gamma \vdash M_i : \theta_i$ is:



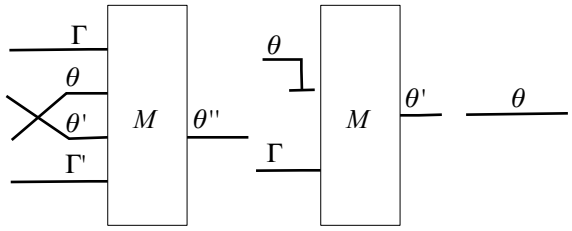
The diagonal circuit is behaviourally similar to a stateful multiplexer-demultiplexer. It routes an input signal from the interfaces on the right to the shared interface on the left while storing the source of the signal in a *set-reset* register. From the semantic model we know that any output signal in the shared interface is followed by an input signal in the same interface, which is routed to the originating component using the demultiplexer T. SR registers are needed for all initial questions and T blocks use the registers for the matching question.

In the simplest case, for δ_{com} the circuit looks like this:



Note that this diagonal introduces a unit delay, which is not strictly necessary for correctness. A lower-latency diagonal that correctly handles instant feedback without the delay D can be implemented, but is more complex.

Structural rules Finally, we give constructions for commutativity, weakening and identity. They are represented by the circuits below:



Commutativity is rearranging ports in the interface, weakening is the addition of dummy ports and identities are typed buses.

Example. The GoS approach allows the compilation of higher-order, open terms. Consider for example a program that executes in-place map on a data structure equipped with an iterator: $\lambda f : \mathsf{exp} \to \mathsf{exp}.\mathsf{init}; \mathsf{while}(\mathsf{more})(\mathsf{curr} := \mathsf{f}(!\mathsf{curr}); \mathsf{next}) : \mathsf{com}$, where $\mathsf{init} : \mathsf{com}, \mathsf{curr} : \mathsf{var}, \mathsf{next} : \mathsf{com}, \mathsf{more} : \mathsf{exp}$. The interface of the iterator consists of an initialiser, access to the current element, advance to the next element and test if there are more elements in the store. Since SCI is call-by-name all free identifiers are thunks. The block diagram of the circuit is given in Fig. 1: The full schematic of the circuit for in-place map is also given in Fig. 1; for clarity we have identified what ports correspond to what identifiers. The ports on the right correspond to the term type com. Note that we can optimise away the diagonal for variable identifier curr because the first instance is used for writing while the second one for reading.

4. Unfolding finite recursion in space

In its simplest instance recursion can be seen simply as an unfolding of a circuit definition. Such recursive definitions can only apply to well-founded definitions as infinite unfoldings cannot be synthesised. To support finite recursion via unfolding we augment the SCI type system with a rudimentary form of dependent typing.

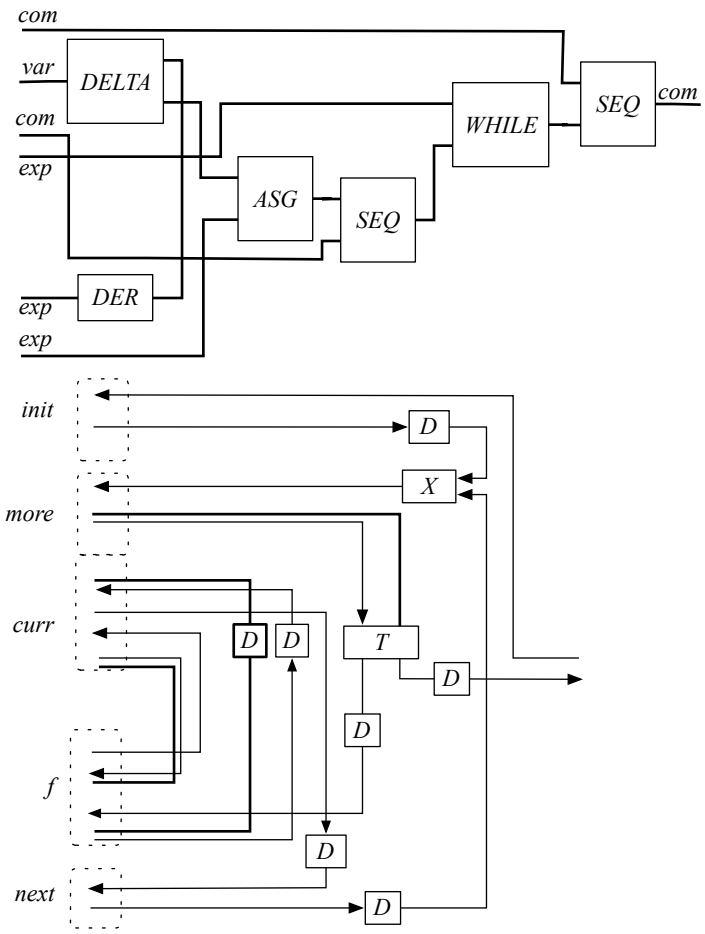


Figure 1. In-place map schematic and implementation

First, it is convenient to add linear product (tensor) explicitly to the type system:

$$\frac{\Gamma \vdash M : \theta \qquad \Gamma' \vdash M' : \theta'}{\Gamma, \Gamma' \vdash M \otimes M' : \theta \otimes \theta'}$$

We also add a very simple form of dependent types, $\theta\{N\}$ which is defined as

$$\theta\{0\} = \mathbf{1}, \qquad \theta\{N\} = \theta \otimes \theta\{N-1\},$$

where $\mathbf{1}$ is the unit type (the empty interface).

