FUNDIO: A Lambda-Calculus With letrec, case, Constructors, and an IO-Interface: Approaching a Theory of unsafePerformIO

Technical Report Frank-16

Manfred Schmidt-Schauß

Institut für Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt, Germany
E-mail: schauss@ki.informatik.uni-frankfurt.de

Date: September 25, 2003

Abstract. This paper proposes a non-standard way to combine lazy functional languages with I/O. In order to demonstrate the usefulness of the approach, a tiny lazy functional core language "FUNDIO", which is also a call-by-need lambda calculus, is investigated. The syntax of "FUNDIO" has case, letrec, constructors and an IO-interface: its operational semantics is described by small-step reductions. A contextual approximation and equivalence depending on the input-output behavior of normal order reduction sequences is defined and a context lemma is proved. This enables to study a semantics of "FUNDIO" and its semantic properties.

The paper demonstrates that the technique of complete reduction diagrams enables to show a considerable set of program transformations to be correct. Several optimizations of evaluation are given, including strictness optimizations and an abstract machine, and shown to be correct w.r.t. contextual equivalence. Correctness of strictness optimizations also justifies correctness of parallel evaluation.

Thus this calculus has a potential to integrate non-strict functional programming with a non-deterministic approach to input-output and also to provide a useful semantics for this combination.

It is argued that monadic IO and unsafePerformIO can be combined in Haskell, and that the result is reliable, if all reductions and transformations are correct w.r.t. to the FUNDIO-semantics. Of course, we do not address the typing problems the are involved in the usage of Haskell's unsafePerformIO.

The semantics can also be used as a novel semantics for strict functional languages with IO, where the sequence of IOs is not fixed.

Table of Contents

1 Introduction

The aim of this paper is to describe an alternative way for investigating the properties of a lazy functional program with side effects and the correctness of program transformations and evaluation strategies. It is based on work on sharing in lazy functional programs [AFM$^+$95,AF97,MOW98], on the experience with a side-effecting lazy functional programming language [HNSSH97], and on investigations of non-deterministically extended lambda-calculi [KSS98,MSC99,Kut00]. The common method to relieve the programming language designer from the inherent IO-problems is to shift responsibility to the programmer who has to sequentialize all IO-requests. This is also true for the monadic approach implemented in Haskell [ABB$^+$99]. The approach in this paper tries to demonstrate that there is a possibility to combine the advantages of lazy functional programming and a more declarative IO where the semantics of the program allows to change the sequence of IO-requests.

The basic question is: "What is a correct (operational) semantics of such a programming language?"

As language we use a functional core-language which is an untyped lambda-calculus extended with `letrec`, `case`, constructors, and an IO-function. The basic IO-action is that the program prints a question, and the user (the operating system) answers by inputting the answer. Questions and answers are modelled by constants, which may encode finitely many alternatives. This restrictions keep reasoning simple and retain sufficient expressiveness.

The IO-behavior of a single IO-action is described by an IO-pair (question-answer-pair), and the IO-behavior of an evaluation is captured in a multiset of IO-pairs.

The operational semantics is defined by a normal order reduction. Based on this, program equivalence is defined by contextual equivalence similar to [GP98,Pit97], but extended by the observation of the multiset of IO-pairs.

Our approach models the input as well as the output part of IO and also the correlation between question and answer in contrast to a combination of an erratic non-deterministic `choice` with a lambda-calculus respecting sharing as in [Kut00,MSC99] which has turned out to lose too much information (see Example 6.10). Our approach also proposes a separation into a deterministic program and a non-deterministic user or environment, respectively, which avoids the need for search during evaluation. This is an advantage over the work in [Mor98,MS99,MSC99], where non-deterministically extended lambda calculi are analyzed for their use in modelling IO by encoding the stream processor in a non-determinisitc calculus. The fudgets as IO-concepts are then coded using the stream processor.

In investigations in non-deterministic programs an often referred-to example is the comparison of two programs P_1, P_2, the first evaluates to a constant a, the second is a non-deterministic choice between a and $\bot$. The natural encodings of these programs in FUNDIO are obviously non-equivalent, since non-deterministic choices in FUNDIO can only be done by asking the environment, and so P_1 terminates without querying the environment, whereas P_2 has to make at least one query (see Example 6.17).

A further example is the double-example $(\lambda x.x+x)\ (1 \oplus 2)$, where $\oplus$ means non-deterministic choice. If usual beta-reduction is performed, then the expression $(1 \oplus 2)+(1 \oplus 2)$ may result in the set $\{2, 3, 4\}$; if non-deterministic choice is applied first, then the possible results are in the set $\{2, 4\}$. Using beta-reduction for the expression $(\lambda x.2 * x)\ (1 \oplus 2)$ gives only the set $\{2, 4\}$. However, the programs $(\lambda x.x + x)$ and $(\lambda x.2 * x)$ should be equivalent, which rules out several calculi.

Though we have kept it as simple as possible, the calculus FUNDIO is rather complex, hence it is far from clear that anything sensible can be proved about correct program transformations, standard reduction, optimization, etc. In this paper we show that it is indeed possible to show that a sufficiently rich equa-

tional theory holds for expressions, that the deterministic reductions are correct, and that a majority of the optimizations in [JS98] remain correct. So we have demonstrated that contextual equivalence provides a useful semantics and that operational techniques are adequate to show sufficiently many program transformations to be correct.
The following results are obtained.

1. All reduction rules of FUNDIO are correct program transformations, of course with the exception of the evaluation of IO-functions. Moreover, further reductions like garbage collection and compressing indirections and "unique copy" are also correct program transformations.
2. Modifying the evaluation by intermediate garbage collection, compressing indirections does not change the meaning of programs.
3. Modifying the evaluation exploiting strictness of abstractions does not change the meaning of FUNDIO-programs. Moreover, strictness optimization does not save any reduction. The optimization effect can only be observed as saved heap-update-operations using an abstract machine.
4. FUNDIO is much better suited to model direct-call-IO than a plain nondeterministic lazy functional programming approach.
   For example there are two programs (see also 6.10): An "and" and an "or"-program of two Boolean inputs, which can be distinguished in FUNDIO, whereas their natural nondeterministic encodings cannot be distinguished in the nondeterministic setting.

It is also possible to combine monadic IO and `unsafePerformIO` in a functional encoding (see subsection 26.2), which is consistent with the correctness proof for monadic state in [AS98]. In a Haskell compiler that seriously compiles the `unsafePerformIO` function, the FUNDIO-semantics can be used as a guideline for correct program transformations.
We have tried several other combinations of syntax, reduction rules and contextual equivalence. After identifying lots of wrong choices, designing FUNDIO's rules and equational theory is a kind of natural choice. The interchangability of IOs is essential for permitting parallel evaluation, since different threads may in general produce different sequential appearance of IOs that are independent of each other. The same holds for strictness optimizations, which are only possible if the definition of equality permits interchanging IOs. So the correctness of strictness optimizations is directly connected with the possibility of parallel evaluation.

2 Related Work

Since it is beyond the scope of this paper to survey all relevant papers and related work on the combination and application of combinations of lambda calculus, non-determinism, different evaluation strategies, sharing, IO, etc., we mention papers that proposed similar methods, and though in our opinion all these papers have their value, we will mostly point out the differences to the calculus FUNDIO. We will also give hints on the applications of FUNDIO.

2.1 Lambda Calculi and Non-Determinism

There are approaches to combine lambda-calculi and non-determinism while retaining the copying beta-reduction (cf. [Ong93,DP95,DCdP94,DCTU99]). These are completely different from FUNDIO, since the above mentioned double-problem arises. An early discussion on the properties of different combinations of lambda calculi and non-determinism is in [Cli82], which contains several instructive examples and counter-examples, however the idea of sharing was not taken into account.

2.1.1 Remedy: Fixed Evaluation Order Fixing the evaluation order in a functional programming language, for example to insist on strict evaluation makes the combination of lambda-calculi and non-determinism more useful in practice. Extended lambda-calculi and reasoning along these lines are described in e.g. [Las98,LP00]. For example, this is the way, Lisp, Scheme, and ML perform their communication with the environment. A disadvantage is that the evaluation order is then part of the language's semantics and cannot be changed. Moreover, the equivalence of programs is more or less restricted to expressions that are free of external calls. In the double-example, the consequence is that first the choice is made, and then the beta-reduction, while the other is forbidden.

If a lazy strategy is chosen, while retaining beta-reduction, then the undesired effect is that non-deterministic calls are copied without control, and hence multiplied. Thus we run into the double counter-example. A call-by-need strategy, which modifies beta-reduction by sharing the argument expressions instead of copying, appears promising. To follow this approach means to accept that sharing is now not only an optimization, but a necessary ingredient in the semantics, and as a consequence beta-reduction is not used, but a modofied variant. This approach is followed in FUNDIO.

2.1.2 Remedy: Sharing That sharing could remedy some problems like the double-problem above in the combination of non-determinism and lambda-calculus was already observed in [AC79].

There are related call-by-need calculi that are based on sharing, but without non-determinism: e.g. [Yos93], explicit substitutions [ACCL91], and let-based lambda-calculi [AFM$^+$95,MOW98,AF97]. These approaches have no non-deterministic operators, which explains that they permit equations that are wrong in FUNDIO, e.g. shifting let over lambda [ACCL91,AC79]. That this rule is wrong in FUNDIO is explained in Example 6.21. The notion of equality of programs is often the equivalence closure of the reduction rules ([AFM$^+$95,MOW98,AF97]), which is weaker than contextual equivalence, and also heavily depends on the formulation of the reduction rules. The calculus FUNDIO can be seen as a non-deterministic extension of a call-by-need lambda calculus presented in [AFM$^+$95,MOW98], however, the FUNDIO-semantics is based on contextual equivalence. It can also be seen as an extension of the calculus in [MSC99].

It should be noted that sharing of subexpressions (i.e. using a graph) is not sufficient; it is also required that the exact scope of every shared term is known. For example, the let-over-lambda reduction does not change the sharing structure of the term, but only the scope of the sharing of the term. The `let`-construct opens a scope and identifies the shared term, hence it is the appropriate method to model sharing for our purposes.

2.1.3 Lambda-calculi, Non-determinism and Sharing Usually, a denotational semantics for nondeterministic programming languages is given by powerdomain constructions [Plo76,Smy78,Bro86]. However, these papers do not take into account the sharing technique. In a paper of Søndergard and Sestoft [SS92] on semantics of nondeterminism, twelve different kinds of non-determinism in non-strict functional languages are identified. Concerning their categories, FUNDIO has an erratic, restrained, singular nondeterminism (up to IO-multisets). However, we use a modified notion of unfoldability since FUNDIO employs (lbeta) exploiting sharing, instead of (beta), such that in FUNDIO there is no incompatibility between singular semantics and unfoldability.
The paper [PS92] uses lazy PCF plus sharing via explicit substitutions. This appears to fit into the FUNDIO-framework, though it is not clear whether the paper has inherited the let-over-lambda rule of explicit substitutions.

2.1.4 Different Kinds of Non-Deterministic Operators There are different views of the non-deterministic choice. One is the bottom-avoiding choice operator `amb`, which can be seen as a choice between two possibilities, however if there is a choice between a non-terminating argument and a terminating one, then the terminating one has to be chosen (see for example [HO90,HO89,HM92]). The operational net effect is that search and backtracking are required by evaluation. The non-determinism in FUNDIO is more like a committed choice, which was also considered in [KSS98,Kut00,MSC99].

2.1.5 Strict functional languages, effects and I/O The treatment of effects and I/O in strict functional programming languages using operational semantics based on the syntax was done by several authors: [MT91,CG94,MST96,Fel78,FH92]. The usage of contextual equivalence was advocated in [MT91,CG94,MST96,Pit97,PS98]. The approach of using operational semantics and contextual equivalence as a basis for proof techniques is also chosen for the non-strict calculus FUNDIO since it permits the most general notion of equality. In the case of I/O, the difference between FUNDIO and other approaches like [CG94] is that FUNDIO has a more declarative view of I/O: it is possible to interchange I/Os, whereas this is not possible in the operational theories of I/O in ML-like languages.

2.1.6 Non-Strict Functional I/O and Non-Deterministic Operators The intended application of FUNDIO is to provide a theory of equality of expressions for the treatment of I/O in non-strict functional languages, like the call to `unsafePerformIO` in Haskell [ABB$^+$99], There are several proposals for I/O in non-strict functional languages (cf. [Gor94]), for example, monadic I/O ([Wad92,Wad95,PJW93], continuation passing I/O, and unique typing ([Ach96]. For example, Gordon writes: "…, as is well-known, side-effecting I/O does not combine well with lazy languages. ". In this paper we want to argue that it is not that bad, in particular if `unsafePerformIO` is used. The monadic approach appears to be not the end of the story: there are several papers that demand more freedom and/or functionality: [PJGF96,JME99,Jon01], which also propose a "pure use" of `unsafePerformIO`. In [Jon03] Simon Peyton Jones proposed to use commutative monads to obtain a possibility to interchange independent I/Os.

A (commercial) non-strict functional language with side-effecting I/O [HNSSH97] provided as interface to the external world a possibility like Haskell's `unsafePerformIO`. Experience from the implementation shows that some standard optimizations do not work in such a language, or that they only work as expected if slightly restricted. For example, lambda-lifting as described in [PJ87] has to be simplified to lifting variables only. The functional language in [HNSSH97] is an example for a language that could be based on FUNDIO.

In [KSS98,Kut00] the lambda-calculus is built upon a combination of call-by-need and non-determinism, but there are no constructors. The equality is a contextual equivalence based on termination and non-termination. It contains a thorough analysis of the equational theory and develops a proof technique based on reduction diagrams.

The paper [MSC99] is a non-deterministic lambda calculus with `letrec` and constructors intended as a basis for the use of an I/O-interface with fudgets implemented by a stream processor. The equality is a contextual equivalence also based on termination and non-termination. The language is restricted such that only applications $(s\ x)$ for variables x are permitted. Their proof technique is based on an abstract machine. FUNDIO does not have the language restriction, however, the results in this report imply that the abstraction rule $(s\ t) \rightarrow (\texttt{letrec}\ x = s\ \texttt{in}\ t)$ is correct, and hence the equational theories in FUNDIO and [MSC99] are strongly related.

The approaches in [KSS98,Kut00,MSC99] are unable to distinguish the two (intuitively different) programs in example 6.10, which is remedied in FUNDIO by adding a notion of trace into the semantics.

Haskell-98 uses monadic I/O, which can be seen as a device that enforces a predictable sequence of I/Os, together with other "unsafe" I/O-operators like `unsafePerformIO`. A current evaluation of the compatibility of the transformations done by the Haskell-98 compiler and the FUNDIO-semantics is worked out in [Sab03], the result being that after switching off several optimizations, Haskell-98 is safe in the sense of the FUNDIO-semantics.

The non-strict functional programming language Clean combines lazy functional programming with IO and therefore uses a unique type system to ensure a single-threaded use of external entities. One disadvantage of this approach is that program transformations and optimizations can only transform a program into another program that has the same normal-order reduction. A rigid notion of contextual equality of programs under a different sequence of evaluation is missing in Clean. As mentioned above, graph-rewriting alone is insufficient, as long as the scope of the sharing is not in the syntax.

2.1.7 Functional-Logic Programming and Non-Deterministic Operators There is work on functional-logic programs with non-determinism and sharing (see e.g. [Liu93]). [AHH$^+$02] describe a natural semantics for a functional logic programming language, which is compatible with the FUNDIO-semantics. However, there is no treatment of I/O and equality of expressions, in particular no contextual semantics. A non-deterministic lambda calculus with constraints is modelled in [MS94].

2.1.8 Constrained Lambda-Calculus The work of L. Mandel [Man95,CMW96,MS94] on a combination of constraints and lambda-calculus can also be seen as a variant of combining non-determinism with lambda-calculus, where non-determinism stems from constraints which may have several solutions. The calculi in this work are mostly call-by-value (i.e. strict) ones. The application of the FUNDIO-calculus and its methods to a lazy variant of the constrained lambda-calculus seems promising and remains to be explored.

3 Overview

FUNDIO is a call-by-need lambda-calculus extended with `letrec`, `case`, constructors and an `IO`-function. The evaluation is based on a normal order reduction, which is deterministic for all reductions with the exception of an IO-reduction, which may have several outcomes. The reduction rules are designed carefully. The guidelines are that copying is avoided where possible, and instead a sharing using `letrec` is used; the first principle is that abstraction can be (syntactically) copied. Copying constructor expressions like (`cons` s t) is prohibited. Only if a `case`-reduction requires the information which constructor is present, there is a desharing that transforms (`cons` s t) more or less into (`letrec` $x = s, y = t$ `in` `cons` x y) and then permits to copy (`cons` x y).
Contextual approximation and equivalence are defined as

$$s \leq_c t \text{ iff } \forall C[\cdot], (\forall P : C[s]{\Downarrow}(P) \Rightarrow C[t]{\Downarrow}(P)) \wedge (C[t]{\Uparrow} \Rightarrow C[s]{\Uparrow})$$
$$s \sim_c t \text{ iff } s \leq_c t \wedge t \leq_c s$$

where P denotes the multi-set of IO-pairs made in an evaluation. This means that equivalence is based on termination for IO-multisets. The second condition means that either both terms have an error (i.e. there is a reduction to a term that has no terminating reductions) or both are error-free.
The proofs for correctness of program transformations are based on this contextual equivalence and rely on an analysis of overlappings of left hand sides of reduction rules and on reduction diagrams.

The main method for proving program transformations to be correct is a context lemma which reduces the requirement for all contexts to reduction contexts; and complete sets of reduction diagrams for the different rules. Finally we show that all deterministic reduction rules are correct plus several other equalities like garbage collection. It is also shown that different evaluation strategies can be used, e.g. modifications using strictness information and also using a strategy "eager-copy" that connects the evaluation to an abstract machine.

From a Haskell perspective, the `IO`-call is like a `unsafePerformIO` combined with `getChar` and/or `putChar`. The treatment of the world is that every time such a call occurs, there is a fetch of a new world, and after the call, the world is discarded. This means that FUNDIO-semantics does not model the outside world and treats all I/Os as independent from each other. Programs which are different in FUNDIO are also different w.r.t. a more sophisticated theory also modelling an outside world with a memory. This view also shows how to correctly connect a lazy functional programming language with a world that may change during the calls independently from the functional program. Thus this can be used for measuring physical entities, for random events outside of the program, and other changing environments.

4 Syntax of FUNDIO

The syntax of the language FUNDIO is as follows:
There are finitely many constructors c, every constructor has an arity $ar(c)$. Let $N \geq 2$ be the number of all constructors. The constructors are indexed, and c_i denotes the i^{th} constructor. The last constructor c_N is the constructor called `lambda` of arity 0, which however is only permitted in patterns.
The syntax for expressions E, case alternatives Alt and patterns Pat is as follows:

$$\begin{aligned} E &::= V \mid (c\ E_1 \dots E_{\mathrm{ar}(C)}) \mid (\texttt{IO}\ E) \mid (\texttt{case}\ E\ Alt_1 \dots Alt_N) \mid (E_1\ E_2) \\ &\quad\mid (\lambda\ V.E) \mid (\texttt{letrec}\ V_1 = E_1, \dots V_n = E_n\ \texttt{in}\ E) \\ Alt &::= (Pat \to E) \\ Pat &::= (c\ V_1 \dots V_{ar(c)}) \end{aligned}$$

where E, E_i are expressions, V, V_i are variables, where c in expressions E may be c_i for $i \in \{1, \dots, N-1\}$, and c in patterns Pat may be c_i for $i \in \{1, \dots, N\}$. The variables in a pattern Pat must be different, and also new ones. In a `case`-expression, for every constructor $c_i, i = 1, \dots, N$ there is exactly one alternative with a pattern of the form $(c_i\ y_1 \dots y_n)$. The expressions $(c\, E_1 \dots E_{\mathrm{ar}(C)}) \mid (\texttt{IO}\, E) \mid (\texttt{case}\, E\ Alt_1 \dots Alt_N) \mid (E_1\ E_2) \mid (\lambda\, V.E) \mid (\texttt{letrec}\, V_1 = E_1, \dots V_n = E_n\ \texttt{in}\ E)$ are called *constructor application*, `IO`-*expression*, `case`-*expression*, *application*, *abstraction*, or `letrec`-*expression*, respectively.

The constructs `case`, `IO` and the constructors c_i can only occur in special syntactic constructions. Thus expressions where `case`, `IO`, or a constructor is applied to a wrong number of arguments are not allowed.
The structure `letrec` obeys the following conditions: The variables V_i in the bindings are all distinct. We also assume that the bindings in `letrec` are commutative, i.e. `letrec`s with interchanged bindings are assumed to be syntactically equivalent. We also allow `letrec`s without bindings. `letrec` is recursive: I.e., the scope of x_i in ($\texttt{letrec}\ x_1 = E_1, \ldots x_j = E_j, \ldots\ \texttt{in}\ E$) is E and all expressions E_i. This defines closed, open expressions and α-renamings. For simplicity we use the distinct variable convention. I.e., all bound variables in expressions are assumed to be distinct. The reduction rules are assumed to implicitly rename bound variables in the result by α-renaming if necessary to obey this convention. In FUNDIO, this is only necessary for the copy rule (cp). We also use the convention to omit parenthesis in denoting nested applications: $(s_1 \ldots s_n)$ denotes $(\ldots(s_1\ s_2)\ldots s_n)$.
To abbreviate the notation, we will sometimes use ($\texttt{case}\ E\ \mathit{alts}$) instead of ($\texttt{case}\ E\ \mathit{Alt}_1 \ldots \mathit{Alt}_N$). Sometimes we abbreviate the notation of `letrec`-expression ($\texttt{letrec}\ x_1 = E_1, \ldots x_n = E_n\ \texttt{in}\ E$), as ($\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ E$), where $\mathit{Env} \equiv \{x_1 = E_1, \ldots x_n = E_n\}$. This will also be used freely for parts of the bindings. An empty `letrec` is written as ($\texttt{letrec}\ \{\}\ \texttt{in}\ t$).
Contexts are defined as follows.

$$\begin{aligned} C ::= \ & [\cdot] \mid (C\ E) \mid (E\ C) \mid (\texttt{IO}\ C) \mid (c\ E \ldots E\ C\ E \ldots) \mid \lambda x.C \mid \\ & (\texttt{case}\ C\ \mathit{alts}) \mid (\texttt{case}\ E\ \mathit{Alt}_1, \ldots, (\mathit{Pat} \to C), \ldots, \mathit{Alt}_n) \mid \\ & (\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ C) \mid \\ & (\texttt{letrec}\ x_1 = E_1, \ldots, x_{i-1} = E_{i-1}, x_i = C, x_{i+1} = E_{i+1}, \ldots, x_n = E_n\ \texttt{in}\ E) \end{aligned}$$

where c is a constructor c_i for $i \in \{1, \ldots, N-1\}$.

Definition 4.1. *The following special context classes are defined:* Reduction contexts R*, and* weak reduction contexts R^-*, the latter has no* `letrec`*-expressions above the hole*

$$\begin{aligned} R^- &::= [\cdot] \mid (R^-\ E) \mid (\texttt{case}\ R^-\ \mathit{alts}) \mid (\texttt{IO}\ R^-) \\ R\ &::= R^- \mid (\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ R^-) \mid \\ &\qquad (\texttt{letrec}\ x_1 = R_1^-, \ldots, x_j = R_{j-1}^-[x_{j-1}], \ldots\ \texttt{in}\ R^-[x_j]) \\ &\qquad\quad \textit{where } R, R_j^- \textit{ are contexts of class } R, R^- \textit{, respectively} \end{aligned}$$

R is called a reduction context *and R^- is called a* weak reduction context*.*
For a term t with $t = R^-[t_0]$, we say R^- is maximal (for t)*, iff there is no larger weak reduction context with this property. For a term t with $t = C[t_0]$, we say C is a* maximal reduction context *iff C is either*

- *a maximal weak reduction context, or*
- *of the form* ($\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ R^-$) *where R^- is a maximal weak reduction context and $t_0 \neq x_j$ for all j, or*

- *of the form* ($\texttt{letrec}\ x_1 = R_1^-, x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], \ldots\ \texttt{in}\ R^-[x_j]$), *where* $t =$ ($\texttt{letrec}\ x_1 = t_1, \ldots\ \texttt{in}\ R^-[x_j]$), R_1^- *is a maximal weak reduction context for* t_1, *and the index* j *of involved bindings is maximal.*

Searching for a maximal reduction context can be seen as an algorithm walking over the term structure. In implementations of functional programming this is usually called "unwind".
For example the maximal reduction context of ($\texttt{letrec}\ x_2 = \lambda x.x, x_1 = x_2\ x_1\ \texttt{in}\ x_1$) is ($\texttt{letrec}\ x_2 = [\cdot], x_1 = x_2\ x_1\ \texttt{in}\ x_1$), in contrast to the non-maximal reduction context ($\texttt{letrec}\ x_2 = \lambda x.x, x_1 = x_2\ x_1\ \texttt{in}\ [\cdot]$).

