# A Partial Rehabilitation of Side-Effecting I/O: Non-Determinism in Non-Strict Functional Languages

Manfred Schmidt-Schauß
Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32, D-60054 Frankfurt, Germany
Tel: (49) 69 798 28597
E-mail: schauss@ki.informatik.uni-frankfurt.de

June 14, 1996

**Abstract**

We investigate the extension of non-strict functional languages like Haskell or Clean by a non-deterministic interaction with the external world. Using call-by-need and a natural semantics which describes the reduction of graphs, this can be done such that the Church-Rosser Theorems 1 and 2 hold. Our operational semantics is a base to recognise which particular equivalencies are preserved by program transformations.

The amount of sequentialisation may be smaller than that enforced by other approaches, and the programming style is closer to the common one of side-effecting programming. However, not all program transformations used by an optimising compiler for Haskell remain correct in all contexts.

Our result can be interpreted as a possibility to extend current I/O-mechanism by non-deterministic memoryless function calls. For example, this permits a call to a random number generator. Adding memoryless function calls to monadic I/O is possible and has a potential to extend the Haskell I/O-system.

## 1 Introduction

A useful implementation of I/O is an essential part of a lazy functional language, since every application written in a functional language must perform some I/Oś. There are very sophisticated solutions to this problem which retain referential transparency, for example monadic I/O [Wad90, PJW93], as supported by the current release of Haskell 1.3 [HAB+96], continuation based I/O, synchronised stream I/O (see [Gor94, Ach96, JS91]), or the use of unique typing [SBvEP93a, PvE93].

The programming style in a lazy functional language is heavily influenced by the supported I/O-mechanism. Modifying the I/O-behaviour or debugging some lazy functional program that uses I/O is a black art. It is interesting that novices in lazy functional programming in general expect that there is some direct (side-effecting) I/O using a function call. As mentioned in [Gor94], this side-effecting

I/O is the most widely-used I/O-mechanism in eager functional languages, and it was also used in an industrial implementation of a lazy functional language [HNMH96].
In this paper we investigate side-effecting I/O using call-by-need. In Glasgow Haskell, this method is permitted by marking it as "unsafe I/O". There are a several papers that denounce this method as non-referentially transparent [HO89, HO90], and hence discard it as a serious method for implementing I/O in non-strict functional languages. The implementation of Clean [NSvP91, PvE95] uses the method of direct calls via the operating system with a static analysis method to ensure safe use of this unsafe I/Os. The underlying method is a unique type system. The direct call method was also used in the industrial lazy functional programming language Natural EL [HNMH96, SS91] however, only with minor precaution.
This paper investigates the foundations for non-deterministic lazy functional languages, such that implementors of lazy functional languages, who are already using such non-deterministic calls, or who want to use them in the future, shall have a criterion for recognising the correct program transformations thus avoiding pitfalls and incorrect program transformations for compiler optimisations. There are several papers discussing a modelling of sharing in functional languages and $\lambda$-calculus [AFM$^+$95, Lau93, PS92, Yos93, GH90]. A difference to these works is that our approach is based on a calculus of supercombinators. We model sharing using an environment like [Lau93], and reduction as in the G-machine [PJ87, Aug84, Joh84]. Our first main result concerns non-deterministic I/O without memory, like a call to a random number generator, or a simple interaction with a user using a pop-up window to ask for some input. The basic notions that we use in this paper are:

1. Graph expressions, i.e., expressions that also reflect sharing properties
2. A non-deterministic choice primitive `natchoice` that yields some natural number on reduction.
3. Sets of possible graph expressions that may result from the reduction of some redex.

Based on this notions, the first result in 2.2 is that reduction of graphic expressions is confluent for sets of expressions. The consequences are that we can say exactly, which program transformations are valid or not, and that such a program can be parallelised. The valid program transformation can roughly be described as the subset of the usual ones which do not disturb the sharing structure. Going a bit further, we investigate the explicit I/O-behaviour considering the multi-set of question-answer pairs that characterise a certain reduction. It will turn out, that using memoryless non-deterministic I/O, we have that for every reduction sequence there is also one that uses normal order reduction, has the same result, and uses a submultiset of the question-answer pairs. This shows that memoryless non-deterministic interaction of a lazy functional program is safe.
The next step is to consider I/O with external memory. Since arbitrary use of I/O in connection with external memory is considered as unsafe, we define

the notion of I/O-correct functional programs, which roughly means that calls that modify the world, are in the right sequence. For example, monadic programming style yields I/O-correct programs. We can show that I/O-correctness implies set-confluence. This again permits concurrent execution and gives a criterion for the validity of program transformations. The power of I/O-correct programs is much more useful and flexible, if it is possible to have programming primitives that sequentialise certain reductions. We consider two possibility: `seq` and $\gg$. The `seq`combinator first evaluates its first argument and then evaluates the second. The $\gg$ is restricted to fix the normal order evaluation strategy as the only permitted one for certain redexes. The combination of both permits to write flexible programs and also to reason about the possible program transformations in a natural way.
As a simple introductory example, consider the well-known "counterexamples" to referential transparency for non-deterministic I/O [HO90], which shows that there is a difference between call-by-name and call-by-need. Consider the two supercombinators `double` and `choice`, with the defintion.

> `double` $x = x + x$
> `choice` $x\ y = z$
> $z$ may be x or y in a non-deterministic fashion.

Note that choice is not like McCarthy's `amb`, which is in addition bottom-avoiding. Then the usual argumentation is that the different reductions of *tdouble*(`choice`12) have different results:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1. | `double(choice 1 2)` | $\rightarrow$ | `double` 1 | $\rightarrow$ | 2 | | | |
| | | $\rightarrow$ | `double` 2 | $\rightarrow$ | 4 | | | |
| 2. | `double(choice 1 2)` | $\rightarrow$ | (`choice` 1 2) + (`choice` 1 2) | $\rightarrow$ | $1+1$ | $\rightarrow$ | 2 | |
| | | | | $\rightarrow$ | $1+2$ | $\rightarrow$ | 3 | |
| | | | | $\rightarrow$ | $2+1$ | $\rightarrow$ | 3 | |
| | | | | $\rightarrow$ | $2+2$ | $\rightarrow$ | 4 | |

It is obvious, that the second reduction results in different values. Our argumentation is that the printed (flat) representation of expressions is not appropriate for reduction in a non-deterministic lazy functional language, since this may duplicate non-deterministic calls. An important ingredient of a correct solution is to preserve the sharing properties. The example above under these restrictions now behaves as follows:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1. | `double(choice 1 2)` | $\rightarrow$ | `double` 1 | $\rightarrow$ | 2 | | | |
| | | $\rightarrow$ | `double` 2 | $\rightarrow$ | 4 | | | |
| 2. | `double(choice 1 2)` | $\rightarrow$ | (`let` $x =$ `choice` 1 2) `in` $x + x$) | | | | | |
| | | $\rightarrow$ | (`let` $x = 1$ `in` $x + x$) | $\rightarrow$ | $1+1$ | $\rightarrow$ | 2 | |
| | | $\rightarrow$ | (`let` $x = 2$ `in` $x + x$) | $\rightarrow$ | $2+2$ | $\rightarrow$ | 4 | |

Now the results are equivalent. In this paper we shall show that this is not an accident, but that a rigorous treatment shows that this method of reduction preserves equivalence of the possible outcomes. As an aside, current implementations of lazy functional languages always perform the reduction that preserves sharing.
Instead of permitting choice as a primitive in the language, we assume that (unrestricted) natural numbers are given as an algebraic data type, and that there is

a primitive function natchoice without arguments, which non-deterministically delivers some natural number on every call. Then `choice` is implementable as

$$\texttt{choice}\ x\ y = \text{if}\ \texttt{natchoice} > 1\ \text{then}\ x\ \text{else}\ y$$

The following table indicates the hierarchy of the languages that we will consider:

| | available syntax | non-deterministic syntax additions |
|---|---|---|
| $FP_0$ | `let(rec)` x = ... | `natchoice` |
| | $case_T\ x$ `of alts` | |
| | algebraic datatypes, selectors | |
| $FP_1$ | | interface functions |
| $FP_2$ | `seq`, $\gg$ | |
| $FP_3$ | | storage interface functions |
| $FP_G$ | full `let(rec)` | |
| | pattern match | |
| | list comprehensions | |

## 2 Non-Deterministic Choices: The Language $FP_0$

This section considers a functional core language extended by a non-deterministic choice operator.
We use a functional core language $FP_0$. The syntax for expressions $E$ and super-combinator definitions *scdef* is as follows:

$$\begin{array}{lll}
<E> & ::= & <constant> | (<E><E>) | (\texttt{case}_A <E> \text{of} <alts>) \\
 & & | (\texttt{let} <defs> \texttt{in} <E> | (\texttt{letrec} <defs> \texttt{in} <E>) \\
<defs> & ::= & (<var> = <E>)^+ \\
<alts> & ::= & (<constant> \rightarrow <E>)^+ \\
<scdef> & ::= & <constant> <var>^* = <E>
\end{array}$$

There are algebraic data types with constructors. Every algebraic data type $A$ has a fixed set of constructors, every constructor has a fixed arity. For every algebraic data type $A$ there is a case-constant $\texttt{case}_A$. There are selectors for every argument position of every constructor. The selector $\texttt{sel}_{A,i,j}$ is the selector that extracts the $j^{th}$ component from an expression starting with the constructor $c_i$, which belongs to algebraic type $A$. We assume that integers are already defined as an algebraic data type.
The usual restrictions and conventions apply: in a supercombinator definition, every free variable in the expression is also an argument in cases, the discriminating constant is a constructor and furthermore, for an algebraic data type $A$, the $\texttt{case}_A$-expressions have an alternative for every constructor of $A$. Applications can be written without brackets, where we assume that $(a\ b\ c)$ is the same as $(a(b\ c))$. A *program* consists of a set of supercombinator definitions and an expression to be evaluated.
A *data object* is an expression that is built from constructors only, and all applications are saturated, i.e. every constructor of arity $n$ is applied to $n$ arguments. There is one non-deterministic primitive supercombinator `natchoice` without

arguments that delivers some arbitrary natural number, when evaluated. The language can be polymorphically typed [Mil78] and we assume that all expressions are well-typed. A term is in weak head normal form (WHNF), if it is of the form $(f\ t_1 \ldots t_n)$, and $f$ is a constructor, or a supercombinator of arity greater than $n$. A WHNF is a constructor WHNF (CWHNF), if the constant in front is a constructor, otherwise it is a partial application. A CWHNF is *saturated*, if the term is $(c\ t_1 \ldots t_n)$ where c is a constructor of arity n.
A remark on $\texttt{case}_A$ is necessary. Instead of permitting $\texttt{case}_A$ to be of the form $(\texttt{case}\ s\ \texttt{of}\ c_1\ x_1 \ldots \to s_1; \ldots; c_m\ x_{m,1} \ldots x_{m,n(m)} \to s_m)$ we instead use $(\texttt{case}e\ \texttt{of}\ c_1 \to \texttt{let}\ (x_1 = \texttt{sel}_{\ldots}\ e; \ldots)\texttt{in}\ e_1; \ldots)$, where `sel` is some selector. Note that selectors are strict in their argument, i.e., they can only reduce, if their argument is in WHNF.

