Geometry of Synthesis

A structured approach to VLSI design

Dan R. Ghica
University of Birmingham
drg@cs.bham.ac.uk

Abstract

We propose a new technique for hardware synthesis from higher-order functional languages with imperative features based on Reynolds's *Syntactic Control of Interference*. The restriction on contraction in the type system is useful for managing the thorny issue of sharing of physical circuits. We use a semantic model inspired by game semantics and the geometry of interaction, and express it directly as a certain class of digital circuits that form a cartesian, monoidal-closed category. A soundness result is given, which is also a correctness result for the compilation technique.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Denotational semantics; B.7.1 [*Types and Design Styles*]: VLSI

General Terms Design, Languages, Theory

Keywords Syntactic Control of Interference, Geometry of Interaction, game semantics, synthesis

1. Introduction

In this paper we propose a new technique for VLSI design that allows the synthesis of digital circuit specifications from a generic, higher-order functional programming language with imperative features.

The main innovative feature of this technique is the use of a semantic model inspired by and related to game semantics [3, 11] and Geometry of Interaction [10], a semantic model that can be expressed directly as a certain class of digital circuits. This source of inspiration is acknowledged, but the paper is self-contained and familiarity with these two topics is not required. On the contrary, this paper could serve as a motivating introduction to their theoretical considerations.

Another innovation is the choice of the programming language, *Basic Syntactic Control of Interference (bSCI)* [19, 16]. It is an Algol-like language [20] with affine typing. This turns out to be a highly suitable language for two reasons. First, the affine type system is a precise tool for the control of *sharing of resources* in the programming language, an issue which is highly relevant in the context of hardware synthesis. Second, the call-by-name procedure mechanism of Algol does not require closures and can be, therefore, synthesised inexpensively.

POPL'07 January 17–19, 2007, Nice, France.

We present a class of digital circuits which we call *handshake circuits*, and which are shown to form a closed monoidal category, and therefore provide the appropriate structure for interpreting the affine features of the language. We then show how a further refined class of circuits which we call *simple-handshake circuits* forms a Cartesian sub-category of the previous, and therefore has the right structure for the interpretation of products. This feature is needed for modelling *sharing* of resources in the language. We show that this model of bSCI is sound relative to an operational definition of the language. This is also a proof of correctness for the synthesis of digital circuits from bSCI programs.

The most striking feature of this synthesis technique is the simplicity of the resulting circuitry. Abstraction and application are synthesised simply as wiring of the circuit representing the body of the function to that of the argument. Some of the other operations, such as sequential composition, assignment and dereferencing of variables reduce also to wiring. The rest of the constructs of the language require only a handful of basic circuits.

An important issue in hardware synthesis is that of concurrency. Because hardware is naturally concurrent, the implementation of concurrent programs is no more expensive than that of sequential programs. Concurrency in the framework of bSCI does not allow shared-variable inter-process communication, and we examine several pragmatic options for overcoming this limitation. Some of these techniques are potentially unsound.

As a proof of concept, we have implemented a prototype compiler from bSCI to Verilog specifications of VLSI circuits, based on a naive realisation of handshake circuits.

2. The base language

The issue of *physical resource* has long been recognised as crucial in current approaches to programming languages. This notion most commonly refers to *memory locations*, especially in conjunction with heap management, but in its most general instance it refers to any computationally and logically meaningful interaction between software and the underlying machinery. Managing the use of resources in programs, through type systems and logics, has been an active area of research for some time.

The importance of resource management in hardware synthesis becomes paramount. Synthesis, in its purest form, represents a complete shift from the logical realm of software to the physical world of circuitry. Every sub-term of the program, when synthesised, becomes a physical resource which must be managed.

Physical resources, unlike their logical counterparts, cannot be as easily created, duplicated and especially shared. This last point is painfully obvious in writing programs that deal with dynamic memory. Writing such programs correctly is difficult precisely because of the subtle issues that arise in sharing physical resources, memory

locations in this case. In synthesis, as every sub-term of a program becomes a physical entity, every program interaction, more precisely every procedure call, involves some potentially dangerous sharing of circuitry.

These considerations motivate our choice of *Basic SCI (bSCI)* [16, Sec. 7.1]. We first introduce this language, then we show a different but equivalent presentation intended to make the issues related to sharing even more explicit.

The primitive types of the language are commands, memory cells and (boolean) expressions: $\sigma ::= \mathsf{com} \mid \mathsf{cell} \mid \mathsf{exp}$. The static nature of hardware forces us to use a bounded data type. For simplicity we only deal with booleans, but bounded integers can be added in a straightforward way.

Additionally, the language contains function types and products:

$$\theta ::= \sigma \mid \theta \times \theta' \mid \theta \to \theta.$$

What is peculiar about the types above is that pairs of terms *may* share identifiers but functions *may not* share identifiers with their arguments. This is made explicit by the following typing rules (also known as the affine λ-calculus).

Terms have types, described by typing judgments of the form $\Gamma \vdash M : \theta$, where $\Gamma = x_1 : \theta_1, \dots x_n : \theta_n$ is a variable type assignment, M is a term and θ the type of the term.

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

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

$$\frac{\Gamma \vdash F : \theta' \to \theta \qquad \Delta \vdash M : \theta'}{\Gamma, \Delta \vdash FM : \theta} \to \text{Elimination}$$

$$\frac{\Gamma \vdash M : \theta' \qquad \Gamma \vdash N : \theta}{\Gamma \vdash \langle M, N \rangle : \theta' \times \theta} \times \text{Introduction}$$

The language also contains a number of (functional) constants for state manipulation and (structured) control.

$1 : \mathsf{exp}$	constant
$0 : \mathsf{exp}$	constant
$\mathsf{skip} : \mathsf{com}$	no-op
$\mathsf{asg} : \mathsf{cell} \times \mathsf{exp} \to \mathsf{com}$	assignment
$\mathsf{der} : \mathsf{cell} \to \mathsf{exp}$	dereferencing
$\mathsf{seq} : \mathsf{com} \times \mathsf{com} \to \mathsf{com}$	sequencing
$\mathsf{seq} : \mathsf{com} \times \mathsf{exp} \to \mathsf{exp}$	sequencing with boolean
$\mathsf{op} : \mathsf{exp} \times \mathsf{exp} \to \mathsf{exp}$	logical operations
$\mathsf{if} : \mathsf{exp} \times \mathsf{com} \times \mathsf{com} \to \mathsf{com}$	branching
$\mathsf{while} : \mathsf{exp} \times \mathsf{com} \to \mathsf{com}$	iteration
$\mathsf{newvar} : (\mathsf{cell} \to \mathsf{com}) \to \mathsf{com}$	local variable
$\mathsf{newvar} : (\mathsf{cell} \to \mathsf{exp}) \to \mathsf{exp}$	local variable.

Product has syntactic precedence over arrow, which associates to the right. This "functionalised" syntax may seem peculiar but a more conventional syntax can be readily encoded into it.

For now we are omitting parallel composition of commands and recursion, but we shall consider them in later sections.

2.1 Operational semantics

We call terms $\Gamma \vdash M : \theta$ semi-closed if all free identifiers are of type cell. The operational semantics of the language is given by a big-step rule of the form $M, s \Downarrow T, s'$ where M is a semi-closed term, $s : \mathtt{dom}\,\Gamma \to \{0, 1\}$ a *state* and T a *terminal* (0, 1, skip, lambda abstraction).

$$\frac{B, s \Downarrow b, s' \qquad V, s' \Downarrow v, s''}{\mathsf{asg}\langle V, B\rangle, s \Downarrow \mathsf{skip}, (s'' \mid v \mapsto b)}$$

$$\frac{V, s \Downarrow v, s'}{\mathsf{der}\, V \Downarrow s'(v), s'}$$

$$\frac{C, s \Downarrow \mathsf{skip}, s' \qquad M, s' \Downarrow T, s''}{\mathsf{seq}\langle C, M\rangle, s \Downarrow T, s''}$$

$$\frac{M, s \oplus (v \mapsto 0) \Downarrow T, s' \oplus (v \mapsto b)}{\mathsf{newvar}(\lambda v.M), s \Downarrow T, s'}$$

$$\frac{B_1, s \Downarrow b_1, s_1 \qquad B_2, s_1 \Downarrow b_2, s_2}{\mathsf{op}\langle B_1, B_2\rangle, s \Downarrow b, s_2}\ b = b_1\ \mathsf{op}\ b_2$$

$$\frac{B, s \Downarrow b, s' \qquad M_i, s' \Downarrow i, s''}{\mathsf{if}\langle B, M_1, M_0\rangle, s \Downarrow T, s''}$$

$$\frac{B, s \Downarrow 0, s'}{\mathsf{while}\langle B, C\rangle, s \Downarrow \mathsf{skip}, s'}$$

$$\frac{B, s \Downarrow 1, s' \qquad C, s' \Downarrow \mathsf{skip}, s'' \qquad \mathsf{while}\langle B, C\rangle, s'' \Downarrow \mathsf{skip}, s'''}{\mathsf{while}\langle B, C\rangle, s \Downarrow \mathsf{skip}, s'''}$$

$$\frac{M, s \Downarrow \lambda x.M', s}{MM'', s \Downarrow M'[M''/x], s}$$

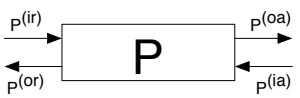
If a term has no free variables we say it is closed. If for a closed term $M, \emptyset \Downarrow T, \emptyset$ we write $M \Downarrow$.