Definition 4.2. *We define* surface contexts*, meaning that the hole is not in the body of an abstraction. Let* S *be the context class of* surface contexts *defined as follows:*

$$\begin{aligned} S ::= &\ [\cdot] \mid (S\ E) \mid (E\ S) \mid (c\ E \ldots E\ S\ E \ldots) \mid (\texttt{case}\ S\ alts) \\ &\mid (\texttt{case}\ E\ \ldots (p \to S) \ldots) \mid (\texttt{IO}\ S) \\ &\mid (\texttt{letrec}\ \ldots\ \texttt{in}\ S) \mid (\texttt{letrec}\ \ldots, x_i = S, \ldots\ \texttt{in}\ E) \end{aligned}$$

where c *may be* c_i *for* $i = 1, \ldots, N-1$.

Note that every reduction context is also a surface context.
Sometimes we will also use *multicontexts*, which are like contexts, but have several holes $\cdot_i$, and every hole occurs exactly once in the term. We write a multicontext as $C[\cdot_1, \ldots, \cdot_n]$, and if the terms s_i for $i = 1, \ldots, n$ are plugged into the holes $\cdot_i$, then we denote the resulting term as $C[s_1, \ldots, s_n]$.
The (call-by-need) reduction rules defined in Definition 4.3 follow the principle of minimizing copying at the cost of perhaps following several indirections. This holds for the copy rule (cp) as well as (case). The technical reason is that this principle assures well-behaved reduction diagrams.

Definition 4.3. *The reduction rules are defined in figures 1 and 2. The union of the rules (cp-in) and (cp-e) is called (cp), the union of (llet-in) and (llet-e) is called (llet), the union of (case-c), (case-lam), (case-in), (case-e) is called (case), and the union of (IOr-c), (IOr-in), (IOr-e) is called (IOr).*
Note that the `case`*-reduction for a constant produces a* `letrec` *without variable bindings.*

If the context is important, then we denote it as a label of the reduction.
It will turn out in later sections that, viewed as correct program transformations, the reductions with the exception of (IOr) may also be used in an arbitrary context.
Reductions are denoted using an arrow with super and/or subscripts: e.g. $\xrightarrow{llet}$.
Transitive closure is denoted by a $+$, reflexive transitive closure by a $*$. E.g. $\xrightarrow{*}$ is the reflexive, transitive closure of $\to$. If necessary, we attach more information to the arrow.

(lbeta)	$((\lambda x.s)\ t) \to (\texttt{letrec}\ x = t\ \texttt{in}\ s)$
(cp-in)	$(\texttt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_j = x_{j-1}, \text{Env}\ \ \texttt{in}\ C[x_j])$
	$\to (\texttt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_j = x_{j-1}, \text{Env}\ \ \texttt{in}\ C[s_1])$
	where s_1 is an abstraction
(cp-e)	$(\texttt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_j = x_{j-1}, x_{j+1} = C[x_j], \text{Env}\ \ \texttt{in}\ s)$
	$\to (\texttt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_j = x_{j-1}, x_{j+1} = C[s_1], \text{Env}\ \ \texttt{in}\ s)$
	where s_1 is an abstraction
(llet-in)	$(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ (\texttt{letrec}\ y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ r))$
	$\to (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ r)$
(llet-e)	$(\texttt{letrec}\ x_1 = s_1, \ldots, x_i =$
	$(\texttt{letrec}\ y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ s_i), \ldots, x_n = s_n\ \texttt{in}\ r)$
	$\to (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ r)$
(lapp)	$((\texttt{letrec}\ x_i = t_i\ \texttt{in}\ t)\ s) \to (\texttt{letrec}\ x_i = t_i\ \texttt{in}\ (t\ s))$
(lcase)	$(\texttt{case}\ (\texttt{letrec}\ E\ \texttt{in}\ t)\ \mathit{alts}) \to (\texttt{letrec}\ E\ \texttt{in}\ (\texttt{case}\ t\ \mathit{alts}))$
(case-c)	$(\texttt{case}\ (c_i\ t_1 \ldots t_n)\ \ \ldots ((c_i\ y_1 \ldots y_n) \to t) \ldots)$
	$\to \quad (\texttt{letrec}\ y_1 = t_1 \ldots y_n = t_n\ \texttt{in}\ t)$
(case-lam)	$(\texttt{case}\ \lambda x.s\ \ \ldots (\texttt{lambda} \to t) \ldots) \to (\texttt{letrec}\ \{\}\ \texttt{in}\ t)$
(case-in)	$\texttt{letrec}\ x_1 = (c_i\ t_1 \ldots t_n),$ where $n = \text{ar}(c_i)$
	$\qquad x_2 = x_1, \ldots x_m = x_{m-1}, \ldots$
	$\texttt{in} \qquad C[\texttt{case}\ x_m\ \ \ldots ((c_i\ z_1 \ldots z_n) \to t)]$
	$\longrightarrow$
	$\texttt{letrec}\ x_1 = (c_i\ y_1 \ldots y_n), y_1 = t_1, \ldots y_n = t_n,$
	$\qquad x_2 = x_1, \ldots x_m = x_{m-1}, \ldots$
	$\texttt{in} \qquad C[(\texttt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \texttt{in}\ t)]$
(case-e)	$\texttt{letrec}\ x_1 = (c_i\ t_1 \ldots t_n),$ where $n = \text{ar}(c_i)$
	$\qquad x_2 = x_1, \ldots x_m = x_{m-1}, \ldots$
	$\qquad u = C[\texttt{case}\ x_m\ \ \ldots ((c_i\ z_1 \ldots z_n) \to r_1)]$
	$\texttt{in} \qquad r_2$
	$\longrightarrow$
	$\texttt{letrec}\ x_1 = (c_i\ y_1 \ldots y_n), y_1 = t_1, \ldots y_n = t_n,$
	$\qquad x_2 = x_1, \ldots x_m = x_{m-1}, \ldots$
	$\qquad \ldots$
	$\qquad u = C[(\texttt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \texttt{in}\ r_1)]$
	$\texttt{in} \qquad r_2$
	where y_i are fresh variables

Fig. 1. Reduction rules of FUNDIO

5 Normal Order Reduction

Definition 5.1. *Let t be a (closed) expression. Let R be the maximal reduction context such that $t \equiv R[t']$ for some t'. The normal order reduction $\xrightarrow{n}$ is defined by one of the following cases:*

1. *t' is a* `letrec`*-expression* (`letrec` Env_1 `in` t'')*, and R is not trivial. Then there are 5 cases, where R_0 is a reduction context:*
   - *(a)* $R = R_0[(\texttt{IO}\ [\cdot])]$*. Reduce* $(\texttt{IO}\ t')$ *using (IOlet).*
   - *(b)* $R = R_0[([\cdot]\ s)]$*. Reduce* $(t'\ s)$ *using (lapp).*
   - *(c)* $R = R_0[(\texttt{case}\ [\cdot]\ alts)]$*. Reduce* $(\texttt{case}\ t'\ alts)$ *using (lcase).*
   - *(d)* $R = \texttt{letrec}\ Env_2\ \texttt{in}\ [\cdot]$*. Reduce t using (llet-in) by flattening t' resulting in* $(\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ t'')$*.*
   - *(e)* $R = \texttt{letrec}\ x = [\cdot], Env_2\ \texttt{in}\ t'''$*. Reduce t using (llet-e) by flattening t' resulting in* $(\texttt{letrec}\ x = t'', Env_1, Env_2\ \texttt{in}\ t''')$*.*
2. *t' is a constructor application. There are the following cases:*
   - *(a)* $R = R_0[\texttt{case}[\cdot]\ \ldots]$*. Then apply (case-c) to* $(\texttt{case}\ t'\ \ldots)$*.*
   - *(b)* $R = \texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ R_0^-[\texttt{case}\ x_m\ alts]$ *where R_0^- is a weak reduction context. Then apply (case-in) to the indicated* `case`*-expression.*
   - *(c)* $R = \texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, y = R_0^-[\texttt{case}\ x_m\ alts], Env\ \texttt{in}\ t''$ *where R_0^- is a weak reduction context, and y is in a reduction context. Then apply (case-e) to the indicated* `case`*-expression.*
3. *t' is a constant, and case 2 does not apply. There are the following cases:*
   - *(a)* $R = R_0[(\texttt{IO}\ [\cdot])]$*. Then apply (IOr-c) to* $(\texttt{IO}\ t')$*.*
   - *(b)* $R = \texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ R_0^-[(\texttt{IO}\ x_m)]$ *where R_0^- is a weak reduction context. Then apply (IOr-in) to the indicated* `IO`*-expression.*
   - *(c)* $R = \texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, y = R_0^-[(\texttt{IO}\ x_m)], Env\ \texttt{in}\ t''$ *where R_0^- is a weak reduction context, and y is in a reduction context. Then apply (IOre) to the indicated* `IO`*-expression.*
4. *t' is an abstraction. There are the following cases:*
   - *(a)* $R = R_0[\texttt{case}[\cdot]\ \ldots]$*. Then apply (case-lambda) to* $(\texttt{case}\ t'\ \ldots)$*.*
   - *(b)* $R = R_0[([\cdot]\ t'')]$ *where R_0 is a reduction context. Then apply (lbeta) to* $(t'\ t'')$*.*
   - *(c)* $R = \texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ R_0^-[x_m]$ *where R_0^- is a weak reduction context. Then apply (cp-in) such that $R_0^-[x_m]$ is changed into $R_0^-[t']$*
   - *(d)* $R = \texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, y = R_0^-[x_m\ s], Env\ \texttt{in}\ t''$ *where R_0^- is a weak reduction context, and y is in a reduction context. Then apply (cp-e) such that $R_0^-[x_m\ s]$ is reduced to $R_0^-[t'\ s]$.*
   - *(e)* $R = \texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, y = R_0^-[\texttt{case}\ x_m\ alts], Env\ \texttt{in}\ t''$ *where R_0^- is a weak reduction context, and y is in a reduction context. Then apply (cp-e) such that* $R_0^-[\texttt{case}\ x_m\ alts]$ *is reduced to* $R_0^-[\texttt{case}\ t'\ alts]$*.*

(IOlet)	$(\texttt{IO}\ (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s)) \to (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{IO}\ s))$
(IOr-c)	$(\texttt{IO}\ c) \to d$
(IOr-in)	$\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots x_m = x_{m-1}, \ldots, \texttt{in}\ C[(\texttt{IO}\ x_1)]$
	$\to \texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots x_m = x_{m-1}, \ldots \texttt{in}\ C[d]$
(IOr-e)	$\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots x_m = x_{m-1}, u = C[(\texttt{IO}\ x_1)], \mathit{Env}\ \texttt{in}\ r$
	$\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots x_m = x_{m-1}, u = C[d], \mathit{Env}\ \texttt{in}\ r$
	where c, d are constants
	The IO-pair of the IOr-reductions is(c, d)

Fig. 2. IO-Reduction rules of FUNDIO

The normal order redex *is defined as the subexpression to which the reduction rule is applied. This includes the* `letrec` *-expression that is mentioned in the reduction rules, for example in (case-e).*

The normal order reduction implies that the IO-function `IO` behaves as a strict function, and that the case-construct is strict in its first argument. I.e., these rules can only be applied if the corresponding argument is a constant or a constructor applicatio, respectively.

Definition 5.2. *A* value *is a constructor application or an abstraction.*

The notion of weak head normal form will be required.

Definition 5.3. *A* weak head normal form (WHNF) *is one of the cases:*

- *A value*
- *A term of the form* $(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t)$*, where* t *is a value*
- *A term of the form* $\texttt{letrec}\ x_1 = (c\ t_1 \ldots t_{\mathrm{ar}(c)}), x_2 = x_1, \ldots, x_m = x_{m-1}, \mathit{Env}\ \texttt{in}\ x_m$*.*

Lemma 5.4. *For every term* t*:*
if t *has a normal-order redex, then this redex is unique.*
If the normal-order reduction is not an (IOr), then the normal order reduction is also unique.
If the normal-order reduction is an (IOr), and the IO-pair is given, then the normal-order reduction is unique.

Definition 5.5. *An* error-term t *is a term that*

1. *is not a WHNF, and*
2. *has no normal order redex, and*
3. *is not of the form* $R[x]$*, where* R *is a reduction context, and* x *a free variable in* t*.*

An error-term t satisfies one of the following conditions:

- The search for a normal order redex using an unwind-like algorithm does not terminate. In general, this is the case if $t = R_1[x] = R_2[x]$ where x is a bound variable and R_1, R_2 are two different reduction contexts.
- A term of the form $R[\texttt{IO}\ t']$, where t' is a non-constant constructor application or an abstraction; or it is bound to a non-constant constructor application or to an abstraction, and where R is a reduction context
- A term of the form $R[t'\ t'']$, where t' is a constructor application or is bound to a constructor application

Definition 5.6. *Let a bot-term be defined as a closed expression that has no normal order reduction that ends in a WHNF.*

Later we will show that all bot-terms are equivalent and that their equivalence class is the least element in the contextual preorder. We will use bot as a representative of a bot-term, if the exact sytactic form is not important.
Note that there are terms t that are neither WHNFs nor have a normal order redex. For example $(\texttt{IO}(\lambda x.x))$ or `((cons 1 2) 3)`, where `cons` is a constructor of arity 2. These terms are error-terms and could be considered as violating type conditions. In FUNDIO these terms will be shown to be equivalent to non-terminating ones.
Consider the "cyclic term" $(\texttt{letrec}\ x = x\ \texttt{in}\ x)$. The maximal reduction context for this term is $(\texttt{letrec}\ x = [\cdot]\ \texttt{in}\ x)$. It is easily seen that there is no normal order reduction defined for this term.

6 Contextual Equivalence

We define contextual equivalence w.r.t. the IO-behavior of terminating normal order sequences. The intuition is that terms are equivalent if the IO-actions of terminating normal order reduction sequences are equal as multiset, and if either both have an error or both are error-free. I.e. have a normal order reduction to a bot-term, or both don't permit such a reduction. Using multisets enables the commutation of IO-actions.

Definition 6.1. *An* IO-pair *is a pair* (a, b) *of constants. An* IO-sequence *is a finite sequence of IO-pairs. An* IO-multiset *is a finite multiset of IO-pairs.*
The IO-sequence $IOS(s_1 \to s_2 \to \ldots \to s_n)$*, and the IO-multiset* $IOM(s_1 \to s_2 \to \ldots \to s_n)$ *of a reduction sequence* $s_1 \to s_2 \to \ldots \to s_n$ *are defined as follows:*

- *The IO-pair of a single (IOr)-reduction* $C[\texttt{IO}\ c] \to C[d]$ *(analogously for the other IOr-reductions) is the pair of output-input-values:* (c, d)*. Non-(IOr)-reductions have no IO-pair.*

- *Let $s_1 \to s_2 \to \ldots \to s_n$ be a reduction sequence.*
  If $s_1 \to s_2$ is not an (IOr)-reduction, then $IOS(s_1 \to s_2 \to \ldots \to s_n) := IOS(s_2 \to \ldots \to s_n)$.
  If (a, b) is the IO-pair for the (IOr)-reduction $s_1 \to s_2$, then $IOS(s_1 \to s_2 \to \ldots \to s_n) \;:=\; (a, b), IOS(s_2 \to \ldots \to s_n)$.
- *Let $s_1 \to s_2 \to \ldots \to s_n$ be a reduction sequence.*
  Then $IOM(s_1 \to s_2 \to \ldots \to s_n)$ is defined as the multiset of the elements of $IOS(s_1 \to s_2 \to \ldots \to s_n)$.

For a term t and a finite IO-multiset P, we write $t{\Downarrow}(P)$ iff there is a normal order reduction sequence to WHNF starting from t with IO-multiset P. Otherwise, we write $t{\Uparrow}(P)$. If $t{\Downarrow}(P)$, we say that t is terminating for IO-multiset *P.*

We say a closed term t has a bot-reduction *(notation: $t{\Uparrow}$), iff there is a normal order reduction $t \xrightarrow{n,*} t'$, and t' is a bot-term.*

Definition 6.2. *(contextual preorder and equivalence) Let s, t be terms. Then:*

$$s \le_c t \text{ iff } \forall C[\cdot] \; \Big(\forall P : C[s]{\Downarrow}(P) \Rightarrow C[t]{\Downarrow}(P)\Big) \;\; \wedge (C[t]{\Uparrow} \Rightarrow C[s]{\Uparrow})$$
$$s \sim_c t \text{ iff } \; s \le_c t \wedge t \le_c s$$

Note that we permit contexts such that $C[s]$ may be an open term.
We define some tools and notions to deal with bot-reductions.

Definition 6.3. *Let $\overrightarrow{P}$ be a finite IO-sequence and let t be a term. We say that $\overrightarrow{P}$ is* valid *for t, iff there is a normal order reduction $t \xrightarrow{n,\overrightarrow{P}} t'$ for some t'. This reduction is called the normal order reduction of t* along *$\overrightarrow{P}$.*

For a term t and an IO-sequence $\overrightarrow{P}$, we write $t{\downarrow}(\overrightarrow{P})$ iff $t \xrightarrow{n,\overrightarrow{P}} t'$ and t' is a WHNF.
For a closed term t and a finite IO-sequence $\overrightarrow{P}$, we write $t{\Downarrow}(\overrightarrow{P})$ iff the following holds: if $\overrightarrow{P}$ is valid for t then $t \xrightarrow{n,\overrightarrow{P}} t'$ where t' has a normal-order reduction to a WHNF.

Lemma 6.4. *$t{\Uparrow}$ holds iff there is an IO-sequence $\overrightarrow{P}$, such that $t \xrightarrow{n,\overrightarrow{P}} t'$, and t' is a bot-term.*

Note that given a term t, and a valid IO-sequence $\overrightarrow{P}$, the normal-order reduction along $\overrightarrow{P}$ is unique. Note also that an open term t may not have any normal order reduction to a WHNF, but we do not consider this as equivalent to bot, e.g. x is such a term.
Given an IO-multiset P, and an expression t, there may be different normal order reduction sequences starting with t with IO-multiset P, such that the normal order reductions are not prefixes of each other. The divergence of a term can also

be formulated as follows: it is possible to reduce the program such that there is no escape: there are no more terminating normal-order reductions. If a term has an infinite normal-order reduction with an infinite IO-multiset, this is not treated as an error as long as there are remaining possibilities to reach a WHNF by normal order reduction.

The intuition behind the contextual preorder is that not only the termination of the program is used as observation, but also the question-answer pairs of an evaluation and the error-behavior (or the reductions to bot-terms). Since we use multisets, this concept will permit interchanging question-answer pairs. It has similarities to a trace semantics, though we allow commutativity.

Lemma 6.5. *The second condition for contextual preorder $\forall C[\cdot] : C[t]\Uparrow \Rightarrow C[s]\Uparrow$ can be reformulated as*

$$\forall C[\cdot] : (\forall \overrightarrow{P} : C[s]\Downarrow\overrightarrow{P}) \Rightarrow (\forall \overrightarrow{P} : C[t]\Downarrow\overrightarrow{P})$$

Lemma 6.6. *$s \leq_c t$ is equivalent to:*

$$\forall C[\cdot]\ C[s], C[t] \text{ are closed} \Rightarrow \Big(\forall P : C[s]\Downarrow(P) \Rightarrow C[t]\Downarrow(P)\Big) \wedge (C[t]\Uparrow \Rightarrow C[s]\Uparrow)$$

Proof. One direction is trivial. The proof of the other direction will appear in section 23. □

A *precongruence* $\leq_c$ is a partial order on expressions, such that $s \leq_c t \Rightarrow C[s] \leq_c C[t]$ for all contexts C. A *congruence* is a precongruence that is also an equivalence relation.

Proposition 6.7. *$\leq_c$ is a precongruence, and $\sim_c$ is a congruence.*

Proof. Let $s \leq_c t, t \leq_c r$, let C be a context, and P be an IO-multiset such that $C[s]\Downarrow(P)$. Then $C[t]\Downarrow(P)$. Since $t \leq_c r$, we have also $C[r]\Downarrow(P)$.

For the other part let $C[r]\Uparrow$. Then $C[t]\Uparrow$ holds, and from $s \leq_c t$, we also have $C[s]\Uparrow$. Hence $s \leq_c r$.

To show the congruence property, let $s \leq_c t$ and let C be a context. To show $C[s] \leq_c C[t]$, let D be a further context. If $D[C[s]]\Downarrow(P)$ for an IO-multiset P, we can use the context DC for $s \leq_c t$, and see that $D[C[t]]\Downarrow(P)$.

To show that $D[C[t]]\Uparrow \Rightarrow D[C[s]]\Uparrow$ for all contexts D, use the context DC and the assumption $s \leq_c t$.

Concluding, this shows $C[s] \leq_c C[t]$. □

The following lemma shows that for terms s, t with $s \sim_c t$, the first condition in the definition implies that either both terms have an infinite normal order reduction with infinitely many IOs, or both terms don't have such a reduction.

Lemma 6.8. *Let t be a term such that for every $n > 0$ there is a P with $|P| \geq n$ and $t\Downarrow(P)$. Then t has a non-terminating n-reduction with infinitely many IOs.*

Proof. It is no restriction to assume that t has a normal order (IOr)-redex by perhaps reducing until such a term is reached. Consider the tree of all reductions, structured according to the IO-sequences, where every node is a term with an normal order redex that is an IO-expression, and the connection from a node to its sons corresponds to different answers. The leaves may be $\bot$, if the n-reduction is non-terminating or results in an error-term without IO-reduction, or it is a term in WHNF.
Then this tree has finite branching, and by the assumption, there is no bound on the depth of the tree, hence by Königs Lemma there is an infinite branch, hence an infinite reduction. □

6.1 Examples and Discussion

In this subsection we discuss the usefulness, appropriateness and consequences of the definition of contextual equivalence in Definition 6.2 by giving examples and counter-examples.

Programs with commuted IO-actions may be contextually equal:

Example 6.9. Let P_1 be a program that asks for your name and then for your date of birth, and let P_2 be the program that asks first for your date of birth and then for your name. Both programs return a pair of the values. We simply assume that strings are constants.
$P_1 :=$

```
letrec b =  IO "date-of-birth?"
       n =  IO "Your-name?"
   in case b
          (c1 -> case n (c1 -> (b,n) ... ))
          (c2 -> case n (c1 -> (b,n) ... ))
          ...
```

$P_2 :=$

```
letrec b = IO "date-of-birth?"
       n = IO "Your-name?"
   in case n
          (c1 -> case b (c1 -> (b,n) ... ))
          (c2 -> case b (c1 -> (b,n) ...))
```

Then P_1 and P_2 have the same behavior in the empty context, since the IO-multisets $\{(\texttt{"date-of-birth?"}, \text{"01011990"}), (\texttt{"Your-name?"}, \text{"Jim"})\}$ can be used for the normal-order reductions of P_1 as well as P_2. We are sure that these terms are contextually equal, the methods for proving this have to be developed.

Example 6.10. Consider two programs computing the `and, or`, respectively of two boolean inputs.
We assume there are boolean constants `True`, `False`, and that the definitions of `and` and `or` are:

$$\begin{array}{ll} \texttt{and} & := \lambda x, y.\texttt{case}\ x\ (\texttt{True} \to y)\ (\texttt{False} \to \texttt{False})\ldots \\ \texttt{or} & := \lambda x, y.\texttt{case}\ x\ (\texttt{True} \to \texttt{True})\ (\texttt{False} \to y)\ \ldots \end{array}$$

$$\begin{array}{ll} \texttt{and-prog} & := \texttt{and (IO True) (IO True)} \\ \texttt{or-prog} & := \texttt{or}\ \ \texttt{(IO True) (IO True)} \end{array}$$

The two programs `and-prog` and `or-prog` are contextually different in FUNDIO: The IO-multiset

$$M := \{(\texttt{True}, \texttt{True}), (\texttt{True}, \texttt{False}))\}$$

together with the context

$$C := (\texttt{case}\ [\cdot]\ (\texttt{False} \to \texttt{False})\ (\texttt{True} \to \mathit{bot})\ \ldots)$$

distinguishes the two programs. Here we let *bot* be a non-terminating expression. We have $C[\texttt{and-Prog}]\Downarrow M$, since the reduction that uses the multiset M results in `False`, and in this case the program terminates. But $C[\texttt{or-Prog}]\Uparrow(M)$.