The language of indices N consists of natural number constants, subtraction and division (over natural numbers). This will guarantee that recursive definitions have finite unfoldings. Note that since $\otimes$ is adjoint to $\to$ in the type system, the following three types are isomorphic:

$$\theta \otimes \cdots \otimes \theta \to \theta' \simeq \theta \to \cdots \to \theta \to \theta' \simeq \theta\{N\} \to \theta'.$$

For example, an N-ary parallel execution operator can be recursively defined, for example, as:

$$\begin{aligned} \mathsf{par}\{1\} &= \lambda x : \mathsf{com}.x : \mathsf{com} \to \mathsf{com} \\ \mathsf{par}\{N\} &= \lambda x : \mathsf{com}.(x \mid\mid \mathsf{par}\{N-1\}) \\ &\quad : \mathsf{com} \to \mathsf{com}\{N-1\} \to \mathsf{com} \simeq \mathsf{com}\{N\} \to \mathsf{com}. \end{aligned}$$

Recursive definitions in this dependent-type metalanguage are elaborated first into SCI by unfolding the definitions until all in-

dices $\{N\}$ are reduced, after which the normal compilation process applies.

Although this approach is technically very simple it is surprisingly effective. For example, it allows the definition of sorting networks using programming language syntax so that the elaborated SCI programs synthesise precisely into the desired sorting network.

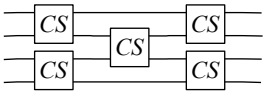
4.1 Batcher's Bitonic Sort

Bitonic Sort [Bat68] is a well known algorithm for generating optimal sorting networks. The definition of the algorithm is *structural*, i.e. it describes how the network is constructed from subcomponents, rather than *behavioural*, i.e. indicating the way input data is processed into output data, as is the case with most "software" algorithms. As a consequence, mapping Batcher's description of a sorting network into a parallelisable program usually needs to change the point of view from the network to the individual element. This is quite subtle and it renders the algorithm unrecognisable[3].

The computational element of a sorting network is a compare-and-swap circuit

$$CS(m,n) = \begin{cases} (m,n) & \text{if } m \leq n \\ (n,m) & \text{if } m > n. \end{cases}$$

A *sorting network* is a circuit formed exclusively from CS circuits. A simple, but inefficient, sorting network is the odd-even transposition network, which for 4 elements is:



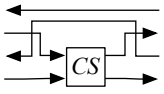
In SCI the CS box can be implemented as

$$CS \stackrel{def}{=} \lambda m{:}\mathsf{exp}.\lambda n{:}\mathsf{exp}.\mathsf{if}\ m < n\ \mathsf{then}\ m \otimes n\ \mathsf{else}\ n \otimes m$$

With $CS : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp} \otimes \mathsf{exp} \simeq \mathsf{exp} \otimes \mathsf{exp} \to \mathsf{exp} \otimes \mathsf{exp}$. Let SC be the converse circuit:

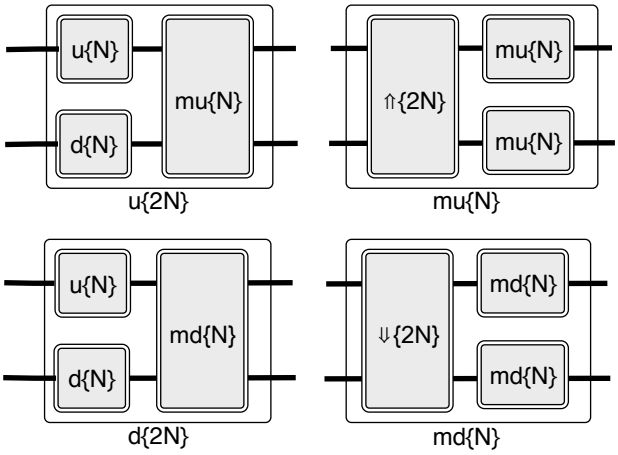
$$SC \stackrel{def}{=} \lambda m{:}\mathsf{exp}.\lambda n{:}\mathsf{exp}.\mathsf{if}\ m > n\ \mathsf{then}\ m \otimes n\ \mathsf{else}\ n \otimes m.$$

Note that the type system allows the $<$ operator to be given a parallel implementation; we can also use the special CS circuit, packaged like this so that it conforms to the type signature:

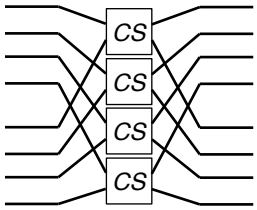


The standard definition of Batcher's algorithm as a recursively specified sorting network is this:

[3] See `http://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm` and the CUDA implementation at `http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html`.



The circuit $\Uparrow\{2N\}$ compares-and-swaps the k-th element in the array against the $N+k$-th element, merging two bitonic sequences; for $N = 4$ is:



$\Downarrow\{2N\}$ is the "upside-down" $\Uparrow\{N\}$ circuit. The other circuits involved in the recursive definition are: u (up-sort), d (down-sort), mu (merge-up) and md (merge-down).