3. A category of digital circuits

We give a denotational semantics for bSCI in terms of digital circuits. The semantics is directly inspired by the game-semantic model for similar languages [4], especially in its automata-theoretic formulation [9]. There are, however important distinctions between the game and digital-circuit semantics, which will be discussed later.

We consider the common conceptual model of (especially asynchronous) VLSI circuits as being defined by an *interface* and by *behaviour*. The interface is a set of *ports*, designated either as *input* or *output*. Ports consume (produce) *signals*, which are called *inputs* (*outputs*). The *behaviour* of a circuit is defined by the way it produces outputs in response to the inputs coming from its environment. Two circuits with the same interface and the same behaviour are considered equal. An input port can be connected to an output port by a *wire*, wich propagates the *signal* after a non-zero bounded *delay*. The notions above should be intuitive and it will help the presentation to maintain a certain level of informality about them. Full formalisations using CSP-like process calculi are quite standard [27], but would make this presentation more opaque for a minimum gain in rigour. We will present such a full formalisation elsewhere.

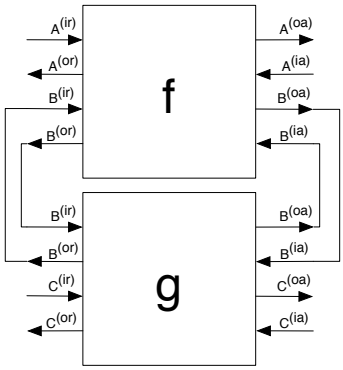
A *handshake circuit* (HC) is a digital circuit where each port has two labels: r(equest) and a(cknowledgement), i(nput) and o(output) $\langle P, l{:}P \to \{i, o\} \times \{r, a\}\rangle$. By convention, we draw such circuits with the r-ports on the left and a-ports on the right; we will denote the input/output polarity by arrows.



We write $A^{(\mathsf{i})} = \{p \in P \mid l(p) \in \{\mathsf{ir}, \mathsf{ia}\}\}$ and so on.

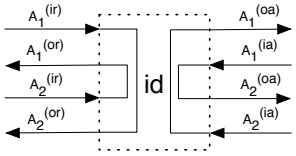
We define a closed-monoidal category of HCs in the following way:

- *Objects* are sets of ports with polarities as defined above.
- *Morphisms* $f : A \to B$ are circuits with sets of ports:
$$f^{(\mathsf{i})} = A^{(\mathsf{o})} \uplus B^{(\mathsf{i})}, \qquad f^{(\mathsf{o})} = A^{(\mathsf{i})} \uplus B^{(\mathsf{o})},$$
$$f^{(\mathsf{r})} = A^{(\mathsf{r})} \uplus B^{(\mathsf{r})}, \qquad f^{(\mathsf{a})} = A^{(\mathsf{a})} \uplus B^{(\mathsf{a})}.$$
- *Composition* of HCs $f : A \to B$ and $g : B \to C$ is the circuit $g \circ f : A \to C$ defined by connecting f and g in the following way:



Note that the ports labeled by B become internal *channels* and are no longer part of the interface.

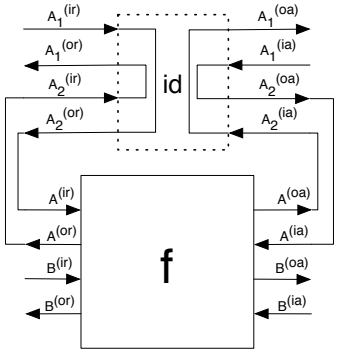
- *Identity* is a HC $id_A : A_2 \to A_1$ (we use tags 1 and 2 to distinguish between argument ports and result ports) of the following shape:



PROPOSITION 1. *HCs form a category.*

Proof:

- Composition is well defined, by inspecting the diagram.
- Composition is associative. $f \circ (g \circ h) = (f \circ g) \circ h$ as they are both equal to the circuit in Fig. 1.
- Identity is an idempotent. The diagram for $f \circ id$ shows that immediately (straightening the wires):



$id \circ f$ has a similar diagram.

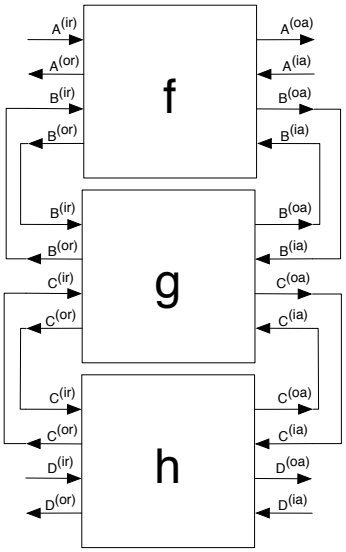
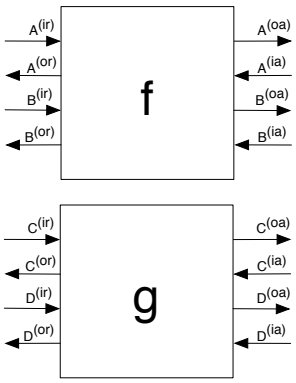


Figure 1. Associativity

We call the category of handshake circuits $\mathbf{HC}$.

The monoidal structure is defined by the functor $- \otimes -$ defined by:

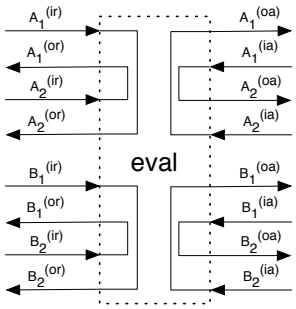
- The unit object I is the empty set of ports.
- On objects, $(A \otimes B)^{(x)} = A^{(x)} \uplus B^{(x)}$ where $x \in \{\mathsf{i}, \mathsf{o}, \mathsf{r}, \mathsf{a}\}$.
- On morphisms $f \otimes g : A \otimes C \to B \otimes D$ is



PROPOSITION 2. $\mathbf{HC}$ *with* $\otimes$ *and* I *is a monoidal category.*

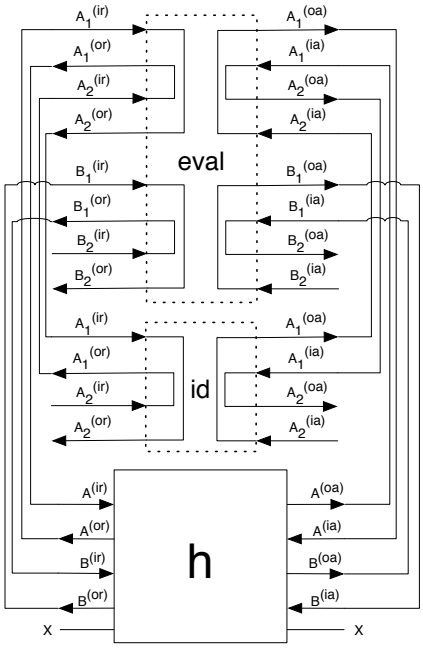
The closed structure is defined as follows:

- On objects, let
$$(A \Rightarrow B)^{(\mathsf{i})} = A^{(\mathsf{o})} \uplus B^{(\mathsf{i})}, \qquad (A \Rightarrow B)^{(\mathsf{o})} = A^{(\mathsf{i})} \uplus B^{(\mathsf{o})},$$
$$(A \Rightarrow B)^{(\mathsf{r})} = A^{(\mathsf{r})} \uplus B^{(\mathsf{r})}, \qquad (A \Rightarrow B)^{(\mathsf{a})} = A^{(\mathsf{a})} \uplus B^{(\mathsf{a})}.$$
- For each A, B let the evaluation morphism $\mathrm{eval}_{A,B} : A_1 \otimes (A_2 \Rightarrow B_1) \to B_2$ be the circuit:



PROPOSITION 3. $\mathbf{HC}$ *with* $-\otimes-$, I, $-\Rightarrow-$, $\mathrm{eval}_{-,-}$ *is a monoidal closed category.*

Proof: The universal property property that for every morphism $f : A \otimes X \to B$ there exists an unique morphism $h : X \to A \Rightarrow B$ such that $f = \mathrm{eval}_{A,B} \circ (\mathrm{id}_A \otimes h)$ is immediate from the diagram of the composition on the right-hand side:



It is obvious, after straightening the wires, that the only h that can satisfy the equality is f itself, with appropriately relabeled ports, and is unique. This relabeling is in fact the currying isomorphism $\Lambda(-)$.