## 2.1 Reduction

We define reduction in a *natural (operational) semantics* as in [Lau93], where an environment-model is used to define the semantics of expressions and to justify the reductions. A flat expression semantics is not compatible with nondeterminism. For simplicity we assume that all bound variables have a different name, which can easily be achieved by using a common static scoping method for renaming variables. Note that as usual, let and letrec differ in their scoping rules.
An *environment* $U$ is a finite set of pairs $(x, e)$, where $x$ is a variable name and $e$ is an expression. We assume that every variable occurs at most once on a left hand side of such a pair in some environment. An environment models the graph used in lazy graph reduction systems and is able to represent sharing and also cycles generated by letrecs. Now we can define a *graphic expression*. It is a pair $(z, U)$, where $z$ is a variable and $U$ is an environment. This term $(z, U)$ directly corresponds to a `letrec` expression:

$$\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ z, \text{ where } U = \{(x_1, e_1), \ldots, (x_n, e_n)\}$$

Note that the $z$ is in general one of the variables $x_i$. In order to simplify the treatment, we insist on the following convention: Every term on the right hand side of a pair is either a variable or an application $(x\ y)$, where $x, y$ are variables, or a `case` of the form $(\texttt{case}_T\ x\ \texttt{of}\ c_1 \to x_1; \ldots)$. Every environment can be brought into this form without changing the meaning by introducing new variables into the environment. A graphic expression is *closed*, if the corresponding */ttletrec* has no free variables.
Given an environment $U$, we let $U(x) := e$ be the direct value of a variable in $U$, if $(x, e) \in U$, and otherwise $U(x) := x$. Let $V_l(U)$ be the set of variables that occur as left hand sides in $U$. The value of a variable in $V_l(U)$ is defined as an iterated application of $U(.)$:

$$\varepsilon_U(x) = \begin{cases} e & \texttt{if} \quad (x, e) \in U \text{and } e \text{ is not a variable} \\ \varepsilon_U(y) & \texttt{if} \quad (x, y) \in U \text{and } y \text{ is a variable} \\ x & \texttt{if} \quad x \notin V_l(U) \end{cases}$$

There is a chance that this function does not terminate. In this case we define the result as $\Omega$, representing non-termination. It is possible to effectively detect non-termination, since the environment is finite. We also inductively define $\varepsilon_U(.)$ for expressions.
A further function is $\texttt{spine}_U(x)$ that computes the spine-term for $x$.

$$\begin{array}{lcll} \texttt{spine}_U(x) & = & \texttt{spine}_U(e) & \text{if}(x,e) \in U \\ \texttt{spine}_U(c) & = & c & \text{if } c \text{ is a constant} \\ \texttt{spine}_U((xy)) & = & (\texttt{spine}_U(x)\ y) & \\ \texttt{spine}_U(t) & = & \text{undefined, otherwise} & \end{array}$$

If $\texttt{spine}_U(t)$ does not terminate, then let $\texttt{spine}_U(t) = \Omega$. Note, that it is decidable, whether $\texttt{spine}_U(.)$ terminates, since $U$ is finite. This may occur, even if the initial program is well-typed, for example the expression ($\texttt{letrec}\ x = x\ \texttt{in}\ x$) gives the environment $(x, \{(x, x)\})$, and $\texttt{spine}_U(x)$ does not terminate.
We call the leftmost path from a node $x$ to the constant the *spine* of a node. Now we define the initial environment and the graphic expression that belongs to every $FP_0$-expression. This environment is the union of all `let`- and `letrec`-defined variables together with their defining expressions, where we add let-bindings such that the conventions are satisfied. I.e., for a term $t$ we define $U_{t,0} = \{(x,t)\} \cup \{(y,e) | y = e$ occurs in some definition of a `let` or `letrec` in $t\}$, and then transform $U_{t,0}$ into $U_{t,1}$ by adding variable definitions for nested terms. The corresponding graphic expression is $(x, U_{t,1})$. Note that we have assumed that different bound variables have different names.
We define the *active* variables in a graphic expression $(e, U)$ as follows: Let $x\ R_U\ y$ for variables $x, y$ hold if $(x, s) \in U$ and $y \in V(s)$ and let $R_U^*$ be the reflexive, transitive closure of $R_U$. Let $(x, U)$ be a graphic expression, then the *gc-simplified* form is: $(x, U')$, where $U' = \{(y, s) \in U | x\ R_U^*\ y\}$.
Now we can start the reduction with the initial expression as a graphic expression $(e, U)$.

**Definition 1.** A *redex* in $(e, U)$ is a right hand side $s$ in $U$ if one of the following holds

- $\texttt{spine}_U(s)$ is of the form $(\ldots(f\ x_1)\ldots x_n)$ and $f$ is a supercombinator of arity $n$.
- $\texttt{spine}_U(s) = (\texttt{case}_A\ x\ \text{of}\ c_1 \to x_1; \ldots; c_m \to x_m)$ if $\texttt{spine}_U(x)$ is a saturated WHNF with a constructor belonging to $A$.
- $\texttt{spine}_U(s) = (\texttt{sel}_A\ x)$ if $\texttt{spine}_U(x)$ is a saturated WHNF with a constructor belonging to $A$.
- if $\texttt{spine}_U(s) = \Omega$

It is decidable whether s is a redex, since the spine is effectively computable.

**Definition 2.** Reduction: The reduction of redexes in the graphic expression $(e, U)$ is as follows. The reductions operate either by replacing some right hand side in $U$, or by leaving $U$ unchanged, or by removing some pair from $U$.

- If $s$ is a redex, and $\texttt{spine}_U(s) = (\ldots(f\ y_1)\ldots y_n)$, and $f$ is a supercombinator of arity $n$, and $f \neq \texttt{natchoice}$. ($\delta$-reduction). Then replace $s$ by $r'$, where $f\ x_1 \ldots x_n = r$ is the definition of $f$ with completely fresh variables, where $(r', U_r)$ is the graphic expression corresponding to $r$. Furthermore add $\{(x_i, y_i) | i = 1, \ldots, n\}$ and $U_r$ to $U$.

- If $s$ is a redex, and $s = \texttt{natchoice}$, then replace $s$ by an arbitrary natural number $n$.

- If $\texttt{spine}_U(s) = \texttt{case}_A\ x\ \texttt{of}\ c_1 \to y_1; \ldots; c_m \to y_m$, and $\texttt{spine}_U(t) = (c_i\ z_1 \ldots z_{m(i)})$. Then replace $s$ by $y_i$.

- If $s$ is a redex, and $spine_U(s) = (\texttt{sel}_{A,i,j}\ y)$ and $\texttt{spine}_U(y) = (c_i\ z_1 \ldots z_m(i))$. Then replace $s$ by $z_j$.

- If $\texttt{spine}_U(x) = \Omega$, then we have the reduction relation $(e, U) \to (e, U)$.

We assume that after every reduction step, the resulting graphic expressions is gc-simplified. We can assume that there is no error situation, since we have assumed that programs are well-typed. We assume This reduction relation is extended to sets of graphic expressions as follows: $\{e\} \cup A \to E \cup A$, where $\{e\} \to E$ means that one redex is chosen in $e$, and $E$ is the set of all possible reducts. In the case of an `natchoice`-redex, the set $E$ contains all possible graphic expressions to which $e$ can be reduced by different choices of a natural number. In all other cases, the set $E$ is a singleton.

**Lemma 3.** *A closed graphic expression can only be reduced to a closed graphic expression.*

*Example 1.*

1. Let $(x, U)$ be a graphic expression with $U = \{(x, \texttt{constr}\ y\ z), (y, u), (z, u), (u, 1 + 1)\}$. Reducing $1 + 1$ to 2 gives $U' = \{(x, \texttt{constr}\ y\ z), (y, u), (z, u), (u, 2)\}$

2. Consider the example from [PJ87] showing that the G-machine requires indirections, where we assume that * (times) is a built-in strict function.

   $\texttt{id}x = x$
   $f\ y = (\texttt{id}\ y) * y$
   $\texttt{main} = f\ (\texttt{square}\ 4)$
   $\texttt{square}\ x = x * x$

   Instead of the pairs $(x, c)$, we replace $x$ by the constant in this example. The initial environment is:

   $U = \{(x, (f\ x_1)), (x_1, (\texttt{square}\ 4))\}$
   Reducing the topmost redex corresponding to $x$ gives:
   $U_1$ =
   $\{(x, (y_1\ y)), (y_1, (*\ y_2)), (y_2, (\texttt{id}y)), (y, x_1), (x_1, (\texttt{square}\ 4))\}$
   Reducing the redex corresponding to $y_2$ gives:

$U_2$ =
$\{(x, (y_1\ y)), (y_1, (*\ y_2)), (y_2, z), (z, y), (y, x_1), (x_1, (\texttt{square}\ 4))\}$
The only redex is now (`square` 4).
$U_3 = \{(x, (y_1\ y)), (y_1, (*\ y_2)), (y_2, z), (z, y), (y, x_1), (x_1, 16)\}$
Now the redex is $(y_1\ y)$, and we obtain:
$U_4 = \{(x, 256), (y_1, (*\ y_2)), (y_2, z), (z, y), (y, x_1), (x_1, 16)\}$
Now garbage collection reduces this graphic expression to:
$(x, \{(x, 256)\})$

## 2.2 Church-Rosser Property for Sets of Graphic Expressions

We shall show that the reduction as defined above preserves an equivalence relation between sets of graphic expressions. This is done by showing confluence for the set of possible reductions w.r.t. to this equivalence.

