Program Transformation for Functional Circuit Descriptions

Manfred Schmidt-Schauß and David Sabel

Fachbereich Informatik und Mathematik,
Institut für Informatik, Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany,
{schauss,sabel}@ki.informatik.uni-frankfurt.de

Technical Report Frank-30

15. February 2007

Abstract. We model sequential synchronous circuits on the logical level by signal-processing programs in an extended lambda calculus L_{por} with letrec, constructors, case and parallel or (por) employing contextual equivalence. The model describes gates as (parallel) boolean operators, memory using a delay, which in turn is modeled as a shift of the list of signals, and permits also constructive cycles due to the parallel or. It opens the possibility of a large set of program transformations that correctly transform the expressions and thus the represented circuits and provides basic tools for equivalence testing and optimizing circuits. A further application is the correct manipulation by transformations of software components combined with circuits. The main part of our work are proof methods for correct transformations of expressions in the lambda calculus L_{por}, and to propose the appropriate program transformations.

1 Introduction

Sequential synchronous circuits can be modeled on the logical level by functional programs in Haskell-style using the Lava-approach [BCSS98,CSS03,She05] where wires are modelled as variables, signals as a stream of Boolean values, gates as elementwise list-combining functions, memory by a delay operator, and the circuit as a functional letrec-expression. Using this model, the issue of correctly modifying a sequential circuit or detecting equality of two sequential circuits can make use of the tool of program transformations of non-strict functional programs.
There are lots of useful extensions. (i) As in the Lava-approach, one extension is to have functional programs as circuit-generators; (ii) adding functions modeling the behavior of sequential circuits as black boxes; (iii) adding functions that can be interpreted as software-components in a circuit; (iv) extending the model to cover also synchronous circuits with combinational cycles.

The Lava-system[1] was developed to specify and manipulate hardware-descriptions in a functional language, in fact it is a variant of Haskell. It is designed to specify sequential circuits as programs. There are an interpreter and further tools to manipulate the descriptions and test their functionality. The Lava-system does not really exploit and investigate modifying the nets by program transformations as a tool, since it emphasizes other aspects. There were lots of conjectures on correctness of program transformations, but often the proofs could not be provided. In a paper on the Hawk-system[2], which aims at processor verification, the authors wrote: " Engineers close to current processor design teams inform us that designers purposefully forgo promising optimizations because they cannot guarantee the optimizations preserve correctness."

Our work on extended call-by-need lambda-calculi with letrec, their equational theories and program transformations where we adopt contextual preorder and equivalence providing a maximal set of equations w.r.t. the observations (see [SSSS05,SS06,SS03,SSS07a,SS07a]), match the formal needs. The calculi come in different variants: a lambda-calculus with letrec; a lambda-calculus with letrec, case and constructors which matches the model without combinational cycles if the logical operators are strict; and a lambda-calculus L_{por} with letrec, case, constructors and parallel or, which matches the model if combinational cycles are allowed (see [SBT96,CP02]); and a lambda-calculus with letrec, case, constructors, and an amb (or choice) (see [SSS07a,Mor98,KSS98]), which can be used to encode parallel or and to derive equations in L_{por}. It is easy to guess lots of correct program transformations, however, it is an issue to rigorously prove their correctness within each calculus, which is indispensable for their safe use. We explored several methods for proving an increasing set of program transformations as correct, among them the basic ones [SSSS05,SSS07a]. Recently, we proved that the unrestricted copy rule, (also called instantiation rule) is correct for call-by-need calculi with letrec [SS06], also for a calculus with case and constructors [SS07a] and as a further recent extension, also for L_{por}, i.e., a letrec-calculus with case, constructors and por [SS07b].

Among the circuit- and program transformations that can be justified by our approach are retiming, sharing introduction, partial evaluation, constant folding, constant introduction and Boolean laws; this holds for all of the above mentioned variants of calculi, and hence for the corresponding hardware-software combinations modeled in L_{por}.

Future work is to investigate verification tools based on program transformations for the functional descriptions, to extend the formal methods to prove correctness of more program transformations and to enhance the methods and tools for proving equality of functional expressions. To switch between a por-free lambda calculus (i.e. Haskell), and L_{por} during transformation appears to be inappropriate, since the corresponding semantics is different, and since in the case that a proper asynchronous cycle is introduced using the por-free calculus it is not correct to switch to a different semantics. So we propose to stick to the

[1] Lava: see `http://www.cs.chalmers.se/~koen/Lava/index.html`
[2] Hawk: see `http://www.cse.ogi.edu/PacSoft/projects/Hawk`

L_{por}-theory and transformations, even if there are no cycles. Modeling hardware using lazy functional programming languages such as Haskell [Pey03] as done in Lava, see also [BCSS98,CSS03,She05] opens the possibility after extending it with por to combine hardware descriptions, abstract hardware descriptions and software components, and to exploit the rich set of program transformations of the call-by-need calculi L_{por} to transform the hardware description, and also the software part, perhaps also optimize them, where only correct transformations are used. This may enhance and extend the verification and transformation methods based on Boolean algebra.

1.1 Overview

In section 2 we introduce the calculus $L_{\mathtt{por}}$ and the equational theory based on contextual equivalence. After proving correctness of the reductions involving "parallel or", we show how por can be implemented in an extended variant of Haskell. In section 3 we present our encoding of hardware descriptions in $L_{\mathtt{por}}$ and motivate the introduction of parallel boolean operators on the basis of an exemplary sequential circuit with a constructive cycle. In section 4 we define program transformations for circuit descriptions and demonstrate their usefulness by showing the equivalence of two synchronous circuits.