The category $\mathbf{HC}$ is similar to other diagram-based models of monoidal closed categories, for example in quantum computation [2].

Note that the axioms for a monoidal closed category are satisfied in a purely "structural" way, by considering only the ports and the wirings. The behaviour of the circuits is not important up to this point, as it is safe to assume that structurally equal circuits are also behaviourally equal.

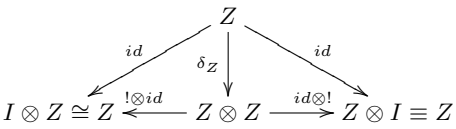
3.1 Cartesian product

To model bSCI we also need a notion of *product*. We will find a sub-category of $\mathbf{HC}$ for which:

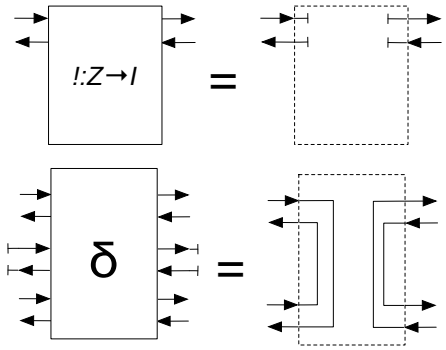
- the unit of the monoidal product is a terminal object;
- for each object there is a diagonal morphism.

It is known that such categories have Cartesian products [17].

DEFINITION 4 (Diagonal). *For an object Z in a monoidal category where the unit is terminal, a diagonal is a morphism $\delta_Z : Z \to Z \otimes Z$ such that the diagram below commutes:*

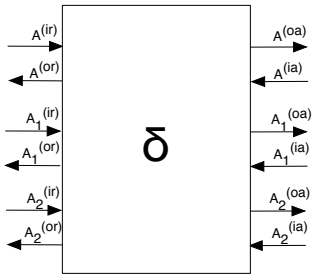


Structurally, this means that the circuits in this sub-category need to satisfy the following equations:



In words, all circuits which are morphisms to I should be equivalent to a circuit with no (open) ports: I does not have ports by designs, and the circuits associated with the domain are left "disconnected." A diagonal $\delta_A : A \to A_1 \times A_2$ with the ports associated with A_1 disconnected (by composition with !) should behave like the identity $id_A : A \to A_2$ (similarly for A_2). For these equations to hold, the behaviour of the circuits becomes relevant.

Let $\delta_A : A \to A_1 \times A_2$ be defined by the circuit



That behaves in the following way:

1. after an input on a port associated with A_i remember the value of i and produce an output on the equivalent port associated with A
2. after an output on a port associated with A produce an output on the equivalent port associated with the memorised i.

It is obvious that the diagonal construction is not well defined for all HCs. What happens, for example, if two consecutive inputs arrive from the two distinct components? Below we will identify a restricted class of HCs, for which the behaviour of the diagonal is well defined, and which form a Cartesian sub-category of $\mathbf{HC}$.

Before we can prove this lemma we need the following definition.

DEFINITION 5. *For object A there is a designated set of input requests I_A called* initial*, such that:*

$$I_{A\otimes B} = I_{A\times B} = I_A \uplus I_B$$

$$I_{A \Rightarrow B} = I_B.$$

DEFINITION 6. *We say that a circuit is a* simple-handshake circuit *(SHC) if it satisfies the following conditions:*

1. *input and output actions must alternate;*
2. *requests and acknowledgments must be well-nested;*
3. *the outermost request is on an initial port;*
4. *there must be no consecutive actions on a given request port without having an intervening action on the acknowledgment port with the same label.*

"Well-nesting" is the same property that well-formed languages of brackets satisfy, with the request (acknowledgment, respectively) on a port with a given label being seen as an open (closed, respectively) bracket with that label.

Note that the assumptions above are both about the circuits themselves and about the environment in which the circuits operate.

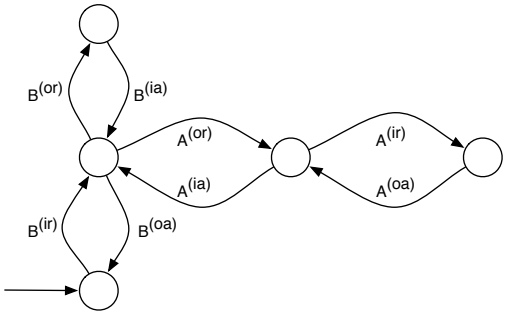
PROPOSITION 7. *Simple-handshake circuits form a Cartesian subcategory of* $\mathbf{HC}$.

Proof: The identity is SHC: The first signal must be on $A_1^{(ir)}$, by Def. 6.3. The following signal is on $A_2^{(or)}$, as Def. 6.1 assumes no other inputs occur before it. After this interaction, there are two possibilities:

- $A_2^{(ir)}$, followed by $A_1^{(or)}$ (by Def. 6.1), then by $A_1^{(ia)}$ (by Def. 6.2) and $A_2^{(oa)}$ (by Def. 6.1). This interaction can repeat (by Def. 6.2), followed by
- a signal on $A_2^{(ia)}$, propagated to $A_1^{(oa)}$,

After which the whole cycle can start again.

We need to show that the composition of SHCs is well defined, i.e. the result is also a SHC. Consider a SHC of shape $f : A \to B$. If we tag any actions of the circuit with $A^{(ir)}, A^{(oa)}, A^{(or)}, A^{(ia)}$, $B^{(ir)}, B^{(oa)}, B^{(or)}, B^{(ia)}$ then the restrictions on the behaviour of the circuit mean that any trace of actions associated with f must belong to the language represented by this automaton:



A SHC $g : B \to C$ has similar traces with appropriately changed labels. The composed circuit $g \circ f : A \to C$ has traces in the composite language given as in Fig. 2. Condition 3 in Def. 6 is satisfied because it is satisfied by g.

Note that the actions associated with B are not *externally observable* since they are no longer ports of the circuit (hence the i/o labels indicate only the relative direction of the action), but they occur on internal channels. This proves that condition 1 in Def. 6 is preserved by composition, and it shows us the general shape of the resulting behaviour.

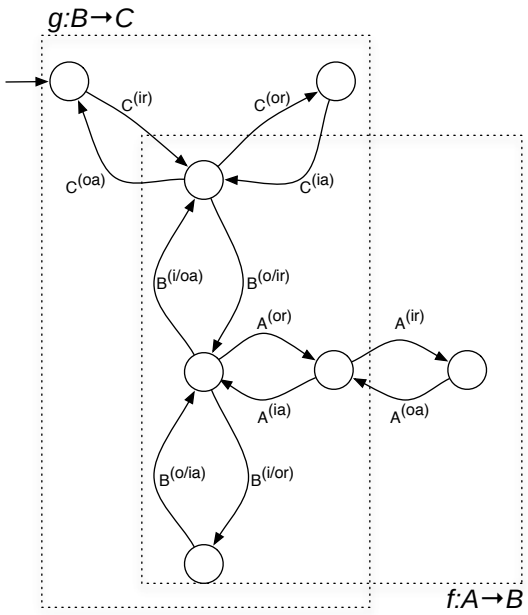


Figure 2. Composition of SHC

For the other conditions, we need to make an additional argument: from the shape of the resulting language we can see that if we project it on the language associated with f (i.e. we remove all other symbols) we must get a string in that language; if we project it on the language associated with g we must get a string in the iterated closure of the language. If the composite language violates conditions 2 or 4 then the projected languages will violate the conditions, which is a contradiction.

The monoidal closed structure is inherited from $\mathbf{HC}$, we only need to show that the product is well defined.

The diagonal morphism and the projections are SHCs (indeed, the very idea of SHCs was intended to accommodate the diagonal morphism!).

The unit I is terminal because all circuits of shape $A \to I$ are equivalent and "inactive" because there is no initial input request port. We denote such morphisms with $! : A \to I$.

We need to show that $(id \otimes !) \circ \delta_A = (! \otimes id) \circ \delta_A = id_A$. The circuit for the composition is shown in Fig. 3. We can see that this simply equates to relabeling all A_1 ports and "hiding" or "blocking" the A_2 ports. δ then propagates the actions between the corresponding A and A_1 ports, just like the identity. The SHC restrictions ensures that δ is always well-behaved.