**Definition 4.** Two graphic expressions $(e_1, U_1)$ and $(e_2, U_2)$ are *$\alpha$-equal*, if they can be transformed into each other by consistently renaming variables. Two sets of graphic expressions $S_1$ and $S_2$ are $\alpha$-equal, if there are mappings $\varphi_1 : S_1 \to S_2$ and $\varphi_2 : S_2 \to S_1$, such that for every graphic expression $e_1 \in S_1$, $\varphi_1(e_1)$ is $\alpha$-equal to $e_1$, and for every $e_2 \in S_2$, $\varphi_2(e_2)$ is $\alpha$-equal to $e_2$

In order to have an adequate description of the non-deterministic reduction process, we describe reduction in terms of sets of graphic expressions. The reduction starts with a singleton set. If there is an evaluation of natchoice, then the resulting set contains more than one graphic expression. These expressions correspond exactly to the reduction possibilities of the initial expression.
In the following we shall show strong confluence for the reduction lifted to sets of graphic expressions. Note that strong confluence is the property that if $a \to b$ and $a \to c$, then there is some $d$, such that $b$ and $c$ can be reduced to $d$ in at most one step. Furthermore strong confluence implies confluence [Bar84]. The reduction is lifted to sets of graphic expression as follows: $\{(e, U)\} \cup R \to S \cup R$, where $S$ contains all possible graphic expressions that can result from reduction of a single redex. In the case that natchoice is reduced, $S$ is an infinite set, and in the other cases, $S$ is a singleton.

**Theorem 5.** *Reduction is confluent on sets of graphic expressions, where we compare sets using $\alpha$-equality.*

*Proof.* We show that strong confluence holds. It is sufficient to show this for a reduction of a singleton set, since then strong confluence can be lifted to sets. We assume that there is some graphic expression $(x_0, U)$ and that there are two different redexes $(x_1, e_1)$ and $(x_2, e_2)$ in $U$. The corresponding reductions are $(x_0, U) \to (x_0, U_1)$ and $(x_0, U) \to (x_0, U_2)$.
First we have to analyse the possible overlapping of redexes.

i.) First assume that the two redexes are equal. Then the only interesting possibility is that the redex expression is natchoice. In this case the sets of possible reducts are the same, hence we have strong confluence.

ii.) We show that the situation that $x_1$ becomes garbage after reduction of $x_2$ and $x_2$ becomes garbage after reduction of $x_1$ implies that $x_1$ and $x_2$ are both garbage before the reduction. Suppose, this is false. Then there is one relational path for $R$ from the initial variable to say $x_1$, which does not use the variable $x_2$. But then the reduction of $x_2$ cannot change this path, hence $x_1$ is not garbage.

iii.) Now we consider the case that $e_1$ and $e_2$ are different. The only case of an overlap in the spines may be that $\mathtt{spine}_U(e_1) = \Omega = \mathtt{spine}_U(e_2)$. In this case, the reductions do not change the expression, hence the outcome is the same.

In all other cases, there is no overlap of the spines of $e_1$ and $e_2$: The variable $x_2$ does not occur in the spine-computation of $e_1$ (and vice versa), since the arities of supercombinators and $\mathtt{sel}_{A,i,j}$ are fixed, and $e_1$ and $e_2$ are different. Furthermore, one reduction does not modify the redex and the corresponding spine computations of the other redex. The only difference after one reduction may be that the other redex becomes garbage. Now we have to consider the different possibilities. We consider in depth the case of two $\delta$-reductions. Let $\mathtt{spine}_U(e_i) = ((\ldots((f_i\ y_{i,1})y_{i,2})\ldots)y_{i,n(i)})$ for $i = 1, 2$, and let $f_i\ z_{i,1}\ldots z_{i,n(i)} = r_i$ be the two supercombinator definitions, where we assume that variables in the definitions are new. Then the two different reductions are: $U \cup \{(x_1, e_1), (x_2, e_2)\} \to U \cup \{(x_1, r_1), (x_2, e_2)\} \cup U_{r_1} \cup \{(z_{1,1}, y_{1,1}), \ldots, (z_{1,n(1)}, y_{1,n(1)})\}$
$\to U \cup \{(x_1, e_1), (x_2, r_2)\} \cup U_{r_2} \cup \{(z_{2,1}, y_{2,1}), \ldots, ((z_{2,n(2)}, y_{2,n(2)})\}$
If there is no garbage collection, then we can reduce the other redex, since there is no interference in computing the spine. This gives the environment: $U \cup \{(x_1, r_1), (x_2, r_2)\} \cup U_{r_1} \cup \{(z_{1,1}, y_{1,1}), \ldots, (z_{1,n(1)}, y_{1,n(1)})\}$ $\cup U_{r_2} \cup \{(z_{2,1}, y_{2,1}), \ldots, ((z_{2,n(2)}, y_{2,n(2)})\}$.

It is obvious, that any garbage collection that does not remove $(x_1, r_1)$ and $(x_2, r_2)$ does not influence the strong confluence. If one redex, say $x_2$, becomes garbage after a reduction of $x_1$, but not vice versa, then the computation shows, that the final expression can be obtained by garbage collecting the final environment. Hence this proves strong confluence in this case.

iv.) The other cases, where no `natchoice`-redex is involved, are treated analogously.

v.) Now let one redex be natchoice. Since the argumentation above shows that a redex is not modified after reduction of the other redex, we get that the common successor is an (infinite) set, where all possibilities to replace `natchoice` with a natural number are present.

vi.) If both redex are `natchoice`-redexes, then again there is no overlap, and the common reduct is a set where the two redexes are replaced by all possible pairs of numbers.

It is interesting to note, that it is not necessary to consider parallel reductions. This is a hint that this complication in the common proof of confluence is an artefact that has its roots in the common flat representation of expressions. Our reduction on graphic expressions is close to graph rewriting systems and term graph rewriting systems (see [PvE93, BvEG$^+$87, KJMdVF93]. However, since graph rewriting systems are not confluent in general, and Theorem 5 states confluence for reduction of graphic expressions, there must be a difference. This difference is in the used data-structure and in definition of reduction. We will illustrate this by an example, which is used to show non-confluence of graph rewriting systems.

*Example 2.* . Let there be two supercombinators, $A, B$ with definitions $A\ x = x, B\ x = x$. Consider reducing the expressions

$$(\texttt{letrec}\ x = A\ y; y = B\ x\ \texttt{in}\ (x, y))$$

As graph rewriting systems, there are two different reducts, corresponding to $(\texttt{letrec}\ x = y; y = B\ x\ \texttt{in}\ (x, y))$ and $(\texttt{letrec}\ x = A\ y; y = x\ \texttt{in}\ (x, y))$, which reduce only to themselves in the definition of graph rewriting systems (see [PvE93], p. 165), hence confluence does not hold.
If we make the same reductions for graphic expressions, then the reductions are as follows:

- $(\texttt{letrec}\ x = A\ y; y = B\ x\ \texttt{in}\ (x, y)) \to (\texttt{letrec}\ x = z_1; z_1 = y; y = B\ x\ \texttt{in}\ (x, y)) \to (\texttt{letrec}\ x = z_1; z_1 = y; y = z_2; z_2 = x\ \texttt{in}\ (x, y))$, which has a cyclic variable reference.
- $(\texttt{letrec}\ x = A\ y; y = B\ x\ \texttt{in}\ (x, y)) \to (\texttt{letrec}\ x = A\ y; y = z_2; z_2 = x\ \texttt{in}\ (x, y)) \to (\texttt{letrec}\ x = z_1; z_1 = y; y = z_2; z_2 = x\ \texttt{in}\ (x, y))$

Hence there is a common reduct. The problem of graph rewrite systems appears to be the compact representation of nodes, which does not work in the presence of so-called black holes.

Now we argue that the second Church-Rosser Theorem also holds for WHNF's, i.e., that a normal order strategy will find a weak head normal form if one exists at all. Therefore, we have to define the normal-order redex (n-o-redex) of a graphic expression $(e, U)$. First we need an algorithm to check a node for weak head normal form:

**Definition 6.** Algorithm: $\texttt{is\_whnf}_U(e, n)$. Let $(x, U)$ be a graphic expression and $e$ be some variable that represents a node. The algorithm starts with $\texttt{is\_whnf}_U(e, 0)$.
$\texttt{is\_whnf}_U(e, n) =$

| | | |
|---|---|---|
| If $e$ is a constructor | then | true |
| If $e$ is a supercombinator and its arity is $> n$ | then | true |
| | otherwise | false |
| If $e$ is a variable | then | $\texttt{is\_whnf}_U(s, n)$ for $(e, s) \in U$ |
| If $e = (\texttt{case}\ldots)$ | then | false |
| If$e = (e_1\ e_2)$ | then | $\texttt{is\_whnf}_U(e_1, n + 1)$ |

In the cases, where this does not terminate, we simply let the result be false.

The topmost redex of $(e, U)$ can simply be found by the following algorithm "unwind", where we start with $\texttt{unwind}_U(e)$.

**Definition 7.** The algorithm `unwind`: The algorithm $\texttt{unwind}_U(s)$ computes the left-most, topmost redex for the expression $s$ in the environment $U$: If $\texttt{spine}_U(s)$ is a redex, then return $s$. Otherwise:

- if $s$ is a constructor, then Fail.
- if $s = (x_1 x_2)$
  - If $\texttt{spine}_U(s) = \Omega$, then return $s$.
  - If$\texttt{spine}_U(s)$ is a constructor application, then Fail.
  - If $\texttt{spine}_U(s)$ is an application to too few arguments, then Fail.
  - If $\texttt{spine}_U(s)$ is an application on too many arguments, compute $\texttt{unwind}_U(x_1)$.
- If $s = (\texttt{case}\ x\ \texttt{of}\ \ldots)$ then: if $\texttt{is\_whnf}_U(x)$, then the result is $s$; otherwise, compute $\texttt{unwind}_U(x)$.
- If $s = (\texttt{sel}\ x)$, where `sel` is a selector: if $\texttt{is\_whnf}_U(x)$, then the result is $s$, otherwise, compute $\texttt{unwind}_U(x)$.
- If $s = x$ is a variable, then there is some $(x, t) \in U$. If $\varepsilon_U(x) = \Omega$, the return $x$ else compute $\texttt{unwind}_U(t)$

The function $\texttt{unwind}_U$ is terminating. The redexes, where either a cyclic variable dependency or an infinite unrolling of a spine occurs (so-called black holes), are treated as redexes that reduce the term, but leave it unchanged, which results in a non-terminating reduction for this term.
We call a reduction on the topmost redex an *n-o-reduction*. Furthermore, we denote a one-step reduction relation that reduces the n-o-redex by $\rightarrow_N$, Furthermore, we use $\stackrel{*}{\rightarrow}$ and $\stackrel{*}{\rightarrow}_N$ for transitive closure of $\rightarrow$ and $\rightarrow_N$, respectively, and $\stackrel{\leq 1}{\Rightarrow}$ and $\stackrel{\leq 1}{\Rightarrow}_N$ for denoting the relation generated by zero or one reduction $\rightarrow$ or $\rightarrow_N$, respectively.