2 A Call-by-Need Calculus With Letrec, Case, Constructors, and Parallel Or

We describe in detail a calculus that extends the fragment of Haskell used in Lava by a parallel-or. The call-by-need calculus $L_{\mathtt{por}}$ has as primitives binary application, recursive `let`, `por`, `lambda`, `seq`, `case` and constructors, with a normal-order reduction that defines evaluation to weak head normal forms. The calculus $L_{\mathtt{por}}$ is non-deterministic, where the non-deterministic primitive is weaker than `amb` or `choice`, see also [SSSS04,MSC99] for non-deterministic calculi with case and constructors.
We describe the syntax. There is an infinite set V of variables and a finite set $K = \{c_1, \ldots, c_n\}$ of constructors with arities $\mathrm{ar}(c_i)$. The 0-ary constructors `True`, `False`, `Nil` and the binary constructor `(:)` for lists are among the constructors. The constant `Nil` is not used in the further development. The syntax is as follows, where E means expressions:

$$\begin{aligned} E ::= {} & V \mid (E\ E) \mid \lambda x.E \mid (\mathtt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \mathtt{in}\ E) \\ & \mid (\mathtt{por}\ E\ E) \mid (\mathtt{seq}\ E\ E) \mid (c_i\ E_1 \ldots E_{\mathrm{ar}(c_i)}) \\ & \mid (\mathtt{case}\ E\ ((c_1\ V_1 \ldots V_{\mathrm{ar}(c)}) \to E) \ldots ((c_n\ V_1 \ldots V_{\mathrm{ar}(c_n)}) \to E)) \end{aligned}$$

The `case`-construct is assumed to have an alternative pattern $(c_i\ x_1 \ldots x_{\mathrm{ar}(c_i)})$ for every constructor $c_i \in K$, where the variables in a pattern have to be distinct. The scoping rules are as usual, where `letrec` is recursive, and hence the scope of x_1 in $(\mathtt{letrec}\ x_1 = s_1, x_2 = s_2\ \mathtt{in}\ t)$ is the terms s_1, s_2 and t. The sequence

$$\begin{array}{ll}
(s\ t)^M & \rightarrow (s^S\ t)^V \\
(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t)^T & \rightarrow (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t^S)^V \\
(\texttt{letrec}\ x = s, \mathit{Env}\ \texttt{in}\ C[x^S]) & \rightarrow (\texttt{letrec}\ x = s^S, \mathit{Env}\ \texttt{in}\ C[x^V]) \\
(\texttt{letrec}\ x = s, y = C[x^S], \mathit{Env}\ \texttt{in}\ r) & \rightarrow (\texttt{letrec}\ x = s^S, y = C[x^V], \mathit{Env}\ \texttt{in}\ r) \\
 & \quad \text{if } C[x] \neq x \\
(\texttt{por}\ s\ t)^M & \rightarrow (\texttt{por}\ s^S\ t)^V \text{ (non-deterministically)} \\
(\texttt{por}\ s\ t)^M & \rightarrow (\texttt{por}\ s\ t^S)^V \text{ (non-deterministically)} \\
(\texttt{seq}\ s\ t)^M & \rightarrow (\texttt{seq}\ s^S\ t)^V \\
(\texttt{case}\ s\ \mathit{alts})^M & \rightarrow (\texttt{case}\ s^S\ \mathit{alts})^V
\end{array}$$

Fig. 1. Unwinding using labels

of the bindings in the let-environment may be interchanged. We assume that expressions satisfy the distinct variable assumption before reduction is applied, which can be achieved by a renaming of bound variables.
We use labels indicating the normal order redex, where T means the top-term, S means a subterm reduction, V means visited, and M matches S as well as T. The shifting algorithm in figure 1 uses the rules exhaustively, where it fails, if a loop is detected, which happens if a to-be-labelled position already is labeled V, and otherwise it succeeds. The contexts where the hole will be labeled with S are also called *reduction contexts*. In figure 2, a cv-expression is an expression of the form $(c\ x_1 \dots x_n)$ where c is a constructor and x_i are variables. A *value* is an abstraction or a constructor-expression $(c\ t_1 \dots t_n)$. Normal-order reduction rules are defined in figure 2, where we assume that the nondeterministic labeling algorithm was used before. The normal-order reduction is non-deterministic due to the label-shift, however, the result of a normal order reduction sequence is deterministic under fairness assumptions for normal-order reductions.
A weak head normal form (WHNF) is value v or an expression $(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ v)$, where v is a value. A term s *converges*, iff $s \xrightarrow{*} v$ for some WHNF v by a normal order reduction, denoted as $s{\Downarrow}$.

Remark 2.1. An evaluator for $L_{\texttt{por}}$ should ensure *fairness* when evaluating a `por`-expression, i.e. no argument of `por` is always avoided during the unwinding when finding the next redex; this can be reformulated as: in every fair normal-order reduction, every normal-order redex will eventually be reduced or eliminated by a normal-order reduction. Nevertheless, using the same argumentation as in [SSS07a] the predicate $\Downarrow$ remains the same under fair evaluation.