□

The projections are $\pi_0 = id \otimes ! : A_0 \times A_1 \to A_0$ and $\pi_1 = ! \otimes id : A_0 \times A_1 \to A_1$.

We will refer to the Cartesian, monoidal-closed category of simple handshake circuits as $\mathbf{SHC}$. It offers all the necessary structure that is required to interpret bSCI.

4. Interpreting bSCI

Types of bSCI are interpreted by objects in the category **SHC**. For the base types, the interpretations are:

$$[\![\mathsf{com}]\!] = \{R \mapsto (ir), D \mapsto (oa)\}$$

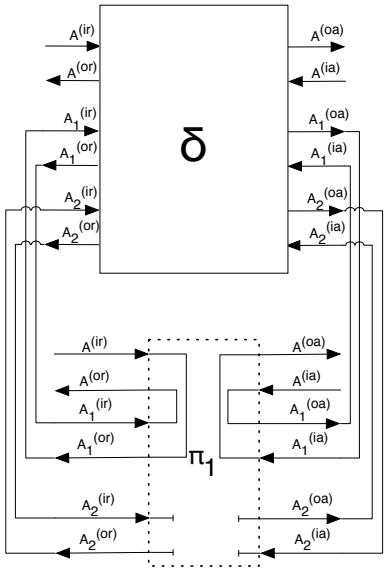


Figure 3. Projections

$$\begin{aligned}[\![\mathsf{exp}]\!] =& \{Q \mapsto (ir), T \mapsto (oa), F \mapsto (oa)\}\\ [\![\mathsf{cell}]\!] =& \{WT \mapsto (ir), WF \mapsto (ir), Q \mapsto (ir), D \mapsto (oa),\\ & T \mapsto (oa), F \mapsto (oa)\}.\end{aligned}$$

with $I_{[\![\mathsf{com}]\!]} = \{R\}, I_{[\![\mathsf{exp}]\!]} = \{Q\}, I_{[\![\mathsf{cell}]\!]} = \{WT, WF, Q\}$. For other types, the interpretations are:

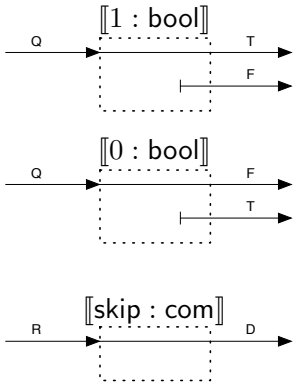
$$[\![\theta \to \theta']\!] = [\![\theta]\!] \Rightarrow [\![\theta']\!], \quad [\![\theta \times \theta']\!] = [\![\theta]\!] \times [\![\theta']\!].$$

The reader familiar with game semantics will note that the notion of "port" in a HC corresponds to that of a "move" in game semantics. An action on a port corresponds to a "move occurrence."

Terms $x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta$ are interpreted by morphisms

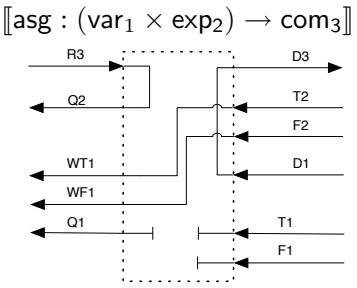
$$\bigotimes_{1 \le i \le n} [\![\theta_i]\!] \xrightarrow{[\![x_1:\theta_1,\ldots,x_n:\theta_n \vdash M:\theta]\!]} [\![\theta]\!].$$

The constants of the language are interpreted by the following circuits:



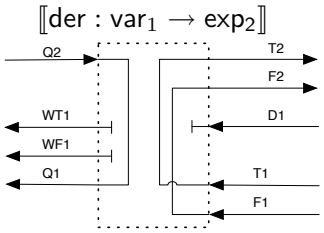
Intuitively, the constant 1 (0, respectively) is interpreted by immediately responding to an input request on the (only such) port Q with an action on port T (F, respectively). The command skip is interpreted by immediately responding to the request. Such behaviour can be most simply interpreted by wires alone. These are proper SHC circuits because Def. 6.1 guarantees that the input signal will propagate to the output before the next input signal will occur.

Assignment has the following interpretation:

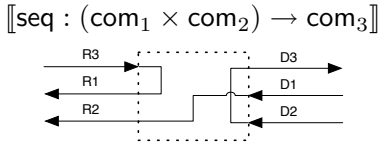


Intuitively, an action on R3, indicating the beginning of execution, leads to an evaluation of the second argument (output request on Q2) which, by Def. 6.2, must followed by an acknowledgment on T2 (F2, respectively) if the value of the expression is 1 (0, respectively). The next output request is "write true", WT, ("write false", WF, respectively) into the first argument variable. Once the write operation is acknowledged, D1, the assignment acknowledges completion, D3. The ports that manage reading from the variable (Q1, T1, F1) are not used in the assignment.

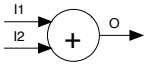
Dereferencing, in contrast, ignores the ports writing to the variable and simply relays the read request and the acknowledged value:



Another functional constant which can be interpreted by wiring only is sequential composition



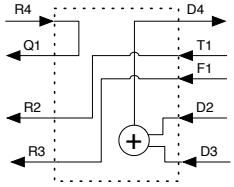
Intuitively, running the sequential composition is done as follows: send a run request (R1) to the first argument. When the first argument acknowledges completion (D1) send a run request (R2) to the second argument. When it completes (D2) the sequential composition acknowledges termination (D3).

In order to interpret branching and iteration we need an auxiliary JOIN circuit with two inputs I1, I2 and one output O. Its behaviour is simply to relay any action on I1 or I2 to O. We denote this circuit by 

.

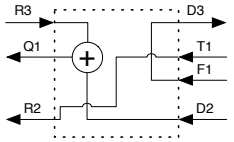
Branching is:

$$[\![\mathsf{if} : (\mathsf{exp}_1 \times \mathsf{com}_2 \times \mathsf{com}_3) \to \mathsf{com}_4]\!]$$



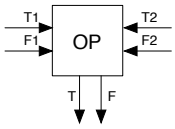
An acknowledgment of true (T1) from the guard of the branching triggers the second argument (R2), whereas a false (F1) triggers the third argument (R3). The branch acknowledges termination (D4) when either of the command arguments terminates (D2, D3). Note that the SHC rules prevent D3 responding to R2 or D2 to R3. For iteration we use:

$$[\![\mathsf{while} : (\mathsf{exp}_1 \times \mathsf{com}_2) \to \mathsf{com}_3]\!]$$



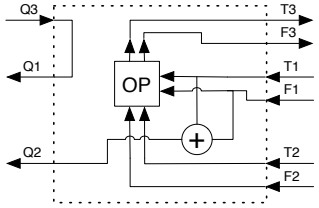
A true (T1) acknowledgment from the guard executes the body of the loop (R2) whereas a false (F1) terminates the loop (D3).

For logical operators we assume the existence of circuits of shape



that produce output on T (respectively F) if and only if the last two inputs were on ports X1 and Y2 and $\mathsf{op}(x_1, y_2) = 1$ (respectively 0); after it produces the output OP must revert to its initial state. Note that circuit OP needs to be stateful, since its inputs are not simultaneous and need to be remembered. Also note that this auxiliary circuit is not itself SHC. The interpretation of logical operator op in the language is the following:

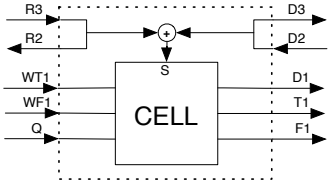
$$[\![\mathsf{op} : (\mathsf{bool}_1 \times \mathsf{bool}_2) \to \mathsf{bool}_3]\!]$$



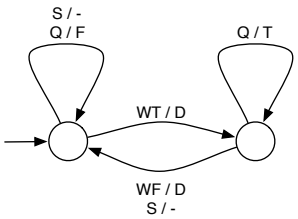
Above we use the same JOIN circuit which was used for branching and iteration. Its role is to propagate any of T1 or F1 input signals to Q2.