This example is a hint that FUNDIO with the defined contextual equivalence is better suited for functional IO-programs, since it distinguishes more intuitively different programs with I/O than the non-deterministic approach in [Kut00]. In this approach, the expression `IO True` has to be encoded as a nondeterministic choice: `(choice True False)`. The contextual equivalence in [Kut00] cannot distinguish the two programs, since it is based on the sets of possible outcomes, which is $\{\texttt{True}, \texttt{False}\}$ in both cases.
The calculus in [MSC99] is also not able to distinguish the `and` from the `or`-program.

Example 6.11. This example shows that it makes sense to require that the normal order reduction terminates with a WHNF in the definition of contextual equivalence.
Suppose we have an appropriate encoding of non-negative integers and $*$. Consider an expression $s * t$, where $*$ is infix product, and s, t may contain IO-calls in their normal order evaluation. Since product is strict in both arguments, it should not matter whether s or t is evaluated first. However, if s, t are both non-terminating, making different IOs, the reduction "first s then t" and "first t then s" were distinguishable under a definition of contextual equivalence that takes infinite reductions into account. Our definition 6.2 prevents this by comparing the IO-multisets only for terminating normal order reductions. The IO-multisets of the two different reductions of $s * t$ are in this case the union of the two IO-multisets for s and t, and thus equal.

The contextual equivalence as defined above is able also to compare potentially non-terminating programs w.r.t. their IO-behavior. It also shows that it is crucial for comparing programs that there are enough possibilities to regularly exit the program:

Example 6.12. As an example assume there is a program P_1 that computes the decimal representation of π, and outputs the digits one by one, in the right

sequence, and asking after every printed digit “n: more? Y/N”. If Y is the answer, then the next digit and question is printed, otherwise the program reduces to a WHNF.
Let P_2 be a program that also outputs the digits of π, also asking : “more? Y/N” after every digit, however, the sequence of digits is different, say $2, 1, 4, 3, 6, 5, 8, 7, \ldots$.
The definition of contextual equivalence shows that these two programs are different, since P_2 has no normal order reduction with IO-multiset $\{(\text{“3 : } \mathit{more?Y/N}\text{”}, N)\}$.
This means that a program that interchanges the output of the digits of π is different from P_1. The reason is that P_1 is able to interrupt after every printed digit.

Example 6.13. The so-called η-rule is not a correct program transformation in FUNDIO: `case True (True ->True)(False->True) ...` and `case (\x-> True x) (True->True)(False->True) ... (lambda->bot)` are contextually different, since the first has a normal order reduction to a WHNF, while the second doesn’t.

Definition 6.14. *Define the expression* `choice` *as follows:*

```
choice := \x,y . if (IO 0) then x else y
```

where the if-expression is a `case` *of boolean values.*

This expression behaves like a non-deterministic selection between two expressions, since the answer may be `True` or `False`. Concerning equality of programs, it is different than in other approaches, since the number of choices is counted, even whether the right or left expression was selected is significant. Thus the examples below should not only rely on counting the choices to permit a fair comparison with program-equivalence in other approaches.
The following examples justify the clause on divergence in the definition of contextual equivalence.

Example 6.15. This example shows that FUNDIO should also look for non-terminating reductions in the definition of contextual equality of expressions.
The programs

```
s1 = (choice (choice 1 bot) (choice 2 3))
s2 = (choice (choice 1 2) (choice 2 3))
```

are not distinguishable using a contextual equivalence that is based only on convergence, since e.g. in the empty context, the expressions have the same results.
However, they are clearly $\not\sim_c$, since `s1`$\Uparrow$, but not `s2`$\Uparrow$.

Example 6.16. This example shows that the contextual equivalence should not use the criterion that equal expressions should have the same non-termination behavior for all IO-multisets. Let $*$ be multiplication, strict in both arguments and appropriately encoded in FUNDIO.

```
s  := choice 1 1
t  := (choice (choice 2 bot) (choice 2 2))
p1 := s*t
p2 := t*s
```

The program p2 has a loop after the sequence of answers True,False, whereas the program p1 has a loop after the answer-sequences True,True,False and False,True,False.
The definition of $\sim_c$ is such that this is no problem. We are sure that p1 $\sim_c$ p2 can be proved.

Example 6.17. In non-deterministic programs an often referred-to example is the comparison of two programs P1, P2, the first evaluates to a constant a, the second is a non-deterministic choice between a and $\perp$.

```
P1  := a
P2  := choice a bot
```

In FUNDIO, these two programs are clearly different: In the empty context, P1 has a terminating normal-order reduction with empty IO-multiset, whereas P2 has no terminating normal-order reduction with empty IO-multiset.
Making the example fair, i.e. independent of counting choices, we modify it to

```
P1'  := choice a a
P2'  := choice a bot
```

This doesn't make the two terms equal modulo $\sim_c$, since P2$'\Uparrow$, but not P1$'\Uparrow$.

Example 6.18. Simon Peyton Jones in [Jon01] discusses the comparison of two programs loop and loopX, the first is nonterminating, the second prints x forever.
Programs with this behavior are all equivalent to $\perp$ in FUNDIO, since there is no escape from both programs. Adding a legal escape in the print program, e.g. between printing looking for an interrupt-bit and then going to a stop, giving the program loopx' would make the two programs different, since now there is a legal possibility to stop, and hence to reach a WHNF.

Example 6.19. The same arguments hold for an operating system. FUNDIO only requires that the operating system always has a legal possibility to stop. However, if the operating system is in a loop with IO, and there is no other escape than switching the power off, then this is treated as $\perp$.

Example 6.20. Consider the programs

```
p1 := choice (choice p1 p1) (choice 1 1)
p2 := choice (choice p2 p2) (choice 1 bot)
```

These programs are equal w.r.t. comparing them only for convergence on all multisets, and also both have a non-terminating reduction. However, we have p1$\not\Uparrow$, but p2$\Uparrow$, thus they are not equivalent in FUNDIO.

Example 6.21. This example shows that the let-over-lambda rule is not correct in FUNDIO. Consider the programs

```
p1 := let z = (let x =  choice 1 2 in \y.x) in (z 0) + (z 0)
p2 := let z = (\y.choice 1 2) in (z 0) + (z 0)
```

These programs are not contextually equivalent, since `p1` has only the possibilities $\{2, 4\}$, whereas `p2` has the possibilities $\{2, 3, 4\}$.

7 Context Lemma

The so-called Context Lemma restricts the criterion for contextual equivalence to reduction contexts. These are also infinitely many contexts, but it is of great value in proving the conservation of contextual equivalence by certain reductions, since there is no need to introduce parallel reductions as a generalization of Barendregt's 1-reduction.
We split the proof of the context lemma into two lemmas.

Lemma 7.1. *Let s, t be terms. If for all reduction contexts R and all IO-multisets P $(R[s]{\Downarrow}(P) \Rightarrow R[t]{\Downarrow}(P))$, then*
$\forall C, \forall P : (C[s]{\Downarrow}(P) \Rightarrow C[t]{\Downarrow}(P))$.

Proof. In this proof we will use multicontexts, which are generalizations of contexts having several holes, and every hole is mentioned in the argument list of the multicontext.
We prove the more general claim:

> For $i = 1, \dots, n$, let s_i, t_i be expressions. Let the following hold:
> $\forall i : \forall$ reduction contexts $R :\ \forall P$: $(R[s_i]{\Downarrow}(P) \Rightarrow R[t_i]{\Downarrow}(P))$.
> Then $\forall C, \forall P : C[s_1, \dots, s_n]{\Downarrow}(P) \Rightarrow C[t_1, \dots, t_n]{\Downarrow}(P)$.

Assume the claim is false. Then there is a counterexample: I.e., there is a multicontext C, an IO-multiset P, a number $n \geq 1$ and terms s_i, t_i for $i = 1, \dots, n$, such that $\forall i : \forall$ reduction contexts $R :\ \forall P'$: $(R[s_i]{\Downarrow}(P') \Rightarrow R[t_i]{\Downarrow}(P'))$, and $C[s_1, \dots, s_n]{\Downarrow}(P)$, but $C[t_1, \dots, t_n]{\Uparrow}(P)$.
We select the counterexample minimal w.r.t. the following lexicographic ordering:

1. the number of normal order reduction steps of a terminating reduction of $C[s_1, \dots, s_n]$ which has P as corresponding IO-multiset.
2. the number of holes of C.

The distinction is whether or not some hole of $C[\cdot_1, \dots, \cdot_n]$ is in a reduction context. The definition of reduction contexts and some reasoning shows that the unwind applied to $C[\cdot_1, \dots, \cdot_n]$ either hits some hole, or doesn't hit a hole, and moreover, this does not change if the holes are filled.

If one hole of $C[\cdot_1, \dots, \cdot_n]$ is in a reduction context, then we assume wlog that it is the first one.

Then $C[\cdot, t_2 \ldots, t_n]$ is a reduction context. Let $C' := C[s_1, \cdot_2, \ldots, \cdot_n]$. Since $C'[s_2, \ldots, s_n] \equiv C[s_1, \ldots, s_n]$, they have the same normal-order reduction for the IO-multiset P. Since the number of holes is smaller, we obtain $C'[t_2, \ldots, t_n]\Downarrow(P)$, which means $C[s_1, t_2, \ldots, t_n]\Downarrow(P)$. Since $C[\cdot, t_2, \ldots, t_n]$ is a reduction context, the preconditions of the lemma applied to s_1, t_1 imply $C[t_1, t_2, \ldots, t_n]\Downarrow(P)$, a contradiction.

If no hole of $C[\cdot_1, \ldots, \cdot_n]$ is in a reduction context, then $C[s_1, \ldots, s_n]$ as well as $C[t_1, \ldots, t_n]$ can be reduced using the same normal order reduction at the same position. To verify this, we have to check that for a normal order redex, the parts that are modified are also in a reduction context.

- in a (cp) normal order reduction, every superterm of the to-variable position is in a reduction context.
- For normal order reductions (llet), (lapp), (lcase), (IOlet), the inner `letrec` is in a reduction context.
- The constructor application in a (case) is in a reduction context.
- the constant in the (IO)-reduction is in a reduction context.

The following may happen to the terms s_i, t_i in the holes:

- If the hole is in the wrong alternative of a (case)-expression that is reduced, then the hole is eliminated after reduction.
- If the hole is not in the wrong alternative of a (case)-reduction, and if the reduction is not a (cp), then the terms s_i, t_i in the holes are unchanged and also not copied, but perhaps at a different position in the result.
- if the reduction is a (cp), and the hole is not in the copied expression, then again the terms s_i, t_i in the holes are unchanged and also not copied.
- if the reduction is a (cp), and the hole is within the copied expression, then the terms s_i, t_i in the holes may be duplicated giving s'_i, t'_i. Since the reduction is a normal order reduction, and since we have assumed the “distinct bound variable convention”, the renaming concerns the free variables in s_i, t_i which are bound in C. For a fixed i, we can use the same renaming ρ_i for the bound variables in s_i and t_i, so we have $\rho_i(s_i) = s'_i, \rho_i(t_i) = t'_i$. This means that the assumption holds also for the new pair of terms:

$$\forall i : \forall \text{ reduction contexts } R : \ \forall P : (R[s'_i]\Downarrow(P) \Rightarrow R[t'_i]\Downarrow(P)).$$

If the normal order reduction is non-(IOr), then we can use induction on the number of $\xrightarrow{n}$-reductions.
If the normal order reduction is an (IOr)-reduction for the pair (c, d), then the IO-multisets have to be checked:
A new smaller example is constructed with an IO-multiset $P' := P \setminus \{(c, d)\}$. The reduction can be written as $C[s_1, \ldots, s_n] \xrightarrow{n,(c,d)} C'[s_1, \ldots, s_n]$ and $C[t_1, \ldots, t_n] \xrightarrow{n,(c,d)} C'[t_1, \ldots, t_n]$ with $C'[s_1, \ldots, s_n]\Downarrow(P')$. Since the number of normal order reductions is strictly smaller, we have also $C'[t_1, \ldots, t_n]\Downarrow(P')$. But

then we have $C[t_1, \ldots, t_n]{\Downarrow}(P)$, which contradicts the assumption that this is a counterexample.

Now we look at the base case. If C has no holes, then a counterexample is impossible.

If the number of normal order reduction steps is 0, then $C[s_1, \ldots, s_n]$ is already a WHNF. Since we can assume that no hole is in a reduction context, the context itself is a WHNF, and thus this holds for $C[s_1, \ldots, s_n]$ as well as $C[t_1, \ldots, t_n]$, which is impossible.

Concluding, we have proved that there is no counterexample to the general claim, hence the lemma holds, since it is a specialization. □

Lemma 7.2. *Let s, t be terms. If for all reduction contexts R:* $(\forall \overrightarrow{P} : R[s]{\Downarrow}(\overrightarrow{P})) \Rightarrow (\forall \overrightarrow{P} : R[t]{\Downarrow}(\overrightarrow{P}))$*, then* $\forall C : (\forall \overrightarrow{P} : C[s]{\Downarrow}(\overrightarrow{P})) \Rightarrow (\forall \overrightarrow{P} : C[t]{\Downarrow}(\overrightarrow{P}))$.

Proof. The proof can be generated from the proof of Lemma 7.1 with some modifications.

We prove the more general claim:

> For $i = 1, \ldots, n$, let s_i, t_i be expressions. Let the following hold:
> If $\forall i : \forall$ reduction contexts $R : \quad (\forall \overrightarrow{P} : R[s_i]{\Downarrow}(\overrightarrow{P})) \Rightarrow (\forall \overrightarrow{P} : R[t_i]{\Downarrow}(\overrightarrow{P}))$,
> then $\forall C : (\forall \overrightarrow{P} : C[s_1, \ldots, s_n]{\Downarrow}(\overrightarrow{P})) \Rightarrow (\forall \overrightarrow{P} : C[t_1, \ldots, t_n]{\Downarrow}(\overrightarrow{P}))$.

Assume that the claim is false. Then there is a counterexample: I.e., there is a multicontext C, a number $n \geq 1$ and terms s_i, t_i for $i = 1, \ldots, n$, such that $\forall \overrightarrow{Q} : C[s_1, \ldots, s_n]{\Downarrow}(\overrightarrow{Q}))$, but for some $\overrightarrow{P}$: $\neg(C[t_1, \ldots, t_n]{\Downarrow}(\overrightarrow{P}))$.

This means that $\overrightarrow{P}$ is valid for $C[t_1, \ldots, t_n]$, and that $C[t_1, \ldots, t_n] \xrightarrow{n,\overrightarrow{P}} \mathit{bot}$ where bot is a bot-term.

We select the counterexample minimal w.r.t. the lexicographic ordering with the following two components:

1. If $\overrightarrow{P}$ is valid for $C[s_1, \ldots, s_n]$, then the minimal number of normal order reduction steps of $C[s_1, \ldots, s_n]$ along $\overrightarrow{P}$ and then to a WHNF.
   If $\overrightarrow{P}$ is not valid for $C[s_1, \ldots, s_n]$, then the maximal number of normal order reduction steps of $C[s_1, \ldots, s_n]$ along $\overrightarrow{P'}$ where $\overrightarrow{P'}$ is a prefix of $\overrightarrow{P}$. I.e. until there is an (invalid (IOr)-redex.
2. the number of holes of C.

If a hole of $C[\cdot_1, \ldots, \cdot_n]$ is in a reduction context, then we assume wlog that it is the first one.

Then $C[\cdot, t_2 \ldots, t_n]$ is a reduction context. Let $C' := C[s_1, \cdot_2, \ldots, \cdot_n]$. Note that $C'[s_2, \ldots, s_n] \equiv C[s_1, \ldots, s_n]$, and that the first component of the induction measure is the same. Since the number of holes of C' is smaller, we obtain $\forall \overrightarrow{Q} : C'[t_2, \ldots, t_n]{\Downarrow}(\overrightarrow{Q})$, which means $\forall \overrightarrow{Q} : C[s_1, t_2, \ldots, t_n]{\Downarrow}(\overrightarrow{Q})$. Since $C[\cdot, t_2, \ldots, t_n]$ is a reduction context, the preconditions of the lemma applied to s_1, t_1 imply in particular: $C[t_1, t_2, \ldots, t_n]{\Downarrow}(\overrightarrow{P})$, a contradiction.

If no hole of $C[\cdot_1, \ldots, \cdot_n]$ is in a reduction context, then $C[s_1, \ldots, s_n]$ as well as $C[t_1, \ldots, t_n]$ can be reduced using the same normal order reduction at the same position.
From here on we can use the same considerations as in the proof of the previous lemma.
If the normal order reduction is non-(IOr), then we can use induction on the number of $\xrightarrow{n}$-reductions.
If the normal order reduction is an (IOr)-reduction for the pair (c, d), which is the first element of $\overrightarrow{P}$, then the IO-sequences have to be checked:
A new smaller example is constructed with an IO-sequence $\overrightarrow{P'}$ with $(c, d), \overrightarrow{P'} := \overrightarrow{P}$. The reduction can be written as $C[s_1, \ldots, s_n] \overset{n,(c,d)}{\longrightarrow} C'[s_1, \ldots, s_n]$ and $C[t_1, \ldots, t_n] \overset{n,(c,d)}{\longrightarrow} C'[t_1, \ldots, t_n]$ with $C'[s_1, \ldots, s_n]\Downarrow(\overrightarrow{P'})$. Since the first component of the measure is strictly smaller, we have also $C'[t_1, \ldots, t_n]\Downarrow(\overrightarrow{P'})$. But then we have $C[t_1, \ldots, t_n]\Downarrow(\overrightarrow{P})$, which contradicts the assumption that this is a counterexample.
Now we look at the base case. If C has no holes, then a counterexample is impossible.
If the number of normal order reduction steps is 0, then there are two cases:

1. $C[s_1, \ldots, s_n]$ is already a WHNF. Since no hole is in a reduction context, the context itself is a WHNF, and thus this holds for $C[s_1, \ldots, s_n]$ as well as $C[t_1, \ldots, t_n]$, which is impossible.
2. The next normal order reduction is impossible, since the next reduction would be an invalid IOr-reduction. This is not possible, since then there would be no normal order reduction from $C[t_1, \ldots, t_n]$ to a bot-term.

Concluding, we have proved that there is no counterexample to the general claim, hence the lemma holds, since it is a specialization. □
Combining the two lemmas 7.1 and 7.2, we obtain:

Lemma 7.3. *(Context Lemma) Let s, t be terms. If for all reduction contexts R: $\forall P : (R[s]\Downarrow(P) \Rightarrow R[t]\Downarrow(P))$, and if for all reduction contexts R: $(\forall \overrightarrow{P} : R[s]\Downarrow(\overrightarrow{P})) \Rightarrow (\forall \overrightarrow{P} : R[t]\Downarrow(\overrightarrow{P}))$, then $s \leq_c t$.*

In the following, we will show that several reductions keep contextual equivalence. However, all these reductions only change the terms and their normal order reductions in a controlled way. The following sufficient criterion for contextual equivalence allows easy proofs and is more appropriate for proving correctness of these reductions.

Lemma 7.4. *Let s, t be terms. A sufficient condition for $s \leq_c t$ is:*

$$\begin{aligned} \forall R[\cdot]\forall \overrightarrow{P} : \quad & (R[s]\downarrow(\overrightarrow{P}) \Rightarrow R[t]\downarrow(\overrightarrow{P})) \\ & \wedge (\overrightarrow{P} \textit{ valid for } R[t] \Rightarrow \overrightarrow{P} \textit{ valid for } R[s]) \end{aligned}$$

Proof. We show that both conditions of the two parts of the context lemma hold. If $\forall R[\cdot]\forall \overrightarrow{P} : R[s]{\downarrow}(\overrightarrow{P}) \Rightarrow R[t]{\downarrow}(\overrightarrow{P})$ holds, then of course also $\forall R[\cdot]\forall P : R[s]{\Downarrow}(P)) \Rightarrow R[t]{\Downarrow}(P)$. Hence we can apply Lemma 7.1.
For the second part, assume that the precondition

$$(\forall \overrightarrow{P} : R[s]{\Downarrow}(\overrightarrow{P})) \Rightarrow (\forall \overrightarrow{P} : R[t]{\Downarrow}(\overrightarrow{P}))$$

is wrong.
This means $\forall \overrightarrow{P} : R[s]{\Downarrow}(\overrightarrow{P})$ holds, but there is a $\overrightarrow{P_0}$, such that $R[t]{\Downarrow}(\overrightarrow{P_0})$ is wrong. Then the normal-order reduction of $R[t]$ along $\overrightarrow{P_0}$ reaches a bot-term. The assumption on validity implies that the normal-order reduction of $R[s]$ along $\overrightarrow{P_0}$ also reaches a term without a normal order reduction. This contradicts the assumption.
Summarizing, this means that the condition of the second context lemma 7.2 holds. From the context lemmas we can conclude that $s \leq_c t$. □

8 Correctness of Reductions

We show that non-IO-reductions of FUNDIO keep contextual equivalence.

Definition 8.1. *We define different kinds of internal reductions.*
An internal reduction *is any non-normal order reduction. We denote such a reduction as* $\xrightarrow{i}$*, perhaps with further labels.*

An R-internal reduction *is a non-normal order reduction, which takes place in a reduction context, i.e. the redex is in a reduction context. We denote such a reduction as* $\xrightarrow{iR}$*.*
An S-internal reduction *is a non-normal order reduction, which takes place in a surface context. We denote such a reduction as* $\xrightarrow{iS}$*.*

Note that we mainly use iR-reductions, which is different from the definition of internal reductions in [Bar84], however, it is sufficient to show correctness of a lot of program transformations if the context lemma is used.

Lemma 8.2. *There are no R-internal reductions of type (lbeta), (lapp), (lcase), (case-c), (case-lam), (IOlet).*

Proof. Whenever such a reduction is in a reduction context, it is already a normal order reduction, which can be checked by going through all the cases. □
The missing reductions are (llet), (cp), and (case-e), (case-in), which require a special treatment.

Example 8.3. There are (case)-reductions that are in a reduction context, but not normal order:
$(\texttt{letrec}\ x = a, y = \texttt{case}\ x\ (a \ \rightarrow\ b)\ alts\ \texttt{in}\ x) \xrightarrow{iR} (\texttt{letrec}\ x = a, y = (\texttt{letrec}\ \{\}\ \texttt{in}\ b)\ \texttt{in}\ x)$. The same for (IOr).

Definition 8.4. *A program transformation T from terms to terms is* correct*, iff for all t, t': $t\ T\ t'$ implies that $t \sim_c t'$*

Proposition 8.5. *The reductions (lbeta), (lapp), (lcase), (case-c), (case-lam), (IOlet) are correct program transformations.*

Proof. We use the fact that normal order reductions of type (lbeta), (lapp), (lcase), (case-c), (case-lam), (IOlet) are unique. Let $s \xrightarrow{a} t$, where $a \in \{$(lbeta), (lapp), (lcase), (case-c),(case-lam), (IOlet)$\}$. We have to show that $s \sim_c t$. Using the context lemma 7.3 and lemma 7.4, we see that it is sufficient to show the following for all reduction contexts R:
For all IO-sequences $\overrightarrow{P}$:
$\overrightarrow{P}$ is valid for $R[s]\ \Leftrightarrow \overrightarrow{P}$ is valid for $R[t]$ and
$R[s]{\downarrow}(\overrightarrow{P}) \Rightarrow R[t]{\downarrow}(\overrightarrow{P})$ and also the reverse: $R[t]{\downarrow}(\overrightarrow{P}) \Rightarrow R[s]{\downarrow}(\overrightarrow{P})$.
Lemma 8.2 shows that $s \xrightarrow{i} t$ implies $s \xrightarrow{n,a} t$, and that the normal-order reduction is unique. Since R is a reduction context, we have $R[s] \xrightarrow{n,a} R[t]$. We have that $R[s]{\downarrow}(\overrightarrow{P})$ implies $R[t]{\downarrow}(\overrightarrow{P})$, since the normal-order reduction step is unique and the first reduction does not use IO-pairs. If $R[t]{\downarrow}(\overrightarrow{P})$, then we also have $R[s]{\downarrow}(\overrightarrow{P})$, since $R[s]$ normal-order reduces to $R[t]$ without using IO-pairs.
The equivalence of validity also holds, since the reduction a does not use an IO-pair.

□

Note that (IOr) is not correct as a program transformation, since $(IO\ d)$ may reduce to different constants, which are clearly not contextual equivalent.