First let us introduce syntactic sugar for function composition, $f \circ g = \lambda x : \theta.f(g(x))$. In SCI the Batcher's bitonic sorting network is defined by the following program:

$$\begin{aligned}
\mathrm{u}\{1\} &= \mathrm{d}\{1\} = \lambda x{:}\mathsf{exp}.x \\
\mathrm{mu}\{1\} &= \mathrm{md}\{1\} = \lambda x{:}\mathsf{exp}.x \\
\mathrm{u}\{N\} &= \mathrm{mu}\{N\} \circ (\mathrm{u}\{N/2\} \otimes \mathrm{d}\{N/2\}) \\
\mathrm{d}\{N\} &= \mathrm{md}\{N\} \circ (\mathrm{u}\{N/2\} \otimes \mathrm{d}\{N/2\}) \\
\mathrm{mu}\{N\} &= (\mathrm{mu}\{N/2\} \otimes \mathrm{mu}\{N/2\}) \circ \mathrm{up}\{N\} \\
\mathrm{md}\{N\} &= (\mathrm{md}\{N/2\} \otimes \mathrm{md}\{N/2\}) \circ \mathrm{down}\{N\} \\
\mathrm{up}\{N\} &= (\lambda x{:}\mathsf{exp}\{N/2{-}1\}.\lambda z{:}\mathsf{exp}\{N/2{-}1\}.\lambda y{:}\mathsf{exp}.\lambda u{:}\mathsf{exp}. \\
&\qquad x \otimes y \otimes z \otimes u) \\
&\quad \circ (CS \otimes \mathrm{up}\{N/2-2\}) \\
&\quad \circ (\lambda x{:}\mathsf{exp}.\lambda y{:}\mathsf{exp}\{N/2{-}1\}.\lambda z{:}\mathsf{exp}.\lambda u{:}\mathsf{exp}\{N/2{-}1\}. \\
&\qquad x \otimes z \otimes y \otimes u) \\
\mathrm{down}\{N\} &= (\lambda x{:}\mathsf{exp}\{N/2{-}1\}.\lambda z{:}\mathsf{exp}\{N/2{-}1\}.\lambda y{:}\mathsf{exp}.\lambda u{:}\mathsf{exp}. \\
&\qquad x \otimes y \otimes z \otimes u) \\
&\quad \circ (SC \otimes \mathrm{up}\{N/2-2\}) \\
&\quad \circ (\lambda x{:}\mathsf{exp}.\lambda y{:}\mathsf{exp}\{N/2{-}1\}.\lambda z{:}\mathsf{exp}.\lambda u{:}\mathsf{exp}\{N/2{-}1\}. \\
&\qquad x \otimes z \otimes y \otimes u)
\end{aligned}$$

Note that above the $\circ$ operator needs to be elaborated with the proper type, which is a trivial exercise in type inference. A bitonic sorter is $\mathrm{u}\{N\}$ for N a power of 2.

This program elaborates to a SCI program which then synthesises to the standard bitonic sorting network. Note that this approach is very similar to the approach that structural functional

hardware description languages such as Lava[4]. However, because our syntax is based on compact-closed combinators they match conventional programming language syntax based on abstraction and application. In the case of Lava, the underlying syntax is set (albeit not explicitly) in a traced-monoidal framework which is somewhat more rigid [JSV96].

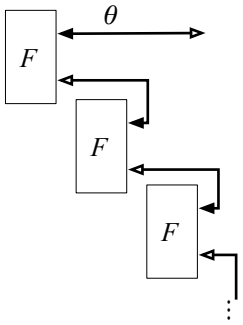
5. Unfolding affine recursion in time

Sometimes a recursive unfolding of a circuit is not desired or not possible and we may wish to have genuine run-time recursive behaviour.

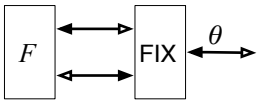
First note that unrestricted recursion in SCI is not possible as the unfolding of the recursive calls can violate the typing rule requiring disjointness of free identifiers between a function and its argument, as in $\mathsf{fix}(F) \longrightarrow F(\mathsf{fix}(F))$. Therefore fix-point can only be applied to closed terms:

$$\frac{\vdash F : \theta \to \theta}{\vdash \mathsf{fix}(F) : \theta} \text{ fix-point}$$

The recursive unfolding of the fix-point combinator suggests that the circuit representation of the fix-point circuit should be equivalent to the following infinite circuit:



The solid/hollow arrows indicate the change of polarity of the bus $[\![\theta]\!]$. Obviously, what we would like is something like this:



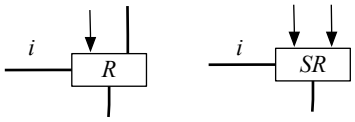
A tempting idea is to fold the circuit for $[\![F]\!]$ onto itself by using a FIX circuit very similar to the diagonal. However, this is not possible: some of the constituent circuits of $[\![F]\!]$ are stateful and their states can be different in different instances of F. The relevant stateful circuits are:

R used in the implementation of operators and local variables.

SR used in the implementation of the diagonal.