Finally, the local state is interpreted by the circuit

$$[\![\mathsf{newvar} : (\mathsf{cell}_1 \to \mathsf{com}_2) \to \mathsf{com}_3]\!]$$



The circuit CELL above is a two-state memory cell: If the input is on WT1 ("write true") it goes to state 1, if it is WF1 ("write false") it goes to state 0. Then it produces output on D1. After a Q request it produces T1 if it is in state 1, F1 if it is in state 0. An input on the S port resets the circuit to its initial state. This behaviour is specified by the following (Mealy-style) automaton:



The structural elements of bSCI are interpreted in the standard way in the category **SHC**:

$$[\![x : \theta \vdash x : \theta]\!] = id_{[\![\theta]\!]}$$
$$[\![\Gamma, x : \theta' \vdash M : \theta]\!] = [\![\Gamma \vdash M : \theta]\!] \circ \pi_1$$
$$[\![\Gamma \vdash \lambda x.M : \theta' \to \theta]\!] = \Lambda([\![\Gamma, x : \theta' \vdash M : \theta]\!])$$
$$[\![\Gamma, \Delta \vdash FM : \theta]\!] = \mathrm{eval} \circ \big([\![\Delta \vdash M{:}\theta']\!] \otimes [\![\Gamma \vdash F{:}\theta' \to \theta]\!]\big)$$
$$[\![\Gamma \vdash \langle M, N\rangle : \theta \times \theta']\!] = \big([\![\Gamma \vdash M{:}\theta]\!] \otimes [\![\rho(\Gamma \vdash N{:}\theta')]\!]\big) \circ \delta_{[\![\Gamma]\!]},$$

where $\rho(\Gamma \vdash N : \theta')$ is the syntactic operation of substituting all free variables from Γ with fresh ones.

We can state that:

LEMMA 8. *The interpretations of bSCI constants are SHC circuits.*

The following property is not required of SHCs, but it holds for all circuits introduced so far:

PROPOSITION 9 (Reset). *For any SHC $[\![\Gamma \vdash M : \theta]\!]$ the internal state of the circuit before an initial input request is the same as after the final output request.*

Proof: Immediate, by structural induction on the syntax of M.

□

We show that this compilation technique is correct through the following soundness theorem.

THEOREM 10 (Soundness). *If $M : \mathsf{com}$ is a closed term and $M \Downarrow$ then $[\![M : \mathsf{com}]\!]$ is equivalent to $[\![\mathsf{skip} : \mathsf{com}]\!]$.*

This is an immediate corollary of a more general following Lemma.

We say that a CELL is *in state B* if a Q input request would produce a B output acknowledgment. We write CELL^B for a cell which is in initial state B.

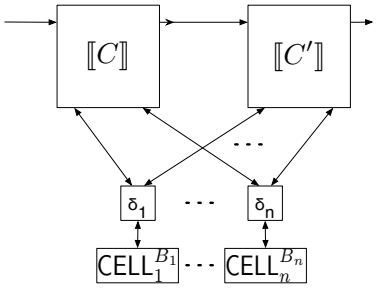
LEMMA 11. *If $\Gamma \vdash M : \sigma$, $\sigma \in \{\mathsf{exp}, \mathsf{com}\}$ is a semi-closed term then for all states $s : \mathrm{dom}\,\Gamma \to \mathbf{B}$ if $M, s \Downarrow c, s'$ then circuit $[\![\Gamma \vdash M : \sigma]\!] \circ \big(\mathsf{CELL}_1^{B_1} \otimes \cdots \otimes \mathsf{CELL}_n^{B_n}\big)$ is equivalent to $[\![c : \sigma]\!]$ and it leaves CELL_j in state B'_j, where $\mathrm{dom}\, s = \{x_1, \ldots, x_n\}$ and $s(x_i) = B_i, s'(x_i) = B'_i$.*

Proof: The proof is by structural induction on the evaluation rules in the operational semantics. Abstraction, application, product and projection rules hold because of the structural properties of **SHC**.

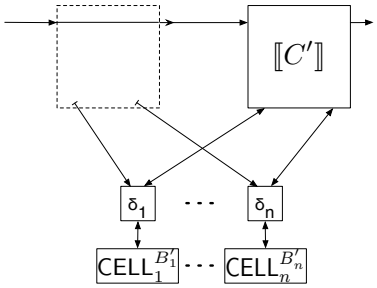
Most constructs have routine proofs. We illustrate the proof for the case of sequential composition of commands. The rule is

$$\frac{C, s \Downarrow \mathsf{skip}, s' \qquad C', s' \Downarrow \mathsf{skip}, s''}{\mathsf{seq}\langle C, C'\rangle, s \Downarrow \mathsf{skip}, s''}$$

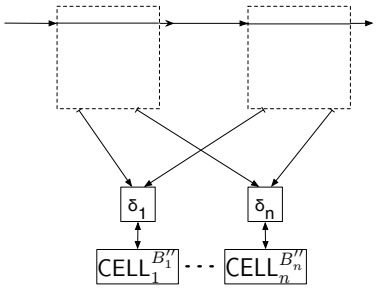
The interpretation of the sequential composition in state s is the following circuit:



We apply the induction hypothesis on C and obtain the equivalent circuit, but with memory cells in new state B'_i.



We then apply the induction hypothesis on C' and obtain

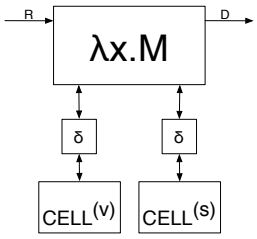


This proves the inductive step for sequential composition, as this circuit is equivalent to skip and leaves the each cell i in state $s''(i)$.

In the case of the local-variable binder:

$$\frac{M, s \oplus (v \mapsto 0) \Downarrow T, s' \oplus (v \mapsto b)}{\mathsf{newvar}(\lambda v.M), s \Downarrow T, s'}$$

we can see that both the hypothesis and the conclusion turn out to be modelled by the same circuit:

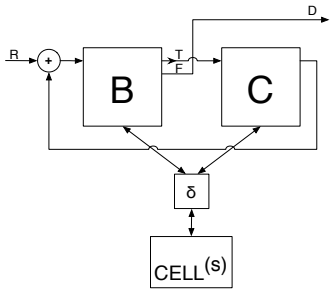


We also sketch out the case of iteration, which is more interesting. The rules for iteration are:

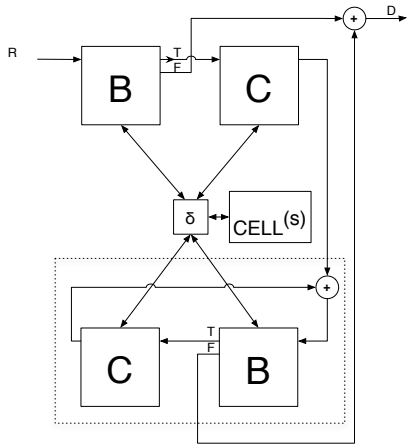
$$\frac{B, s \Downarrow 0, s'}{\mathsf{while}(B, C), s \Downarrow \mathsf{skip}, s'}$$

$$\frac{B, s \Downarrow 1, s' \qquad C, s' \Downarrow \mathsf{skip}, s'' \qquad \mathsf{while}\langle B, C\rangle, s'' \Downarrow \mathsf{skip}, s'''}{\mathsf{while}\langle B, C\rangle, s \Downarrow \mathsf{skip}, s'''}$$

Let us sketch the interpretation of iteration in state s:



The reset property (Prop. 9) ensures that we can rewrite the circuit as below, without changing its behaviour (δ here shares four copies of an identifier):



The equivalent circuit above is actually

$$[\![\Gamma \vdash \mathsf{if}\langle B, \mathsf{seq}\langle C, \mathsf{while}\langle B, C\rangle\rangle\rangle : \mathsf{com}]\!].$$

This leads to an immediate proof by applying the induction hypothesis.

□

The soundness result is a proof of correctness for the compiler from bSCI to SHCs.

5. Concurrency

5.1 Safe concurrency

The language bSCI also has a construct for concurrent composition of commands, which we shall consider now:

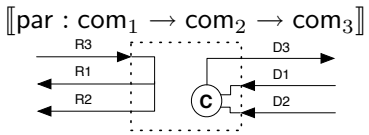
$$\mathsf{par} : \mathsf{com} \to \mathsf{com} \to \mathsf{com}.$$

The contrast between its type and the type of sequential composition ($\mathsf{com} \times \mathsf{com} \to \mathsf{com}$) reflects the restriction that the two arguments may not share identifiers. The operational semantics for this rule is

$$\frac{C_1, s_1 \Downarrow \mathsf{skip}, s'_1 \qquad C_2, s_2 \Downarrow \mathsf{skip}, s'_2}{\mathsf{par}\, C_1\, C_2, s_1 \oplus s_2 \Downarrow \mathsf{skip}, s'_1 \oplus s'_2}$$

Where by $s \oplus s'$ we mean the union of two function with disjoint domains. The rule makes it explicit that the two commands operate on disjoint stores.