9 Complete Sets of Commuting and Forking Diagrams

For proving correctness of further program transformations, we require the notions of a complete set of commuting diagrams and of a complete set of forking diagrams.
A *reduction sequence* is of the form $t_1 \to \ldots \to t_n$, where t_i are terms and $t_i \to t_{i+1}$ is a FUNDIO-reduction, if not specified otherwise, as defined in definition 4.3. In the following definition we describe transformations on reduction sequences. Therefore we use the notation

$$\xrightarrow{iX,red,P_0} . \xrightarrow{n,a_1,P_1} \ldots \xrightarrow{n,a_k,P_k} \quad \rightsquigarrow \quad \xrightarrow{n,b_1,P'_1} \ldots \xrightarrow{n,b_m,P'_m} . \xrightarrow{iX,red_1,P''_1} \ldots\ldots \xrightarrow{iX,red_h,P''_h}$$

for transformations on reduction sequences. Here the notation $\xrightarrow{iX,red,P}$ means a reduction with $\mathtt{iX} \in \{iC, iR, iS\}$, *red* is a reduction from FUNDIO, and P, P_i are the IO-pairs of the reductions, if necessary. P_i may also be variables to indicate that certain IO-pairs are retained during the transformation.
The above transformation rule can be applied to the prefix of the reduction sequence RED, if the prefix is: $s \xrightarrow{iX,red,P_0} t_1 \xrightarrow{n,a_1,P_1} \ldots t_k \xrightarrow{n,a_k,P_k} t$. Since we will

use sets of transformation rules, it may be the case that there is a transformation rule in the set, where the pattern matches a prefix, but it is not applicable, since the right hand side cannot be constructed.
We will say, a transformation rule is *applicable* to the prefix of the reduction sequence RED, where the prefix is: $s \xrightarrow{iX,red,P_0} x_1 \xrightarrow{n,a_1,P_1} \ldots x_k \xrightarrow{n,a_k,P_k} t$, iff the following holds:

$$\begin{array}{l} \exists y_1,\ldots,y_m,z_1,\ldots,z_{h-1}: \\ s \xrightarrow{n,b_1,P'_1} y_1 \ldots \xrightarrow{n,b_m,P'_m} y_m \xrightarrow{iX,red_1,P''_1} z_1 \ldots z_{h-1} \xrightarrow{iX,red_h,P''_h} t \end{array}$$

The transformation consists in replacing this prefix with the result:

$$s \xrightarrow{n,b_1,P'_1} t'_1 \ldots t'_{m-1} \xrightarrow{n,b_m,P'_m} t'_m \xrightarrow{iX,red_1,P''_1} t''_1 \ldots t''_{h-1} \xrightarrow{iX,red_h,P''_h} t$$

where the terms in between are appropriately constructed.

Definition 9.1.
• *A* complete set of commuting diagrams for the reduction $\xrightarrow{iX,red,P_0}$ *is a set of transformation rules on reduction sequences of the form*

$$\xrightarrow{iX,red,P_0} \cdot \xrightarrow{n,a_1,P_1} \ldots \xrightarrow{n,a_k,P_k} \rightsquigarrow \xrightarrow{n,b_1,P'_1} \ldots \xrightarrow{n,b_m,P'_m} \cdot \xrightarrow{iX,red_1,P''_1} \ldots\ldots \xrightarrow{iX,red_{k'},P'_{k'}} ,$$

where $k,k' \geq 0, m \geq 1$, such that in every reduction sequence $t_0 \xrightarrow{iX,red,P_0} t_1 \xrightarrow{n} \ldots \xrightarrow{n} t_h$, where t_h is a WHNF, at least one of the transformation rules is applicable to a prefix of the sequence.
In the special case $h = 1$, we require that in $t_0 \xrightarrow{iX,red,P_0} t_1$, the term t_1 is a WHNF, and the term t_0 is not a WHNF.

• *A* complete set of forking diagrams for the reduction $\xrightarrow{iX,red,P}$ *is a set of transformation rules on reduction sequences of the form*

$$\xleftarrow{n,a_1,P_1} \ldots \xleftarrow{n,a_k,P_k} \cdot \xrightarrow{iX,red,P_0} \rightsquigarrow \xrightarrow{iX,red_1,P'_1} \ldots\ldots \xrightarrow{iX,red_{k'},P'_{k'}} \cdot \xleftarrow{n,b_1,P''_1} \ldots \xleftarrow{n,b_m,P''_m} ,$$

where $k,k' \geq 0, m \geq 1$, such that for every reduction sequence $t_h \xleftarrow{n} \ldots t_2 \xleftarrow{n} t_1 \xrightarrow{iX,red,P_0} t_0$, where t_h is a WHNF, at least one of the transformation rules from the set is applicable to a suffix of the sequence. In the special case that $h = 1$, we require that in $t_1 \xrightarrow{iX,red,P_0} t_0$, the term t_1 is a WHNF, and that t_0 is not a WHNF.

The two different kinds of diagrams are required for two different parts of the proof for the contextual equivalence of two terms.
In the following, the verification of the reduction diagrams is done mostly on paper. It is necessary to extend the work on checking and testing the diagrams automatically as already done for a simpler language in [Hub00].

10 Correctness of (llet)

We define the reduction (lll) as the union of $\{llet, lapp, lcase, IOlet\}$. The notation $\xrightarrow{(lll)^*}$ then means a reduction sequence of an arbitrary number of reductions from the set $\{llet, lapp, lcase, IOlet\}$.

Lemma 10.1. *The reduction (lll) cannot be applied infinitely often.*

Proof. Let the depth of a position p ignoring intermediate `letrec`-expressions be the function $dpl(p)$. In every (lll)-reduction, the following number is strictly reduced:

> The sum of the number of `letrec` -expressions in t plus the sum of all $dpl(p)$ where p is the position of a `letrec`-subexpression.

Since this number is strictly reduced by every (lll)-reduction, termination holds. □

Lemma 10.2. *If a* `letrec` *occurs in a reduction context, then it can be shifted upwards to the top using* $\xrightarrow{iR,lll}$ *- and normal-order (lll)-reductions:*

1. $R^-[$`letrec` Env `in` $t] \xrightarrow{(n,lll)^*}$ `letrec` Env `in` $R^-[t]$.
2. *If* $R =$ (`letrec` $x_1 = R_1^-[\cdot], \ldots, x_j = R_{j-1}^-[x_{j-1}], Env_1$ `in` $R^-[x_j]$) *is a reduction context, then*
   (`letrec` $x_1 = R_1^-[$`letrec` Env_2 `in` $t], \ldots, x_j = R_{j-1}^-[x_{j-1}], Env_1$ `in` $R^-[x_j]$)
   $\xrightarrow{(n,lll)^*}$
   (`letrec` $x_1 = R_1^-[t], \ldots, x_j = R_{j-1}^-[x_{j-1}], Env_1, Env_2$ `in` $R^-[x_j]$).
3. (`letrec` Env_1 `in` $R^-[$`letrec` Env_2 `in` $t]$) $\xrightarrow{(n,lll)^*}$ (`letrec` Env_1, Env_2 `in` $R^-[t]$)

Proof. This follows by checking all the cases of the reductions $\{llet, lapp, lcase, IOlet\}$. □

In the following $\xrightarrow{(n,a,P)}$ means a normal-order reduction with an arbitrary reduction a, and if it is an (IOr)-reduction then P denotes the IO-pair of the reduction.

Lemma 10.3. *A complete set of commuting diagrams for* $\xrightarrow{iR,llet}$ *is:*

$$\xrightarrow{(iR,llet)} \cdot \xrightarrow{(n,a,P)} \quad \rightsquigarrow \quad \xrightarrow{(n,a,P)} \cdot \xrightarrow{(iR,llet)}$$
$$\xrightarrow{(iR,llet)} \cdot \xrightarrow{(n,lll)^+} \quad \rightsquigarrow \quad \xrightarrow{(n,lll)^+}$$
$$\xrightarrow{(iR,llet)} \cdot \xrightarrow{(n,lll)^+} \quad \rightsquigarrow \quad \xrightarrow{(n,lll)^+} \cdot \xrightarrow{(iR,llet)}$$

Proof. An (iR,llet)-reduction has the following possibilities:

1. it is a (llet-e) in the empty context, and the inner `letrec`-term is not in a reduction context. Then we can only have the commuting case.
2. it is in a nontrivial reduction context and the normal order reduction is a (llet). There are 4 combinations of the subcases of (llet), but there may be six reduction completions.

$$\begin{array}{lcl}
(iR, \mathit{llet\text{-}in}).(n, \mathit{llet\text{-}e}) & \rightsquigarrow & (n, \mathit{llet\text{-}e}).(n, \mathit{llet\text{-}in}) \\
(iR, \mathit{llet\text{-}in}).(n, \mathit{llet\text{-}in}) & \rightsquigarrow & (n, \mathit{llet\text{-}in}).(n, \mathit{llet\text{-}in}) \\
(iR, \mathit{llet\text{-}e}).(n, \mathit{llet\text{-}e}) & \rightsquigarrow & (n, \mathit{llet\text{-}in}).(iR, \mathit{llet\text{-}in}) \\
(iR, \mathit{llet\text{-}e}).(n, \mathit{llet\text{-}e}) & \rightsquigarrow & (n, \mathit{llet\text{-}in}).(n, \mathit{llet\text{-}in}) \\
(iR, \mathit{llet\text{-}e}).(n, \mathit{llet\text{-}in}) & \rightsquigarrow & (n, \mathit{llet\text{-}in}).(iR, \mathit{llet\text{-}e}) \\
(iR, \mathit{llet\text{-}e}).(n, \mathit{llet\text{-}in}) & \rightsquigarrow & (n, \mathit{llet\text{-}in}).(n, \mathit{llet\text{-}e})
\end{array}$$

An illustration of the third and fourth case is:

$$\begin{array}{ll}
 & \texttt{letrec}\ x = (\texttt{letrec}\ y = (\texttt{letrec}\ z = t_z\ \texttt{in}\ t_y)\ \texttt{in}\ t_x), \mathit{Env}\ \texttt{in}\ R^-[x] \\
\xrightarrow{iR,\mathit{llet\text{-}e}} & \texttt{letrec}\ x = (\texttt{letrec}\ y = t_y, z = t_z\ \texttt{in}\ t_x), \mathit{Env}\ \texttt{in}\ R^-[x] \\
\xrightarrow{n,\mathit{llet\text{-}e}} & \texttt{letrec}\ x = t_x, y = t_y, z = t_z, \mathit{Env}\ \texttt{in}\ R^-[x] \\
\hline
\xrightarrow{n,\mathit{llet\text{-}e}} & \texttt{letrec}\ x = t_x, y = (\texttt{letrec}\ z = t_z\ \texttt{in}\ t_y), \mathit{Env}\ \texttt{in}\ R^-[x] \\
\xrightarrow{iR\vee n,\mathit{llet\text{-}e}} & \texttt{letrec}\ x = t_x, y = t_y, z = t_z, \mathit{Env}\ \texttt{in}\ R^-[x] \\
\hline
\end{array}$$

The last reduction is an (n) if $t_x = R_1^-[y]$, otherwise it is an (iR).
3. it is in a nontrivial reduction context and the normal order reduction is a (cp), (lbeta),(case), or (IOr). In all the cases, there is no interference, hence the commutation of the reductions holds.
4. it is in a nontrivial reduction context and the normal order reduction is a (lcase), (lapp), or (IOlet). We have to distinguish the cases (iR,*llet-e*) and (iR,*llet-in*). The easy case is (iR,*llet-e*), which requires two diagrams:

$$\begin{array}{lcl}
(iR, \mathit{llet\text{-}e}).(n, \mathit{lll}) & \rightsquigarrow & (n, \mathit{lll}).(iR, \mathit{llet\text{-}e}) \\
(iR, \mathit{llet\text{-}e}).(n, \mathit{lll}) & \rightsquigarrow & (n, \mathit{lll}).(n, \mathit{llet\text{-}e})
\end{array}$$

In the case (iR,*llet-in*) the required diagram is
$\xrightarrow{(iR,llet)} \cdot \xrightarrow{(n,lll)^+} \ \rightsquigarrow\ \xrightarrow{(n,lll)^+}$.
We show a typical case:

$$\begin{array}{ll}
 & R[(\texttt{IO}\ (\texttt{letrec}\ x = t_x\ \texttt{in}\ \texttt{letrec}\ y = t_y\ \texttt{in}\ t))] \\
\xrightarrow{iR,\mathit{llet\text{-}in}} & R[(\texttt{IO}\ (\texttt{letrec}\ x = t_x, y = t_y\ \texttt{in}\ t))] \\
\xrightarrow{n,IOlet} & R[(\texttt{letrec}\ x = t_x, y = t_y\ \texttt{in}\ (\texttt{IO}\ t))] \\
\hline
\xrightarrow{n,IOlet} & R[(\texttt{letrec}\ x = t_x\ \texttt{in}\ \texttt{IO}\ (\texttt{letrec}\ y = t_y\ \texttt{in}\ t))] \\
\xrightarrow{(n,lll)^*} & (\texttt{letrec}\ x = t_x\ \texttt{in}\ R[\texttt{IO}\ (\texttt{letrec}\ y = t_y\ \texttt{in}\ t)]) \\
\xrightarrow{(n,IOlet)} & (\texttt{letrec}\ x = t_x\ \texttt{in}R[(\texttt{letrec}\ y = t_y\ \texttt{in}\ (\texttt{IO}\ t))]) \\
\xrightarrow{(n,lll)^*} & (\texttt{letrec}\ x = t_x, y = t_y\ \texttt{in}\ R[\texttt{IO}\ t])
\end{array}$$

The reductions that shift the `letrec`s to the top are possible according to Lemma 10.2.

□

Note that the diagrams would change, if empty `letrec`s are forbidden, since then a `case` on a constant can interfere with a (llet).

Lemma 10.4. *A complete set of forking diagrams for* $\xrightarrow{iR,llet}$ *is:*

$$\begin{array}{l}
\xleftarrow{n,a,P} \cdot \xrightarrow{iR,llet} \rightsquigarrow \xrightarrow{iR,llet} \cdot \xleftarrow{n,a,P} \\
\xleftarrow{(n,lll)^+} \cdot \xrightarrow{iR,llet} \rightsquigarrow \xrightarrow{iR,llet} \cdot \xleftarrow{(n,lll)^+} \\
\xleftarrow{(n,lll)^+} \cdot \xrightarrow{iR,llet} \rightsquigarrow \xleftarrow{(n,lll)^+}
\end{array}$$

Proof. We have to check the overlappings of (iR, llet)-reductions with normal-order reductions. The cases are the same as for the commuting diagrams, hence the diagrams can be obtained using the commuting diagrams as guide. □

Lemma 10.5. *If* $s \xrightarrow{iR,llet} t$*, then* s *is a WHNF iff* t *is a WHNF.*

Proposition 10.6. *If* $s \xrightarrow{llet} t$*, then* $s \sim_c t$*.*

Proof. The context lemma 7.4 shows that it is sufficient to analyze the situation where the (llet)-reduction is in a reduction context. We mainly argue that the termination equivalence holds for every IO-sequence. The proof for equivalence of validity can be done in the same way.
We assume that $s \equiv R[s']$ and s' is the (llet)-redex and R is a reduction context. Let $s' \to t'$ be the (llet)-reduction. Then either $R[s'] \xrightarrow{iR,llet} R[t']$ or $R[s'] \xrightarrow{n,llet} R[t']$. In the latter case the assumption of the context lemma holds, since $\xrightarrow{n,llet}$ is unique and does not use an IO-pair. For the rest of the proof assume that $R[s'] \xrightarrow{iR,llet} R[t']$.
Let s have a normal order reduction to the WHNF s'' with an IO-sequence $\overrightarrow{P}$. Induction on the length k of the normal order reduction and using the complete set of forking diagrams for (llet) in Lemma 10.4 shows that we can construct a normal order reduction sequence for t with the same IO-sequence. Here we use the fact that the transformations guarantees that at most one R-internal (llet)-reduction during transformation has o be considered. The number of normal-order reductions to the right of the (iR,llet)-reduction is strictly decreased. Furthermore, the IO-sequence is not changed by the transformation. The following diagram shows one possibility during the induction:

$$\begin{tikzcd}
s \arrow[r, "iR{,}llet"] \arrow[dd, "n"', "k"] & t & {} \arrow[r, dotted] & {} & s & t \arrow[d, "n"', "1"] \\
& & & & s' \arrow[r, "iR{,}llet"] \arrow[d, "n"', "k-1"] & t' \\
s'' & & & & {} &
\end{tikzcd}$$

Lemma 10.5 shows the constructed normal-order reduction terminates in a WHNF.

Let t have a normal order reduction to a WHNF with an IO-sequence $\overrightarrow{P}$. Then induction on the length of a normal order reduction and using the commuting diagrams for (llet) in Lemma 10.3 again is sufficient to show that s has a normal order reduction sequence to a WHNF with the same IO-sequence.
The following diagram schematically shows one possible step in the induction.

$$\begin{tikzcd}
s \arrow[r, "iR{,}llet"] & t \arrow[dd, "n"', "k"] & & & s \arrow[d, "n"', "1"] & t \\
& & {} \arrow[r, dotted] & {} & s' \arrow[r, "iR{,}llet"] & t' \arrow[d, "n"', "k-1"] \\
& t'' & & & & t''
\end{tikzcd}$$

Lemma 10.5 shows that the constructed normal-order reduction terminates in a WHNF. □

Proposition 10.7. *(llet) is a correct program transformation.*

11 Correctness of (cp)-Reductions

To show that the (cp)-reduction is correct as a program transformation, we have to split the reduction into two different reductions, depending on the position of the target variable.
(cpt) = (cp) where the replaced position of the variable is in a surface context.
(cpd) = (cp) where the replaced position of the variable is not in a surface context.
Note that the rule (cp) can be applied infinitely often:
`letrec x = \y. x y  in t` → `letrec x = \y. (\z. x z) y  in t` → ....
This also means that (cpd) may cause an infinite reduction sequence.
This is not possible for the rule (cpt):

Lemma 11.1. *The reduction (cpt) cannot be applied infinitely often.*

Proof. We show that (cpt) strictly reduces a well-founded measure of terms. The measure of t is the number of variable occurrences at surface positions in the term t. The (cpt)-reduction replaces such a variable occurrence by an abstraction. It removes one such variable position without introducing new ones. □

Lemma 11.2. *A complete set of commuting diagrams for* $\xrightarrow{iR,cpt}$ *is:*

$$\xrightarrow{(iR,cpt)} \cdot \xrightarrow{(n,a,P)} \rightsquigarrow \xrightarrow{(n,a,P)} \cdot \xrightarrow{(iR,cpt)}$$

$$\xrightarrow{(iR,cpt)} \cdot \xrightarrow{(n,a,P)} \rightsquigarrow \xrightarrow{(n,a,P)} \cdot \xrightarrow{(n,cp)}$$

$$\xrightarrow{(iR,cpt)} \cdot \xrightarrow{(n,case)} \rightsquigarrow \xrightarrow{(n,case)}$$

Proof. A reduction (iR,cpt) replaces a single variable by an abstraction. Since it is not a normal order reduction, the positions that are in reductions contexts do not change with one exception:
`letrec` $x_1 = s, x_2 = x_1, \ldots$ `in` $R[x_m] \rightarrow$ `letrec` $x_1 = s, x_2 = s, \ldots$ `in` $R[x_m]$
where s is an abstraction in a reduction context. There is also no normal order reduction that has the effect that a `letrec`-expression is moved out of a reduction context. Going through the possibilities, the (iR,cpt)-reduction may be superfluous if the target is in a lost alternative of a `case`, there may be commutation of the (iR,cpt) and the (n,a)-reduction, where the (iR,cpt)-reduction may remain internal or turn into a normal order (cp)-reduction. □

Lemma 11.3. *A complete set of forking diagrams for* $\xrightarrow{iR,cpt}$ *is:*

$$\xleftarrow{n,a,P} \cdot \xrightarrow{iR,cpt} \rightsquigarrow \xrightarrow{iR,cpt} \cdot \xleftarrow{n,a,P}$$

$$\xleftarrow{n,case} \cdot \xrightarrow{iR,cpt} \rightsquigarrow \xleftarrow{n,case}$$

$$\xleftarrow{n,cp} \cdot \xleftarrow{n,a,P} \cdot \xrightarrow{iR,cpt} \rightsquigarrow \xleftarrow{n,a,P}$$

Lemma 11.4. *A complete set of commuting diagrams for* $\xrightarrow{iR,cpd}$ *is:*

$$\xrightarrow{(iR,cpd)} \cdot \xrightarrow{(n,a,P)} \rightsquigarrow \xrightarrow{(n,a,P)} \cdot \xrightarrow{(iR,cpd)}$$

$$\xrightarrow{(iR,cpd)} \cdot \xrightarrow{(n,cp)} \rightsquigarrow \xrightarrow{(n,cp)} \cdot \xrightarrow{(iR,cpd)} \cdot \xrightarrow{(iR,cpd)}$$

$$\xrightarrow{(iR,cpd)} \cdot \xrightarrow{(n,case)} \rightsquigarrow \xrightarrow{(n,case)}$$

$$\xrightarrow{(iR,cpd)} \cdot \xrightarrow{(n,lbeta)} \rightsquigarrow \xrightarrow{(n,lbeta)} \cdot \xrightarrow{(iR,cpt)}$$

$$\xrightarrow{(iR,cpd)} \cdot \xrightarrow{(n,lbeta)} \rightsquigarrow \xrightarrow{(n,lbeta)} \cdot \xrightarrow{(n,cp)}$$

Proof. The first and third cases occur als in the diagrams for (cpt); the second occurs, if the (iR,cpd) copies into the abstraction that is copied by the other reduction. The fourth and fifth cases are required, if the target variable of the copy is in the body of an abstraction that is turned into a `letrec` by an (lbeta). □

Lemma 11.5. *A complete set of forking diagrams for* $\xrightarrow{iR,cpd}$ *is:*

$$\begin{array}{lll}
\xleftarrow{n,a,P} \cdot \xrightarrow{iR,cpd} & \leadsto & \xrightarrow{iR,cpd} \cdot \xleftarrow{n,a,P} \\
\xleftarrow{n,a,P} \cdot \xrightarrow{iR,cpd} & \leadsto & \xrightarrow{iR,cpd} \cdot \xrightarrow{iR,cpd} \cdot \xleftarrow{n,a,P} \\
\xleftarrow{n,case} \cdot \xrightarrow{iR,cpd} & \leadsto & \xleftarrow{n,case} \\
\xleftarrow{n,lbeta} \cdot \xrightarrow{iR,cpd} & \leadsto & \xrightarrow{iR,cpt} \cdot \xleftarrow{n,lbeta} \\
\xleftarrow{n,cp} \cdot \xleftarrow{n,lbeta} \cdot \xrightarrow{iR,cpd} & \leadsto & \xleftarrow{n,lbeta}
\end{array}$$

Lemma 11.6. *If* $s \xrightarrow{iR,cp} t$*, then* s *is a WHNF iff* t *is a WHNF.*

Proposition 11.7. *The reduction (cp) is a correct program transformation*

Proof. Due to the context lemma, it is sufficient to consider the situation $s = R[s'], t = R[t'], s' \xrightarrow{cp} t'$, such that $s \xrightarrow{iR,cp} t$.
Again we omit the argumentation for equivalence wrt validity, since it can be derived from the proof below.
First assume that t has a normal order reduction RED to WHNF with IO-sequence $\overrightarrow{P}$. We assume that the reduction $s \xrightarrow{iR,cp} t \cdot RED$[1] is transformed into a normal-order reduction for s terminating with a WHNF, where the transformations are used that correspond to the complete set of commuting diagrams in Lemmas 11.4,11.2.
We have to show that the transformation terminates. We have to give a well-founded measure for reduction sequences RED' where $\xrightarrow{(iR,cpd)}$, $\xrightarrow{(iR,cpt)}$ and normal order reductions are mixed.
A single (iR, cpd) or (iR, cpt) in RED' has as measure the triple consisting of

1. the number of (n,lbeta)-reductions right of it;
2. the number of all (n,cp)- and (iR,cpt)-reductions right of it before the next (n,lbeta)-reduction;
3. the number of normal-order reductions before the next (n,cp)- or (iR,cpt)-reduction.