**Lemma 8.** *Let $r, s, t$ be graphic expressions such that $r \rightarrow s \rightarrow_N t$, and $r \rightarrow s$ is not an n-o-reduction. Then there exists a graphic expression $s'$ such that $r \rightarrow_N s' \stackrel{\leq 1}{\Rightarrow} t$.*

*Proof.* Let the reduction be $r \rightarrow s \rightarrow_N t$ and $r \rightarrow s$ be no n-o-reduction. Let $r = (x_r, U_r), s = (x_s, U_s), t = (x_t, U_t)$, and let $e_r, e_s$ be the redex in $r$ and $s$, respectively. This is only possible, if $e_s$ is also an n-o-redex in $r$. The arguments in the proof of strong confluence in Theorem 5 and the fact that $e_s$ cannot be garbage, show that it is possible to commute the reductions. Let $s'$ be such that $r \rightarrow_N s'$ using redex $e_s$, and in the case of a redex `natchoice`, choosing the same number. Either $s'$ and $t$ have the same gc-simplified form, if the redex $e_r$ is garbage in $s'$, or $s'$ can be reduced to $t$ using the redex $e_r$ and in the case of `natchoice`, selecting the same number. Hence we have $r \stackrel{n}{\rightarrow} s' \stackrel{\leq 1}{\Rightarrow} t$.

**Theorem 9.** *Second Church-Rosser theorem.*
*Let $(e, U)$ be a graphic expression. Then every reduction of $(e, U)$ to a WHNF can be rearranged, such that first there is a normal order reduction to some WHNF, and afterwards a reduction to the final term.*

*Proof.* Let $(e, U) \xrightarrow{*} (e', U')$, where $e'$ is in WHNF. The last reduction before reaching a WHNF must be a n-o-reduction, which can be shifted to the start of the reduction. Using induction on the number of the reductions, we can rearrange this reduction as claimed.

An interesting property is that the length of an n-o-reduction sequence is not longer than an arbitrary one to a WHNF, and is hence the shortest one using our definition of reduction.
There is a surprising application of the result to the confluence properties of a constrained lambda-calculus [Man95]. This is a lambda-calculus, where constraints are permitted in the syntax. These can be seen as external functions that either fail, if constraints are not solvable, or that may instantiate variables in a non-deterministic way, if there is more than one solution. Our result is then that reduction is confluent without prescribing a reduction strategy. However, there is a small gap to the full lambda-calculus with constraints, since we consider a combinator reduction system without local definitions.

# 3 Computational Adequacy of the Natural Semantics

In this section we show that the natural semantics is adequate for deterministic expressions, i.e., expressions in the language $FP_0$ without `natchoice`.
We define a reduction system on the flat expressions, which we assume to be standardized, such that all bound variables have different names.

**Definition 10.** Reduction $\rightarrow_{flat}$ In these reductions we use replacement of a variable by an expression. In every case of a replacement, the bound variables in the expression are renamed to avoid name-clashes.

i.) $(\ldots(f\ t_1)\ldots t_n) \rightarrow r[t_1/x_1, \ldots, t_n/x_n]$ if the definition is: $f\ x_1 \ldots x_n = r$

ii.) $(\texttt{case}_A\ (\ldots(c_i\ t_1)\ldots t_n)\ \texttt{of}\ c_1 \rightarrow s_1;\ \ldots c_n \rightarrow s_n) \rightarrow s_i$.

iii.) $(\texttt{let}\ x = s;\ \texttt{binds in}\ t) \rightarrow (\texttt{let binds in}\ t[s/x])$

iv.) $(\texttt{letrec}\ x = s;\ \texttt{binds in}\ t) \rightarrow (\texttt{letrec}\ x = s;\ \texttt{binds in}\ t[s/x])$

v.) $(case_A\ (\texttt{letrec binds in}\ s)\ \texttt{of}\ c_1 \rightarrow s_1; \ldots c_n \rightarrow s_n) \rightarrow (\texttt{letrec binds in}\ (\texttt{case}\ s\ \texttt{of}\ c_1 \rightarrow s_1; \ldots c_n \rightarrow s_n)$

These reductions can be performed for any subterm of some term, including the terms on the right hand side of a `let` or a `letrec`.

In the following we intend to describe the correspondence between flat and graphic expressions and between $\rightarrow_{flat}$ and $\rightarrow$.
The notion "infinite printed representation" or infinite unravelling (in the terminology of [PvE93]) and the equality shall be the base for the comparison between reductions of flat and graphic representation of deterministic expressions. The printed representation is a potentially infinite ground expression (i.e. without let and `letrec`'s), which may also have the symbol $\bot$ at leaf nodes. In order to be able to compare different infinite print trees and to avoid a non-terminating printing algorithm, we write the definition of the algorithm such that it produces the print tree up to some given depth $m$.

**Definition 11.** Given a flat expression, its print-tree of depth $m$ can be computed as follows: This is a function $[\![.]\!]$ taking the expressions as argument, with three further arguments: A set of variables to indicate cyclic references, the depth, and an environment. The initial call is $\texttt{flat\_tree}\ (s) := [\![s]\!]\ \emptyset\ m\ \emptyset$

$$
\begin{array}{lcl}
[\![c]\!]\ V\ m\ \rho & = & c \\
[\![x]\!]\ V\ m\ \rho & = & \text{if } x \in V \text{ then } \bot \\
 & & \text{else } [\![\rho(x)]\!]\ V \cup \{x\}\ m\ \rho \\
[\![(st)]\!]\ V\ m\ \rho & = & \text{if } m = 0 \text{ then } \texttt{Bot} \\
 & & else([\![s]\!]\ \emptyset\ (m-1)\ \rho)\ ([\![t]\!]\ \emptyset\ (m-1)\ \rho) \\
[\![\texttt{case}\ s\ \texttt{of}\ c_i \rightarrow s_i]\!]\ V\ m\ \rho & = & \text{if } m = 0 \text{ then } \texttt{Bot} \\
 & & \text{else } (\texttt{case}\ ([\![s]\!]\ \emptyset\ (m-1)\ \rho)\ \texttt{of} \\
 & & \qquad c_1 \rightarrow [\![s_1]\!]\ \emptyset\ (m-1)\ \rho; \\
 & & \qquad \ldots; \\
 & & \qquad c_n \rightarrow [\![s_n]\!]\ \emptyset\ (m-1)\ \rho \\
[\![\texttt{let(rec)}\ x = t\ \texttt{in}\ s]\!]\ V\ m\ \rho & = & [\![s]\!]\ V\ m\ \rho \cup \{(x,t)\}
\end{array}
$$

**Definition 12.** The print tree for graphic expressions can be computed using the function `graph_tree` and $[\![.]\!]$ as above:

$$\texttt{graph\_tree}(x, U)\ m = [\![x]\!]\ \emptyset\ m\ U$$

**Definition 13.** i.) Two print-trees are *equivalent*, if for all depths $m$, the computed trees are identical.

ii.) Two expressions (flat or graphic) are *pt-equivalent*, iff the corresponding print trees are equivalent

**Lemma 14.** *Let-reductions do not influence the pt-equality of deterministic expressions.*

*Proof.* It is sufficient to argue that the print tree does not change after a `let(rec)`-reduction: If the let-reduction replaces a variable with a non-variable term, then the argumentation is easy. In case a variable is replaced by a variable, then nothing changes, since this is either a $\bot$-variable or a non-bot node in the tree.

**Lemma 15.** *Let $g$ be a graphic expression and let $e$ be a flat expression, such that $g$ and $e$ are pt-equivalent.*

*i.) If $g \to g'$, then there is some $g''$ and some $e'$, such that $e \xrightarrow{*} e'$, and $g' \xrightarrow{*} g''$ and $e'$ and $g''$ are pt-equivalent.*

*ii.) If $e \to e'$, then there is some $e''$ and some $g'$, such that $g \xrightarrow{*} g'$, and $e' \xrightarrow{*} e''$ and $g'$ and $e''$ are pt-equivalent.*

*Proof.* It is sufficient to consider $\delta$-reductions or `case`-reductions. Now, every redex in the expressions maps to a (possibly infinite) set of redexes in the print tree We can define the minimal set of the redexes in $g$ and $e$, such that the corresponding sets of redexes in the original print tree are the same. Reducing these redexes will give the same tree, where a path in the tree that shrinks from an infinite one to a finite one will have Bot at a leaf node.

**Theorem 16.** *Let $e$ be an expression and $g$ be the corresponding graphic expression. Then*

*i.) $g$ and $e$ are pt-equivalent i*

*ii.) $g$ has a normal form iff $e$ has a normal form. Furthermore, if both $g$ and $e$ have normal forms, then these normal forms are identical.*

# 4 Observational Properties of Functional Programs ($FP_1$)

If we look at functional programs as black boxes that interact with the environment, or with the operating system via interface functions, then the question of behavioural equivalence of programs must be expressed in terms of interaction sequences with the environment. In this section we consider only interface functions without memory, i.e., The set of possible values that an interface function can return is completely determined by the (fully evaluated) arguments. Therefore, we have to extend the core language in order to make the interface functions explicit and hence we will extend the syntax of sets of graphic expression, such that also the I/O-history can be compared. We do not use the Unix-model of a standard-input and output stream, but a simple blocking call-mechanism that transfers some data to the outside world, and then waits, until some data comes back, which are then consumed by the functional program.

## 4.1 Church-Rosser Properties

**Definition 17.** We add a new class of supercombinators: *interface supercombinators.* These combinators have a fixed arity and a monomorphic type, the arguments and the output are data objects, and they are hyperstrict, i.e., the arguments must be in normal form, before the interface supercombinator can be reduced. These interfaces have no defining body in the functional language. It is assumed that the mapping from arguments to the result is done externally. Furthermore the result may be non-deterministic, however, the functions have no memory, i.e., the set of possible values is only determined by the input arguments. The definition of redex and reduction is adapted to these interface supercombinators.

We can simulate the interface supercombinators using the `natchoice`-function as far as the resulting graphic expression is concerned. Thus we already can inherit the properties like the Church-Rosser Theorems, if we are not interested in the interaction with the operating system. I.e., we have:

**Theorem 18.** *Reduction on Graph expression has the following properties.*

*i.) Reduction is confluent on sets of graphic expressions*

*ii.) Every reduction to a WHNF can be rearranged into a normal-order reduction that requires an equal number of less reductions steps than the original reduction.*

*Proof.* The behaviour of the interface functions can be simulated using `natchoice`. This requires a potentially infinite number of cases in the definition of the interface function, however, this does no obstacle. The argumentation on the lengths of reduction steps can be inherited from section 2, if we do not count the reductions that are necessary to simulates the interface.