The circuit for $[\![\mathsf{par} : \mathsf{com} \to \mathsf{com} \to \mathsf{com}]\!]$ is:



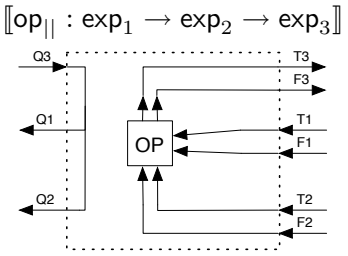
Where the auxiliary circuit C produces output after both input ports have received data (the behaviour of a *Mueller* C-element).

We can see that the circuit above is not a simple handshake circuit, because it sends two output signals (R1, R2) without any intervening input. The semantic model is still sound, and therefore the compiler is still correct, but the characterisation of circuits and their environments is more complex and will not be given here.

Concurrency is very important for hardware synthesis. Computers, even multi-processor versions, are essentially sequential devices, and the execution of concurrent programs raises various theoretical and engineering challenges. In contrast, hardware is inherently concurrent and, as we can see from the circuit above, the synthesis of the parallel execution operator does not raise any difficulties. In fact, it is as easy to synthesise concurrent version of the logical operators as well:

$$\mathsf{op}_{||} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}$$

synthesised as

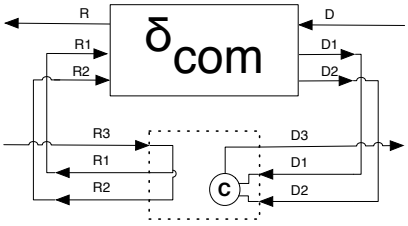


Note that the concurrent version of the operator is even simpler than the sequential version.

5.2 Unsafe concurrency

The bSCI type system prevents race conditions by preventing sharing of identifiers between concurrent sub-programs. In compilation and execution the dangerous consequence of sharing variables in concurrent contexts is that of race conditions: the value stored in a variable is not well determined. For example, shared-variable concurrency is commonly modelled so that programs such as $x := 1; (x := 7 \,||\, x := x - 3)$ may leave variable x, nondeterministically, with values 4, 7, or even -2 (e.g. [6]). More realistic analyses that take into account low-level concurrency issues [22] show that in such programs x may end up with *any* value at all.

If race conditions have dire consequences in programs, they have even more severe consequences in synthesis, because they affect not just variables but whole terms! Consider for example the (untypable) term for $\lambda c : \mathsf{com}.\mathsf{par}\ \mathsf{c}\ \mathsf{c}$. It would lead to the synthesis of circuit

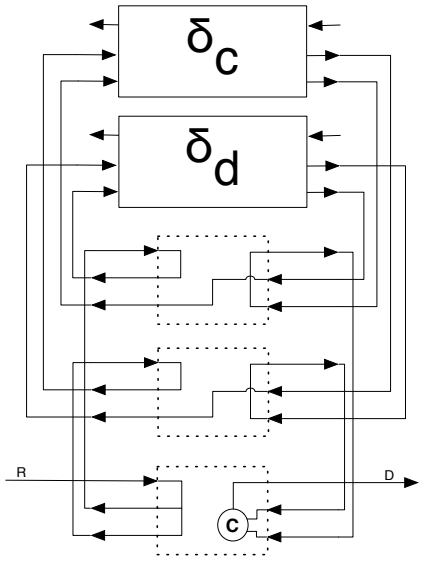


The initial input request on R3 will lead to two simultaneous input requests on δ_{com}, which is illegal, so the behaviour of δ is undefined, therefore the behaviour of the entire program is undefined!

However, programs that cannot be typed do not necessarily have race conditions, and programs that have race conditions are not necessarily misbehaved! Consider, for example, the synthesis of

$$\lambda c : \mathsf{com}.\lambda d : \mathsf{com}.(c; d) \,||\, (d; c)$$

which is both untypeable and it can introduce race conditions between circuits bound to c and d.



However, as it can be seen in the circuit, if c and d are bound to circuits that do not, in turn, share variables *and* c and d have equal input/output delays (or are somehow synchronised) then the diagonals δ_c, δ_d will function correctly and the circuit will behave correctly.

The significant additional expressivity of shared-variable concurrency over "safe" concurrency probably justifies allowing it in the language even at the risk of undefined behaviour in the presence of racing conditions. The onus for correctness will lay heavier on the programmer. Perhaps resource-management logics could be helpful in this regard [18].

It is also possible to find a middle way between totally unsafe and totally safe concurrency. For example, the diagonal morphism could be "safely" synthesised to function as a semaphore: if two input requests arrive simultaneously one of them would have to wait until the first input receives its acknowledgment. This, of course, would restrict the amount of overall concurrency in the presence of race conditions by replacing it with nondeterminism. This solution would also require much more sophisticated and expensive implementations of the diagonal morphism.

5.3 More on affine typing

In the previous section we have seen that concurrency raises important issues, and that affine typing is only one of the possible solutions. It is a safe and elegant solution, but at the cost of greatly reduced expressiveness, thus not entirely satisfactory.

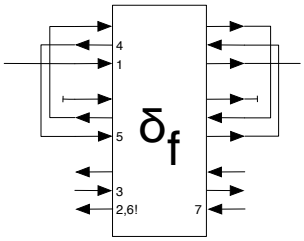
So why is affine typing really needed?

The most serious problem to handle in synthesis, perhaps unsurmountable, is that of nested function self-application, as it involves subtle interleavings of inputs and output on the same physical cir-

cuit. Consider this term (known as a *Kiersetead term*):

$$\lambda f : (\mathsf{com} \to \mathsf{com}) \to \mathsf{com}.f(\lambda x : \mathsf{com}.f(\lambda y : \mathsf{com}.x))$$

This term does not have affine typing. If we apply the synthesis procedure and straighten all the loops in the wiring we obtain the following circuit:



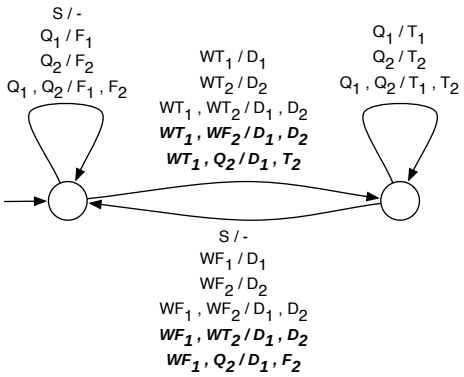
We label the ports of the diagonal circuit that shares the two occurrences of $f : (\mathsf{com} \to \mathsf{com}) \to \mathsf{com}$ with numbers indicating the order in which various inputs and outputs are triggered. (The succession of events 2, 3 is allowed and it will happen if f the function is applied to a term that is non-strict.) The problem is that output requests 2 and 6 occur on the same port without any intervening acknowledgment. This is a fundamental violation of the SHC or DHC requirements and will lead to undefined behaviour in the circuit. Also, if event 7 occurs it is impossible to tell whether it is an acknowledgment for 2 or for 6.

Unlike the problems related to concurrency, which can have pragmatic solutions that make sense at least from an engineering point of view, this seems to be a fundamental difficulty. The question is open whether affine typing is not too strict, since there are terms with nested self-application (e.g. $\lambda f.f(f(\mathsf{skip}))$) that seem to lead to well-behaved circuits.

Finally, if one wishes to relax the constraints of the typing system for concurrency but not for nested self-application it is possible to replace the new-variable binder $\mathsf{newvar} : (\mathsf{cell} \to \mathsf{com}) \to \mathsf{com}$ with a family of new-variable binders

$$\mathsf{newvar}_n : (\mathsf{cell} \to \cdots \to \mathsf{cell} \to \mathsf{com}) \to \mathsf{com},$$

that bind n *distinct* variable-identifiers in (possibly concurrent) contexts to the *same* physical memory cell. This trick can also allow the introduction of semaphores (which must be shared between concurrent contexts) without breaking the affine typing rules. The behaviour is the usual one for a memory cell if the read and write requests do not race, and some arbitrary behaviour if there are race conditions. For example, the (Mealy-style) automaton below illustrates such behaviour for a CELL circuit used to implement $[\![\mathsf{newvar}_2 : (\mathsf{cell}_1 \to \mathsf{cell}_2 \to \mathsf{com}) \to \mathsf{com}]\!]$:



The transitions that correspond to race conditions are highlighted.

6. Recursion

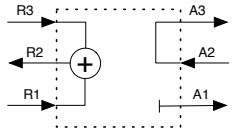
The restrictions of the type system together with the finite-state restriction on the circuits allow only for limited forms of recursion: *mutual ground-type tail-recursion*. The recursion construct is

$$\mathsf{rec}_\theta : (\theta \to \theta) \to \theta,$$

where $\theta ::= \sigma \mid \theta \times \theta$.