The pairs are ordered lexicographically. The measure μ of the whole reduction sequence is the multiset of the triples for all iR-reductions, ordered by the multiset-ordering. Now every transformation rule of the commuting diagrams for (iR,cpd) and (iR,cpt) strictly decreases the measure μ. This can be verified by going through the 8 possible transformation rule applications. That the measure is decreased must also be checked for (iR,cpd) and (iR,cpt)- reductions that are not directly involved in the transformation.
There are the following critical ones: 1. The rule that duplicates (i,cpd)-reductions replaces a pair by two smaller ones, which makes the multiset strictly

[1] We use $s \xrightarrow{iR,cp} t \cdot RED$ as a notation for the combined reduction sequence.

smaller. 2. Changing (i,cpd) into (i,cpt) or (n,cp) can only be done by jumping over an (n,lbeta)-reduction, hence also in this case the multiset is strictly decreased. The other rules are easy.
In summary, the rules transform the reduction sequence into a reduction sequence $s \xrightarrow{n,*} s'' \xrightarrow{iR,*} s'''$, where the normal order reduction $s \xrightarrow{n,*} s''$ has the same IO-sequence as RED. Due to Lemma 11.6 the normal-order reduction terminates in a WHNF.

The other case is that s has a normal order reduction RED with IO-sequence $\overrightarrow{P}$. We have to show that t has a normal order reduction with the same IO-sequence $\overrightarrow{P}$. Now we use the forking diagrams in Lemmas 11.3, 11.3 .
The mechanism of the proof is similar as above. It is sufficient to show that the transformation terminates. The well-founded measure for reduction sequences Q' where $(iR, cpd), (iR, cpt)$ and normal order reductions are mixed is as follows: An (iR, cpd) or (iR, cpt) in Q' has as measure the triple consisting of

1. the number of $\xleftarrow{(n,lbeta)}$-reductions left of it;
2. the number of $\xleftarrow{(n,cp)}$ and $\xrightarrow{(iR,cpt)}$-reductions left of it before the next $\xleftarrow{(n,lbeta)}$-reduction.
3. the number of normal-order reduction left to it before the next $\xleftarrow{(n,cp)}$ or $\xrightarrow{(iR,cpt)}$-reduction.

The triples are ordered lexicographically. The measure μ of the whole reduction sequence is the multiset of the triples for all iR-reductions, ordered by the multiset-ordering. Lemma 11.6 shows that the constructed normal-order reduction terminates in a WHNF.

Now we can conclude by applying the context lemma for the two directions that $s \sim_c t$. □

12 Correctness of Garbage Collection (gc)

In this and the following sections, we use diagrams, where also other reductions are permitted, not only FUNDIO-reductions.
There are two forms of garbage collections:

(gc) • $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, Env\ \texttt{in}\ t) \rightarrow (\texttt{letrec}\ Env\ \texttt{in}\ t)$
if for all $i : x_i$ does not occur in Env nor in t
• $(\texttt{letrec}\ \{\}\ \texttt{in}\ t) \rightarrow t$

The first is able to collect cyclic references; the second eliminates `letrec`s with empty environment.

Lemma 12.1. *The rule (gc) cannot be applied infinitely often.*

Again we have to find complete sets of commuting and forking diagrams.

Lemma 12.2. *A complete set of commuting diagrams for* $\xrightarrow{(iR,gc)}$ *is:*

$$\begin{array}{lcl}
\xrightarrow{(iR,gc)} \cdot \xrightarrow{(n,a,P)} & \rightsquigarrow & \xrightarrow{(n,a,P)} \cdot \xrightarrow{(iR,gc)} \\
\xrightarrow{(iR,gc)} \cdot \xrightarrow{(n,a,P)} & \rightsquigarrow & \xrightarrow{(n,lll)^*} \cdot \xrightarrow{(n,a,P)} \cdot \xrightarrow{(iR,gc)} \\
\xrightarrow{(iR,gc)} \cdot \xrightarrow{(n,a,P)} & \rightsquigarrow & \xrightarrow{(n,lll)^*} \cdot \xrightarrow{(n,a,P)} \\
\xrightarrow{(iR,gc)} & \rightsquigarrow & \xrightarrow{(n,llet)}
\end{array}$$

Proof. A case analysis shows the validity of the diagrams. We give a typical example

$$\begin{array}{ll}
 & (\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{IO}\ (\texttt{IO}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ a)))) \\
\xrightarrow{iR,gc} & (\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{IO}\ (\texttt{IO}\ a))) \\
\xrightarrow{n,IOr,(a,b)} & (\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{IO}\ b)) \\
\hline
\xrightarrow{n,lll} & (\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{IO}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ (\texttt{IO}\ a)))) \\
\xrightarrow{n,lll} & (\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ (\texttt{IO}\ (\texttt{IO}\ a)))) \\
\xrightarrow{n,lll} & (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ (\texttt{IO}\ (\texttt{IO}\ a))) \\
\xrightarrow{n,IOr,(a,b)} & (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ (\texttt{IO}\ b)) \\
\xrightarrow{iR,gc} & (\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{IO}\ b))
\end{array}$$

If the environment Env_2 is empty, we get the third transformation rule. The last transformation rule is the special case of the term $\texttt{letrec}\ \{\}\ \texttt{in}\ \texttt{letrec}\ Env\ \texttt{in}\ t$, where t is a value, and the reduction $\xrightarrow{(iR,gc)}$ transforms the term into a WHNF. □

Lemma 12.3. *A complete set of forking diagrams for* $\xrightarrow{(iR,gc)}$ *is:*

$$\begin{array}{lcl}
\xleftarrow{n,a,P} \cdot \xrightarrow{iR,gc} & \rightsquigarrow & \xrightarrow{iR,gc} \cdot \xleftarrow{n,a,P} \\
\xleftarrow{(n,a,P)} \cdot \xleftarrow{(n,lll)^*} \cdot \xrightarrow{iR,gc} & \rightsquigarrow & \xrightarrow{iR,gc} \cdot \xleftarrow{n,a,P} \\
\xleftarrow{(n,a,P)} \cdot \xleftarrow{(n,lll)^*} \cdot \xrightarrow{iR,gc} & \rightsquigarrow & \xleftarrow{n,a,P} \\
\xleftarrow{(n,llet)} \cdot \xrightarrow{iR,gc} & \rightsquigarrow & \emptyset
\end{array}$$

Proof. A case analysis using the same cases as in the previous lemma is sufficient. □

Lemma 12.4. *Let* $s \xrightarrow{iR,gc} t$.

- *If s is a WHNF, then t is a WHNF.*

- *If t is a WHNF, but s is not a WHNF, then we have the special case that $s \equiv \texttt{letrec}\ \{\}\ \texttt{in}\ \texttt{letrec}\ Env\ \texttt{in}\ s'$, where s' is a value, or s' is a variable which is bound in Env to a constructor application.*

Proposition 12.5. *The rule (gc) is a correct program transformation.*

Proof. Due to the context lemma, it is sufficient to consider the situation $s = R[s'], t = R[t'], s' \xrightarrow{gc} t'$, such that $s \xrightarrow{iR,gc} t$.
First assume that t has a normal order reduction RED with IO-sequence $\overrightarrow{P}$. We assume that the reduction $s \xrightarrow{iR,gc} t\ RED$ is transformed into a normal-order reduction using the complete set of commuting diagrams in Lemma 12.2. This transformation is a shift of $\xrightarrow{iR,gc}$ to the right, where it is possible that $(n, lll)^*$-reductions have to be inserted. The number of transformation steps is at most the length of the reduction sequence RED. Eventually, a normal order reduction with the same IO-sequence $\overrightarrow{P}$ is constructed. Lemma 12.4 shows that the constructed normal order reduction sequence terminates with a WHNF.
Now assume that s has a normal order reduction sequence RED from right to left and with IO-sequence $\overrightarrow{P}$. Now we shift $\xrightarrow{iR,gc}$ to the left in the reduction $RED\ s \xrightarrow{iR,gc} t$ using the complete set of forking diagrams in Lemma 12.3. Lemma 12.4 shows that the constructed normal order reduction sequence terminates with a WHNF.

□

13 Correctness of (cpx)-Reductions

The further development requires to consider the reduction that shortens indirections.

(cpx-in) $(\texttt{letrec}\ x = y, \ldots, Env\ \texttt{in}\ C[x])$
$\rightarrow (\texttt{letrec}\ x = y, Env\ \texttt{in}\ C[y])$ where y is a variable and $x \neq y$
(cpx-e) $(\texttt{letrec}\ x = y, z = C[x], Env\ \texttt{in}\ t)$
$\rightarrow (\texttt{letrec}\ x = y, z = C[y], Env\ \texttt{in}\ t)$ where y is a variable and $x \neq y$

We do not allow the (useless) reduction $\texttt{letrec}\ x = x\ \texttt{in}\ t \rightarrow \texttt{letrec}\ x = x\ \texttt{in}\ t$. The union of (cpx-in) and (cpx-e) is denoted as (cpx).
Note that the reduction $\xrightarrow{iR,cpx}$ may not terminate:
$\texttt{letrec}\ x = y, y = x\ \texttt{in}\ C[x] \xrightarrow{iR,cpx} \texttt{letrec}\ x = y, y = x\ \texttt{in}\ C[y] \xrightarrow{iR,cpx} \texttt{letrec}\ x = y, y = x\ \texttt{in}\ C[x]$.
A further example for non-termination is: $\texttt{letrec}\ x = y, y = x, z = x\ \texttt{in}\ t \xrightarrow{iR,cpx} \texttt{letrec}\ x = y, y = x, z = y\ \texttt{in}\ t \xrightarrow{iR,cpx} \texttt{letrec}\ x = y, y = x, z = x\ \texttt{in}\ t$

Lemma 13.1. *A complete set of commuting diagrams for $\xrightarrow{iR,cpx}$ is:*

$$\xrightarrow{(iR,cpx)} \cdot \xrightarrow{(n,a,P)} \leadsto \xrightarrow{(n,a,P)} \cdot \xrightarrow{(iR,cpx)}$$

$$\xrightarrow{(iR,cpx)} \cdot \xrightarrow{(n,cp)} \leadsto \xrightarrow{(n,cp)} \cdot \xrightarrow{(iR,cpx)} \cdot \xrightarrow{(iR,cpx)}$$

$$\xrightarrow{(iR,cpx)} \cdot \xrightarrow{(n,a)} \leadsto \xrightarrow{(n,a)}$$

The second case happens if the target of the (cpx)-reduction is in the copied abstraction of the (cp). The third case may happen if the reduction is a (case) or (cp). An example for the last case is $\texttt{letrec}\ x = s, y = x\ \texttt{in}\ C[y] \xrightarrow{iR,cpx} \texttt{letrec}\ x = s, y = x\ \texttt{in}\ C[x] \xrightarrow{n,cp} \texttt{letrec}\ x = s, y = x\ \texttt{in}\ C[s]$, and also $\texttt{letrec}\ x = s, y = x\ \texttt{in}\ C[y] \xrightarrow{n,cp} \texttt{letrec}\ x = s, y = x\ \texttt{in}\ C[s]$.

Lemma 13.2. *A complete set of forking diagrams for* $\xrightarrow{iR,cpx}$ *is:*

$$\xleftarrow{n,a,P} \cdot \xrightarrow{iR,cpx} \leadsto \xrightarrow{iR,cpx} \cdot \xleftarrow{n,a,P}$$

$$\xleftarrow{n,cp} \cdot \xrightarrow{iR,cpx} \leadsto \xrightarrow{iR,cpx} \cdot \xrightarrow{iR,cpx} \cdot \xleftarrow{n,cp}$$

$$\xleftarrow{n,a} \cdot \xrightarrow{iR,cpx} \leadsto \xleftarrow{n,a}$$

Lemma 13.3. *If* $s \xrightarrow{iR,cpx} t$*, then* s *is a WHNF iff* t *is a WHNF.*

Proposition 13.4. *The reduction (cpx) is a correct program transformation.*

Proof. We only show the non-standard parts of the proof, which is termination of the transformation process. There are two cases for the transformation. First consider the transformation of $s \xrightarrow{iR,cpx} t \cdot RED$ into a normal order reduction sequence from s to WHNF, where RED is a normal order reduction to a WHNF. Intermediate steps have a sequence of normal-order reductions mixed with (iR,cpx)-reductions. We measure the sequences by the multiset consisting of the following numbers: for every (iR,cpx)-reduction, the number of normal-order reductions to the right of it. This is a well-founded order, and it is easy to see that the transformations strictly reduce this measure in every step using the commuting diagrams.

The other case is the transformation of $\overline{RED} \cdot s \xrightarrow{iR,cpx} t$ to a normal order reduction of t, where RED is a normal order reduction sequence of s to WHNF, and $overlineRED$ the inverted sequence. Now the measure is the multiset consisting of the following numbers: for every (iR,cpx)-reduction, the number of normal-order reductions to the left of it. This is a well-founded order, and it is easy to see that the transformations strictly reduce this measure in every step using the forking diagrams. □

14 Correctness of (cpcx)-Reductions

The correctness of the reductions (cpcx) may be helpful to show correctness of more copy-operations, partial evaluation, in particular case-reductions which are embedded in a deeper `letrec` environment.

$$\begin{array}{ll}
\text{(cpcx-in)} & (\texttt{letrec}\ x = c\ t_1 \dots t_m, Env\ \texttt{in}\ C[x]) \\
 & \to (\texttt{letrec}\ x = c\ y_1\ \dots\ y_m, y_1 = t_1, \dots, y_m = t_m, Env\ \texttt{in}\ C[c\ y_1\ \dots\ y_m]) \\
\text{(cpcx-e)} & (\texttt{letrec}\ x = c\ t_1\ \dots\ t_m, z = C[x], Env\ \texttt{in}\ t) \\
 & \to (\texttt{letrec}\ x = c\ y_1\ \dots\ y_m, \\
 & \quad y_1 = t_1, \dots, y_m = t_m, z = C[c\ y_1\ \dots\ y_m], Env\ \texttt{in}\ t)
\end{array}$$

The union of (cpcx-in) and (cpcx-e) is denoted as (cpcx).

14.1 Correctness of (xch)-Reductions

Definition 14.1. *We need a reduction rule that is required for further proofs:*

$$\textit{(xch)}\ (\texttt{letrec}\ x = t, y = x, Env\ \texttt{in}\ r) \ \to\ (\texttt{letrec}\ y = t, x = y, Env\ \texttt{in}\ r)$$

Lemma 14.2. *The (xch)-reduction commutes with normal-order reductions. I.e.*

$$\xrightarrow{xch} \cdot \xrightarrow{n,a,P} \quad \rightsquigarrow \quad \xrightarrow{n,a,P} \cdot \xrightarrow{xch}$$

This is also true for the restricted reduction $\xrightarrow{iR,xch}$.

Proof. It is easy to verify that this holds for the different kinds of reductions. Only for (case) and a specific type of interference we show the concrete transformation:

$$\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t, y = x\ \texttt{in}\ \texttt{case}\ x\ ((c\ u) \to r)) \\
\xrightarrow{xch} & (\texttt{letrec}\ y = c\ t, x = y\ \texttt{in}\ \texttt{case}\ x\ ((c\ u) \to r)) \\
\xrightarrow{n,case} & (\texttt{letrec}\ y = c\ z, z = t, x = y\ \texttt{in}\ (\texttt{letrec}\ u = z\ \texttt{in}\ r)) \\
\hline
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ z, z = t, y = x\ \texttt{in}\ (\texttt{letrec}\ u = z\ \texttt{in}\ r)) \\
\xrightarrow{xch} & (\texttt{letrec}\ y = c\ z, z = t, x = y\ \texttt{in}\ (\texttt{letrec}\ u = z\ \texttt{in}\ r))
\end{array}$$

□

Lemma 14.3. *The (xch)-reduction has trivial forking diagrams with normal order reductions. I.e.*

$$\xleftarrow{n,a,P} \ \cdot \ RRAPxch \rightsquigarrow \ \xrightarrow{xch} \ \cdot \ \xleftarrow{n,a,P}$$

This is also true for the restricted reduction $\xrightarrow{iR,xch}$.

14.2 Properties of (cpcx)

We conjecture that there are no infinite reduction sequences consisting only of (cpcx) reductions.

Lemma 14.4. *A complete set of commuting diagrams for* $\xrightarrow{iR,cpcx}$ *is:*

$$\xrightarrow{(iR,cpcx)} \cdot \xrightarrow{(n,a,P)} \leadsto \xrightarrow{(n,a,P)} \cdot \xrightarrow{(iR,cpcx)}$$

$$\xrightarrow{(iR,cpcx)} \cdot \xrightarrow{(n,cp)} \leadsto \xrightarrow{(n,cp)} \cdot \xrightarrow{(iR,cpcx)} \cdot \xrightarrow{(iR,cpcx)} \cdot \xrightarrow{(iR,cpx)^*} \cdot \xrightarrow{(iR,gc)^*}$$

$$\xrightarrow{(iR,cpcx)} \cdot \xrightarrow{(n,a)} \leadsto \xrightarrow{(n,a)}$$

$$\xrightarrow{(iR,cpcx)} \cdot \xrightarrow{(n,case)} \leadsto \xrightarrow{(n,case)} \cdot \xrightarrow{(iR,cpcx)} \cdot \xrightarrow{(iR,cpx)^*} \cdot \xrightarrow{(iR,xch)^*}$$

Proof. Instead of a complete proof, we only show the typical cases:

$$\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t, y = \lambda u.C[x]\ \texttt{in}\ y) \\
\xrightarrow{iR,cpcx} & (\texttt{letrec}\ x = c\ z, z = t, y = \lambda u.C[c\ z]\ \texttt{in}\ y) \\
\xrightarrow{n,cp} & (\texttt{letrec}\ x = c\ z, z = t, y = \lambda u.C[c\ z]\ \texttt{in}\ \lambda u'.C'[c\ z]) \\ \hline
\xrightarrow{n,cp} & (\texttt{letrec}\ x = c\ t, y = \lambda u.C[x]\ \texttt{in}\ \lambda u'.C'[x]) \\
\xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ z, z = t, y = \lambda u.C[c\ z]\ \texttt{in}\ \lambda u'.C'[x]) \\
\xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ z', z' = z, z = t, y = \lambda u.C[c\ z]\ \texttt{in}\ \lambda u'.C'[c\ z']) \\
\xrightarrow{cpx} & (\texttt{letrec}\ x = c\ z', z' = z, z = t, y = \lambda u.C[c\ z]\ \texttt{in}\ \lambda u.C[c\ z]) \\
\xrightarrow{cpx} & (\texttt{letrec}\ x = c\ z, z' = z, z = t, y = \lambda u.C[c\ z]\ \texttt{in}\ \lambda u.C[c\ z]) \\
\xrightarrow{gc} & (\texttt{letrec}\ x = c\ z, z = t, y = \lambda u.C[c\ z]\ \texttt{in}\ \lambda u.C[c\ z])
\end{array}$$

$$\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t\ \texttt{in}\ \texttt{case}\ x\ (c\ y \to s)) \\
\xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ z, z = t\ \texttt{in}\ \texttt{case}\ (c\ z)\ ((c\ y) \to s)) \\
\xrightarrow{cn,case} & (\texttt{letrec}\ x = c\ z, z = t\ \texttt{in}\ (\texttt{letrec}\ y = z\ \texttt{in}\ s)) \\ \hline
\xrightarrow{cn,case} & (\texttt{letrec}\ x = c\ z, z = t\ \texttt{in}\ (\texttt{letrec}\ y = z\ \texttt{in}\ s))
\end{array}$$

In the following example we use a multi-context $C[.,.]$ that may have different holes, every hole is mentioned as an argument.

$$\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t\ \texttt{in}\ C[\texttt{case}\ x\ (c\ y \to s), x]) \\
\xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ z, z = t\ \texttt{in}\ C[\texttt{case}\ x\ (c\ y \to s), c\ z]) \\
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ z', z' = z, z = t\ \texttt{in}\ C[(\texttt{letrec}\ y = z'\ \texttt{in}\ s), c\ z]) \\ \hline
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ z', z' = t\ \texttt{in}\ C[(\texttt{letrec}\ y = z'\ \texttt{in}\ s), x]) \\
\xrightarrow{cpcx} & (\texttt{letrec}\ x = c\ z, z = z', z' = t\ \texttt{in}\ C[(\texttt{letrec}\ y = z'\ \texttt{in}\ s), c\ z]) \\
\xrightarrow{cpx} & (\texttt{letrec}\ x = c\ z', z = z', z' = t\ \texttt{in}\ C[(\texttt{letrec}\ y = z'\ \texttt{in}\ s), c\ z]) \\
\xrightarrow{xch} & (\texttt{letrec}\ x = c\ z', z' = z, z = t\ \texttt{in}\ C[(\texttt{letrec}\ y = z'\ \texttt{in}\ s), c\ z])
\end{array}$$

□

Lemma 14.5. *A complete set of forking diagrams for* $\xrightarrow{iR,cpcx}$ *is:*

$$\xleftarrow{n,a,P} \cdot \xrightarrow{iR,cpcx} \rightsquigarrow \xrightarrow{iR,cpcx} \cdot \xleftarrow{n,a,P}$$
$$\xleftarrow{n,cp} \cdot \xrightarrow{iR,cpcx} \rightsquigarrow \xrightarrow{iR,cpcx} \cdot \xrightarrow{iR,cpcx} \cdot \xrightarrow{(iR,cpx)^*} \cdot \xrightarrow{(iR,gc)^*} \cdot \xleftarrow{n,cp}$$
$$\xleftarrow{n,a} \cdot \xrightarrow{iR,cpcx} \rightsquigarrow \xleftarrow{n,a}$$
$$\xleftarrow{n,case} \cdot \xrightarrow{iR,cpcx} \rightsquigarrow \xrightarrow{iR,cpcx} \cdot \xrightarrow{(iR,cpx)^*} \cdot \xrightarrow{(iR,xch)^*} \cdot \xleftarrow{n,case}$$

Lemma 14.6. *If* $s \xrightarrow{iR,cpcx} t$*, then* s *is a WHNF iff* t *is a WHNF.*

Proposition 14.7. *The reduction (cpcx) is a correct program transformation.*

Proof. The non-standard part of the proof is the termination part.

First consider the transformation of $s \xrightarrow{iR,cpcx} t \cdot RED$ to a normal order reduction to WHNF of s. The used Lemmas are 14.4, 14.2, 13.1, 12.2. Intermediate steps have a sequence of normal-order reductions mixed with (iR,cpcx)-, (iR,cpx), (iR,xch) and (iR,gc)- reductions. We measure the sequences by the multiset consisting of the following triples of numbers:
for every iR-reduction: (n_1, n_2, n_3), where

1. n_1 is the number of normal-order (case)- or (cp)-reductions to the right of it,
2. n_2 is the number of (iR,gc)-reduction to the right of it before the next (n,cp) or (n, case)-reduction.
3. n_3 is the number of other normal-order reductions to the right of it.

The triples are compared lexicographically. This is a well-founded order on multisets. The commuting diagrams show that the transformations corresponding to (iR,cpcx), (iR,cpx), and (iR,xch) strictly reduce this multiset-measure in every step, if always the rightmost iR-reduction before a WHNF is transformed. For (iR,gc) we assume that we shift the rightmost (iR,gc)-reduction until it vanishes or is shifted over a (n,case) or (n,cp)-reduction. Hence it strictly reduces the multiset-measure.

The other case is the transformation of $\overline{RED} \cdot s \xrightarrow{iR,cpcx} t$ to a normal order reduction of t. We measure the sequences by the multiset consisting of the following triples of numbers: for every iR-reduction: (n_1, n_2, n_3), where

1. n_1 is the number of normal-order (case)- or (cp)-reductions to the left of it,
2. n_2 is the number of (iR,gc)-reduction to the left of it before the next (n,cp) or (n, case)-reduction.
3. n_3 is the number of other normal-order reductions to the left of it.

The triples are compared lexicographically.