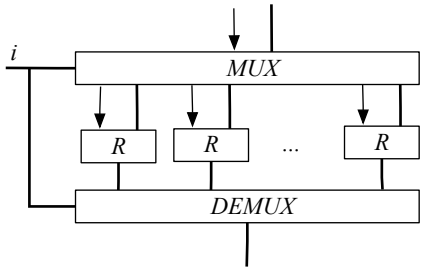
The other stateful circuit is D, but while the FIX block is executing, its state is the same for all instances of $[\![F]\!]$.

As a first step towards implementing recursion we replace all occurrences of registers R and *SR* with *indexed* versions Ri and *SRi*:



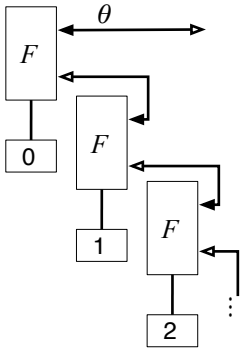
[4] See `http://raintown.org/lava/` for a Bitonic sort implementation.

Ri is implemented as follows:



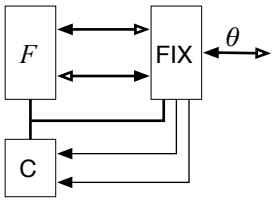
and *SRi* analogously. The registers must be replaced with small addressable memory elements.

Now we can rewrite our infinite unfolding of the fix-point like this:



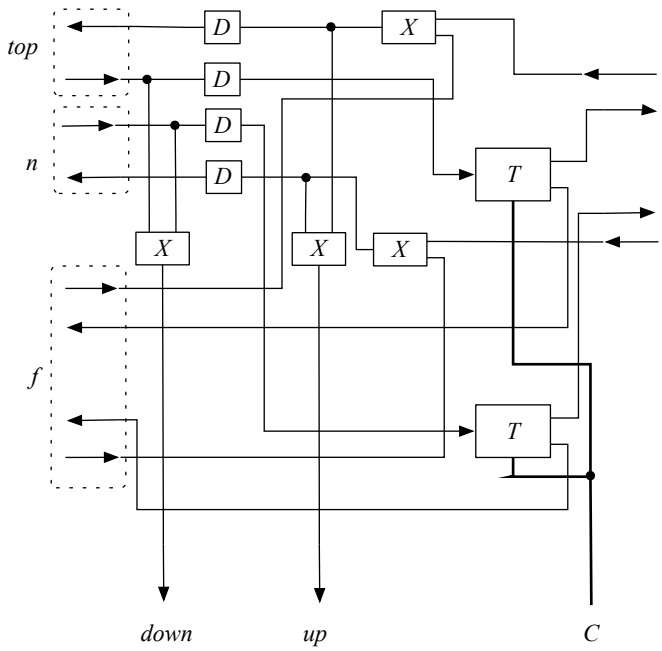
Every instance of $[\![F]\!]$ now uses a different index, but is otherwise identical. This means that we can replace the fixed indices with a counter and fold all the instances into one single instance, indexed by the counter. The value of the counter will indicate what "virtual" instance of $[\![F]\!]$ is active and will be used as an index into the registers.

The fix-point circuit will increase this global counter whenever a recursive call is made and decrease it when a recursive return is made. When the counter is 0 the recursive return will be to the environment.

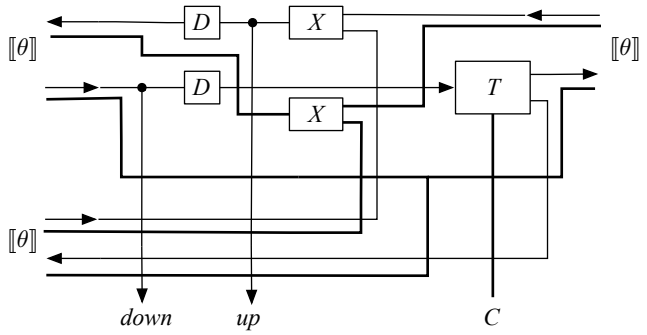


Circuit C is an up-down counter.

For example, for $\mathsf{exp} \to \mathsf{exp}$ the fix-point combinator which can be applied to function $\lambda f : \mathsf{exp} \to \mathsf{exp}.\lambda n : \mathsf{exp}.M$ is (data lines not shown):



In the general case, for an arbitrary θ, the FIX_θ combinator is a replication of the circuit below



for each input and output port in θ. If the port is pure control then the data line can be omitted.

Correctness and limitations. Except for recursion, the implementation of the compiler is a hardware instantiation of the synchronous representation of the game semantics and is correct by construction as explained in a series of papers [McC02, GM10, GM11]. The detailed proof of correctness of the fix-point constructor is beyond the scope of this paper, but the step-by-step construction given in this section mirrors the structure of the proof of correctness. There are two limitations which play an important role in the correct implementation of fix-point:

Concurrency. The SCI type system allows concurrency in general, but concurrency inside the term to which the fix-point constructor is applied must be disallowed. It is crucial for the correctness of the implementation that only one "virtual" instance of the recursive function, as indicated by the recursion depth counter, is active at any one moment in time. If the recursive call is used in a concurrent context this can no longer be guaranteed. Note that recursively implemented functions can run in a parallel environment, only internally parallelism is disallowed. For simplicity we disallow all parallelism, but a more careful analysis which only bans the use of the recursive call in a parallel context is possible.