Now we consider the input-output behaviour of functions.

**Definition 19.** A *question-answer pair* consists of: i) the question, i.e., the interface function together with its arguments and ii) the result. A *reduction* $red$ from $a$ to $c$, where $a$ and $c$ are graphic expressions is a sequence $a \to_{r_1} \ldots \to_{r_n} c$, where $\to_{r_i}$ contains all the information to execute the reduction.
We are interested in the question-answer pairs of a reduction. If we consider them as a multiset, then we denote them by $QA_{MS}(red)$, and considered as a list, we denote them by $QA_L(red)$.

Now we can prove an improvement of the second Church-Rosser Theorem that takes into account the QA-multisets of the reduction.

**Theorem 20.** *Let* $red = a \to_{r_1} \ldots \to_{r_k} c$ *be a reduction, where* $c = f\ c_1 \ldots c_n$ *is a WHNF. Then there is a normal order reduction* $red_{no} = a \to \ldots \to d$*, such that* $d$ *is in WHNF and* $d = f\ d_1 \ldots d_n$*, and* $d_i$ *is reducible to* $c_i$*. Moreover we have* $QA_{MS}(red) \subseteq QA_{MS}(red_{no})$*.*

*Proof.* In a similar way as in the proof of the Church-Rosser Theorem 9 above, we can commute reductions until a reduction in normal order is obtained. The question-answer pairs are either permuted, or dropped, or shifted in the reduction after a WHNF has been reached.

This theorem states that for every reduction that is performed in an arbitrary order, the interactions with the world can be divided into necessary ones and redundant ones, where the necessary ones are exactly those that are required in the corresponding normal order reduction. However, the sequence of the questions may be permuted.
Note that the condition on hyperstrictness is only a pragmatical one, which may be dropped without loosing any nice properties. However, if hyperstrictness

does not hold, then we have to specify exactly to which extent every interface has to evaluate its arguments. This would complicate specification of interface functions and reasoning about the language. In general, the hyperstrictness restriction does not limit the expressiveness of the language, since functions that behave like non-hyperstrict interface functions can in general be implemented in the functional language using some hyperstrict interface functions.

### 4.2 Equivalence Relations on Programs

We shall define a notion of equivalent programs with memoryless interface functions. We assume that programs have a type that corresponds to a data object.

**Definition 21.** Let $A$ be an $FP_1$-program. Let $R_W(A) = \{(QA_M S(red), c) | red$ is a normal order reduction terminating with a CWHNF starting with constructor $c\}$.
Let $R_S(A) = \{(QA_L(red), c) | red$ is a normal order reduction terminating with a CWHNF starting with constructor $c\}$.

i.) Two programs $A$ and $B$ are called *weakly behaviourally equivalent*, iff $R_W(A) = R_W(B)$.

ii.) Two programs $A$ and $B$ are called *strongly behaviourally equivalent*, iff $R_S(A) = R_S(B)$.

The pairs for the non-terminating normal-order reductions: $(QA_M S(red), \perp)$ and $(QA_L(red), \perp)$ are not added to the definition. The intuition is that I/O contributes to the computation of a value, not vice versa. If the emphasis of the program is on side-effecting, then it might be necessary to modify these equivalencies to include these pairs (see also the strictness transformation in section 4.3). Unfortunately, it is hard to distinguish weakly equivalent programs on the base of their I/O-behaviour. If we have two programs in two black boxes that both use normal order reduction, where every call to an interface function is observable, and if our task is to prove that the programs are different, then we have a hard job. If both $A$ and $B$ yield the same result, and the multiset of question-answer pairs is different for A and B, we do not know whether the programs are different. Another situation, where we cannot conclude that A and B are different, is that the multiset of question-answer pairs is equal, but the result is different.

*Example 3.* . This example demonstrates the problems of weak equivalence. Let `ask_int` be an interface function that has no arguments and returns some integer

i.) `main = if (ask_int) > (ask_int) then 1 else ask_int`

reduction 1: The input sequence is 1,2,1; the result is 1

reduction 2: The input sequence is 2,1; the result is 1.

ii.) `main = if (ask_int) > (ask_int) then 1 else 2`

reduction 1: The input sequence is 3,2, the result is 1

reduction 2: The input sequence is 2,3, the result is 2

**Lemma 22.** .

*i.) Given a program $P$, the set $R_S(A)$ can be viewed as a (partial) function from lists of question-answer-pairs to results.*

*ii.) If for two programs $A$ and $B$, $R_S(A)$ contains a pair $(L, c)$, and $R_S(B)$ contains a pair $(L, d)$, where $c \neq d$, then $A$ is not strongly equivalent to $B$.*

This lemma means that strong equivalence of programs can be tested using the black box model of programs. If the same list of question-answer pairs yields different results for the two programs, then the programs are not strongly equivalent.
However, nice as it may be, strong equivalence means more or less that sequential order of normal order reductions has to be respected by an implementation. This prevents for example to exploit the information of a strictness analyser, it also prevents parallelisation of functional programs.
I am of the opinion that it is more desirable to permit concurrent execution and a lot of optimisations, and thus to use weak equivalence as a base for program transformations.

### 4.3 Transformations on $FP_1$-Programs

Now we have criteria to judge the validity of program transformations w.r.t. the defined equivalencies. Note that now all program transformations must be seen as transformations on graphic expressions rather than flat expressions.
We will have a look at the transformations mentioned in [PJS94]

*Transformation*: $\delta$-reduction, `case`-reduction (partial evaluation)
The $\delta$-reduction and `case`-reduction preserve strong equivalence of programs, as can be seen by inspecting the proof of the second Church-Rosser Theorem. However, it is not permitted to evaluate interface functions. The question-answer behaviour is not affected.

*Transformation*: interface reduction
This reduction does not preserve strong or weak equivalence, since after the reduction one question-answer pair is missing.
Now we consider some simple (humble) program transformation [PJS94].

*Transformation* `case`over `case`with functions as globally defined.
`case` (`case` $e$ `of` $c_1 \to s_1, \ldots, c_n \to s_n$) `of` $d_1 \to t_1, \ldots, d_m \to t_m$ transforms to:

$$\begin{array}{ll} (\texttt{let}\ x_1 = t_1, \ldots, x_m = t_m\ \texttt{in}\ (\texttt{case}\ e\ \texttt{of} & c_1 \to (\texttt{case}\ s_1\ \texttt{of}\ d_1 \to x_1, \ldots, \ldots, d_m \to x_m); \\ & \ldots; \\ & c_n \to (\texttt{case}\ s_n\ \texttt{of}\ d_1 \to x_1, \ldots, \ldots, d_m \to x_m)) \end{array}$$

where $x_i$ are new variables. This transformation preserves equivalence, since the normal order reduction sequence is not affected.

*Transformation*: Using strictness information, evaluate an argument of a function before evaluating the body.
This transformation is not correct w.r.t. strong equivalence, since question-answer pairs may occur in a different sequence after transformation. If it is known, that an argument will be evaluated if the function application is evaluated, then the evaluation of the argument can be done before evaluation of the body. In the case, that the function application does not terminate, but the argument is not evaluated, then the transformation is not correct.

*Example 4.* Consider the `length` function, then it is correct to evaluate the spine of the list before evaluating `length`.

*Example 5.* Consider a variant of iterative `length`

> `itlength` $xs$ $s$ $=$ `case` $xs$ `of` `nil` $\rightarrow$ $s$;`cons` $\rightarrow$
> `itlength` (`tail` $xs$)$(1+s)$

A strictness analyser will tell that this function is strict in both arguments. Evaluating $s$ before the `itlength` body is correct w.r.t. to weak equivalence, however, this may have some strange effects for infinite lists: Let `nat` $= [1..]$ and `seq` be a function that first evaluates the first argument, then the second, and results in the evaluated second argument. The expression

> (`itlength nat` (`seq`(`print` "list has terminated") 0 ))

prints nothing and runs into a loop before the transformation, but after the transformation, it first prints "list has terminated" and then runs into a loop.

*Transformation.* Common subexpression elimination: Is in general incorrect, since it destroys the sharing structure of the expression.

*Transformation.* function inlining
Is in general not correct w.r.t. the equivalencies. It is only correct, if the sharing structure is not changed. This is the case, if for some function to be inlined, the arguments occur at most once in the body.

**Definition 23.** Deterministic expressions.
This is an expression of a data object type, and there are no reductions of interface redexes necessary to evaluate it to normal form.

Within these expressions, all program transformations as in deterministic lazy functional languages can be used.
In the following we give some counter examples that demonstrate some pitfalls in transforming programs, and also in transforming functions from lambda-calculus into our language.

*Example 6.* Partial evaluation is in general not correct
Consider the following definition:
$f\ x\ y = ((\texttt{ask\_int}) + x) * y$
The evaluation of $(\texttt{map}\ (f\ 1)[1,2])$ in the permitted reduction sequence gives: $(\texttt{map}\ (f\ 1)[1,2]) \rightarrow [(f\ 1\ 1), (f\ 1\ 2)] \rightarrow [(a{+}1){*}1, (b{+}1){*}2]$, where $a$ and $b$ are the two different answers to the question $(\texttt{ask\_int})$. Using partial evaluation gives: $(\texttt{map}(f\ 1)[1,2]) \rightarrow (\texttt{map}(\lambda y.((\texttt{ask\_int}){+}1){*}y)[1,2]) \rightarrow (\texttt{map}\ (\lambda y.(a{+}1){*}y)\ [1,2]$ $\stackrel{*}{\rightarrow} [(a+1)*1, (a+1)*2]$, where $a$ is the answer to the one question.
This is different from the first if interpreted as a set of results.