Informally, tail-recursion means that the recursive call must occur "last" in the body of the procedure. This effectively reduces the recursion to iteration. We will not formalise the notion of tail recursion, but will only make an informal argument for the soundness of the recursion circuit:

$$[\![\mathsf{rec}_\theta : (\theta_1 \to \theta_2) \to \theta_3]\!]$$

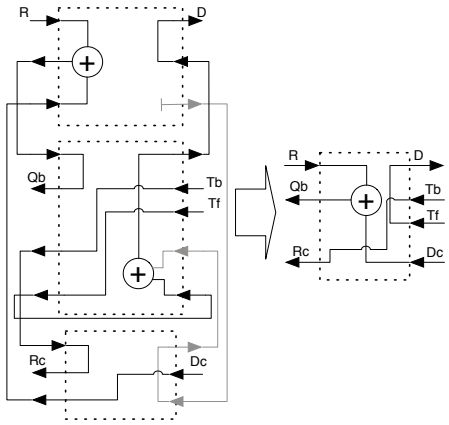


The ground type restriction ensures that each type has only input (or output) requests (or acknowledgments). Intuitively, the recursive call is from R1 to R2 and the tail-return is from A2 to A3; it is a tail return because termination of the argument results in immediate termination of the recursion operator, rather than a return to the calling function.

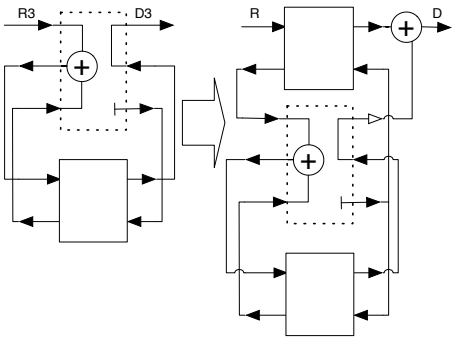
We can see how this recursion operator, applied to the circuit for

$$b : \mathsf{exp}, c : \mathsf{com} \vdash \lambda x.\mathsf{if}\langle b, \mathsf{seq}\langle c, x\rangle, \mathsf{skip}\rangle$$

gives a circuit equivalent to iteration (the grayed-out wires are never active):



Recursion is guaranteed to be sound only when applied to closed terms. Informally, the argument for the soundness of the recursion operator is similar to that for the soundness of the iteration rule: the reset rule allows us to make a copy of the circuit being iterated over, and we can apply the induction hypothesis on the resulting, equivalent circuit. In the diagram below, note that a general recursion operator (not tail recursive) would return (the output port marked with an outline arrow) into the previous instance of the circuit representing the body of the function, rather than make a global return. Also note that it is important that the duplicated circuit does not have open ports.



An example of term that leads to unsound recursion is

$$b : \mathsf{exp}, c : \mathsf{com} \vdash \lambda x.\mathsf{if}\langle b, \mathsf{par}\langle c, x\rangle, \mathsf{skip}\rangle,$$

as it can lead to a race condition on the port associated with c.

It is an open question what kind of more expressive recursion operators are compatible with hardware synthesis.

7. Hardware synthesis

For actual synthesis, the definition of HCs needs to be refined. We need to define what constitutes a signal on a port and implement the δ, CELL, OP and JOIN circuits accordingly. HCs can be designed to be either synchronous or asynchronous; in the former case they only need to be *locally synchronous*, i.e. HCs can be composed without requiring a global clock. This class of circuits is especially well suited for compositional designs, and their comparative advantages and disadvantages are well studied [23].

We will take a naive and straightforward approach, refining the notion of "action on port P" to a voltage transition along P. This naive approach has a series of well-documented disadvantages [26] (it requires a circuit to remember the state of each input, which is extravagantly expensive) but it is functionally correct, and it gives a proof-of-concept for the technique. The circuits have locally synchronous designs.

We make one simple optimisation in the representation of ports that deal with boolean data (T and F, WT and WF). Instead of using two data ports we use one data port (BD, WD) and one control port (BC, WC). The data port indicates the value and the voltage transition on the control port indicates an action. This is less expensive and can be extended to integers in a simple way.

A naive Verilog implementation for JOIN is:

```
module hsJoin(I1, I2, O, clock);
  input I1, I2, clock;
  output O;
  reg I1p, I2p, O;

    always @(posedge clock)
    begin
      if (I1p != I1)  begin O <= !O; I1p <= I1; end
      if (I2p != I2)  begin O <= !O; I2p <= I2; end
    end
endmodule
```

A naive Verilog implementation for and is:

```
module hsAnd(BD1, BC1, BD2, BC2, BD, BC, clock);
  input BD1, BC1, BD2, BC2, clock;
  output BD, BC;
  reg BC1p, BC2p, BC, BD;

    always @(posedge clock)
    begin
      if (BC1 != BC1p)
        begin BC1p <= BC1; BD <= BD1; end
      if (BC2 != BC2p)
        begin BC2p <= BC2; BC <= !BC;
        BD <= BD && BD2; end
    end
endmodule
```

Other logical and arithmetic operations are similar by replacing the final *and* operation (`&&`) with the desired operation.

A naive Verilog implementation for CELL is:

```
module hsCell(WD, WC, D, Q, BD, BC, S, clock);
  input WD, WC, Q, S, clock;
  output D, BD, BC;
  reg WCp, Qp, Sp, D, BC, BD;

    always @(posedge clock)
    begin
      if (WC != WCp)
        begin WC <= WCp; BD <= WD; D <= !D end
      if (Q != Qp)
        begin Q <= Qp; BC <= !BC; end
    end
endmodule
```

For δ each type requires a different implementation, we only show the simplest one, for com.

```
module hsDiagCom(R, D, R1, D1, R2, D2, clock)
  output R, D1, D2, clock;
  input D, R1, R2;
  reg R1p, R2p, Dp, R, D1, D2, state;

    always @(posedge clock)
    begin
      if (R1p != R1)
        begin state <= 1; R1p <= R1; R <= !R; end
      if (R2p != R2)
        begin state <= 0; R2p <= R2; R <= !R; end
      if (D != Dp && state)
        begin Dp <= D; D1 <= !D1; end
      if (D != Dp && !state)
        begin Dp <= D; D2 <= !D2; end
    end
endmodule
```

These circuits have the advantage that they do not need initialisation, often a problem in hardware design.

For realistic implementation it is required to use cleverer and more efficient refinements for actions, such as multi-phase encodings as well as various optimisations [26].

We have implemented a prototype compiler from bSCI to Verilog using this technique. For example, the circuit synthesised for program $\lambda f.\lambda g.\lambda x.f(g(x)); g(f(x))$ has block-schematic and technology schematics as in Fig. 4. The two larger block circuits are diagonals for com $\rightarrow$ com and the smaller block circuits are the diagonal for com and the implementation of skip (it only contains wires). Verilog synthesis has been executed using the Xilinx ISE package.

8. Related work

Compilation techniques usually rely on operational semantic ideas, but denotational-based techniques have been proposed before,

e.g. Reynolds's work on compiling Algol using functor categories [21]. Although the two underlying models have little in common, dissimilarity reflected by the target architecture, one of the principal objectives is shared, providing a compelling and accessible computational intuition of the essential features of the semantic model.

The category of simple-handshake circuits is obviously related to and inspired by the idea of strategy in game semantics [3, 11], the notion of *action* we use corresponds to that of *move occurrence* and the notion of *port* to that of *move*. The main technical difference is that game models are usually targeted towards *definability* results, ensuring that all semantic objects correspond to terms. This requires tighter constraints on what is considered acceptable behaviour of the environment and also more precise descriptions of what is the possible behaviour of a program. Many of these considerations are not relevant if the aim is soundness only.

Although no precise connections are drawn in this work, there are obvious parallels between our circuit semantics and work on abstract machines based on game semantics (such as the *Token Abstract Machine* [8]) and Geometry of Interaction [13]. The emphasis of this paper is more practical though. We aimed for a technique that starts with a (fairly realistic) programming language and ends with VLSI-synthesisable circuitry. However, a closer inspection of these connection may be useful from an applied point of view, especially in regards to devising less restrictive type systems for the programming language.

Even more closely related in this sense is Mackie's work on compiling functional programs using ideas from the Geometry of Interaction [12, 13]. Also, Abramsky's recent work on structural approaches to reversible computation [1] and quantum computation [2] shares similar aims, although they all look at different target architectures.

There is a vast amount of literature concerning hardware synthesis from higher-level languages. One of the most successful approaches is Mycroft and Sharp's work on *statically allocated functional languages* [14, 15]. This line of work uses fundamentally different techniques than ours, but it shares an identical aim. It would be a useful exercise to compare circuits synthesised from similar programs using these two techniques.