Two terms s, t are related by contextual preorder: $s \leq_c t$, iff $\forall C : C[s]{\Downarrow} \implies C[t]{\Downarrow}$, and s, t are contextually equivalent, $s \sim_c t$, iff $s \leq_c t$ and $t \leq_c s$.
This contextual equivalence is w.r.t. may-convergence, which is no restriction, since we will argue below that must-convergence is identical to may-convergence: A term t must-converges: $t{\Downarrow}_{must}$, iff $\forall t' : t \xrightarrow{*} t' \implies t'{\Downarrow}$. Our definition of contextual equivalence is compatible with definitions using may- and must-convergence as above, and also with may- and must-convergence using fair

(lbeta)	$C[((\lambda x.s)^S\ r)] \to C[(\texttt{letrec}\ x = r\ \texttt{in}\ s)]$
(cp)	$(\texttt{letrec}\ x = v^S\ \texttt{in}\ C[x^V]) \to (\texttt{letrec}\ x = v\ \texttt{in}\ C[v])$
	where v is an abstraction, a variable or a cv-expression
(abs)	$(\texttt{letrec}\ x = (c\ t_1 \dots t_n)^S, \mathit{Env}\ \texttt{in}\ r) \to$
	$(\texttt{letrec}\ x_1 = t_1, \dots, x_n = t_n, x = (c\ x_1 \dots x_n), \mathit{Env}\ \texttt{in}\ r)$
	if $(c\ t_1 \dots t_n)$ is not a cv-expression, where x_i are fresh variables
(case)	$C[(\texttt{case}\ (c\ t_1 \dots t_n)^S \dots ((c\ y_1 \dots y_n) \to s) \dots)]$
	$\to\ C[(\texttt{letrec}\ y_1 = t_1, \dots, y_n = t_n\ \texttt{in}\ s)]$
(seq)	$C[(\texttt{seq}\ v^S\ t)] \to C[t]$ if v is a value
(porlT)	$C[(\texttt{por}\ \texttt{True}^S\ t)^V] \to C[\texttt{True}]$
(porrT)	$C[(\texttt{por}\ s\ \texttt{True}^S)] \to C[\texttt{True}]$
(porlF)	$(\texttt{por}\ \texttt{False}^S\ s) \to s$
(porrF)	$(\texttt{por}\ t\ \texttt{False}^S) \to t$
(llet-e)	$(\texttt{letrec}\ \mathit{Env}_1, x = (\texttt{letrec}\ \mathit{Env}_2\ \texttt{in}\ s)^S\ \texttt{in}\ t)$
	$\to (\texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2, x = s\ \texttt{in}\ t)$
(llet-in)	$(\texttt{letrec}\ \mathit{Env}_1\ \texttt{in}\ (\texttt{letrec}\ \mathit{Env}_2\ \texttt{in}\ s)^S)$
	$\to (\texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ s)$
(lapp)	$C[((\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s)^S\ t)] \to C[((\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (s\ t))]$
(lseq)	$C[(\texttt{seq}\ (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s)^S\ t)] \to C[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{seq}\ s\ t))]$
(lporl)	$C[(\texttt{por}\ (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s)^S\ t)] \to C[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{por}\ s\ t))]$
(lporr)	$C[(\texttt{por}\ s\ (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t)^S)] \to C[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{por}\ s\ t))]$
(lcase)	$C[(\texttt{case}\ (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t)^S\ \mathit{alts})] \to C[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{case}\ t\ \mathit{alts}))]$

Fig. 2. Normal-order rules

normal-order reductions (for a similar discussion in another non-deterministic calculus, see [SSS07a]). Note, however, that using strong must-convergence – i.e. all normal order reduction sequences have to terminate with success – in the definition would have the strange consequence that `por True` $\bot$ is not equivalent to `True`, whereas w.r.t. our definition `por True` $\bot \sim_c$ `True` holds.

There is a rich set of transformations for the calculus L_{por}:

Partial evaluation: i.e. all reduction rules from figure 2 in any context ignoring the S-labels can be used, including the por-rules. There are further transformation rules in figure 3.

Proving correctness of the reduction rules as transformations can be done as e.g. in [SSSS04] using diagrams and a context lemma. We are confident that this will work out without problems. The *context-lemma* for $L_{\texttt{por}}$ states that it is sufficient to take into account the class of reduction contexts for the proof of contextual equivalence: $(\forall R : R[s]\Downarrow \implies R[t]\Downarrow) \implies s \leq_c t$. Fortunately $L_{\texttt{por}}$ meets the generic properties of [SSS07b] and hence the context lemma follows immediately from the results in [SSS07b]. The correctness of (gc1), (gc2) can be proved using the operational approach by diagrams as exemplified in [SSS07a]. The correctness of (cpall1), (cpall2) follows from [SS07b], and the correctness of the por-rules is proved in the next subsection. Correctness of common subexpression elimination (csexel) follows by combining a reversed (*cpall*1) and (*gc*).

(cpall1) $(\texttt{letrec}\ x = s, Env\ \texttt{in}\ t) \rightarrow (\texttt{letrec}\ x = s, Env[s/x]\ \texttt{in}\ t[s/x])$
(cpall2) $(\texttt{letrec}\ x = s, Env\ \texttt{in}\ t) \rightarrow (\texttt{letrec}\ x = s[s/x], Env\ \texttt{in}\ t)$
(gc1) $(\texttt{letrec}\ Env\ \texttt{in}\ t) \rightarrow t$
if the defined variables in Env do not occur in t
(gc2) $(\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ t) \rightarrow (\texttt{letrec}\ Env_2\ \texttt{in}\ t)$
if the defined variables in Env_1
do not occur in t nor in Env_2
(csexel) $C[s,s] \rightarrow \texttt{letrec}\ x = s\ \texttt{in}\ C[x,x]$, where s is closed