Nested recursion. For simplicity we assume that recursive definitions do not contain other recursive definitions. This is technically possible, by applying the same methodology and transforming the recursion counter into an indexed array of recursion counters.

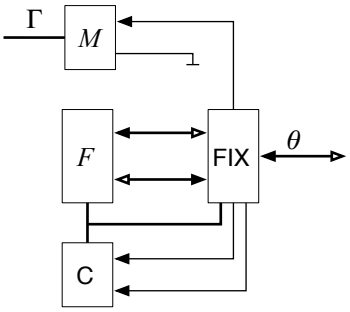
5.1 Overflow detection

Recursive calls assume an idealized machine model with no bound on resources. In physical machines that cannot be the case so recursion calls that run too deep can overflow resources. On a conventional CPU-based architecture this leads to an abnormal termination of the program, triggered either by the run-time environment (memory management modules or the operating system). A synthesised program, in contrast, runs in isolation and runtime errors cannot be detected or processed. In our implementation, overflow would manifest simply by the counter rolling over, which would lead to active instances of the recursive functions being misidentified. This is problematic because it will not give an error but will produce the wrong result (similar to integer overflow in C).

It is important therefore to build into the language an error detection and handling mechanisms by providing the fix-point operator with an overflow handler. The syntax is

$$\frac{\vdash F : \theta \to \theta \qquad \Gamma \vdash M : \mathsf{com}}{\Gamma \vdash \mathsf{fix}\, F \,\mathsf{with}\, M : \theta}$$

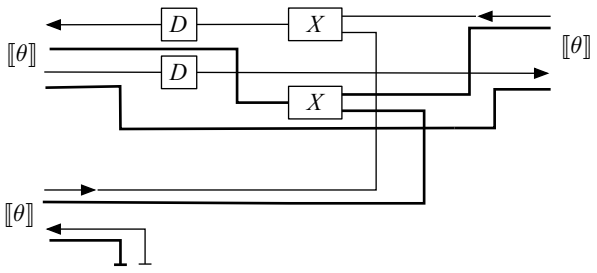
The implementation:



The fix-point operator is, by design, aware of the maximum size of the counter. When a recursive call is about to increase beyond the maximum size, instead of propagating the signal back to $[\![F]\!]$ it will issue a special error signal to command $[\![M]\!]$ which is the error handler. The control output of $[\![M]\!]$ is either ignored (grounded) or connected to a special global error port.

5.2 Tail recursion

Iteration (while) is included in the recursion-free fragment of SCI and it can be readily generalised to a higher-order tail-recursion operator. Because tail-recursive calls do not need to percolate back through each instance of the function the counter C and the instance-management apparatus are no longer necessary. The tail-recursive operator TAIL_θ becomes simply:



Note that the return from the recursive call is immediately propagated to the calling environment.

Also, there is no need for the tail recursion operation to be applied to closed terms, as the tail unfolding is sequential rather than nested application:

$$\frac{\Gamma \vdash F : \theta \to \theta}{\Gamma \vdash \mathsf{tail}_\theta F : \theta}$$

6. Performance and benchmarks

At this early stage our research is mainly qualitative, demonstrating the possibility of compiling recursive functions in static hardware.

We will give two benchmark-style examples to show that even in the absence of any compiler optimisations the performance is promising. We choose a naive Fibonacci number calculation as a farily common benchmark of recursion performance and we examine the call-by-name and the call-by-value versions. This is not a realistic efficient implementation but is an excellent contrived example for creating a large number of recursive calls in an arbitrarily nested pattern.

As this is the first, to our knowledge, attempt to give a general method for synthesising recursion in time (recursion in space has been explored thoroughly by Lava) we are forced to compare against execution on a conventional CPU-based architecture. It is difficult to abstract away from the fact that we use not only different devices (CPU versus FPGA) but also different compilers and runtime environments.

CPU We use an Intel Core 2 Duo processor running at 2.4 GHz on a machine with the Mac OS X 10.5 operating system, the Marst[5] Algol 60 compiler and the gcc 4.0.1 C compiler.

FPGA We use a Altera Stratix III EPSL150 device, programmed using Quartus 10.0, set on a Terasic DE3 board, operating at a default 50 MHz. We compute the maximum safe clock frequency using the TimeQuest timing constraint tool, part of the `Quartus` tool-set, and we scale execution time accordingly.

In order to normalise the performance comparison we will consider two main metrics:

Relative throughput We measure the relative throughput between the two devices computing from exactly the same source code. We take into account execution time and resource utilisation, which indicate the amount of parallelism that the device can support. Note that on the FPGA parallel execution has no additional overhead or contention over resources, even in the case of recursive functions, as no global stack is used in the implementation.

Relative throughput per transistor We further normalise the relative throughput of the two computations relative to the transistor count, in order to obtain a fairer measure of performance.

On both these measures the circuits we synthesise perform better than CPU-based execution.

[5] `http://www.gnu.org/software/marst/`

6.1 Fibonacci, call-by-name

We use the simple-minded implementation of Fibonacci numbers in order to generate a very large number of recursive calls from small and simple code. This is appropriate as we do want to focus our benchmarking on the effectiveness of the implementation of recursion.