*Example 7.* . The example above also shows that the translation of the lambda-calculus in the combinator calculus is not unique. Consider the expression

$$\texttt{map}((\lambda x.\lambda y.(\texttt{ask\_int} + x) * y)1)[1,2]$$

One possibility of translation is in the example above. The other one is to use lambda-lifting:

$$\texttt{map}(\lambda x.(\lambda z.\lambda y.(\texttt{ask\_int} + z) * y)\ x\ 1)[1,2]$$

This translates into a supercombinator definition and a modifed expression:

$$\texttt{F}\ z\ y = (\texttt{ask\_int} + z) * y$$

$$\texttt{map}\ (\lambda x.F\ x\ 1)\ [1,2]$$

A further translation gives:

$$\texttt{F}\ z\ y = (\texttt{ask\_int} + z) * y$$
$$\texttt{map}\ \texttt{G}\ [1,2]$$
$$\texttt{G}\ x = \texttt{F}\ x\ 1$$

### 4.4 McCarthy's `amb`

.
It is not possible to simulate McCarthy's bottom-avoiding, non-deterministic choice operator `amb`, which behaves as follows

$$\begin{array}{lcl} \texttt{amb}\ x\ \bot & = & x \\ \texttt{amb}\ \bot\ y & = & y \\ \texttt{amb}\ x\ y & = & \texttt{choice}\ x\ y \end{array}$$

This would permit to have a parallel disjunction $\vee_p$, which is implementable using `amb`:

$$x \vee_p y = \texttt{amb}\ (x \vee y)\ (y \vee x)$$

However, it is not hard to see that the existence of $\vee_p$ violates the second Church-Rosser theorem, hence `amb` cannot be defined in our language.
It is possible to simulate `choice` using `natchoice`, however, the other direction is not true, which can simply be seen as follows: natchoice can generate a non-deterministic choice between a countable number of elements in on step. However, choice must generate this set step by step, with at most duplicating the set in every reduction. Infinity can only be reached at the cost of non-termination.

*Example 8.* External store cannot be simulated using non-deterministic interfacing: As mentioned above, the `natchoice` functions permits us to simulate (the internal effects of) external interfacing functions. For example, the NEL-function `ASK_INT` [HNMH96] that prompts the user to input some integer can be simulated by the `natchoice` functions. More complicated functions can also be simulated. However, it is interesting to note, that it is not possible to simulate an external modifiable global variable:
A sequential execution that first writes $n$ in external store that later can be retrieved can not be simulated by memoryless interface functions, since this would contradict confluence:
Suppose the functionality is as follows: `OUTPUT` is a function that puts its argument in an external store, and returns `T`, and let `INPUT` be a function that retrieves that store. Assume the value in the external store is 0 before evaluation. Then consider the triple (`OUTPUT1`, `OUTPUT2`, `INPUT`).
It has set-reductions as follows:
1) (`T`, `OUTPUT2`, `INPUT`) $\rightarrow$ (`T`, `OUTPUT2`, 1) $\rightarrow$ (`T`, `T`, 1)
2) (`OUTPUT1`, `T`, `INPUT`) $\rightarrow$ (`T`, `OUTPUT2`, 2) $\rightarrow$ (`T`, `T`, 2)
Since we can determine all reduction sequences, we see that reduction 1 produces a different resulting (unit-) multiset than reduction 2. Thus the functionality that permits external store enforces non-confluence, hence such a functionality cannot be simulated in the language $FP_1$.

# 5 Enforcing Order of Evaluation: $FP_2$

The programming language $FP_1$, though at a low level, has sufficient expressiveness. However, in practice it may be helpful to have some sequentiality restrictions for the reduction. For interfacing with the external world, the sequentialisation primitives are not necessary for $FP_1$, but they are necessary for the languages $FP_3$ that permits to interface external memory and files. Thus this section can be seen as a preparation for the next section.
In this section we shall add two primitives to the language that are intended to control evaluation. The `seq` is a combinator of two arguments that first evaluates its first argument, then forgets the value and proceeds evaluating the second. The $\gg$ primitive indicates that for two (potential) redexes $x$ and $y$, only the reduction sequences are permitted that first perform one reduction step for $x$ and then reduce $y$. From a programmers point of view, `seq` permits to change the normal order sequence of programs, whereas $\gg$ does not change the normal order sequence, but enforces the compiler to take care that the normal order sequence of reduction for two redexes holds for all executions.

- There is an extra function `seq` with the following behaviour:
  (`seq` $x$ $y$): first it evaluates $x$ to WHNF, then the result is (the WHNF of) $y$.

- `let(rec)` is extended, respectively modified as follows:

$$\begin{array}{lll} <E> & ::= & \ldots | \texttt{let(rec)} <defs> <\texttt{seq}>^* \texttt{ in } <E> | \ldots \\ <\texttt{seq}> & ::= & <var> \gg <var> \end{array}$$

Environments $U$ have to be adapted to suit these sequentialisation pairs. We simply add them as marked pairs to the environment, denoted by $x \gg y$. Note that `seq` can be implemented using `strict` ([BW88]:
Let `K2` $x\ y = y$. Then `seq` = (`strict` $K2$)
The primitive $\gg$ cannot be used without restriction. In order to formulate these restrictions, we need some preparation. The sequentiality primitive $\gg$ can only be used in the body of super combinators, hence we will investigate this situation. First we define the notion of a potential redex in a super combinator body.

**Definition 24.** A *potential redex* in a body is

- a redex in the body
- a `case`-expression
- an expression $(\ldots(f\ x_1)\ldots)x_n)$, where $f$ is a strict super combinator like `seq` or a selector, and the arity of $f$ is $n$.

A body of a supercombinator can be represented as an environment with free variables, where the free variables are the argument variables. Let $U$ be the environment corresponding to the body of a super combinator, and $e$ be a non-variable expressions, then let the set of *aliasing variables* $AV_U(e)$ be the set of all variables that directly or indirectly point to $e$: $AV_U(e) = \{x | x\ R^*\ e\}$. For variables $x$, we define a similar notion: $A_U(x) = \{y | y\ {}^*R^*\ x\}$, where ${}^*R^*$ denotes the symmetric, transitive and reflexive closure of $R$. Now we can formulate the restrictions for $\gg$:

- `seq` and $\gg$ can only occur in the body of a supercombinator definition,
- For every supercombinator body $U$ and variables $x \gg y$, the variables $x$ and $y$ must be aliasing variables for potential redexes, and furthermore $x \in A_U(y)$.
- For every supercombinator $f$ of arity $n$ with body $U$ and variables $x, y$ with $x \gg y$ in $U$, no normal order evaluation of $f\ t_1 \ldots t_n$ evaluates the expression corresponding to $y$ before $x$, where normal order evaluation is meant ignoring the $\gg$-pairs.

The usage of $\gg$-pairs is stronlgy restricted. The restrictions are sufficient for programming purposes, such as to enforce evaluation of a condition before the evaluation of an expression in some alternative. The first-order restriction on potential redexes is not too severe a restriction. The restrictions are strong enough to prevent $\gg$-cycles. Furthermore, a proof of confluence is possible. It is not hard to find simple static analysis methods that are sufficient to ensure the constraints.

**Definition 25.** Redex and reduction.

- A *$FP_2$-redex* is a non-variable right hand side $e$ of a pair $(x, e)$ in an environment, Furthermore, for all $y \in A_U(x)$, there is no further variable $z$ with $z \gg y$. In addition, (`seq` $u$ $v$) is a redex, if $\texttt{spine}_U(u)$ is a WHNF.
- A reduction does not only replace the redex $s$ by a term, but also removes all pairs $x \gg y$, where $x \in AV_U(s)$.

The algorithm `unwind` now has to be adapted to the `seq`-redexes, but there is no change for the $\gg$-pairs, since these pairs have no influence on the sequence of the normal order reduction. The influence of the $\gg$-pairs are the permitted program transformations and the permitted concurrent executions. There is a difference between `seq`and $\gg$: The `seq`-constructs sequentialises evaluation by enforcing the normal order reduction to take a different path, whereas the $\gg$ does not influence the normal order reduction sequence, but is only an affirmation of the normal order sequence. In effect, the $\gg$ is only a special notation for preventing certain transformations.

*Example 9.* The $\gg$ is a method that is able to sequentialise the condition and alternatives of a `case`:

$$f\ x = \texttt{if}\ x > 0\ \texttt{then print}\ \text{“}+\text{”}\ \texttt{else print}\ \text{”}-\text{”}.$$

First executing the prints, and then computing the comparison, then deciding which alternative, is a valid execution sequence for this program. However, it is an undesired one. Using `seq` does not really help:

$$f\ x = \texttt{let}\quad x_1 = x > 0;\ x_2 = \texttt{print}\ \text{“}+\text{”};\ x_3 = \texttt{print}\ \text{“}-\text{”};\ x_4 = \texttt{if}\ x_1\ \texttt{then}\ x_2\ \texttt{else}\ x_3 \\ \texttt{in}\ (\texttt{seq}\ x_1\ x_4).$$

This permits the same reduction sequences as before. Any execution according to the semantics has no hints on the intended sequencing of the function. Using $\gg$, it is possible and easy to program the intended and correct reduction sequence:

$$f\ x = \texttt{let}\quad x_1 = x > 0;\ x_2 = \texttt{print}\ \text{“}+\text{”};\ x_3 = \texttt{print}\ \text{”}-\text{”};\ x_4 = \texttt{if}\ x_1\ \texttt{then}\ x_2\ \texttt{else}\ x_3; \\ x_1 \gg x_4;\ x_4 \gg x_3;\ x_4 \gg x_2 \\ \texttt{in}\ x_4$$

The only valid reduction sequence is to reduce $x_1 > 0$, then the `if`, and finally one of the `print` expressions.

**Theorem 26.** *Reduction in $FP_2$ is Church-Rosser.*

*Proof.* Adding the `seq`primitive is no problem, hence we concentrate on the $\gg$-pairs. We have to consider the case that there are two redexes $(x_1, e_1)$ and $(x_2, e_2)$. The restrictions enforce that for all variables $z$ in $A_U(x_1)$ and $A_U(x_2)$, there is no variable $y$, such that $y \gg z$. Reducing the redexes, we can find a common reduct, such that strong confluence holds, provided the reductions are possible in $FP_2$. If the reduction of $e_1$ is a $\delta$-reduction, then there can be some new $\gg$-pairs, but it is not possible, that $x_2$ is constrained, since all

new variables point also to new potential redexes. It is also not possible, that the reduction adds a variable to $AV_U(e_2)$, such that $x_2$ is sequentialised after another reduction. The reason is that reduction in $FP_2$ is defined, such after a reduction of a redex corresponding to y, all pairs y $\gg$ z are removed.
In summary, the reduction in $FP_2$ is strongly confluent, and hence confluent.

*Example 10.* Consider the following function:
`wrong` $x =$ `let` $y = 1 + x$; $z = 2 + x$; $y \gg z$ `in seq` $z$ $y$
This definition is not correct w.r.t. our restrictions, but it does not conflict with the confluence theorem, since we have not used the restriction that $\gg$ is compatible with normal order. However, it is not possible to reach a WHNF of (`wrong` 1) in a normal order reduction.