Fig. 3. Correct transformations

2.1 Correctness of por-Reductions

The goal of this subsection is to prove that (por) := (porlT)∪(porlF)∪(porrT)∪ (porrF) is a correct program transformation provided the other transformations are already proved as correct, i.e. if $s \xrightarrow{por} t$ then $s \sim_c t$. The proof of correctness splits into two parts:

- If $s \xrightarrow{por} t$, then for all contexts C: $C[s]\Downarrow \implies C[t]\Downarrow$
- If $s \xrightarrow{por} t$, then for all contexts C: $C[t]\Downarrow \implies C[s]\Downarrow$

Due to the context lemma it is sufficient to show

- If $s \xrightarrow{por} t$, then for all reduction contexts R: $R[s]\Downarrow \implies R[t]\Downarrow$
- If $s \xrightarrow{por} t$, then for all reduction contexts R: $R[t]\Downarrow \implies R[s]\Downarrow$

The second part follows easily, since every (por)-reduction inside a reduction context is also a normal order reduction, i.e. if $s \xrightarrow{por} t$ then the reduction $R[s] \rightarrow R[t]$ is always a normal order reduction. If $R[t]\Downarrow$ then there exists a WHNF t' with $R[t] \xrightarrow{no,*} t'$. Appending this sequence of normal order reductions to $R[s] \xrightarrow{R,por} R[t]$ we have $R[s]\Downarrow$.
The other direction is harder to prove. We need to compute the overlappings of a normal order reduction with a (por)-reduction inside a reduction context. Unfortunately the class of reduction contexts is to small for closing the overlappings, since normal-order redexes are not unique (see [SSS07a]). Therefore we compute the overlappings w.r.t. to (por)-reduction inside *surface contexts S*, where a surface context is a context where the hole is not inside the body of abstraction.
A case analysis gives the following nontrivial overlappings, where full arrows are given transformations and dashed arrows are existentially quantified:

$$\begin{tikzcd}
\cdot \arrow[r, "S{,}por"] \arrow[d, "no{,}a"'] & \cdot \arrow[d, dashed, "no{,}a"] \\
\cdot \arrow[r, dashed, "S{,}por"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "no{,}por"] \arrow[d, "no{,}a"'] & \cdot \\
\cdot \arrow[ur, dashed, "no{,}por"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "S{,}por"] \arrow[d, "no{,}a"'] & \cdot \arrow[dl, dashed, "no{,}a"] \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[rr, "S{,}por"] \arrow[dd, "no{,}lpor"'] & & \cdot \\
& \cdot \arrow[ur, dashed, "S{,}gc"'] & \\
\cdot \arrow[ur, dashed, "S{,}por"'] & &
\end{tikzcd}$$

The first diagram describes the commuting cases. The second diagram covers the cases where the (S, por)-transformation removes the normal order redex, where the (S, por)-reduction must be a normal order reduction. The third diagram describes the cases where a normal order reduction removes the redex of the (S, por)-transformation. An example for the last diagram is

$$\begin{tikzcd}
\texttt{por (letrec } x_1 = s_1 \texttt{ in } s_2\texttt{) True} \arrow[rr, "S{,}por"] \arrow[dd, "no{,}lpor"'] & & \texttt{True} \\
& \texttt{letrec } x_1 = s_1 \texttt{ in True} \arrow[ur, dashed, "S{,}gc"'] & \\
\texttt{letrec } x_1 = s_1 \texttt{ in (por } s_2 \texttt{ True)} \arrow[ur, dashed, "S{,}por"'] & &
\end{tikzcd}$$

Now let $s \xrightarrow{por} t$, S be a surface context with $S[s] \Downarrow$. By induction on the length of a sequence of normal order reductions $S[s] \xrightarrow{no,*} s'$ where s' is a WHNF we show that there exists a sequence of normal order reductions starting with $S[t]$ and ending in a WHNF. The base case of the induction is easy, since a (S, por)-transformation preserves WHNFs. For the induction step we apply a (forking) diagram from above to the first reduction of $S[s] \xrightarrow{no,*} s'$ and have the following cases: If the first or the second diagram is applied, the existence of normal order reduction for $S[t]$ to a WHNF follows using the induction hypothesis. In case of the third diagram the normal order reduction for $S[t]$ obviously exists. If the last diagram is applied, then let $S[s] \xrightarrow{no,a} s''$ be the first reduction of $S[s] \xrightarrow{no,*} s'$ and t', t'' be the terms with $s'' \xrightarrow{S,por} t' \xrightarrow{S,gc} S[t]$. Using the induction hypothesis we have $t' \Downarrow$. Finally the correctness of (gc) shows $S[t] \Downarrow$.
Summarizing we have shown, that if $s \xrightarrow{por} t$, then for all surface contexts S: $S[s] \Downarrow \implies S[t] \Downarrow$. Since every reduction context is also a surface context the context lemma implies that if $s \xrightarrow{por} t$ then $s \leq_c t$.