In traditional Algol60 syntax the program is:

```
integer procedure fib(n);
  integer n;
  begin
    if n-1 = 0 then fib := 1
    else if n-2 = 0 then fib := 1
    else fib := fib(n-1) + fib(n-2)
  end fib;
```

The benchmark results are as follows:

	CPU	FPGA
Time	35.5 s	50 s
Clock	2.4 GHz	137 MHz
Cycles	85.2 B	6.85 B
Transistors	$\simeq$ 300 M	$\simeq$ 400 M
Utilisation	50%	2%
Tx (rel.)	5.62%	1,774.9%
Tx/trans (rel)	7.49%	1,331.1%

Note that the execution time on the FPGA is larger than on the CPU. This is to be expected, as the code is purely sequential and uses very little memory. This is precisely the code that CPUs are highly optimised to execute fast, and this is on top of the CPU working with a much faster clock.

However, one core of this dual-core CPU is fully occupied by this computation, leading to a 50% utilisation of resources. We could compute two instances of fib in parallel with no extra overhead. On the FPGA, the area occupied by the synthesised circuit is only 2%, which means we can run up to 50 instances of this circuit in parallel, with no overhead. Note that the utilisation bound (2%) is on total ALUTs rather than on memory (1%), which means that the overhead needed to handle recursion is manageable. Using spatial unfolding we can only run fib to a depth of recursion of about 5-6 (the expansion is exponential) whereas our temporal unfolding has a depth of 256 (and reduced use of resources).

A relative comparison of execution time versus utilisation gives the FPGA 1,774.9% (almost 18×) total throughput compared to the CPU (conversely, the CPU has 5.62% of the FPGA throughput).

Images of block diagram and resource utilisation density for the synthesised circuit are in Fig. 2. Note the row of RAM blocks, three of which are used to implement the stateful elements of recursion– they are the only memory overhead needed for implementing recursion.

It is fair to take into account the fact that this particular FPGA has a larger transistor count than the corresponding CPU. Normalising throughput by this factor still gives the FPGA over 13× total computational throughput per transistor.

6.2 Fibonacci, call-by-value

The CBN evaluation makes the purely functional Fibonacci function staggeringly inefficient. An improvement on the naive algorithm forces the evaluation of the argument inside the function:

```
integer procedure fib(n);
  integer n;
  begin
    integer n0, n1, n2;
    n0 := n;
    if n0-1 = 0 then fib := 1
    else if n0-2 = 0 then fib := 1
```

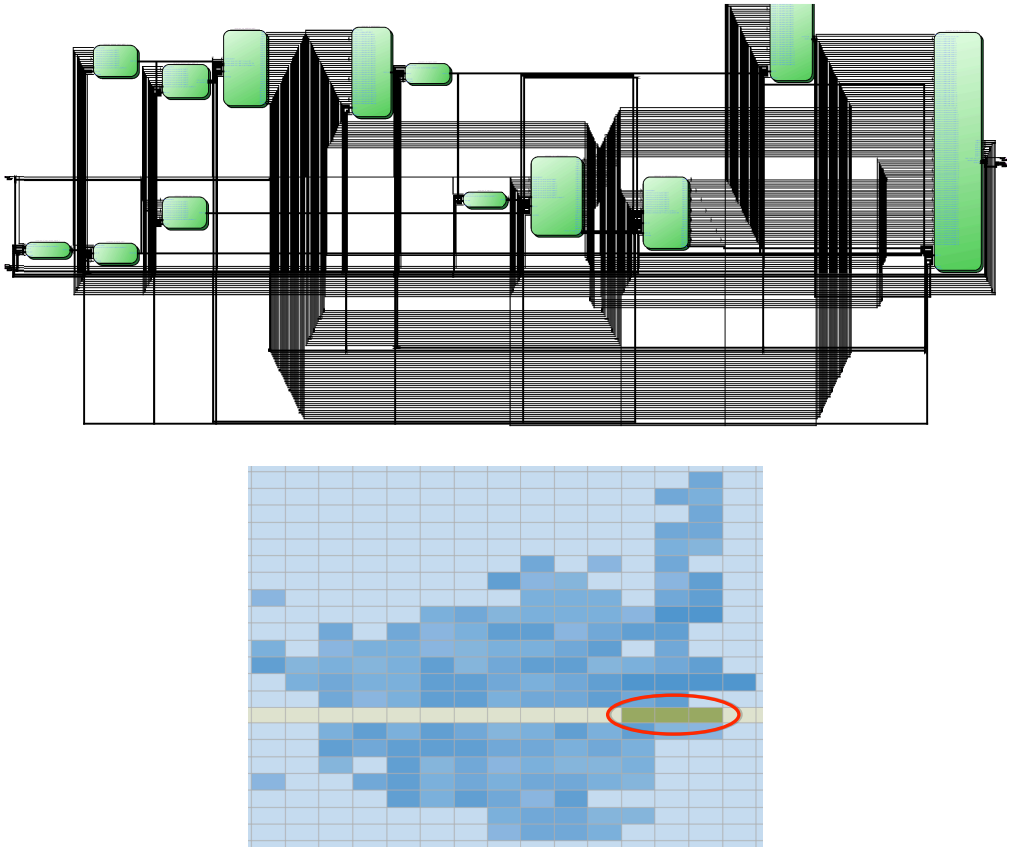


Figure 2. Synthesised FPGA layout for `fib`: block diagram and resource utilisation density (zoom-in)

```
    else begin
      n1 := fib(n0-1);
      n2 := fib(n0-2);
      fib := n1 + n2 end
  end fib;
```

The results are now better in absolute terms both on the CPU and the FPGA. The maximum clock frequency for the CBV `fib` is smaller, but a similar relative throughput advantage for the FPGA version still occurs.