Using the restrictions, we can show the second Church-Rosser Theorem:

**Theorem 27.** *In $FP_2$, for every term that has a WHNF, a WHNF can be reached using normal order reduction.*

*Proof.* . It is easy to see that the Church-Rosser Theorem holds for $FP_1 +$ `seq`. Hence we prove this for the addition of $\gg$ to $FP_1$ extended by `seq`. It is sufficient to prove that every n-o-redex is also an n-o-$FP_2$-redex. Since this holds for the language $FP_1$+`seq`, we can assume there is a normal order reduction ignoring the $\gg$-pairs. This normal order reduction will now sometimes introduce $\gg$-pairs. However, the restrictions guarantee, that the $\gg$-pairs do not influence the normal order restriction, i.e., in the reduction, every used n-o-redex is a redex that is not constrained by $\gg$.

The unique typing method in Clean appears to rely on the normal order reduction [PvE95], which means that there is no solid base for distinguishing valid from non-valid program transformations. The only criterion is that the normal order reduction sequence should not be modified, if this influences the unique objects. The method of adding $\gg$-pairs could be a way to enhance the exactness of the methods for sequentialising programs and recognise the valid program transformations.

# 6 Access to Files and External Store - $FP_3$

The aim of this section is to demonstrate that using the interfacing to access files and other external store can be done by side-effecting functions. Since we are on a rather low language level, we shall give a rather simple but undecidable criterion that retains the Church-Rosser property. In this paper we do not investigate analysis techniques to ensure this property. There are already techniques either to enforce sequentiality and single-threadedness like monads [Wad90] or the system of uniqueness types [PvE95, SBvEP93b], that are sufficient for this condition.

**Definition 28.** Language extensions for $FP_3$:
$FP_3$ extends the language $FP_2$ as follows.

- There is a specific algebraic data type "external object". The attributes are:
  - unique identifier (for example a name)
  - lock-status (may be locked or unlocked)
  - version number (some natural number)
    This version number is used like a time stamp.
- External objects can be of a type copy / nocopy and deterministic / non-deterministic.

  This makes 4 different types. For example, if arrays are treated as external objetcs, then they may have the attribute copy, whereas databases should have the attribute nocopy. External objects like sensors may be candidates for the non-deterministic kind of external object. But also files may be viewed as non-deterministic, since they may answer with I/O-error on a read-request instead of the expected sentence.
- There are the following classes of interface functions on external objects:
  - read functions: locked external object as argument, no external object out
  - update functions: locked external argument in and out
  - lock/unlock functions
    `lock`: unlocked in and locked out
    `unlock`: locked in and unlocked object out.
  - copy: one copyable external object as input, a copy of the object as output.
- There is an automatic run-time bookkeeping of version-numbers. For every identifier of type external object, there is an entry in the bookkeeping table that keeps the actual version-number of the external objects and their lock-status.
- There are the following restrictions on functions and expressions:
  - The algebraic data type"external object" cannot be cased, i.e., there is no `case`-constant for this type.
  - there are interface-functions that may have external objects as a part of the input argument or as part of the output. We assume, that the input and the output, respectively, contain at most one such external object.
  - The object-id and the version-number determine the results of an interface function (either deterministically or non-deterministically). In the deterministic case, the result is always the same for the same arguments and the same version-number. In the non-deterministic case, the possible outcomes are in a set that is determined by the external object and the version-number.

- Update, `lock` and unlock have as result always the same external object with the version-number increased by 1.
- Initially, all external objects are locked.
- A copy function should have a new object-id in the output
- `update`, `read`, `copy` and `unlock` can only operate on an external object that is marked "locked" in the book keeping table. Lock can only operate on an external object that is marked as not locked.

For the next definition we assume that graphic expressions are gc-simplified.

**Definition 29.** An $FP_2$-program is *I/O-correct* iff for every possible reduction:

- every read, update, and unlock always uses the external object with the actual version-number and the lock-status "locked".
- every lock always uses the external object with the actual version-number and the lock-status "unlocked".

There is an alternative to this definition, requiring only for every normal order reduction that there are no accesses to old versions of external stores. This, however, has the disadvantage that it is not compatible with parallelisation and several optimisations done by a compiler.

**Lemma 30.** *If a program is I/O-correct , then for all possible reductions: If there are two different redexes for the same external object, and the redexes are interface-function applications, then these can only be read or copy functions and furthermore the version-numbers must be identical for these redexes.*

*Proof.* If there are two different redexes with interface applications, where one is neither a read nor a copy, then the two possibilities of reducing the redex show that at least one does not satisfy the I/O-correctness restriction.

This, however, is not a sufficient criterion for I/O-correct programs:

*Example 11.* It is not possible to use only criteria for overlapping redexes without looking for the version number:

$$\begin{array}{lll} \texttt{let} \;\; e_0 & = & \texttt{update} \ldots e; \\ \quad (a, e_1) & = & \texttt{update}\ e_0 \\ \quad \texttt{in}\ (\mathit{read}\ a\ e_0) & & \end{array}$$

In this expression, there is only one redex, namely update $\ldots e$. After evaluating this redex, there are two redexes, that violate the criteria above.

*Example 12.* Without the $\gg$-primitive it is not easy to write I/O-correct sensible functional programs without rearranging or reprogramming. Consider the function `w`, where we assume that `read` is some read-function for files and `write` writes something into the file.

$$\begin{array}{ll} \texttt{w}\ \mathit{file}\ a = & \texttt{if}\ (p(\texttt{read}\ \mathit{file}\ a))\ \texttt{then write}\ \mathit{file}\ 1 \\ & \texttt{else write}\ \mathit{file}\ 2 \end{array}$$

If the expression (`w file` $a$) is reduced, then the next expression is:

$$\texttt{if } (p(\texttt{read file } a)) \texttt{ then write file } 1 \texttt{ else write file } 2$$

which contains 3 redexes. Without $\gg$, there is always a wrong path for reductions: Without loss of generality assume that $(p(\texttt{read file } a))$ evaluates to `true`. We write the local version number as an index to the file.

$$\begin{array}{lll}
 & \texttt{if } (p(\texttt{read file } a)) \texttt{ then write file}_n\ 1 \texttt{ else write file}_n\ 2 & (\texttt{vnr} = n) \\
\rightarrow & \texttt{if } (p(\texttt{read file } a)) \texttt{ then write file}_n\ 1 \texttt{ else file}_{n+1} & (\texttt{vnr} = n+1) \\
\rightarrow & \texttt{if true then write file}_n\ 1 \texttt{ else file}_{n+1} & (\texttt{vnr} = n+1) \\
\rightarrow & \texttt{write file}_n\ 1 & (\texttt{vnr} = n+1)
\end{array}$$

There is a conflict: writing to an old version-nr of the same file.
The simple remedy using $\gg$, is to write the program as follows:

$$\begin{array}{lll}
\texttt{w file } a = & \texttt{let} & x_1 = (\texttt{read file } a);\ x_2 = \texttt{write file } 1; \\
 & & x_3 = \texttt{write file } 2;\ x_4 = \texttt{if } p\ x_1 \texttt{ then } x_2 \texttt{ else } x_3;\ x_4 \gg x_2;\ x_4 \gg x_3 \\
 & \texttt{in } x_4 &
\end{array}$$

If the expression is $(w\ \texttt{file}\ a)$, where $a$ is some constant, then the program is I/O-correct.

**Lemma 31.** *I/O-correct programs are strongly confluent on sets of graphic expressions.*

*Proof.* . Consider two different redexes in the same graphic expression. The only new case is that there are two interface functions on external objects involved. If the external objects are different, we have no problem. If there are two `read`- or `copy` -calls, there is no problem in commuting the calls, since we have assumed that the version-numbers are identical in this situation. In the case of other redexes, we have a contradiction to I/O-correctness since at least one of the redexes must be an interface call that updates the version-number, which would lead to inconsistency after this reduction.

**Theorem 32.** *I/O-correct programs are confluent.*

*Proof.* Follows from strong confluence.

Whether a program is I/O-correct is undecidable. However, there are several mechanisms that ensure this property, like monadic programming, or unique types.

*Example 13.* Strictness transformations.
Consider the same variant of iterative length as in section 4.3:

$$\texttt{itlength } xs\ s\ =\ \texttt{case } xs \texttt{ of nil } \rightarrow\ s;\ \texttt{cons } \rightarrow\ \texttt{itlength } (\texttt{tail } xs)\ (1+s)$$

This function is strict in both arguments. Evaluating $xs$ and $s$ before the `itlength` body is correct w.r.t. to weak equivalence. For `nat` $= [1..]$ the expression

(`itlength nat` (`seq` (`write file` "list has terminated") 0 ))

does not write something before the transformation, but after the transformation, it first writes "list has terminated" into the file and then runs into a loop. This appears to be a not intended effect of the strictness transformation.

*Example 14.* Correct version of Example 8.
Let `OUTPUT` $a\ e \to e$ be a function that puts its argument in an external store, and let (`INPUT` $e \to a$) be a function that retrieves that store. Assume the value in the external store is 0 before evaluation. Then consider the following program.

(`let` $x =$ `OUTPUT` $1\ e_0, y =$ `OUTPUT` $2\ e_0, z =$ `INPUT` $y$ `in` $z$)

This program is not I/O-correct, the redexes of (`OUTPUT` $1\ e_0$) and (`OUTPUT` $2\ e_0$) are the critical ones. An I/O- correct program using data dependency is:

(`let` $x =$ `OUTPUT` $1\ e_0, y =$ `OUTPUT` $2\ x, z =$ `INPUT` $y$ `in` $z$)

which is I/O-consistent, and has as only result the integer 2.

*Example 15.* If we extend the functional language to add more concurrency, then this has its limits, if also the Church-Rosser Theorems should hold. We consider the extensions, where `lock` is permitted to lock an (unlocked) external object without looking at its version number. The intuition is that an unlocked external object may be modified externally in an uncontrolled way. Then deadlocks are possible, which may be interpreted as non-termination. We argue that the Church-Rosser theorems are false. Consider the following example:

```
all xs = case xs of   nil → true;
                      y : ys → seq y (all ys)
all                [(lock e), (unlock e), (lock e)]
```

The expression has a normal form evaluation to true, however there is a reduction that first reduces the two `lock`-expressions, which results in a deadlock, hence confluence does not hold. A simple variation makes obvious, that the second Church-Rosser theorem is also violated:
Consider the expression

`all`$[$(`lock` $e$), (`lock` $e$), (`unlock` $e$)$]$

for which the normal order evaluation runs into a deadlock, whereas there is a reduction that results in `true`.