Theorem 2.2. *The transformation (por) is correct, i.e if $s \xrightarrow{por} t$ then $s \sim_c t$.*

Theorem 2.3. *For all expressions s: $s \Downarrow$ if and only if $s \Downarrow_{must}$.*

Proof. The implication $s \Downarrow_{must} \implies s \Downarrow$ obviously follows from the definition of must-convergence. The other direction holds since all normal order reductions preserve contextual equivalence: We assume there exists an expression s with $s \Downarrow$ but $\neg(s \Downarrow_{must})$. The latter assumption implies that there exists an expression t with $s \xrightarrow{no,*} t$ and t is must-divergent, i.e $\neg(t \Downarrow)$, But since $s \sim_c t$ also for the empty context, we have $s \Downarrow \implies t \Downarrow$, which is a contradiction. □

2.2 Implementing Parallel Or in Haskell

Figure 4 shows an ad-hoc encoding of (binary) parallel or in Concurrent Haskell [PGF96] additionally using `unsafePerformIO`. The function creates an empty mutable and synchronizing variable and starts the concurrent evaluation of both arguments using the `forkIO` primitive, where pre-emptive multitasking is necessary to ensure fairness. The main thread now waits until one of the concurrent threads finishes by writing into the `MVar`. The outer `unsafePerformIO` lifts the IO-monadic code to pure code. Although the library documentation forbids to use `unsafePerformIO` in combination with `forkIO`, this use does not break referential transparency, since the value of `por` s t is deterministic.

```
por a b =
 unsafePerformIO $
   do v <- newEmptyMVar
      forkIO (if a == True then putMVar v True else putMVar v b)
      forkIO (if b == True then putMVar v True else putMVar v a)
      takeMVar v
```

Fig. 4. Encoding of `por` in Concurrent Haskell using `unsafePerform`

3 Describing Sequential Circuits with Constructive Cycles in $L_{\texttt{por}}$

Sequential circuits are modeled in $L_{\texttt{por}}$ as in Lava by recursive programs using lists, where an element of a list means a boolean signal at a fixed clock tick. Logical gates are list processing functions and memory is modelled using a function, which delays the signal by one clock tick by adding an element at the head of the list. Figure 5 shows the encodings of *parallel or*, *parallel and* as well as the function `delay2`. The latter function has as first argument a Boolean value which is added at the head of the second argument. We omit `case`-alternatives for some constructors, the right hand sides of these alternatives are $\perp$.

3.1 Why Parallel Or is Required – Constructive Cycles

It is possible to represent constructive cycles without a delay element also in Lava, but a sequential functional programming language, like Haskell 98, does not match the semantics of those circuits. A correct modeling in a letrec calculus has to use parallel Boolean operators like por and pand.
The circuit C_4 of [SBT96] is shown in figure 6 where L_1 is initialized with `True` and L_2 has `False` as initial value. The circuit C_3 has the same layout with the difference that L_1 and L_2 are initialized with `True`. The circuits C_4, C_3 can be modeled in our letrec-calculus L_{por} as follows:

$$\begin{array}{ll}
C_4 = \texttt{letrec} & C_3 = \texttt{letrec} \\
\quad x = a \ \texttt{<\&>} \ (not\ y) & \quad x = a \ \texttt{<\&>} \ (not\ y) \\
\quad a = delay2\ \texttt{True}\ y & \quad a = delay2\ \texttt{True}\ y \\
\quad y = b \ \texttt{<\&>} \ (not\ x) & \quad y = b \ \texttt{<\&>} \ (not\ x) \\
\quad b = delay2\ \texttt{False}\ x & \quad b = delay2\ \texttt{True}\ x \\
\texttt{in}\ (x, y) & \texttt{in}\ (x, y)
\end{array}$$

$$
\begin{array}{ll}
\mathit{head} & = \lambda xs.\texttt{case}\ xs\ (y : ys \rightarrow y) \\
\mathit{tail} & = \lambda xs.\texttt{case}\ xs\ (y : ys \rightarrow ys) \\
\mathit{map} & = \lambda f.\lambda xs.(f\ (\mathit{head}\ xs)) : (\mathit{map}\ f\ (\mathit{tail}\ ys)) \\
\mathit{bid} & = \lambda b.\texttt{case}\ b\ (\texttt{True} \rightarrow \texttt{True})\ (\texttt{False} \rightarrow \texttt{False}) \\
\mathit{tobl} & = \lambda xs.\mathit{map}\ \mathit{bid}\ xs \\
\mathit{zipWith} & = \lambda op.\lambda xs.\lambda ys.\ \texttt{letrec}\ \mathit{out} = (op\ (\mathit{head}\ xs)\ (\mathit{head}\ ys)), \\
 & \qquad\qquad\qquad\qquad\qquad\quad \mathit{res} = (\mathit{zipWith}\ op\ (\mathit{tail}\ xs)\ (\mathit{tail}\ ys)) \\
 & \qquad\qquad\qquad\quad \texttt{in}\ (\mathit{out} : \mathit{res}) \\
\mathit{not}2 & = \lambda b.\texttt{case}\ b\ (\texttt{True} \rightarrow \texttt{False})\ (\texttt{False} \rightarrow \texttt{True}) \\
\mathit{not} & = \lambda xs.(\mathit{map}\ \mathit{not}2\ xs) \\
\mathit{pand} & = \lambda a.\lambda b.\mathit{not}2(\texttt{por}\ (\mathit{not}2\ a)\ (\mathit{not}2\ b)) \\
\texttt{(<|>)} & = \lambda xs.\lambda ys.\mathit{zipWith}\ \texttt{por}\ xs\ ys \\
\texttt{(<\&>)} & = \lambda xs.\lambda ys.\mathit{zipWith}\ \mathit{pand}\ xs\ ys \\
\mathit{delay}2 & = \lambda a.\lambda xs.(a : xs)
\end{array}
$$