	CPU	FPGA
Time	2.8 s	4.0 s
Clock	2.4 GHz	119 MHz
Cycles	6.78 B	.48 B
Transistors	$\simeq$ 300 M	$\simeq$ 400 M
Utilisation	50%	2%
Tx (rel.)	5.7%	1,735.9%
Tx/trans (rel)	7.6%	1,301.9%

As a sanity check, we can compare this implementation with a call-by-value implementation as used by more wide-spread compilers; Ocaml v3.10 computes `fib(36)` on the same CPU in 0.6 seconds, substantially better than Algol60 and Marst. However, the overall throughput of the FPGA remains higher.

In Fig. 3 we can see run-time snapshots taken with the SignalTap tool of the two circuits, indicating current calculated value, current value of the internal recursion counter and next value being calculated. We can notice in the CBN version the recursion counter cycling all the way back to 0 each time an argument needs to be (re)evaluated, whereas in the CBV version the argument is picked up from the local variable.

6.3 Expressiveness

The fact that the recursion operator can only be applied to syntactically closed functions is not a major restriction in terms of expressiveness. Affine recursion has been studied in the context of closed recursion in the lambda calculus [AFFM07] and is in fact the style of recursion used in Goedel's System T [AFFM10], where it is called *iteration*. We prefer "affine recursion" as iteration may incorrectly suggest repeated serial application rather than true, nested, recursion.

As a test of the expressiveness of the compiler we programmed Escardo's implementation of the Bailey-Borwein-Plouffe spigot algorithm for the computation of the n-th digit of π [BBP97]. The implementation is written in Haskell and it uses higher-order functionals to give precise representations to real numbers as streams of digits [Esc09]. The program was successfully compiled to VHDL and synthesised on the same Altera-based FPGA-board. Note that our language does not offer direct support for lists and streams, which needed to be coded up in the standard way into the underlying lambda calculus.

This test was done only to evaluate the expressiveness of the compiler, as without direct semantic support for lists or streams the run-time performance cannot be realistically assessed. Even so, the fact that ALUT utilisation stood at only 12% and the BRAM utilisation at only 8% indicates that the overall footprint is small and the overhead imposed by recursion manageable.

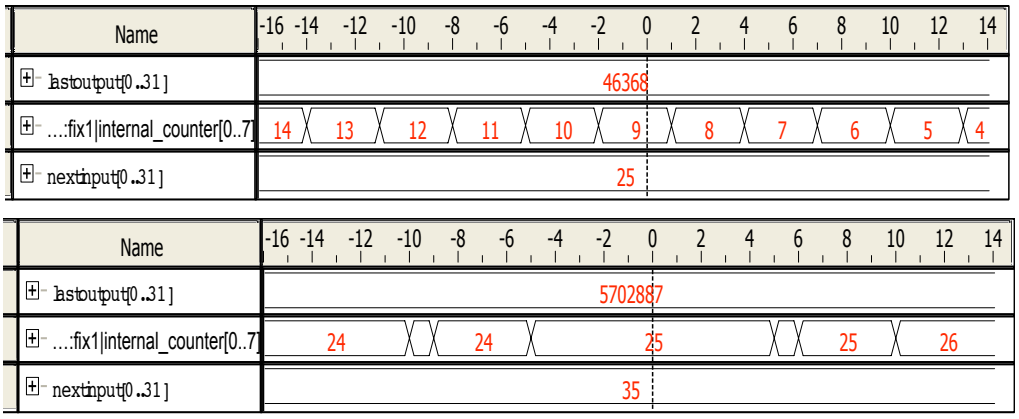


Figure 3. Run-time snapshots of CBN and CBV `fib`

7. Summary

We have shown a systematic approach for compiling programs using affine recursion into correct circuits implemented on FPGAs which operate between the 110 MHz to 140 MHz range (so the critical path is respectable) and these circuits can use far fewer cycles to compute a result than the corresponding software implementation. However, unlike the software version we can systematically explore space and time trade-offs by unrolling function calls through effectively inlining which in turn can increase throughput at the cost of area. Our initial preliminary results are encouraging and we are developing the system further by allowing the processing of off-chip dynamic data-structures and streams. By understanding how to effectively synthesize recursive descriptions we get one step closer to the ability to transform programs into circuits for implementations that have superior computational throughput or reduced energy consumption.

References

[AFFM07] Sandra Alves, Maribel Fernández, Mário Florido, and Ian Mackie. The power of closed reduction strategies. *Electr. Notes Theor. Comput. Sci.*, 174(10):57–74, 2007.

[AFFM10] Sandra Alves, Maribel Fernández, Mário Florido, and Ian Mackie. Gödel's system tau revisited. *Theor. Comput. Sci.*, 411(11-13):1484–1500, 2010.