# 7 Comments on Compiling a High-level Language into $FP_i$

We do not intend to describe in full a high level non-deterministic functional language. Nevertheless, a high level language $FP_G$ should have more expressive power than the core language. It should have

- a `letrec` that can also define functions,
- Lambda-abstraction
- polymorphic typing
- pattern matching
- list comprehensions
- Type classes

In order to define a semantics for $FP_G$, there are always choices, since the transformations into the core language define their semantics. The following conventions are sensible ones.

- (`letrec`... $f\ pat = t\ \ldots$ `in`...) always means that the body $t$ of the function $f$ should construct a new graph.
- (`let(rec)`... $x = t \ldots$ `in`...) means that $x$ is a graph node.
- a function defined as $f\ () = t$ is a function with arity zero, no node will be generated.
- The constant functions defined on top level are nodes in the graph, i.e. they are treated as constants. This can be translated as placing them in a let surrounding the expression to be evaluated rather than as a top-level combinator in $FP_i$.

Functions defined using (`letrec`... $f\ () = t$ `in` $exp$) can now be lambda-lifted using the most simple form of extending $f$ by variable arguments, and defining $f\ x_1 \ldots x_n = t[x_1/y_1, \ldots, x_n/y_n]$. Then shift $f$ to top-level taking care that it has a unique name, and replace $f$ in the `letrec` expression and the right hand sides by $(f\ y_1 \ldots y_n)$, where $y_1, \ldots, y_n$ are the variables in $t$. We can use the same simple form of lambda-lifting for lambda-abstraction, and for other functions as defined in `letrec`ś. Note that lambda-lifting using maximal free expressions as done in [PJ87] is in general not correct in our non-determinisitc setting, since our graph construction convention does not hold. It is only correct for deterministic maximal free expressions.
In the presence of type classes as in Haskell, the compiler should not add dictionary parameters to constants that are defined as graph nodes. However, this should be no problem, since the type checker is able to detect this situation.

# 8 Related Work

There is a lot of research on adding non-determinism to functional languages or to implement I/O facilities. We do not want to comment on strict functional languages like Lisp , Scheme, or ML, where the reduction sequence is applicative order and hence fixed, since the approaches are not comparable.
The proposal for non-determinism and I/O in Id using M-structures [BNA91] is based on a non-strict, first-order functional language. The sequential barrier

introduced there is an interesting method that is able to synchronise concurrent execution of commuting external update operations, it is comparable to our $\gg$, though it puts a stronger condition on the execution. It may be interesting to investigate the compatibility of the approaches.
In the rest of this section, we compare our approach only with proposals for modern (higher order, polymorphic, non-strict, lazy) functional languages. The single-threaded polymorphic lambda-calculus proposed in [GH90] is similar to ours, it also uses sharing of nodes in the reduction, and it proves Church-Rosser properties if the programs are well-typed w.r.t. to a single-threaded polymorphic type system. They do not consider cyclic graphs nor memoryless interfacing. Predetermining the sequence of external updates is the base of many approaches, where the programming methodology may differ either using monadic programming or continuation based programming [Wad90, HPW$^+$92]. The monadic programming is the current standard in Haskell 1.3 [HAB$^+$96]. It is easy to use, since it has an imperative look at the surface, it is functional and safe, it proves I/O-correctness in our sense, and the Church-Rosser Theorems hold. Its disadvantage is that the amount of sequentialisation may be too high. This sequentialisation marks a trend away from declarativeness to procedural.
Another direction that follows the idea of analysing the program, whether the I/O-operations will be executed in a safe sequence is proposed in [PvE93, PvE95, SBvEP93a, GH90]. A so-called unique type system is implemented for the functional language CLEAN [PvE95]. This method allows writing side-effecting interfacing, so long as the type-system can verify that there is no sharing of external references, and that the normal order sequence evaluates reads before updates. The foundational work on this unique type system is in [PvE95, SBvEP93a]. The reduction sequence in Clean appears to be fixed as a normal order reduction, hence there may be a problem in recognising correct program transformations if a strictness analyser and the unique type system are in a conflict.
There is an approach using sets in a functional core language to express non-determinism [HO90, HO89]. This approach is different from ours, since it can express `amb`.
It is interesting that our approach only works, if sharing of expressions is treated correctly. There are several recent papers that discuss a modellig of sharing in the $\lambda$-calculus [AFM$^+$95, Lau93, PS92, Yos93, GH90]. A difference to these works is that our approach is based on a calculus of supercombinators with fixed arities. Furthermore, due to our algorithmic definition of reduction, we have no problems in proving confluence also in the presence of cyclic reduction graphs.
The use of linear types as proposed in [Wad90, WR91] is also a method to prove that the execution of a functional program is safe in the presence of I/O's, and can be used to prove I/O-correctness.

# 9 Conclusion

The non-deterministic functional language $FP_G$ is able to perform interaction with a user or a file-system without being forced to introduce to much unnecessary sequentialisation. The following interactions are possible without loosing the important properties of pure functional languages:

- Opening a filled window and getting back an answer in a synchronous way, i.e., the execution waits, until the window contents are submitted.
- Asking a random number generator (rnd) without sequentialising the program.
- Printing in an arbitrary sequence, if the result of the print is like a non-deterministic function.
- Updating and reading several external files. The necessary sequentialisation is not on the whole external world, but can be done independently for every file. The read accesses have to be sequentialised
- Using an array that can be updated in-place and also copied. In this case, the array is viewed as an external object. This is not problematic for an array of fully evaluated data objects. Our treatment does not directly cover the case of an array where functions or closures are permitted as entries.

A problem may be the discipline by the programmers. For example it is possible to use an interface function to simulate an external store without informing the compiler. This programming style is not justified by our treatment of the topic.

# References


[Ach96] P. Achten. *Interactive functional programs: models, methods and implementation.* PhD thesis, Computer Science Department, University Nijmegen, 1996.

[AFM+95] Z.M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Principles of programming languages*, San Francisco, California, 1995. ACM Press.

[Aug84] Lennart Augustsson. A compiler for Lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 218–227, 1984.

[Bar84] H.P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* North-Holland, Amsterdam, New York, 1984.

[BNA91] P.S. Barth, R.S. Nikhil, and Arvind. M-structures: Extending a parallel non-strict functional language with state. In *Proc. Functional Programming Languages and Computer Architecture 1991*, LNCS 523, pages 538–568, 1991.

[BvEG+87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In *Proc. Parallel Architectures and Languages Europe (PARLE 87)*, LNCS 259 (2), pages 141–158, 1987.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall International, London, 1988.

[GH90] J.C. Guzman and P. Hudak. Single-threaded polymorphic lambda-calculus. In *Proc. of 5th IEEE Symposium on Logic in Computer Science*, pages 333–343, 1990.

[Gor94] A.D. Gordon. *functional programming and Input/Output*. Cambridge University Press, 1994.

[HAB+96] K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fairbairn, J. Fasel, A. Gordon, M. Guzmán, J. Hughes, P. Hudak, T. Johnsson, M. Jones, D. Kieburtz, R. Nikhil, W. Partain, J. Peterson, S. Peyton Jones, and P Wadler. Report on the programming language haskell 1.3. Technical report, Department of Computer Science, University of Glasgow, 1996.

[HNMH96] N.W.O. Hutchison, U. Neuhaus, Schmidt-Schauß M., and C.V Hall. Natural expert: A commercial functional programming environment,. *J. of Functional Programming*, 1996. to appear.

[HO89] J. Hughes and J. O'Donnell. Expressing and reasoning about non-deterministic functional programs. In *Glasgow workshop on functional programming 1989*, Workshops in Computing, pages 308–328. Springer-Verlag, 1989.

[HO90] J. Hughes and J. OD́onnell. Nondeterministic functional programming with sets. In *IV Higher Order Workshop*, Workshops in Computing, pages 11–31. Springer-Verlag, 1990.

[HPW+92] Paul Hudak [ed.], Simon L. Peyton Jones [ed.], Philip Wadler [ed.], Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell. A non-strict purely functional language. version 1.2, 1992.

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM Conference on Compiler Construction, Montreal*, pages 58–69, 1984.

[JS91] S.B. Jones and A.F. Sinclair. On input and output in functional languages. In *Esprit research reports, proj. 302 (1), Prospects for functional programming in software engineering*, pages 139–171. Springer-Verlag, 1991.

[KJMdVF93] J.R. Kennaway, Klop J.W., Sleep M.R., and de Vries F.J. An infinitary church-rosser property for non-collapsing orthogonal rewriting systems. In Sleep M.R. et. al., editor, *Term Graph Rewriting*. John Wiley, 1993.

[Lau93] J Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th Principles of Programming Languages*, 1993.

[Man95] L. Mandel. *Constrained Lambda Calculus*. Verlag Shaker, Aachen, Germany, 1995.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J.Comp.Sys.Sci*, 17:348–375, 1978.

[NSvP91] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Concurrent Clean. In Springer Verlag, editor, *Proc of Parallel Architecture and Languages Europe (PARLE'91)*, number 505 in Lecture Notes in Computer Science, pages 202–219, 1991.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International, London, 1987.

[PJS94] Simon L. Peyton Jones and André Santos. Compilation by transformation in the Glasgow Haskell Compiler. In *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 184–204. Springer, 1994.

[PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages, Charleston, South Carolina,*, pages 71–84. ACM, 1993.

[PS92] S. Purushothaman and J. Seaman. An adequate operational semantics of sharing in lazy evaluation. In *Proc. ESOP 92*, LNCS 582, pages 435–450. Springer-Verlag, 1992.

[PvE93] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, Workingham, 1993.

[PvE95] R. Plasmeijer and M. van Eekelen. Concurrent clean: Version 1.0. Technical report, Dept. of Computer Science, University of Nijmegen, 1995. draft.

[SBvEP93a] Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. Technical Report technical report 93-04, University of Nijmegen, Department of Computer Science, 1993.

[SBvEP93b] Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. Technical report, University of Nijmegen, Department of Computer Science, 1993.

[SS91] M. Schmidt-Schauß. External function calls in a functional language. In *Proc. of the 1991 Glasgow workshop on functional programming*, Workshops in Computing, pages 324–331. Springer-Verlag, 1991.

[Wad90] P. Wadler. Comprehending monads. In *Proceedings of Symposium on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990. ACM.

[WR91] D. Wakeling and C. Runciman. Linearity and laziness,. In *Proc. functional programming languages and computer architecture*, LNCS 523, pages 215–240. Springer-Verlag, 1991.

[Yos93] N. Yoshida. Optimal reductions in weak-$\lambda$-calculus with shared environments. In *Proc. functional programming languages and computer architecture*, pages 243–252. ACM press, 1993.