Fig. 5. Encodings of Parallel Or `<|>`, Parallel And `<&>` and `delay2`

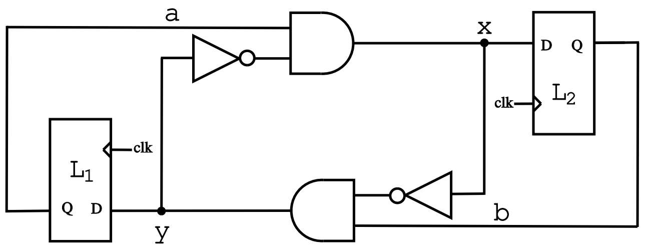

Fig. 6. The Circuit C_4

We consider the circuit C_4: If `<&>` would be implemented in a Haskell-like language as (*zipWith and*) where (*and*) is Boolean *and* being strict in both arguments then the values for x and y are undefined (i.e. $\perp$) (as shown in table 1), although the cycle is constructive. If *and* is the standard Haskell-operator `(&&)`, then the circuit C_4 produces the expected behavior, i.e. table 2 shows how the computation begins. But if we use a simple Boolean law and commute the and-operator in the binding for x (i.e. $x = (\mathit{not}\ y)$ `<&>` a instead of $x = a$ `<&>` $(\mathit{not}\ y)$), then x and y are undefined after the first step, i.e. the beginning of the computation is shown in table 3. The solution is to use an *and*-operator that is non-strict in both of its arguments, i.e. "parallel and". Then the behavior is as expected: the circuit C_4 produces the sequence of states as shown in table 2, where after the initialization all values are defined.

	a	b	x	y
step 1:	True	False	$\perp$	$\perp$
step 2:	$\perp$	$\perp$	$\perp$	$\perp$

Table 1. Execution of C_4 with strict *and*

	a	b	x	y
step 1:	True	False	True	False
step 2:	False	True	False	True
step 3:	True	False	True	False

Table 2. The Expected Behavior of C_4

	a	b	x	y
step 1:	True	False	True	False
step 2:	False	True	$\perp$	$\perp$
step 3:	$\perp$	$\perp$	$\perp$	$\perp$

Table 3. Execution of C_4 with Haskell-like `<&>` after commuting the arguments of an *and*-operator

	a	b	x	y
step 1:	True	True	$\perp$	$\perp$
step 2:	$\perp$	$\perp$	$\perp$	$\perp$

Table 4. The Expected Behavior of C_3

This result does not change if we commute the arguments of the `<&>`-operators. We now consider the circuit C_3 which produces the sequence of states shown in table 4, where due an inappropriate initialization the values for x, y are undefined, which is the expected behavior.

4 Program Transformations for Hardware Descriptions

Apart from classical program transformations used in functional programming languages e.g. during the compilation of functional programs (see e.g. [PS98]), we need transformations operating on the logical layer of circuits descriptions. Since logical gates are implemented as list processing functions, we need equivalences on those functions.
Figure 7 shows program transformations, which are analogous to the classical laws of Boolean algebra: (dne) eliminates a double negation, the transformations (fdo-`<|>`) and (fdo-`<&>`) float a delay over por and pand resp., followed by two distributivity laws; the last two transformations are adapted versions of the law of de Morgan.
Note that the rules (fdo-`<&>`) and (fdo-`<|>`) are correct for $b, d \in \{\texttt{True}, \texttt{False}, \perp\}$ and all expressions r, s, t that are *tfb-lists*, i.e. list of elements of $\{\texttt{True}, \texttt{False}, \perp\}$, which can be achieved by requiring them to be of the form (*tobl xs*) (or equivalent to such an expression). We have *tobl* $s \sim_c s$ for all tfb-lists, and all resulting lists of our operators are also tfb-lists given the arguments are tfb-lists. The correctness of the transformations follows using correctness of partial evaluation and the copying rule. We demonstrate the part of the proof for (fdo-`<|>`) and b, d being `True`:

$$\begin{aligned} \mathit{delay2}\ \texttt{True}(s\ \texttt{<|>}\ t) &\rightarrow \texttt{True} : (s\ \texttt{<|>}\ t) \\ &\leftarrow (\texttt{True} : s)\ \texttt{<|>}\ (\texttt{True} : t) \\ &\leftarrow (\mathit{delay2}\ \texttt{True}\ s)\ \texttt{<|>}\ (\mathit{delay2}\ \texttt{True}\ t) \end{aligned}$$

where $\rightarrow$ are reductions mixed with (cpall) transformations

(dne)	$not\ (not\ s) \to s$
(fdo-`<|>`)	$(delay2\ b\ s)$ `<|>` $(delay2\ d\ t) \to delay2$ (`por` $b\ d)\ (s$ `<|>` $t)$
(fdo-`<&>`)	$(delay2\ b\ s)$ `<&>` $(delay2\ d\ t) \to delay2\ (pand\ b\ d)\ (s$ `<&>` $t)$
(distr-`<|>`)	r `<|>` $(s$ `<&>` $t) \to (r$ `<|>` $s)$ `<&>` $(r$ `<|>` $t)$
(distr-`<&>`)	r `<&>` $(s$ `<|>` $t) \to (r$ `<&>` $s)$ `<|>` $(r$ `<&>` $t)$
(dm-`<|>`)	$not\ (s$ `<|>` $t) \to (not\ s)$ `<&>` $(not\ t)$
(dm-`<&>`)	$not\ (s$ `<&>` $t) \to (not\ s)$ `<|>` $(not\ t)$

Also commutativity, associativity, idempotence,
and absorption transformations are permitted for `<|>`, `<&>`

Fig. 7. Transformations for delay2, `<|>` and `<&>`

The remaining rules of figure 7 are correct for r, s, t being tfb-lists. The proofs of correctness can be achieved using an approximation variant of Bird's *Take-Lemma* ([Bir98]), which is based on induction on lists, additionally approximating infinite lists with lists having $\perp$ as last tail.