Most of the rest of the work concerning higher-level synthesis techniques is not directly comparable to this approach, as it involves either structural layout techniques (such as Lava [5]), design languages based on process calculi (such as Balsa [25]) or lower-level languages more similar in spirit to hardware description language (such as Haendel-C [7] or SystemC [24]).

Acknowledgments Guy McCusker and Peter O'Hearn made important observations regarding an earlier draft of this work. Alan Mycroft patiently explained the drawbacks of the one-phase handshake protocol. Qianyi Zhang and Xu Wang provided much needed guidance and help in using digital design tools. I gratefully acknowledge their contributions to this work.

References

[1] ABRAMSKY, S. A structural approach to reversible computation. *Theor. Comput. Sci. 347*, 3 (2005), 441–464.

[2] ABRAMSKY, S., AND COECKE, B. A categorical semantics of quantum protocols. In *LICS* (2004), pp. 415–425.

[3] ABRAMSKY, S., JAGADEESAN, R., AND MALACARIA, P. Full abstraction for PCF. *Inf. Comput. 163*, 2 (2000), 409–470.

[4] ABRAMSKY, S., AND MCCUSKER, G. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions. *Electr. Notes Theor. Comput. Sci. 3* (1996).

[5] BJESSE, P., CLAESSEN, K., SHEERAN, M., AND SINGH, S. Lava: hardware design in haskell. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 1998), ACM Press, pp. 174–184.

[6] BROOKES, S. D. Full abstraction for a shared variable parallel language. In *LICS* (1993), pp. 98–109.

[7] CELOXICA. *Handel-C Reference Manual*. `http://www.celoxica.com`.

[8] DANOS, V., HERBELIN, H., AND REGNIER, L. Game semantics & abstract machines. In *LICS* (1996), pp. 394–405.

[9] GHICA, D. R., AND MCCUSKER, G. Reasoning about idealized Algol using regular languages. In *ICALP* (2000), pp. 103–115.

[10] GIRARD, J.-Y. Geometry of interaction (abstract). In *CONCUR* (1994), p. 1.

[11] HYLAND, J. M. E., AND ONG, C.-H. L. On full abstraction for PCF: I, II, and III. *Inf. Comput. 163*, 2 (2000), 285–408.

[12] MACKIE, I. *The Geometry of Implementation*. PhD thesis, Imperial College, University of London, 1994.

[13] MACKIE, I. The geometry of interaction machine. In *POPL* (1995), pp. 198–208.

[14] MYCROFT, A., AND SHARP, R. A statically allocated parallel functional language. In *ICALP* (2000), pp. 37–48.

[15] MYCROFT, A., AND SHARP, R. Higher-level techniques for hardware description and synthesis. *STTT 4*, 3 (2003), 271–297.

[16] O'HEARN, P. W. On bunched typing. *J. Funct. Program. 13*, 4 (2003), 747–796.

[17] O'HEARN, P. W., POWER, J., TAKEYAMA, M., AND TENNENT, R. D. Syntactic control of interference revisited. *Theor. Comput. Sci. 228*, 1-2 (1999), 211–252.

[18] O'HEARN, P. W., AND PYM, D. J. The logic of bunched implications. *Bulletin of Symbolic Logic 5*, 2 (1999), 215–244.

[19] REYNOLDS, J. C. Syntactic control of interference. In *POPL* (1978), pp. 39–46.

[20] REYNOLDS, J. C. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages* (1981), North-Holland, pp. 345–372.

[21] REYNOLDS, J. C. Using functor categories to generate intermediate code. In *POPL* (1995), pp. 25–36.

[22] REYNOLDS, J. C. Toward a grainless semantics for shared-variable concurrency. In *FSTTCS* (2004), pp. 35–48.

[23] SHAPIRO, D. M. *Globally-asynchronous locally-synchronous systems*. PhD thesis, Stanford University, 1984.

[24] SYNOPSYS INC. *SystemC Reference Manual*. `http://www.systemc.org`.

[25] THE UNIVERSITY OF MANCHESTER ADVANCED PROCESSOR TECHNOLOGIES. *BALSA Reference Manual*. `http://www.cs.manchester.ac.uk/apt/projects/tools/balsa/`.

[26] VAN BERKEL, K. *Handshake circuits: An intermediary between communicating processes and VLSI*. PhD thesis, Technische Univ., Eindhoven (Netherlands)., 1992.

[27] VAN BERKEL, K., KESSELS, J., RONCKEN, M., SAEIJS, R., AND SCHALIJ, F. The VLSI-programming language Tangram and its translation into handshake circuits. In *EURO-DAC '91: Proceedings of the conference on European design automation* (Los Alamitos, CA, USA, 1991), IEEE Computer Society Press, pp. 384–389.

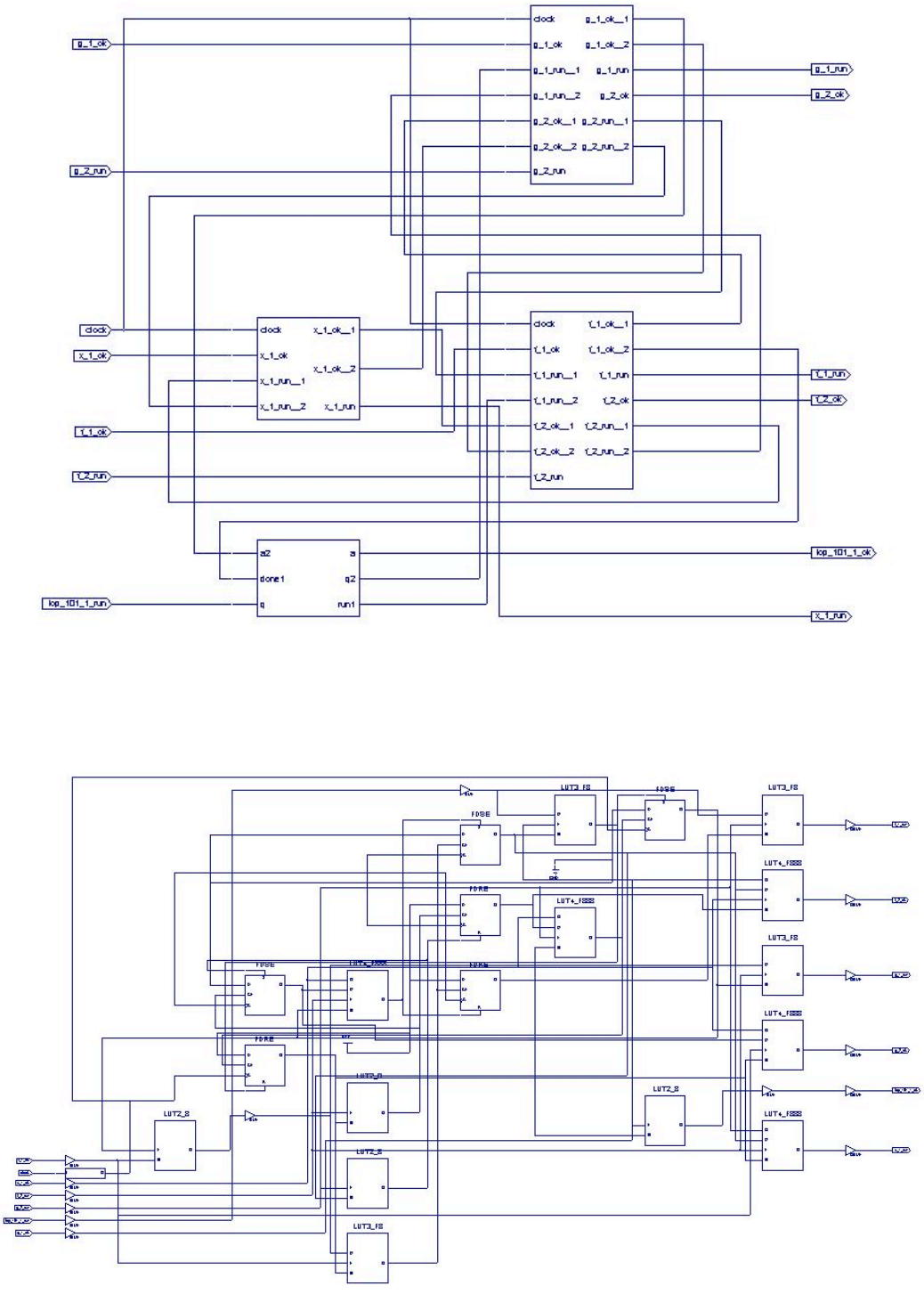


Figure 4. Schematics