This is a well-founded order, and it is easy to see that the transformations strictly reduce this measure in every step using the forking diagrams for the reductions (iR,cpcx), (iR,cpx), and (iR,xch), if always the leftmost i-reduction is shifted to the left. For the transformation involving (iR,gc) we use the same strategy as for the other part of the proof: We only shift them to the left, if there is no other iR-reduction between the (iR,gc) and the final WHNF. Then we see that the measure also in this case is strictly decreased. □

15 Correctness of (case)-Reductions

Proposition 15.1. *The reductions (case-in) and (case-e) are correct program transformations.*

Proof. Proposition 8.5 shows that (case-c) and (case-lam) are correct program transformations, from Lemmas 14.7, 13.4, 12.5, and 11.7, we obtain that (cpcx), (cpx), and (gc) are correct program transformations. We show by induction that a (case-e) and (case-in)-reduction is correct by using the correctness of the reductions (cpt), (cpcx), (case-c), (cpx), and (gc). The induction is on the length of the variable chain in the (case-in) (or (case-e), respectively). We give the proof only for (case-in), the other is a copy of this proof.
The base case is:

$$\begin{aligned}
& (\texttt{letrec}\ x = c\ t, \mathit{Env}\ \texttt{in}\ C[\texttt{case}\ x\ (c\ z \to s)\ \mathit{alts}]) \\
\xrightarrow{cpcx}\ & (\texttt{letrec}\ x = c\ y, y = t, \mathit{Env}\ \texttt{in}\ C[\texttt{case}\ (c\ y)\ (c\ z \to s)\ \mathit{alts}]) \\
\xrightarrow{\text{case-c}}\ & (\texttt{letrec}\ x = c\ y, y = t, \mathit{Env}\ \texttt{in}\ C[(\texttt{letrec}\ z = y\ \texttt{in}\ s)])
\end{aligned}$$

The result after a (case-in) is:

$$\xrightarrow{case-in}\ (\texttt{letrec}\ x = c\ y, y = t, \mathit{Env}\ \texttt{in}\ C[(\texttt{letrec}\ z = y\ \texttt{in}\ s)])$$

We show the induction for a short variable chain:

$$\begin{aligned}
& (\texttt{letrec}\ x_1 = c\ t, x_2 = x_1, \mathit{Env}\ \texttt{in}\ C[\texttt{case}\ x_1\ (c\ z \to s)\ \mathit{alts}]) \\
\xrightarrow{cpcx}\ & (\texttt{letrec}\ x_1 = c\ y, y = t, x_2 = c\ y, \mathit{Env}\ \texttt{in}\ C[\texttt{case}\ x_n\ (c\ z \to s)\ \ \mathit{alts}]) \\
\xrightarrow{case-in}\ & (\texttt{letrec}\ x_1 = c\ y, y = t, x_2 = c\ y_2, y_2 = y, \mathit{Env}\ \texttt{in}\ C[(\texttt{letrec}\ z = y_2\ \texttt{in}\ s)]) \\
\xrightarrow{cpx,cpx,gc}\ & (\texttt{letrec}\ x_1 = c\ y, y = t, x_2 = c\ y, \mathit{Env}\ \texttt{in}\ C[(\texttt{letrec}\ z = y\ \texttt{in}\ s)])
\end{aligned}$$

The other case is:

$$\begin{aligned}
\xrightarrow{case-in}\ & (\texttt{letrec}\ x_1 = c\ y, y = t, x_2 = x_1, \mathit{Env}\ \texttt{in}\ C[(\texttt{letrec}\ z = y\ \texttt{in}\ s)]) \\
\xrightarrow{cpcx}\ & (\texttt{letrec}\ x_1 = c\ y', y' = y, y = t, x_2 = c\ y', \mathit{Env}\ \texttt{in}\ C[(\texttt{letrec}\ z = y\ \texttt{in}\ s)]) \\
\xrightarrow{cpx,cpx,gc}\ & (\texttt{letrec}\ x_1 = c\ y, y = t, x_2 = c\ y, \mathit{Env}\ \texttt{in}\ C[(\texttt{letrec}\ z = y\ \texttt{in}\ s)])
\end{aligned}$$

□

Proposition 15.2. *The reduction (case) is a correct program transformation.*

Proof. Follows from Proposition 15.1 and 8.5. □

16 Correctness of FUNDIO-Reductions: Summary

Theorem 16.1. *All reductions of FUNDIO with the exception of (IOr) are correct program transformations in FUNDIO*

Proof. This follows from Propositions 8.5, 10.7, 11.7 and 15.2. □

It is clear that (IOr) is not a correct program transformation, since this would mean to allow IO-operations at compile-time:

Proposition 16.2. *If there are at least two different constants, then (IOr) is not correct as a program transformation.*

Proof. Assume that (IOr) is a correct program transformation. Then let a_1, a_2 be two different constants. The assumption that (IOr) is a correct program transformation implies that for a constant b the term `(IO b)` can be transformed to a_1 as well as a_2, hence this implies $a_1 \sim_c a_2$. However, the context (`case` $[\cdot](a_1 \to \bot)$ $(a_2 \to a_2)$ *alts*) for a non-terminating expression $\bot$ distinguishes these two constants, which is a contradiction. □

17 Correctness of ucp-Reductions

In this section we show that a binding `letrec` $x = s, Env$ `in` t, where x occurs at most once in Env or t, and this occurrence is at a surface position, can be resolved by copying s to this occurrence. We define the reduction such that the corresponding binding is eliminated after copying.

(ucp)
- `letrec` $x = s, Env$ `in` $S[x] \to$ `letrec` Env `in` $S[s]$
  if x has only an occurrence in $Env, S[x]$ and no occurrence in s.
- `letrec` $x = s, Env, y = S[x]$ `in` $t \to$ `letrec` $Env, y = S[s]$ `in` t
  if x has only an occurrence in $Env, S[x], t$ and no occurrence in s.

Note that this may generate `letrec`s with empty environment. Note also that copying into a non-surface position would be incorrect (see Example 17.6).

Lemma 17.1. *The reduction (ucp) cannot be applied infinitely often.*

Lemma 17.2. *A complete set of commuting diagrams for $\xrightarrow{iR,ucp}$ is:*

$$\xrightarrow{(iR,ucp)} \cdot \xrightarrow{(n,a,P)} \rightsquigarrow \xrightarrow{(n,a,P)} \cdot \xrightarrow{(iR,ucp)}$$

$$\xrightarrow{(iR,ucp)} \cdot \xrightarrow{(n,a)} \rightsquigarrow \xrightarrow{(n,a)}$$

$$\xrightarrow{(iR,ucp)} \cdot \xrightarrow{(n,lll)^+} \rightsquigarrow \xrightarrow{(n,lll)^+} \cdot \xrightarrow{(iR,ucp)}$$

$$\xrightarrow{(iR,ucp)} \cdot \xrightarrow{(n,case)} \rightsquigarrow \xrightarrow{(n,case)} \cdot \xrightarrow{(iR,gc)} \cdot \xrightarrow{(iR,ucp)}$$

$$\xrightarrow{(iR,ucp)} \rightsquigarrow \xrightarrow{(n,cp)} \cdot \xrightarrow{(iR,gc)}$$

Proof. We show the typical overlappings.

$$\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t, \mathit{Env}\ \texttt{in}\ \texttt{case}\ x\ (c\ z \to a)) \\
\xrightarrow{\mathit{ucp}} & (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ \texttt{case}\ (c\ t)\ (c\ z \to a)) \\
\xrightarrow{n,\mathit{case}} & (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{letrec}\ z = t\ \texttt{in}\ a)) \\ \hline
\xrightarrow{n,\mathit{case}} & (\texttt{letrec}\ x = c\ y, y = t, \mathit{Env}\ \texttt{in}\ (\texttt{letrec}\ z = y\ \texttt{in}\ a)) \\
\xrightarrow{\mathit{gc}} & (\texttt{letrec}\ y = t, \mathit{Env}\ \texttt{in}\ (\texttt{letrec}\ z = y\ \texttt{in}\ a)) \\
\xrightarrow{\mathit{ucp}} & (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{letrec}\ z = t\ \texttt{in}\ a))
\end{array}$$

$$\begin{array}{ll}
 & (\texttt{letrec}\ x = (\texttt{letrec}\ y = t\ \texttt{in}\ s), \mathit{Env}\ \texttt{in}\ (x\ a)) \\
\xrightarrow{\mathit{ucp}} & (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ ((\texttt{letrec}\ y = t\ \texttt{in}\ s)\ a)) \\
\xrightarrow{n,\mathit{lapp}} & (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{letrec}\ y = t\ \texttt{in}\ (s\ a))) \\
\xrightarrow{n,\mathit{llet}} & (\texttt{letrec}\ \mathit{Env}, y = t\ \texttt{in}\ (s\ a)) \\ \hline
\xrightarrow{n,\mathit{llet}} & (\texttt{letrec}\ x = s, y = t, \mathit{Env}\ \texttt{in}\ (x\ a)) \\
\xrightarrow{\mathit{ucp}} & (\texttt{letrec}\ y = t, \mathit{Env}\ \texttt{in}\ (s\ a))
\end{array}$$

$$\begin{array}{ll}
 & (\texttt{letrec}\ x = (\texttt{letrec}\ y = t_y\ \texttt{in}\ t_x), z = R'[x], \mathit{Env}\ \texttt{in}\ R[z]) \\
\xrightarrow{\mathit{ucp}} & (\texttt{letrec}\ z = R'[(\texttt{letrec}\ y = t_y\ \texttt{in}\ t_x)], \mathit{Env}\ \texttt{in}\ R[z]) \\
\xrightarrow{n,\mathit{llet},+} & (\texttt{letrec}\ z = R'[t_x], y = t_y, \mathit{Env}\ \texttt{in}\ R[z]) \\ \hline
\xrightarrow{n,\mathit{llet},+} & (\texttt{letrec}\ x = t_x, y = t_y, z = R'[x], \mathit{Env}\ \texttt{in}\ R[z]) \\
\xrightarrow{\mathit{ucp}} & (\texttt{letrec}\ y = t_y, z = R'[t_x], \mathit{Env}\ \texttt{in}\ R[z])
\end{array}$$

$$\begin{array}{ll}
 & (\texttt{letrec}\ x = s, \mathit{Env}\ \texttt{in}\ (x\ a)) \\
\xrightarrow{\mathit{ucp}} & (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (s\ a)) \\ \hline
\xrightarrow{n,\mathit{cp}} & (\texttt{letrec}\ x = s, \mathit{Env}\ \texttt{in}\ (s\ a)) \\
\xrightarrow{\mathit{gc}} & (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (s\ a))
\end{array}$$

□

Lemma 17.3. *A complete set of forking diagrams for* $\xrightarrow{iR,\mathit{ucp}}$ *is:*

$$\begin{array}{lcl}
\xleftarrow{n,a,P} \cdot \xrightarrow{iR,\mathit{ucp}} & \leadsto & \xrightarrow{iR,\mathit{ucp}} \cdot \xleftarrow{n,a,P} \\
\xleftarrow{n,a} \cdot \xrightarrow{iR,\mathit{ucp}} & \leadsto & \xleftarrow{n,a} \\
\xleftarrow{(n,\mathit{lll})^+} \cdot \xrightarrow{iR,\mathit{ucp}} & \leadsto & \xrightarrow{iR,\mathit{ucp}} \cdot \xleftarrow{(n,\mathit{lll})^+} \\
\xleftarrow{(n,\mathit{case})} \cdot \xrightarrow{iR,\mathit{ucp}} & \leadsto & \xrightarrow{(iR,\mathit{gc})} \cdot \xrightarrow{iR,\mathit{ucp}} \cdot \xleftarrow{(n,\mathit{case})} \\
\xleftarrow{n,\mathit{cp}} \cdot \xrightarrow{iR,\mathit{ucp}} & \leadsto & \xrightarrow{iR,\mathit{gc}}
\end{array}$$

Lemma 17.4. *Let $s \xrightarrow{iR,ucp} t$. Then s is a WHNF iff t is a WHNF.*

Proof. If s is a WHNF, then clearly t is a WHNF. For the other direction, note that (ucp) does not remove the `letrec`, but may leave an empty environment, hence t can only be a WHNF, if s is already a WHNF. □

Proposition 17.5. *The reduction (ucp) is a correct program transformation.*

Proof. Let $s \xrightarrow{ucp} t$. It is sufficient to show that the transformations of the reductions $s \xrightarrow{ucp} t\ RED$ and $\overline{RED}\ s \xrightarrow{ucp} t$ to normal order reductions of s and t, respectively, terminate.
We show this for the transformation of $s \xrightarrow{ucp} t\ RED$: The commuting diagrams in Lemma 17.2 show that there is at most one $\xrightarrow{iR,ucp}$-reduction in intermediate reductions. In every transformation step based on $\xrightarrow{iR,ucp}$, the number of normal order reductions on the right strictly decreases. After this, there may be some remaining $\xrightarrow{iR,gc}$-reductions, which can also be shifted to the right (see Lemma 12.2), starting with the rightmost one before a WHNF is reached in the reduction. The same proof is possible for the transformation of the reduction $\overline{RED}\ s \xrightarrow{iR,ucp} t$, where right and left have to be interchanged using Lemmas 17.3 and 12.3 □

Example 17.6. The reduction (ucp) would be not correct, if it were possible to copy into an abstraction: We assume that there is an operation `xor`, from which we only require that it is strict in both arguments, and that `IO` output Booleans and gets Booleans.

$P_1 :=$ `letrec` $x =$ (`IO True`), $y = (\backslash z.x)$ `in` (`IO` (`xor` (y `True`) (y `True`)))
$P_2 :=$ `letrec` $y = (\backslash z.$(`IO True`)) `in` (`IO` (`xor` (y `True`) (y `True`)))

Obviously, we have $P_1 \to P_2$ by a unique copy reduction to a non-surface position, replacing x by (`IO True`).
The programs P_1 and P_2 are contextually different: P_1 has only terminating normal-order reductions for IO-multisets of the form $\{(\texttt{True}, a), (\texttt{False}, b)\}$, whereas P_2 has also normal-order reductions for IO-multisets of the form $\{(\texttt{True}, a), (\texttt{True}, b), (\texttt{True}, c)\}$. The reason is that the abstraction $\backslash z.$(`IO True`) is copied by reduction, and thus the `IO`-expression is duplicated.

18 Further Program Transformations

18.1 Correctness of Lambda-Lifting

Lambda-lifting is the operation used in a compiler that abstracts free variables:

$$C[s[z]] \to C[((\lambda\ x\ .\ s[x])\ z)]$$

In order to show that this is a correct transformation, we have to show that it can be undone by correct program transformations.

$$C[(\lambda\ x\ .\ s[x])\ z] \xrightarrow{lbeta} C[(\texttt{letrec}\ x = z\ \texttt{in}\ s[x])] \xrightarrow{(cpx)} C[(\texttt{letrec}\ x = z\ \texttt{in}\ s[z])] \xrightarrow{(gc,*)} C[s[z]]$$

Proposition 18.1. *Lambda-Lifting is a correct program transformation.*

Proof. Follows from propositions 8.5, 13.4 and 12.5. □

18.2 Correctness of a Beta with Variable Arguments

This rule can be used if the arguments of an abstraction are variables. This rule is used in the STG-machine of Haskell, and can be proved to be correct in FUNDIO:

$$\text{(betavar)}\ \ C[((\lambda x.s)\ y)] \to C[s[y/x]]$$

Proposition 18.2. *(betavar) is a correct program transformation.*

Proof. The reduction can be simulated by correct program transformations:

$$\begin{array}{l} C[((\lambda x.s)\ y)] \xrightarrow{lbeta} C[(\texttt{letrec}\ x = y\ \texttt{in}\ s)] \\ \xrightarrow{cpx^*} C[(\texttt{letrec}\ x = y\ \texttt{in}\ s[y/x])] \xrightarrow{gc} C[s[y/x]] \end{array}$$

See propositions 8.5, 13.4 and 12.5. □

19 Modifying the Normal Order Reduction

There are mainly the following modifications and extensions of the normal order reduction: Intermediate garbage collection, reduction of indirections and exploiting strictness information. Basically, the evaluation by an abstract machine is also a kind of modification of the normal order reduction.

19.1 Garbage Collection and Removing Indirections

First we show that intermediate garbage collection and compressing indirections does not provide any problems for evaluation. It is only necessary to treat (gc) and (cpx) that are performed in surface contexts. The rationale for the restriction is that garbage collections within abstractions can be done at compile time. This is possible, since (gc) as well as (cpx) are correct as program transformations.
So we are left with garbage collections and (cpx)-reductions during run-time, i.e., during reduction to WHNF. The normal order reductions of the calculus FUNDIO never introduce (gc)-redexes or (cpx)-redexes within an abstraction, which can be checked by looking at the rules and the definition of normal order reduction.

However, normal order reduction may generate lots of possibilities to reduce redundant bindings by (gc) and (cpx), which means to free space in an implementation of FUNDIO.
The reduction (cpx) may cause infinite reductions, hence we have to restrict the application to obtain a useful optimization during evaluation. It is not useful to do this reduction as a macro step like $\texttt{letrec}\ x = y, Env[x]\ \texttt{in}\ C[x] \rightarrow \texttt{letrec}\ Env[y]\ \texttt{in}\ C[y]$, since this is unrealistic for an implementation for it can only be achieved by checking all references.
We define the reduction (cpxTrm) consisting of the reductions:

(cpxTrmX) $\texttt{letrec}\ x = y, Env\ \texttt{in}\ t \rightarrow \texttt{letrec}\ x = x, Env\ \texttt{in}\ t$
if $x \neq y$ and there is a cyclic chain of bindings in Env:
$x = y, y = y_1, y_1 = y_2, \ldots, y_k = x$
(cpxTrmC) $\texttt{letrec}\ x = y, z = C[x], Env\ \texttt{in}\ t \rightarrow \texttt{letrec}\ x = y, z = C[y], Env\ \texttt{in}\ t$
if $x \neq y$ and if (cpxTrmX) is not applicable
$\texttt{letrec}\ x = y, E\ \texttt{in}\ C[x] \rightarrow \texttt{letrec}\ x = y, E\ \texttt{in}\ C[y]$
if $x \neq y$ and if (cpxTrmX) is not applicable

The reduction (cpxTrm) is defined as the union of (cpxTrmX) and (cpxTrmC).

Lemma 19.1. *A $\xrightarrow{(iR,cpxTrmC)}$-reduction is also a $\xrightarrow{(iR,cpx)}$-reduction.
A $\xrightarrow{(iR,cpxTrmX)}$-reduction is a sequence of $\xrightarrow{(iR,cpx)}$-reductions. Hence the reductions are correct as program transformations.*

Lemma 19.2. *A reduction sequence of only $\xrightarrow{(cpxTrm)}$-reductions consists of first applying $\xrightarrow{cpxTrmX,*}$ and then $\xrightarrow{cpxTrmC,*}$.*

Proof. (cpxTrmC) can only be applied, if (cpxTrmX) is not applicable, hence we have only to argue that (cpxTrmC) does not enable a (cpxTrmX)-reduction. This holds, since if there is a cyclic chain after a (cpxTrmC)-reduction, then there is also one before the reduction. □

Lemma 19.3. *There are no infinite reduction sequences consisting only of $\xrightarrow{cpxTrm}$-reductions.*

Proof. Lemma 19.2 shows that it is sufficient to show termination of the reductions (cpxTrmC) and (cpxTrmX). The reduction (cpxTrmX) strictly reduces the number of variable occurrences that contribute to cyclic binding chains of more than one variable. The rule (cpxTrmC) is only applicable, if the `letrec`s in reduction contexts have no cyclic chains of more than one variable. We can construct a well-founded measure as follows: The variables that occur as left hand sides in `letrec`s in reduction contexts are ordered as follows: $x > y$, if $x \neq y$ and $\{x = y\}$ is a binding in t. The transitive closure of this partial ordering is also denoted as $>$. The transitive closure operation generates a strict partial ordering, since there are no cycles. The measure of t is a multiset M of variables, where every variable occurs as often in M as there are occurrences of this variable in t. The multisets are ordered by the induced multiset-ordering, which is also a well-founded, strict partial order. Every step of (cpxTrmC) replaces a variable occurrence by a smaller one, hence the resulting term is smaller in the multiset-ordering. Thus the reduction (cpxTrmC) and hence (cpxTrm) terminates. □

19.2 Evaluation Strategies

Definition 19.4. *An* evaluation strategy *S is a (possibly non-deterministic) algorithm which for every term t determines the next reduction to apply. These may be reductions from FUNDIO, perhaps requiring an IO-pair, or also some other reductions like (cpx, gc). We assume in this paper that the normal forms for strategies are WHNFs.*
We say a term t terminates w.r.t. S and IO-multiset P, if there is a terminating S-reduction to a WHNF requiring P as IO-multiset. This is denoted as $t\Downarrow_S(P)$.
We say a term t terminates w.r.t. S and the finite IO-sequence $\overrightarrow{P}$, iff there is a terminating S-reduction to a WHNF requiring $\overrightarrow{P}$ as IO-sequence. This is denoted as $t\downarrow_S(\overrightarrow{P})$.

An IO-sequence $\overrightarrow{P}$ is S-valid for t, iff there is an S-reduction $t \xrightarrow{S,\overrightarrow{P}} t'$ for some t'.

For a term t and a finite IO-sequence $\overrightarrow{P}$, we write $t\downarrow_S(\overrightarrow{P})$ iff $t \xrightarrow{S,\overrightarrow{P}} t'$ and t' is a WHNF.
For a closed term t and a finite IO-sequence $\overrightarrow{P}$, we write $t\Downarrow_S(\overrightarrow{P})$ iff the following holds: if $\overrightarrow{P}$ is S-valid for t then $t \xrightarrow{S,\overrightarrow{P}} t'$ where $t' \neq \bot$.
If $\forall \overrightarrow{P} : t\Downarrow_S(\overrightarrow{P})$, then we say t has no bot-S-reduction. Otherwise, we write $t\Uparrow_S$.

We say the strategy S is correct*, iff for all closed expressions t: for all finite IO-multisets P: $t\Downarrow(P) \Leftrightarrow t\Downarrow_S(P)$ and $t\Uparrow \Leftrightarrow t\Uparrow_S$.*

The second condition can also be reformulated as $(\forall \overrightarrow{P} : t\Downarrow(\overrightarrow{P})) \Leftrightarrow (\forall \overrightarrow{P} : t\Downarrow_S(\overrightarrow{P}))$.

Note that it is not sufficient to define an evaluation strategy as consisting of correct reductions, since there may be terms that have a WHNF for a certain IO-multiset P (IO-sequence $\overrightarrow{P}$), but the strategy does not terminate for t and P.

Lemma 19.5. *Let S be an evaluation strategy. Let the following hold:*
$\forall \overrightarrow{P} : t\Downarrow(\overrightarrow{P}) \Leftrightarrow t\Downarrow_S(\overrightarrow{P})$.
Then S is correct.

Proof. The condition that for all finite IO-multisets P: $t\Downarrow(P) \Leftrightarrow t\Downarrow_S(P)$ follows immediately from the condition. The equivalence of error-freeness follows, since $(\forall \overrightarrow{P} : t\Downarrow(\overrightarrow{P})) \Leftrightarrow (\forall \overrightarrow{P} : t\Downarrow_S(\overrightarrow{P}))$ can be derived from the assumption. □
For proofs about correctness, the following lemma provides a sufficient condition:

Lemma 19.6. *Let S be an evaluation strategy. Let the following hold:*

$$\begin{array}{rl}\forall \overrightarrow{P}: & t{\downarrow}(\overrightarrow{P}) \Leftrightarrow t{\downarrow_S}(\overrightarrow{P}) \\ & \wedge\ \overrightarrow{P} \textit{ is valid for } t \Leftrightarrow \overrightarrow{P} \textit{ is } S\textit{-valid for } t\end{array}$$

Then S is correct.

Proof. Let t be a term, and $\overrightarrow{P}$ be an IO-sequence. Since the normal-order and the strategy S are equivalent w.r.t. validity, we check the cases where $\overrightarrow{P}$ is valid w.r.t. both strategies.
The precondition implies that $t{\downarrow}(\overrightarrow{P})$ is equivalent with $t{\downarrow_S}(\overrightarrow{P'})$. □

Of course, the following should hold for strategies:

Proposition 19.7. *Let $s \sim_c t$, and let S be a correct evaluation strategy. Then for all IO-multisets P: $s{\Downarrow_S}(P) \Leftrightarrow t{\Downarrow_S}(P)$ and $s{\Uparrow_S} \Leftrightarrow t{\Uparrow_S}$.*

Proof. Follows from the definition of correct strategy and from the definition of $\sim_c$. □

19.3 Correctness of the strategy G using (gc) and (cpx)

Now we define a special strategy:

Definition 19.8. *We assume that there is a strategy for selecting after every reduction whether the next step is a garbage collection $\xrightarrow{iR,gc}$, a (cpx)-reduction $\xrightarrow{iR,cpxTrm}$ or a normal-order reduction. The selection may also be random. We call this reduction strategy nG-reduction.*

Note that the definition of a single normal order reduction is not changed.

Theorem 19.9. *Let t be a closed expression. The nG is correct as strategy.*

Proof. We use Lemma 19.6. We argue that the termination criterion holds. That equivalence of validity holds also is a by-product of the reasoning, since the proof below shows that there is a normal-order reduction w.r.t. $\overrightarrow{P}$ iff there is a S-reduction w.r.t. $\overrightarrow{P}$.
First let s have a nG-reduction to WHNF, where the reduction has IO-sequence $\overrightarrow{P}$. We use the commuting diagrams for (iR,gc) and (iR,cpx) (see Lemmas 19.1, 12.2 and 13.1) to transform this reduction sequence into a normal order reduction sequence. The termination of this transformation is easily shown as in the proofs of correctness of the corresponding reductions.