4.1 Equivalence of Circuits – An Example

As an illustrating example figure 8 shows two equivalent logical nets. In the following we demonstrate for the example of the two circuits in figure 8, how program transformations show the equivalence of f and g for tfb-list-arguments, and on the other hand, also reduce the number of required wires.
We transform the expressions $(f\ xs\ ys)$ and $(g\ xs\ ys)$, where xs' and ys' are arbitrary lists and $xs = tobl\ xs'$, $ys = tobl\ xs'$.

Transformation of f xs ys:

```
f xs ys  --(1)-->  letrec                          --(2)-->  letrec
                     d1 = d2 <|> d3                            d2 = delay2 False d4
                     d2 = delay2 False d4                      d3 = delay2 False d5
                     d3 = delay2 False d5                      d4 = not xs
                     d4 = not xs                               d5 = ys <&> d6
                     d5 = ys <&> d6                            d6 = not d7
                     d6 = not d7                               d7 = d2 <|> d3
                     d7 = d2 <|> d3                          in d7
                   in d1

--(3)--> letrec                                --(4)--> letrec
           d4 = not xs                                    d7 = delay2 False
           d5 = ys <&> d6                                          ((not xs) <|> (ys <&> (not d7)))
           d6 = not d7                                  in d7
           d7 = delay2 False (d4 <|> d5)
         in d7

--(5)--> letrec d7 = delay2 False (((not xs) <|> ys) <&> ((not xs) <|> (not d7)))
         in d7
```

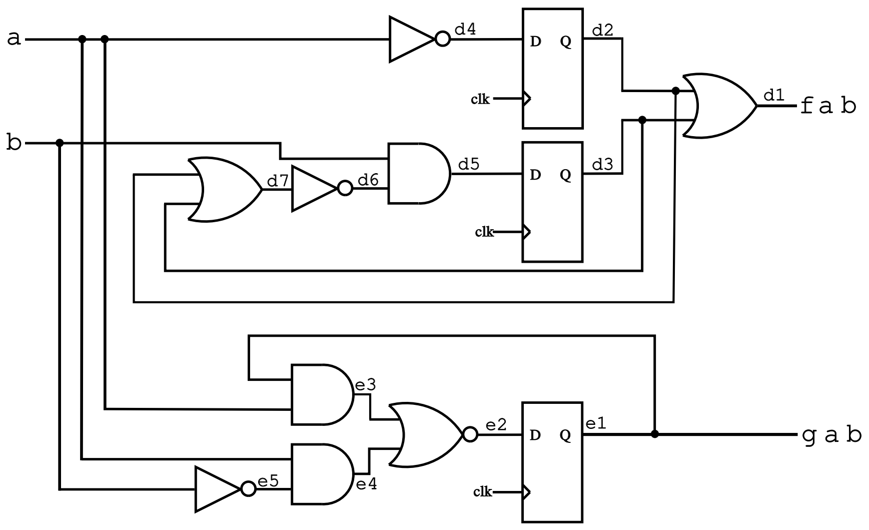

Fig. 8. Two Equivalent Circuits

$f = \lambda a.\lambda b.$ `letrec`
- $d_1 = d_2$ `<|>` d_3
- $d_2 = delay2$ `False` d_4
- $d_3 = delay2$ `False` d_5
- $d_4 = not\ a$
- $d_5 = b$ `<&>` d_6
- $d_6 = not\ d_7$
- $d_7 = d_2$ `<|>` d_3

`in` d_1

$g = \lambda a.\lambda b.$ `letrec`
- $e_1 = delay2$ `False` e_2
- $e_2 = not\ (e_3$ `<|>` $e_4)$
- $e_3 = e_1$ `<&>` a
- $e_4 = a$ `<&>` e_5
- $e_5 = not\ b$

`in` e_1

Fig. 9. The corresponding functional hardware descriptions

(1) (lbeta) and (cpall1) twice

(2) common subexpression elimination for the subexpression (d_2 `<|>` d_3), copying d_1, and garbage collecting d_1.

(3) copying d_2 and d_3, and floating delay over `<|>`

(4) copying d_4, d_5, and d_6

(5) distributivity law

Transformation of $g\ xs\ ys$:

$$g\ xs\ ys \xrightarrow{(1)} \begin{array}{l}\texttt{letrec}\\ \quad e_1 = delay2\ \texttt{False}\ e_2\\ \quad e_2 = not\ (e_3\ \texttt{<|>}\ e_4)\\ \quad e_3 = e_1\ \texttt{<\&>}\ xs\\ \quad e_4 = xs\ \texttt{<\&>}\ e_5\\ \quad e_5 = not\ ys\\ \texttt{in}\ e_1\end{array} \xrightarrow{(2)} \begin{array}{l}\texttt{letrec}\\ \quad e_1 = delay2\ \texttt{False}\ e_2\\ \quad e_2 = (not\ e_3)\ \texttt{<\&>}\ (not\ e_4)\\ \quad e_3 = e_1\ \texttt{<\&>}\ xs\\ \quad e_4 = xs\ \texttt{<\&>}\ e_5\\ \quad e_5 = not\ ys\\ \texttt{in}\ e_1\end{array}$$