[AJ92] Samson Abramsky and Radha Jagadeesan. New foundations for the Geometry of Interaction. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 211–222, 1992.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

[BBP97] D. Bailey, P. Borwein, and S. Plouffe. On the rapid computation of various polylogarithmic constants. *Mathematics of Computation*, 66(218):903–914, 1997.

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *The International Conference on Functional Programming (ICFP)*, New York, NY, USA, 1998. ACM.

[BH98] P. Bellows and B. Hutchings. JHDL: an HDL for reconfigurable systems. In *IEEE Symposium on FPGAs for Custom Computing Machines*, Apr 1998.

[Esc09] Martin H. Escardo. Computing with real numbers represented as infinite sequences of digits in haskell. In *Computability and complexity analysis*, Ljubljana, Slovenia, 2009. (code available at the author's web page).

[FG06] George Ferizis and Hossam El Gindy. Mapping recursive functions to reconfigurable hardware. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, 2006.

[Ghi07] Dan R. Ghica. Geometry of Synthesis: a structured approach to VLSI design. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 363–375, 2007.

[Ghi09] Dan R. Ghica. Applications of game semantics: From software analysis to hardware synthesis. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 17–26, 2009.

[Ghi11] Dan R. Ghica. Functions interface models for hardware compilation. In *ACM/IEEE Ninth International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2011. (forthcoming)

[GM00] Dan R. Ghica and Guy McCusker. Reasoning about Idealized Algol using regular languages. In *The International Colloquium on Automata, Languages and Programming (ICALP)*, pages 103–115, 2000.

[GM10] Dan R. Ghica and Mohamed N. Menaa. On the compositionality of round abstraction. In *The International Conference on Concurrency Theory (CONCUR)*, pages 417–431, 2010.

[GM11] Dan R. Ghica and Mohamed N. Menaa. Synchronous game semantics via round abstraction. In *The International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 350–364, 2011.

[GS10] Dan R. Ghica and Alex Smith. Geometry of Synthesis II: From games to delay-insensitive circuits. *Electr. Notes Theor. Comput. Sci.*, 265:301–324, 2010.

[GS11] Dan R. Ghica and Alex Smith. Geometry of Synthesis III: Resource management through type inference. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 345–356, 2011.

[HD08] Scott Hauck and André DeHon, editors. *Reconfigurable Computing*, chapter Specifying Circuit Layout in FPGAs. Systems on Silicon. Morgan Kaufmann Publishers, 2008.

[JSV96] André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of Cambridge Philosophical Society*, 119:447–468, 1996.

[KL80] G. M. Kelly and M. L. Laplaza. Coherence for compact closed categories. *Journal of Pure and Applied Algebra*, 19:193–213, 1980.

[Mac94] Ian Mackie. *The Geometry of Implementation*. PhD thesis, Imperial College, University of London, 1994.

[Mac95] Ian Mackie. The geometry of Interaction machine. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 198–208, 1995.

[McC02] Guy McCusker. A fully abstract relational model of Syntactic Control of Interference. In *The Conference on Computer Science Logic (CSL)*, pages 247–261, 2002.

[McC07] Guy McCusker. Categorical models of syntactic control of interference revisited, revisited. *LMS Journal of Computation and Mathematics*, 10:176–216, 2007.

[McC10] Guy McCusker. A graph model for imperative computation. *Logical Methods in Computer Science*, 6(1), 2010.

[MTH99] Tsutomu Maruyama, Masaaki Takagi, and Tsutomu Hoshino. Hardware implementation techniques for recursive calls and loops. In *The International Conference on Field Programmable Logic and Applications (FPL)*, pages 450–455, 1999.

[Nik04] Rishiyur Nikhil. Bluespec SystemVerilog: Efficient, correct RTL from high-level specifications. *Formal Methods and Models for Co-Design (MEMOCODE)*, 2004.

[OPTT99] Peter W. O'Hearn, John Power, Makoto Takeyama, and Robert D. Tennent. Syntactic control of interference revisited. *Theor. Comput. Sci.*, 228(1-2):211–252, 1999.

[OT81] Peter O'Hearn and Robert D. Tennent. *Algol-like languages*. Birkhauser, Boston, 1981.

[Red96] Uday S. Reddy. Global state considered unnecessary: An introduction to object-based semantics. *Lisp and Symbolic Computation*, 9(1):7–76, 1996.

[Rey78] John C. Reynolds. Syntactic control of interference. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 39–46, 1978.

[Rey81] John C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[Rey89] John C. Reynolds. Syntactic control of inference, part 2. In *The International Colloquium on Automata, Languages and Programming (ICALP)*, pages 704–722, 1989.

[Sel09] Peter Selinger. *New Structures for Physiscs*, chapter A survey of graphical languages for monoidal categories. Springer Lecture Notes in Physics, 2009.

[Skl04] Valery Sklyarov. FPGA-based implementation of recursive algorithms. *Microprocessors and Microsystems*, 28(5-6):197 – 211, 2004. Special Issue on FPGAs: Applications and Designs.

[TL07] D.B. Thomas and W. Luk. A domain specific language for reconfigurable path-based monte carlo simulations. In *International Conference on Field-Programmable Technology (ICFPT)*, pages 97 –104, 2007.

[Wal04] Matthew Wall. *Games for Syntactic Control of Interference*. PhD thesis, University of Sussex, 2004.