For the other part of the proof let $s{\downarrow}(\overrightarrow{P})$ by a reduction RED. Then the strategy nG has three possibilities:

1. selecting a normal order reduction $s \xrightarrow{n} s'$. In this case the same normal order reduction as for s is to be used, in particular if this is an (IOr).

2. selecting a (iR, gc)
3. selecting a $(iR, cpxTrm)$

The induction is on the number of normal-order reductions.
In case 1 induction can be used.

We treat the second and third case. Let $s \xrightarrow{(iR,gc)} s'$ or $s \xrightarrow{(iR,cpxTrm)} s'$. From Lemmas 12.3 and 13.2 it follows that there is a normal order reduction RED' of s' to a WHNF with the same IO-sequence and the number of normal-order reductions is not greater than in RED.

$$\begin{tikzcd}
s \arrow[r, "iR{,}gc"] \arrow[d, dashed] & s' \arrow[d, "n"', "\overrightarrow{P}"] \\
r_{WHNF} \arrow[r, dashed] & r'_{WHNF}
\end{tikzcd}$$

In order to use induction, we have to show that every combination of (iR, gc) and $(iR, cpxTrm)$ terminates. This follows using the same arguments as in the proof of Lemma 19.3 using a slightly extended measure: the number of occurrences of `letrec`s has to be added to the multiset. Using the observation that (gc) strictly reduces this measure, we obtain termination. Thus the strategy cannot select (iR,gc) or (iR,cpxTrm) infinitely often, and after a finite number of steps G must select a normal order reduction. Then we can use induction on the length of the normal order reduction sequence. □

20 Relativized Normal-Order Reduction

This section is intended as a preparation for analyzing the effect of modifying the normal order evaluation if it is known that a function is strict in its argument. I.e. if in an application $((\lambda x.s)\ t)$ it is known that the lambda-expression is strict, then the evaluation order can be changed such that first the argument t is evaluated and then the application.
In this section we formalize the notion of "evaluating a subexpression to WHNF" and show some properties of the corresponding relativized reduction. This is more complex than for the lambda calculus, since it has to take the `letrec`-structure into account.

Definition 20.1. *The context class LR of right-nested* `letrec`*s is defined by the syntax*

$$LR := [] \mid (\texttt{letrec}\ Env\ \texttt{in}\ LR)$$

It turns out that not every surface position is relevant in the treatment of strictness; in fact the positions in the alternatives of a `case` should be ruled out.

Definition 20.2. *We define* application surface contexts*, meaning that the hole is not in the body of an abstraction and not in an alternative of a case. Let A be* *this context class defined as follows:*

$$\begin{array}{rl} A ::= & [\cdot] \mid (A\ E) \mid (E\ A) \mid (\texttt{case}\ A\ \mathit{alts}) \mid (\texttt{IO}\ A) \\ & \mid (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ A) \mid (\texttt{letrec}\ \mathit{Env}_1, x = A\ \texttt{in}\ E) \end{array}$$

In the following we want to capture the intuition of evaluating a subterm of a term. This is done by labeling this subterm appropriately and inheriting the labeling after reduction steps. The evaluation of the subterm should be a variant of the normal order reduction adapted for subterms. This is different from the easy notion in lambda calculus, since in FUNDIO, the `letrec`-structure may enforce evaluation of terms in an environment.
We define evaluation labels and their treatment:

Definition 20.3. *Let t be a closed expression, and t_0 be a subexpression of t that is in an A-context, and which is intended to be evaluated first. The* evaluation label *is a non-negative integer.*
The labelling starts with the label 0 at subexpression t_0.
First the 0-label is shifted to the rightmost expression in a `letrec` *using the following rule as often as applicable.*

$$\textit{(down0)} \qquad (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s)^0 \rightarrow (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s^0)$$

Then the following rules propagate the evaluation label i.

$$\begin{array}{ll} (s_1\ s_2)^i & \rightarrow (s_1^{i+1}\ s_2)^i \\ (\texttt{case}\ s_1\ \mathit{alts})^i & \rightarrow (\texttt{case}\ s_1^{i+1}\ \mathit{alts})^i \\ (\texttt{IO}\ s_1)^i & \rightarrow (\texttt{IO}\ s_1^{i+1})^i \\ (\texttt{letrec}\ x = s_1, \mathit{Env}\ \texttt{in}\ A[x^i]) & \rightarrow (\texttt{letrec}\ x = s_1^{i+1}, \mathit{Env}\ \texttt{in}\ A[x^i]) \\ (\texttt{letrec}\ x_1 = s_1, x_2 = A[x_1^i], \mathit{Env}\ \texttt{in}\ s_2) & \rightarrow (\texttt{letrec}\ x_1 = s_1^{i+1}, x_2 = A[x_1^i], \mathit{Env}\ \texttt{in}\ s_2) \end{array}$$

The labeling stops with success, if the labelled term is a value, marking it with a maximal label.
The labeling fails, if a subexpression is labeled twice with different labels. This indicates a loop in the labeling, and means that there is no terminating evaluation of the subterm.

Now assume the labeling is successful, i.e. done completely.
We say a variable x with label i is bound to a value *in t, if labeling does not fail, and if in an outer* `letrec`*, there is a binding* $(\texttt{letrec}\ x = s^{i+1}, \mathit{Env}\ \texttt{in}\ C[x])$*, and either s is the value t, or s is a variable bound to the value t .*
The subterm s^0 of t is in relativized WHNF*, if labelling does not fail and s is of one of the following forms:*

- *a value, i.e. an abstraction or a constructor application.*
- *x, where x is bound in t to a constructor application*

Example 20.4. An example for the labeling is

$$\texttt{letrec}y = \lambda u.u \ \texttt{in} \ \texttt{letrec}x = ((\lambda z.z)^3 \ a)^2 \ \texttt{in} \ (x^1 \ b)^0$$

The term $(x \ b)$ is the term which had to be evaluated, the coresponding labelings areindicated as superscripts. The normal order evaluation would be different, since the first reduction is a (llet).
The term `letrec x = x in x^0` is a further example, where propagating the label first gives `letrec x = x^1 in x^0`, and then fails.

The initial shift of the labeling 0 means that relativized normal order reduction cannot trace evaluation of a subterm t_0 of the form $(\texttt{letrec} \ Env_2 \ \texttt{in} \ s)^0$ in a term $(\texttt{letrec} \ Env_1 \ \texttt{in} \ (\texttt{letrec} \ Env_2 \ \texttt{in} \ s)^0)$, since this is relabeled as $(\texttt{letrec} \ Env_1 \ \texttt{in} \ (\texttt{letrec} \ Env_2 \ \texttt{in} \ s^0))$, and hence only the evaluation of the rightmost non-`letrec`-term can be traced. This is not a severe restriction, but makes a proof of correctness easy possible for rearranging the normal order reduction sequences exploiting strictness.

Using the labeling, we define the relativized normal order reduction w.r.t. to a subexpression.

Definition 20.5. *Let t be a closed expression, and s be a subexpression that is in an A-context, and labelled initially with 0.*
If the labeling failed, then there is no relativized normal order reduction step w.r.t. s.
Otherwise, let j be the maximal number used by the labeling. There are several cases for the (rn)-reduction w.r.t s.

- *If s is in relativized WHNF, then there is no relativized normal order reduction.*
- *t is of the form $C[(s_1^j \ s_2)^{j-1}]$, and s_1 is an abstraction. Then apply rule (lbeta) to $(s_1 \ s_2)$.*
- *t is of the form $A[x^{j'}]$, and x is bound in t to an abstraction r^j. Then reduce the term as follows:*

  $$\textit{(cptrn)} \ A[x] \rightarrow A[r]$$

  The rule (cptrn) only copies into A-contexts.
- *t is of the form $C[(\texttt{case} \ (c \ t_1 \dots t_n)^j \ ((c \ y_1 \dots y_n) \rightarrow r) \ alts)]$.*
  Then apply (case-c) and obtain $C[(\texttt{letrec} \ y_1 = t_1, \dots, y_n = t_n \ \texttt{in} \ r)]$.
- *t is of the form $C[(\texttt{case} \ t'^j \ (\texttt{lambda} \rightarrow r) \ alts)]$, and t' is an abstraction.*
  Then apply (case-lam) and obtain $C[(\texttt{letrec} \ \{\} \ \texttt{in} \ r)]$.
- *t is of the form*

  $$\begin{array}{l} C[\texttt{letrec} \ u = (c \ t_1 \dots t_n)^j, Env \\ \quad \texttt{in} \ D[(\texttt{case} \ x^{j'} \ ((c \ y_1 \dots y_n) \rightarrow r) \ alts)]] \end{array}$$

  where D is an A-context, and `x` *is bound to the term $(c \ t_1 \dots t_n)^j$. Then use the following rule that results in:*

  $$\begin{array}{l} \textit{(casern)} \ C[\texttt{letrec} \ u = (c \ z_1 \dots z_n), z_1 = t_1, \dots, z_n = t_n \\ \qquad\qquad \texttt{in} \ D[(\texttt{letrec} \ y_1 = z_1, \dots, y_n = z_n \ \texttt{in} \ r)] \end{array}$$

- *t is of the form*

$$C[\texttt{letrec}\ x = (c\ t_1 \ldots t_n)^j, Env,\\ y = D[(\texttt{case}\ x^{j'}\ ((c\ y_1 \ldots y_n) \to r)\ alts)]\ \texttt{in}\ t']$$

  where D is an A-context, and `x` *is bound to the term $(c\ t_1 \ldots t_n)^j$. Then use the following rule that results in:*

$$\text{(casern)}\ C[\texttt{letrec}\ x = (c\ z_1 \ldots z_n), z_1 = t_1, \ldots, z_n = t_n,\\ y = D[(\texttt{letrec}\ y_1 = z_1, \ldots, y_n = z_n\ \texttt{in}\ r)]\ \texttt{in}\ t']$$

- *t is of the form $C[(\texttt{letrec}\ Env\ \texttt{in}\ r)^j]$, where C is not trivial and $j > 0$. Then apply a rule of (lll) and shift the* `letrec` *one level higher.*
- *t is of the form $C[(\texttt{IO}\ a^j)^{j-1}]$ then apply (IOr) and obtain one of the expressions $C[b]$, where b is a constant. This reduction has IO-Pair (a, b).*
- *t is of the form $C[(\texttt{IO}\ x^{j'})]$ and x is bound in t to a^j, then use the following rule*

$$\text{(IOrrn)}\ C[(\texttt{IO}\ x^j)] \to C[b] \qquad \text{where } b \text{ is a constant}$$

  This reduction has IO-Pair (a, b).

After every (rn)-reduction, it is assumed that the term s remains labelled 0, and then the other labels are computed anew.

Note that the rn-rules above are the (lll)-rules of calculus or the generalized rules (casern), (IOrrn), (cptrn), where the binding of the variable may act over several levels of `letrec`.
Note also that for rn-reductions, the normal-order reduction would prefer (lll)-reductions.
It is an easy exercise to show that the extended rules are correct program transformations. However, this is not an issue here, since we are interested in a modified normal order reduction at run-time.
In the following let ρ be a mapping for reductions as follows: $\rho(cptrn) = (cp), \rho(IOrrn) = (IOr), \rho(casern) = (case)$ and $\rho(a) = a$, otherwise.

Lemma 20.6. *Let t be a closed expression, and s be a subexpression that is in a reduction context. Then a normal order reduction leaves the successor of s according to definition 20.3 in a reduction context w.r.t. t.*

Definition 20.7. *We need a further reduction rule as a restriction of (cpx) to have nice commuting diagrams for the (rn)-reductions:*

$$\text{(cpxx)}\ (\texttt{letrec}\ x = y, z = x, Env\ \texttt{in}\ t)\\ \to (\texttt{letrec}\ x = y, z = y, Env\ \texttt{in}\ t)$$

Lemma 20.8. *The reduction (iA, cpxx) commutes with the n-reductions of the FUNDIO-calculus and the reductions (rn).*

In the following we use a modified notion of complete set of commuting diagrams. The idea is that there are the desired reductions and one disturbing reduction that has to be eliminated by shifting.

Lemma 20.9. *Let t be a closed expression, and s be a subexpression that is in an A-context, and labelled initially with 0.*
The following is a complete set of commuting diagrams for an rn-reduction w.r.t. s in an A-context.

$$\begin{array}{lcl}
\xrightarrow{(rn,a,P_1)} & \leadsto & \xrightarrow{(n,\rho(a),P_1)} \\
\xrightarrow{(rn,a_1,P_1)} \cdot \xrightarrow{(n,a_2,P_2)} & \leadsto & \xrightarrow{(n,a_2,P_2)} \cdot \xrightarrow{(rn,a_1,P_1)} \\
\xrightarrow{(rn,case)} \cdot \xrightarrow{(n,case)} & \leadsto & \xrightarrow{(n,case)} \cdot \xrightarrow{(rn,case)} \cdot \xrightarrow{(iA,cpxx)^*} \cdot \xrightarrow{(iA,xch)^*}
\end{array}$$

where P_1, P_2 are IO-pairs or empty.

Proof. We omit the standard calculations and demonstrate the typical nontrivial commutation for the (case)-over-(case) transformation.

$$\begin{array}{ll}
 & (\texttt{letrec}\ x = c\ t\ \texttt{in}\ C[(\texttt{case}\ x\ ((c\ y) \to r_1))^0, \texttt{case}\ x\ ((c\ z) \to r_2)]) \\
\xrightarrow{rn,case} & (\texttt{letrec}\ x = c\ u, u = t\ \texttt{in}\ C[(\texttt{letrec}\ y = u\ \texttt{in}\ r_1^0), \texttt{case}\ x\ ((c\ z) \to r_2)]) \\
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ v, u = t, v = u\ \texttt{in}\ C[(\texttt{letrec}\ y = u\ \texttt{in}\ r_1^0), (\texttt{letrec}\ z = v\ \texttt{in}\ r_2)]) \\
\hline
\xrightarrow{n,case} & (\texttt{letrec}\ x = c\ v, v = t\ \texttt{in}\ C[(\texttt{case}\ x\ ((c\ y) \to r_1))^0, (\texttt{letrec}\ z = v\ \texttt{in}\ r_2)]) \\
\xrightarrow{rn,case} & (\texttt{letrec}\ x = c\ u, v = t, u = v\ \texttt{in}\ C[(\texttt{letrec}\ y = u\ \texttt{in}\ r_1^0), (\texttt{letrec}\ z = v\ \texttt{in}\ r_2)]) \\
\xrightarrow{cpxx} & (\texttt{letrec}\ x = c\ v, v = t, u = v\ \texttt{in}\ C[(\texttt{letrec}\ y = u\ \texttt{in}\ r_1^0), (\texttt{letrec}\ z = v\ \texttt{in}\ r_2)]) \\
\xrightarrow{xch} & (\texttt{letrec}\ x = c\ v, u = t, v = u\ \texttt{in}\ C[(\texttt{letrec}\ y = u\ \texttt{in}\ r_1^0), (\texttt{letrec}\ z = v\ \texttt{in}\ r_2)])
\end{array}$$

□

Lemma 20.10. *The (iA, xch)-reduction commutes with the n-reductions of the FUNDIO-calculus and the reductions (rn). I.e.*

$$\begin{array}{lcl}
\xrightarrow{iA,xch} \cdot \xrightarrow{rn,a,P} & \leadsto & \xrightarrow{rn,a,P} \cdot \xrightarrow{iA,xch} \\
\xrightarrow{iA,xch} \cdot \xrightarrow{n,a,P} & \leadsto & \xrightarrow{n,a,P} \cdot \xrightarrow{iA,xch}
\end{array}$$

Proof. Is the same as for commutation with normal order reductions. □

Proposition 20.11. *Let t be a closed term, and let s be a subexpression for relativized normal order reduction, which is in a reduction context R, i.e., $t = R[s]$. Assume given a reduction sequence $t \xrightarrow{rn,*} t' \xrightarrow{n,*} t''$ with IO-multiset P, where the first part is the relativized normal order reduction w.r.t. s until a relativized WHNF for s is obtained in the term t', and the second part is a normal order reduction terminating with a WHNF t''.*

Then the reduction sequence can be transformed into a normal order reduction for $t \xrightarrow{n,*} t'''$ *to a WHNF, with* $t''' \xrightarrow{cpxx \vee xch,*} t''$*, where the IO-multiset is* P*, and the number of the reduction is the same; even for every type of reduction: (case), (cp),(lll), (lbeta), (IOr), the number of reductions is the same.*

Proof. Lemmas 20.9, 20.10 and 20.8 show that shifting (rn)-reductions to the right terminates and keeps the number of non-(lll)-reductions.
We have to argue that there remains no (rn)-reduction before a WHNF. Lemma 20.6 shows that the successor of s after a normal order reduction step is in a reduction context. If t is a WHNF, then the subterm s is also in relativized WHNF, and thus there is no remaining (rn)-reduction before a WHNF. Thus the shifting right does not lose any reductions nor shifts them after the terminating n-reduction. Concluding, the claim holds. □

Lemma 20.12. *The (iA, xch)- and (iA,cpxx)-reduction has trivial forking with (rn)-reductions and with n-reductions. I.e.*

$$\begin{array}{lcl}
\xleftarrow{rn,a,P} \cdot \xrightarrow{iA,xch} & \rightsquigarrow & \xrightarrow{iA,xch} \cdot \xleftarrow{rn,a,P} \\
\xleftarrow{n,a,P} \cdot \xrightarrow{iA,xch} & \rightsquigarrow & \xrightarrow{iA,xch} \cdot \xleftarrow{n,a,P} \\
\xleftarrow{rn,a,P} \cdot \xrightarrow{iA,cpxx} & \rightsquigarrow & \xrightarrow{iA,cpxx} \cdot \xleftarrow{rn,a,P} \\
\xleftarrow{n,a,P} \cdot \xrightarrow{iA,cpxx} & \rightsquigarrow & \xrightarrow{iA,cpxx} \cdot \xleftarrow{n,a,P}
\end{array}$$

Lemma 20.13. *Let* t *be a closed term, and let* s *be a subexpression that is in an A-context. The following is a complete set of forking diagrams for an rn-reduction w.r.t.* s *that is complete if the case of a fail does not occur*

$$\begin{array}{ll}
\xleftarrow{(n,\rho(a),P_1)} \cdot \xrightarrow{(rn,a,P_1)} & \rightsquigarrow Id \\
\xleftarrow{(n,a_2,P_2)} \cdot \xrightarrow{(rn,a_1,P_1)} & \rightsquigarrow \xrightarrow{(rn,a_1,P_1)} \cdot \xleftarrow{(n,a_2,P_2)} \\
\xleftarrow{(n,\rho(a_1),P_1)} \cdot \xleftarrow{(n,a_2,P_2)} \cdot \xrightarrow{(rn,a_1,P_1)} & \rightsquigarrow \xleftarrow{(n,a_2,P_2)} \\
\xleftarrow{(n,case)} \cdot \xrightarrow{(rn,case)} & \rightsquigarrow \xrightarrow{rn,case} \cdot \xrightarrow{iA,cpxx,*} \cdot \xrightarrow{iA,xch,*} \cdot \xleftarrow{(n,case)} \\
\xleftarrow{(n,\rho(a),P_1)} \cdot \xrightarrow{(rn,a,P_2)} & \rightsquigarrow fail \\
 & \quad \textit{if the redex is the same, but } P_1 \neq P_2
\end{array}$$

The right side Id *means that both reductions are removed.*

Proposition 20.14. *Let* t *be a closed term, and let* s *be a subexpression that is in a reduction context, i.e.,* $t = R[s]$*. If there is a terminating normal order reduction sequence for* t *with IO-sequence* $\overrightarrow{P}$*:* $t \xrightarrow{n,*,\overrightarrow{P}} t'$*, then there is also a terminating relativized normal order reduction sequence w.r.t.* s*. Moreover, there are IO-sequences* $\overrightarrow{P_1}, \overrightarrow{P_2}$ *with* $P = P_1 \cup P_2$ *as multisets and such that* $\overrightarrow{P_1}$ *is a subsequence of* $\overrightarrow{P}$*, such that there exists a reduction sequence* $t \xrightarrow{rn,*,\overrightarrow{P_1}} t'' \xrightarrow{n,*,\overrightarrow{P_2}} t'''$*,* t'' *is a rWHNF and* $t' \xrightarrow{iA,(xch \vee cpxx)^*} t'''$*.*

$$\begin{tikzcd}
t \arrow[r, "{rn,*,\overrightarrow{P_1}}"] \arrow[d, "{n,*,\overrightarrow{P}}"] & t'' \arrow[d, "{n,*,\overrightarrow{P_2}}"] \\
t' \arrow[r, "{iA,(xch \vee cpxx)^*}"] & t'''
\end{tikzcd}$$

The reduction $t \xrightarrow{rn,*,P_1} t'' \xrightarrow{n,*,P_2} t'''$ *requires the same number of reductions as* $t \xrightarrow{n,*,P} t'$*; even for every type of reduction: (case), (cp),(lll), (lbeta), (IOr), the number of reductions is the same.*

Proof. Note that the normal order reductions leave s in a reduction context.
As a first step we prove by induction on the number k of normal order reductions of $t \xrightarrow{k,P} t'$ the following restricted claim:
If $R[s] = t \xrightarrow{n,k,P} t'$, in k steps to a WHNF t', then either s is in rWHNF , or there is a rn-reduction $t \xrightarrow{rn} t_1$, such that $t_1 \xrightarrow{n,k-1,P} t'''$, where t''' is a WHNF and $t' \xrightarrow{iA,(xch \vee cpxx)^*} t'''$.
We have to construct a rn-reduction w.r.t. s to a rWHNF. If t is already a rWHNF, then there is nothing to show. Assume that $t \xrightarrow{rn} t_1$. We use the complete set of forking diagrams in Lemma 20.13. We treat all cases:

- If $k = 1$, then the second case is not possible, since a WHNF has no rn-reduction for a term s in a reduction context. So only the first forking transformation is possible, and the claim is true. For the other cases, assume that $k > 1$.
- If the first or third forking rule can be used, then the claim is obvious.
- If the forking rule 2 is used, then we get a t_2 with $t \xrightarrow{n} t_3 \xrightarrow{rn} t_2$ and $t_1 \xrightarrow{n} t_2$. induction shows that t_2 has a normal order reduction of length $k - 2$.
- If a fail occurs according to the last rule, then we had a choice among several IO-reductions: we simply use a different (rn)-IOr-reduction using the same redex, but an appropriate IO-pair.
- If the forking rule 4 is used, then we get a diagram as follows:

$$\begin{tikzcd}
t \arrow[rrr, "rn"] \arrow[d, "n"] & & & t_1 \arrow[d, "n"] \\
t_3 \arrow[r, "rn"] \arrow[d, "{n,k-1}"] & t_4 \arrow[r, "{cpxx,*}"] \arrow[d, "{n,k-2}"] & \cdot \arrow[r, "{xch,*}"] & t_2 \arrow[d, "{n,k-2}"] \\
t' \arrow[r, "(xch \vee cpxx)^*"] & \cdot \arrow[rr, "(xch \vee cpxx)^*"] & & t'''
\end{tikzcd}$$

By induction, there is a normal order reduction for t_4 of length $k - 2$. The forking diagrams for (cpxx) and (xch) (see Lemma 20.12) show that diagram with north-west corner t_4 holds. The diagram then contains a normal order reduction of length $k-1$ for t_1.

Tracing the IO-pairs in the diagrams, we see that the IO-multisets of the reduction $t \xrightarrow{n,*} t'$ and $t \xrightarrow{rn,\overrightarrow{P_1}} t_1 \xrightarrow{n,*,\overrightarrow{P_2}} t'''$ are the same, and that moreover the IO-sequence $\overrightarrow{P_1}$ is a subsequence of $\overrightarrow{P}$. Thus the claim is proved.

A further induction on the number of rn-reductions now shows the lemma. □

The following lemma shows that the difference between a normal-order reduction to a WHNF and an rn-reduction of the top level term consists only of (llet-in)-reductions

Lemma 20.15. *Let t be a closed term.*
There is a terminating (rn)-reduction w.r.t. t: $t \xrightarrow{rn,*,P} t'$ where t' is a rWHNF, iff there is a terminating normal order reduction $t \xrightarrow{n} t''$, and $t' \xrightarrow{P,(n,llet)^*} t''$.
The sequence of the reduction types is the same after removing the (llet-in)-reductions.

Proof. This holds, since the the trivial forking diagram for $\xrightarrow{n,llet-in}$ and an rn-reduction holds in the above situation.

$$\xleftarrow{n,llet-in} \cdot \xrightarrow{rn,a,P} \quad \rightsquigarrow \quad \xrightarrow{rn,a,P} \cdot \xleftarrow{n,llet-in}$$

$$\xleftarrow{n,a,P} \cdot \xrightarrow{rn,a,P} \quad \rightsquigarrow \quad Id$$

The inductive construction of the n-reduction (the rn-reduction, respectively) can be driven either by the rn-reduction or by the n-reduction, where in case of an IOr-reduction, we have to use the same IO-pair.