$$\xrightarrow{(3)} \begin{array}{l}\texttt{letrec}\ e_1 = delay2\ \texttt{False}\ ((not\ (e_1\ \texttt{<\&>}\ xs))\ \texttt{<\&>}\ (not\ ((xs\ \texttt{<\&>}\ (not\ ys)))))\\ \texttt{in}\ e_1\end{array}$$

$$\xrightarrow{(4)} \begin{array}{l}\texttt{letrec}\\ \quad e_1 = delay2\ \texttt{False}\ (((not\ e_1)\ \texttt{<|>}\ (not\ xs))\ \texttt{<\&>}\ ((not\ xs)\ \texttt{<|>}\ (not\ (not\ ys))))\\ \texttt{in}\ e_1\end{array}$$

$$\xrightarrow{(5)} \begin{array}{l}\texttt{letrec}\ e_1 = delay2\ \texttt{False}\ (((not\ xs)\ \texttt{<|>}\ (not\ e_1))\ \texttt{<\&>}\ ((not\ xs)\ \texttt{<|>}\ ys))\\ \texttt{in}\ e_1\end{array}$$

$$\xrightarrow{(6)} \begin{array}{l}\texttt{letrec}\ e_1 = delay2\ \texttt{False}\ (((not\ xs)\ \texttt{<|>}\ ys)\ \texttt{<\&>}\ ((not\ xs)\ \texttt{<|>}\ (not\ e_1)))\\ \texttt{in}\ e_1\end{array}$$

(1) (lbeta) and (cpall1) twice
(2) law of de Morgan
(3) copying e_2, e_3, e_4 and e_5
(4) law of de Morgan (two times)
(5) double negation elimination and commutativity of `<|>`
(6) commutativity of `<&>`

Since the results of the transformations of $f\ xs\ ys$ and $g\ xs\ ys$ are equal w.r.t. to α-equivalence, $f\ xs\ ys$ and $g\ xs\ ys$ are contextually equivalent for all lists xs, ys with $xs = tobl\ xs'$ and $ys = tobl\ ys'$.

5 Conclusion

We introduced the calculus $L_{\texttt{por}}$ – a call-by-need lambda-calculus with parallel or – and showed how its semantics correctly models sequential circuits with constructive cycles. We proposed a set of program transformations to manipulate functional circuit descriptions on the source code level and exemplarily demonstrated their applicability. Future work is devoted to develop tools for automated circuit transformation, to work out the proofs of correctness of all the transformations above, and to describe further useful program transformations and to prove their correctness.

Acknowledgements We would like to thank Lars Hedrich, Alexander Jesser, and Sebastian Steinhorst for their valuable advice on questions concerning hardware design.

References

BCSS98. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. In *International Conference on Functional Programming, ICFP*, pages 174–184. ACM, 1998.

Bir98. Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, London, 1998.

CP02. K. Claessen and G. Pace. An embedded language framework for hardware compilation. In *DCC'02, ETAPS, Grenoble, France*, 2002.

CSS03. Koen Claessen, Mary Sheeran, and Satnam Singh. *Functional hardware description in Lava*, volume Fun of Programming of *Cornerstones of Computing*, chapter 8. Palgrave, March 2003.

KSS98. Arne Kutzner and Manfred Schmidt-Schauß. A nondeterministic call-by-need lambda calculus. In *International Conference on Functional Programming 1998*, pages 324–335. ACM Press, 1998.

Mor98. A.K.D. Moran. *Call-by-name, call-by-need, and McCarthys Amb*. PhD thesis, Dept. of Comp. Science, Chalmers university, Sweden, 1998.

MSC99. Andrew K.D. Moran, David Sands, and Magnus Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, volume 1594 of *Lecture Notes in Comput. Sci.*, pages 85–102. Springer-Verlag, 1999.

Pey03. Simon Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003. `www.haskell.org`.

PGF96. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. 23th Principles of Programming Languages, St. Petersburg Beach, Florida*, 1996.

PS98. Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.

SBT96. Thomas R. Shiple, Gerard Berry, and Herve Touati. Constructive analysis of cyclic circuits. In *EDTC '96: Proceedings of the 1996 European conference on Design and Test*, pages 328–333, Washington, DC, USA, 1996. IEEE Computer Society.

She05. M. Sheeran. Hardware design and functional programming: a perfect match. *J.UCS*, 11(7):1135–1158, 2005.

SS03. Manfred Schmidt-Schauß. FUNDIO: A lambda-calculus with a `letrec`, `case`, constructors, and an IO-interface: Approaching a theory of `unsafePerformIO`. Frank report 16, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, 2003.

SS06. Manfred Schmidt-Schauß. Equivalence of call-by-name and call-by-need for lambda-calculi with letrec. Frank report 25, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, September 2006. submitted for publication.

SS07a. Manfred Schmidt-Schauß. Correctness of copy in calculi with letrec, case and constructors. Frank report 28, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main, 2007.

SS07b. Manfred Schmidt-Schauß. Correctness of copy in calculi with letrec, case, constructors and por. Frank report 29, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main, 2007.

SSS07a. David Sabel and Manfred Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 2007. accepted for publication.

SSS07b. Manfred Schmidt-Schauß and David Sabel. On generic context lemmas for lambda calculi with sharing. Frank report 27, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, 2007.

SSSS04. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. On the safety of Nöcker's strictness analysis. Frank report 19, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, 2004.

SSSS05. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. A complete proof of the safety of Nöcker's strictness analysis. Frank report 20, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, 2005. submitted for publication.