□

20.0.1 Non-Terminating rn-Reductions

Corollary 20.16. *Let s be a closed expression without a terminating normal order reduction. Then for all R: $R[s]$ does not have a normal order reduction for all reduction contexts R.*
By the context lemma this means $s \leq_c t$ for all expressions t.

Proof. The claim follows from Proposition 20.14 as follows.
Suppose $R[s]$ has some terminating n-reduction with IO-multiset P. Lemma 20.14 implies that there is also a reduction sequence first rn-reducing s in $R[s]$ to a rWHNF using P_1, and then a further n-reduction sequence with some IO-multiset P_2 with $P = P_1 \cup P_2$. Since s is closed, the rn-reduction is almost the same as a normal order reduction of s, up to some (llet)-reductions by Lemma 20.15. The (rn)-reductions yield an expression $R[s']$, where s' is closed and of the form $LR[s'']$, where s'' is in rWHNF in $LR[s'']$. After a finite number of ($llet$)-reductions, this term is also in WHNF.

This is a contradiction to the assumption that s has no terminating normal order reduction.

Now both preconditions of the context lemma are satisfied, and the context lemma shows $s \leq_c t$ for all expressions t. □

Now the following makes sense:

Definition 20.17. *Let $\bot$ be defined as a closed expression without any terminating normal order reduction.*

Corollary 20.16 shows that $\forall t : \bot \leq_c t$.

A consequence of Corollary 20.16 is that programs that have an infinite normal order reduction with (IOr)-reductions, but without any terminating normal order reduction are also equivalent to $\bot$. This is the view that programs that cannot be stopped legally are equivalent to $\bot$.

Corollary 20.18. *For every reduction context R, the equation $R[\bot] \sim_c \bot$ holds.*

Proof. Follows from Corollary 20.16. □

Lemma 20.19. *If t is a term without any normal-order reduction to a WHNF, and $t \to t'$ by a FUNDIO-reduction, then t' has no terminating normal-order reduction to a WHNF*

Proof. Assume that t' has a terminating normal order reduction. Since all non-IOr-reductions from FUNDIO are correct program transformations, the only case we have to consider is that $t \xrightarrow{IOr} t'$. So we need a complete set of commuting diagrams for $\xrightarrow{IOr}$, where the (IOr) may be in an arbitrary context. These are

$$\begin{array}{l}
\xrightarrow{i,IOr,P_1} \cdot \xrightarrow{n,P_2} \rightsquigarrow \xrightarrow{n,P_2} \cdot \xrightarrow{i,IOr,P_1} \\
\xrightarrow{i,IOr,P_1} \cdot \xrightarrow{n,P_2} \rightsquigarrow \xrightarrow{n,P_2} \cdot \xrightarrow{n,IOr,P_1} \\
\xrightarrow{i,IOr,P} \cdot \xrightarrow{cp,n} \rightsquigarrow \xrightarrow{cp,n} \cdot \xrightarrow{i,IOr,P} \cdot \xrightarrow{i,IOr,P}
\end{array}$$

Moreover, if $t_1 \xrightarrow{IOr} t_2$, and t_2 is a WHNF, then t_1 is a WHNF.

For induction we use as measure a pair of the following measures, where the pairs are ordered lexicographically.

1. The multiset of the following numbers: for every (i,IOr)-reduction the number of (n,cp)-reductions to the right of it. The multiset is ordered by the multiset ordering.
2. The multiset of the following numbers: for every (i,IOr)-reduction the number of reductions to the right of it.

Thus the shifting to the right terminates.

We conclude by induction that $t \xrightarrow{IOr} t' \xrightarrow{n,*} t''$, where t'' is a WHNF, leads to a terminating normal order reduction for t.

We have shown that the lemma holds □

Corollary 20.20. *If $t \sim_c \bot$ and $t \to t'$ by a FUNDIO-reduction, then $t' \sim_c \bot$.*

Proof. This follows from Lemma 20.19 and 20.16. □

20.1 Multiple Relativized Reductions

To treat correctness of strictness optimizations, it is necessary to consider the evaluation of more than one subterm in a single reduction sequence, and also to trace the corresponding reductions.
Therefore we generalize the evaluation labeling from Definition 20.3 to a pair (i, j), where i is the index of the term to be evaluated, and j is the evaluation label, starting from 0. It is possible that a subterm has multiple labels for different indices.
We denote the index i only if necessary. The relativized reduction according to index i is denoted as $\xrightarrow{rn(i)}$, or if the subterm s is clear from the context, we will also write $\xrightarrow{rn(s)}$.

Example 20.21. We give an example for a term with a subterm that has multiple labels for different indices.

$$(\texttt{letrec}\ x = (\texttt{IO}\ a)\ \texttt{in}\ r\ x^{(2,0)}\ (\texttt{case}\ x^{(1,0)}\ alts))$$

We show more evaluation labels:

$$(\texttt{letrec}\ x = (\texttt{IO}\ a)^{(1,2),(2,3)}\ \texttt{in}\ ((r\ x^{(2,0)})^{(2,1)}\ (\texttt{case}\ x^{(1,0)}\ alts)^{(1,1)})^{(1,2),(2,2)})$$

Lemma 20.22. *Let t be a closed term, and let s_1, s_2 be two subexpression in A-contexts. The following is a complete set of forking diagrams for a $rn(s_1)$-reduction and $rn(s_2)$-reductions which is complete if the case of a fail does not occur*

$$\xleftarrow{(rn(1),\rho(a),P_1)} \cdot \xrightarrow{(rn(2),a,P_1)} \quad \rightsquigarrow Id$$

$$\xleftarrow{(rn(1),a_1,P_1)} \cdot \xrightarrow{(rn(2),a_2,P_2)} \quad \rightsquigarrow \xrightarrow{(rn(2),a_2,P_2)} \cdot \xleftarrow{(rn(1),a_1,P_1)}$$

$$\xleftarrow{(rn(1),\rho(a_1),P_1)} \cdot \xleftarrow{(rn(1),a_2,P_2)} \cdot \xrightarrow{(rn(2),a_1,P_1)} \quad \rightsquigarrow \xleftarrow{(rn(1),a_2,P_2)}$$

$$\xleftarrow{(rn(1),case)} \cdot \xrightarrow{(rn(2),case)} \quad \rightsquigarrow \xrightarrow{rn(2),case} \cdot \xrightarrow{iA,cpxx,*} \cdot \xrightarrow{iA,xch,*} \cdot \xleftarrow{(rn(1),case)}$$

$$\xleftarrow{(rn(1),\rho(a),P_1)} \cdot \xrightarrow{(rn(2),a,P_2)} \quad \rightsquigarrow fail$$

if the redex is the same, but $P_1 \neq P_2$

Proof. Can be done by case analysis □

21 Strictness and Strictness Optimization

21.1 Strictness in FUNDIO

In this section we define the notion of strict abstraction, and more general of strict expressions. We define also the corresponding modification of the evaluation strategy w.r.t. to a strict abstraction s, and call this the s-strict evaluation strategy, or strict strategy. We prove that the strict strategy is correct if s is a closed and strict abstraction.

A strictness analysis for terms in FUNDIO appears to be possible. However, a standard denotational semantics is inappropriate since sharing is usually ignored in these semantics, and instead the contextual semantics has to be used. A promising method is the abstract reduction approach (see [Nöc93,vEGHN93,SSPS95]), since it appears to be adaptable to FUNDIO.

A result of this section is that strictness optimization in a call-by-need calculus does not save any of the (non-lll) normal order-reductions. For the (lll)-reductions, the observation is that the number of bindings in a global heap of an abstract machine (e.g. see section 25) is unchanged. An observable optimization effect comes from the knowledge of the order of evaluation, which permits to save heap-accesses, in particular heap-updates.

Definition 21.1. *We say the term s is* strict *iff* $(s\ \bot) \sim_c \bot$.

In this section, we will only consider the s-strict strategy for strict and closed abstractions. The methods in this section are easily applicable also to terms of the form $s = LR[s_0]$ where s_0 is an abstraction.

21.2 Shifting (rn) to the Left

In the following we assume that the terms contain a subterm r^0, which is the term to be evaluated by (rn)-reduction.

In order to shift (rn)-reductions in a reduction sequence to the left, commuting diagrams of the type $\xrightarrow{n} \cdot \xrightarrow{rn} \ \rightsquigarrow\ \xrightarrow{rn} \cdot \ldots$ are required. We have a rule that shows that an n-reduction may also be a (rn)-reduction. This is in particular necessary for the case $t_1 \xrightarrow{n} t_2$ where t_2 is a WHNF, turning this reduction into an (rn)-reduction. We say the set of diagrams is *complete*, if every case is covered.

Lemma 21.2. *We consider the situation where the term for relativized reduction is already available in the (n)-reduction. We fix the (rn)-reductions to this term and the corresponding 0-label. In the diagrams we assume that all reductions are in A-contexts. A complete set of diagrams for shifting (rn) to the left over (n) is as follows:*

$$\xrightarrow{a_1,n,P_1} \cdot \xrightarrow{a_2,rn,P_2} \ \rightsquigarrow\ \xrightarrow{a_2,rn,P_2} \cdot \xrightarrow{a_1,n,P_1}$$

$$\xrightarrow{\rho(a),n,P} \ \rightsquigarrow\ \xrightarrow{a,rn,P}$$

$$\xrightarrow{n,case} \cdot \xrightarrow{rn,case} \ \rightsquigarrow\ \xrightarrow{rn,case} \cdot \xrightarrow{n,case} \cdot \xrightarrow{iA,cpxx,*} \cdot \xrightarrow{iA,xch,*}$$

Proof. A case analysis. The computation of a critical case is the same as in the proof of Lemma 20.9. □

Lemma 21.3. *Let t be a closed term and r be a subexpression that is in rWHNF w.r.t. $rn(r)$. If $t \xrightarrow{iA,lll} t'$, then the successor of r is also in rWHNF.*

The following proposition is a first step in showing that a rearrangement of a normal order reduction sequence according to strictness information is correct, but does not save any reduction steps.

Proposition 21.4. *Let $R[(s\ t)]$ be an expression such that s is a strict, closed abstraction, and R a reduction context. Assume there is a terminating normal order reduction sequence for $R[s\ t]$ with IO-sequence $\overrightarrow{P}$. Then there is a reduction sequence w.r.t. t as follows: $R[s\ t] \xrightarrow{rn(t),*,\overrightarrow{P_1}} r \xrightarrow{n,*,\overrightarrow{P_2}} r'$, such that r is a rWHNF w.r.t. t, $\overrightarrow{P_1}$ is a subsequence of $\overrightarrow{P}$, $P = P_1 \cup P_2$ as multisets, and the number of reductions of every kind is unchanged: (lll), (cp), (lbeta), (IOr), (case).*

Proof. Since s is strict, the reduction can be splitted, such that $R[s\ t] \xrightarrow{n,*} r' \xrightarrow{n,*} r''$, such that t^0 is in a reduction context in r'. Then Proposition 20.14 shows that there is a terminating rn-reduction of r' such that

$$R[s\ t] \xrightarrow{n,*,P_{2,1}} r' \xrightarrow{rn(t),*,P_1} \cdot \xrightarrow{n,*,P_{2,2}} r''.$$

The complete set of reductions in Lemma 21.2 shows that the (rn)-reductions can be moved to the left, where the (cpxx) and (xch) may be shifted to the right out of the reduction sequence. We use the following strategy using the diagrams in Lemma 21.2: Let there be a border between the starting (rn)-reduction sequence and a following (n)-reduction sequence. In the n-reduction sequence find the first one that can be turned into an (rn)-reduction. Then shift this (rn)-reduction completely to the left until the border is reached. The remaining (cpxx) and (xch)-reductions are shifted to the rightmost end of the sequence. After this, the reduction is of the form $R[s\ t] \xrightarrow{rn,t,*,\overrightarrow{P_1}} r \xrightarrow{n,*,\overrightarrow{P_2}} r'$. The term r must be in rWHNF. □

21.3 Shifting (n) to the Left Over rn-Reductions

We require also the diagrams for shifting n-reductions to the left.

Lemma 21.5. *We consider the situation where the term for relativized reduction is already available in the (n)-reduction. We fix the (rn)-reductions to this term and the corresponding 0-label. In the diagrams we assume that all reductions are in A-contexts. A complete set of diagrams for shifting (n) to the left over (rn) is as follows:*

$$\begin{array}{lll}
\xrightarrow{a_1,rn,P_1} \cdot \xrightarrow{a_2,n,P_2} & \rightsquigarrow & \xrightarrow{a_2,n,P_2} \cdot \xrightarrow{a_1,rn,P_1} \\
\xrightarrow{a,rn,P} & \rightsquigarrow & \xrightarrow{\rho(a),n,P} \\
\xrightarrow{rn,case} \cdot \xrightarrow{n,case} & \rightsquigarrow & \xrightarrow{n,case} \cdot \xrightarrow{rn,case} \cdot \xrightarrow{iA,cpxx,*} \cdot \xrightarrow{iA,xch,*}
\end{array}$$

Proof. A case analysis. □

21.4 Shifting (rn) to the Left Over rn-Reductions

To treat the strict strategy, it is necessary to consider shifting rn(1)-reductions over rn(2)-reductions.

Lemma 21.6. *We consider only situations where at least the 0-label for the corresponding indices are available. We use two (rn)-reductions for the indices 1 and 2. In the diagrams we assume that all reductions are in A-contexts.*
A complete set of diagrams for shifting rn(2)-reductions to the left over rn(1)-reductions is as follows:

$$\xrightarrow{a_1,rn(1),P_1} \cdot \xrightarrow{a_2,rn(2),P_2} \;\leadsto\; \xrightarrow{a_2,rn(2),P_2} \cdot \xrightarrow{a_1,rn(1),P_1}$$

$$\xrightarrow{a,rn(1),P} \;\leadsto\; \xrightarrow{a,rn(2),P}$$

$$\xrightarrow{rn(1),case} \cdot \xrightarrow{rn(2),case} \;\leadsto\; \xrightarrow{rn(2),case} \cdot \xrightarrow{rn(1),case} \cdot \xrightarrow{iA,cpxx,*} \cdot \xrightarrow{iA,xch,*}$$

Proof. A case analysis. The computation of a critical case is the same as in the proof of Lemma 20.9. □

Proposition 21.7. *Let $C[(s\ t)]$ be an expression, where s is a strict and closed abstraction, C is an A-context, and the subterm $(s\ t)$ is labeled as follows: $C[(s\ t)^{(1,j)}]$. Assume there is a terminating rn(1)-reduction sequence for $C[(s\ t)]$ with IO-sequence $\overrightarrow{P}$. Then for t being the subterm for index 2, there is a rn(2)-reduction sequence as follows:*

$$R[s\ t] \xrightarrow{rn(2),*,\overrightarrow{P_1}} r \xrightarrow{rn(1),*,\overrightarrow{P_2}} r',$$

such that $\overrightarrow{P_1}$ is a subsequence of $\overrightarrow{P}$, and $P = P_1 \cup P_2$ as multisets. The term r is a rn(2)-rWHNF, and the term r' is a rn(1)-rWHNF. The number of reductions of every kind of reductions (lll), (cp), (lbeta), (IOr), (case) is unchanged.

Proof. The proof is the same as for Proposition 21.4 where a copied lemma and proof of Lemma 20.14 is used. □

21.5 Strictness Optimizations

Definition 21.8. *Let s be a strict, closed abstraction that is a subexpression of a closed expression t_0. We assume that the reduction strategy maintains strictness marks for closed abstractions, where in case of a (cp), the strictness mark is duplicated. Let s have a strictness marker.*
Define the s-strict evaluation strategy (s-strict reduction) *as follows:*
Whenever $(s\ t)$ is in a reduction context, then

- *Use a new index j for t and apply s-strict-rn(j)-reduction w.r.t. t, until a rn(j)-rWHNF is reached.*
- *Apply the normal-order reduction (lbeta) for the subterm $(s\ t')$*

- *Proceed with the s-strict evaluation strategy.*

Define the s-strict-rn(j)-reduction w.r.t. $r^{(j,0)}$, as follows:
Whenever $(s\ t)$ is a subterm to be rn(j)-reduced by (lbeta), then

- *If $(s\ t)$ is already pending for another rn(k)-reduction, then fail: this would cause an infinite loop.*
- *Use a new index j' for t and apply s-strict-rn(j') reduction w.r.t. t, until a rn(j')-rWHNF is reached.*
- *Apply the rn(j)- step (lbeta) for the subterm $(s\ t')$*
- *Then proceed with the s-strict-rn(j)-reduction.*

Example 21.9. As an example for the possibility of a loop, consider the term

```
letrec x = s y, y = s x in y
```

We show that the existence of a terminating normal-order reduction for P implies the existence of a terminating s-strict reduction for P.

Proposition 21.10. *Let s be a strict, closed abstraction that is a subexpression of a closed expression t and that is marked as strict. Assume there is a normal order reduction of t using IO-multiset P to a WHNF. Then there is a s-strict reduction to a WHNF with IO-multiset P that requires the same number of of every kind of reduction: (lll), (cp), (lbeta), (IOr), (case).*

Proof. Let $t{\Downarrow}(P)$, then we have to show that there is a terminating s-strict reduction using IO-multiset P. Let $t \xrightarrow{n,*,P} t_1$ be a normal order reduction to a WHNF t_1. If there is no intermediate term of the form $R[s\ r]$, where R is a reduction context, then we are ready, since the normal-order reduction is already an s-strict reduction. So assume that $t \xrightarrow{n,*,P_1} t' = R[s\ r] \xrightarrow{n,*,P_2} t_1$. Proposition 21.4 shows that the (rn)-reductions w.r.t. r in a sequence $R[s\ r] \xrightarrow{n,*,P_2} t_1$ can be moved to the left until the reduction sequence is of the form $R[s\ r] \xrightarrow{rn,*} t'' \xrightarrow{n,*} t_1$, where t'' is in rWHNF.
To use induction, we have to count the length of a reduction: The length is the same, and additionally, the first reduction in $t'' \xrightarrow{n,*} t_1$ must be an (lbeta)-reduction: The (lbeta)-reduction cannot be required as an (rn)-reduction, since otherwise, the rn-reduction is either non-terminating, or there will be a fail in finding the next rn-redex. Since at least the (lbeta)-reduction is missing, the reductions $R[s\ r] \xrightarrow{rn,*} t''$, as well as $t'' \xrightarrow{n,*} t_1$ have less reductions, and induction can be used.
The same arguments can be used if the reduction $R[s\ r] \xrightarrow{rn,*} t''$ is modified by a strict abstraction s:
Let $t \xrightarrow{rn(j),*,P} t_1$ be a reduction to a rWHNF t_1. If there is no intermediate term of the form $R[s\ r]$, then we are ready, since the $rn(j)$-reduction is already an s-strict-$rn(j)$-reduction. So assume that $t \xrightarrow{rn(j),*,P_1} t' = C[s\ r] \xrightarrow{n,*,P_2} t_1$, where $(s\ r)$

is the subterm to be (lbeta)-reduced by $rn(j)$-reduction. Proposition 21.7 shows that the (rn(j'))-reductions w.r.t. r in a sequence $C[s\ r] \xrightarrow{rn(j),*,P_2} t_1$ can be moved to the left until the reduction sequence is of the form $C[s\ r] \xrightarrow{rn(j'),*} t'' \xrightarrow{rn(j),*} t_1$, where t'' is in rWHNF . Again the number of reductions is the same, and the first reduction of $t'' \xrightarrow{rn(j),*} t_1$ is an (lbeta)-reduction. Thus induction can be used.

□

Proposition 21.11. *Let s be a strict, closed abstraction that is a subexpression of a closed expression t and that is marked as strict. Assume there is an s-strict reduction of t using IO-multiset P to a WHNF. Then there is a normal order reduction to a WHNF with IO-multiset P that requires the same number of non-(lll)-reductions, in particular also the same number of every kind of reductions: (lll), (cp), (lbeta), (IOr), (case).*

Proof. Let $t \xrightarrow{str(s),*,P} t_0$. If this is already a normal order reduction, then we are ready. Let there be an intermediate term, such that $t \xrightarrow{n,*,P_1} t_1 = R[s\ r_1] \xrightarrow{str(s),*,P_2} t_0$ with $P = P_1 \cup P_2$. By induction on the length of the reduction sequence, we can assume that $t = t_1$. The reduction is of the form $t \xrightarrow{str(s),*,P_{2,1}} t_2 = R[s\ r_1'] \xrightarrow{str(s),*,P_2} t_0$, where the first reduction of the second part is the (lbeta) reduction for the redex $(s\ r_1')$. By induction, this can be rearranged as $t \xrightarrow{rn(r_1),*,P_{2,1}} t_2 = R[s\ r_1'] \xrightarrow{str(s),*,P_2} t_0$, where the (cpxx)- and (xch)-reductions are moved out of the way. We can shift the n-reductions to the left using Lemma 21.5, again moving the (cpxx)- and (xch)-reductions out of the way. This shows that there is also a n-reduction to a WHNF.
What remains to be proved is that the (rn)-reductions that are to the right are not after the WHNF is reached, but are all turned into n-reductions. The argument is as follows: If after the shifting, the reduction is of the form $t \xrightarrow{n,*,P_3} t_3 \xrightarrow{rn(r_1),*,P_4} t_4$, where t_3 is a WHNF, then again, we can isolate the $rn(r_1)$-reductions in the reduction $t \xrightarrow{n,*,P_3} t_3$ and shift them to the left. The result would be a reduction $t \xrightarrow{rn(r_1),*,P_5} t_5 \xrightarrow{n,*,P_6} t_3'$, but t_5 is not in rWHNF. So we could also place in a bot-term for the 0-labeled term in t_5, since there is no further reduction on it. This is a contradiction to the strictness of s.

The same arguments as above can be used to show the claim for the case where an $rn(j)$-reduction has to be shifted over an rn(j')-reduction, where Lemma 21.6 has to be used.

□

Theorem 21.12. *Let s be a closed and strict abstraction that is a subexpression of a closed expression t_0, and that is marked as strict. The the s-strict strategy is correct as a strategy.*

Proof. For the proof we have to use the definition of a correct strategy. I.e. we show that $\forall P : t\Downarrow(P) \Leftrightarrow t\Downarrow_{str(s)}(P)$ and $t\Uparrow \Leftrightarrow t\Uparrow_S$. Hence the proof consists of four parts.

1. For $t\Downarrow(P)$, we have to show that there is an s-strict reduction terminating with a WHNF and using P as IO-multiset. This follows from Proposition 21.10 and Theorem 16.1.
2. Assume, that the s-strict strategy terminates with a WHNF using IO-multiset P. Then Lemma 21.11 shows the that there is also a normal order reduction to a WHNF using the same IO-multiset.
3. If $t_0 \xrightarrow{n,*,\overrightarrow{P}} t_1$, and $t_1 = \bot$, then we have to show that there is some s-strict reduction of t_0 to a term t'_1 that has no terminating s-strict reduction (i.e. $t'_1 = \bot$)

   If the normal-order reduction $t_0 \xrightarrow{n,*,\overrightarrow{P}} t_1$ is also a s-strict reduction, then we can use part 1 and 2 of this proof and we are ready.
   Hence we have to consider the case that there is an intermediate term $t_2 = R[s\ t]$ with $t_0 \xrightarrow{n,*\overrightarrow{P_1}} t_2 \xrightarrow{n,\#k,\overrightarrow{P_2}} \bot$. From t_2 we start the (rn)- reduction(s) for s: $t_2 \xrightarrow{rn} t'_2$. The situation is as follows:

$$\begin{tikzcd}
t_2 = R[s\ t] \arrow[r, "rn{,}*"] \arrow[d, "n{,}\#k"] & t''_2 \arrow[d, "?"] \\
t_1 = \bot \arrow[r, "?"] & t''_1 = \bot
\end{tikzcd}$$

   We construct further reduction sequences of t_2 by induction on the number k of normal-order reductions.
   If all rn-reductions w.r.t. t from t_2 are infinite or stop with an error, then we are ready, since then there is also no terminating s-strict reductions of t_2.
   Hence assume there are some terminating rn(t)-reductions of t_2. We argue by using forking diagrams that either we can use induction on the k, or the rn(t)-reduction terminates with a rWHNF that has a n-reduction to $\bot$.
   So the second parameter for the induction is the number of rn-reductions.
   Using the forking diagrams for (rn) in 20.13 as well as for (cpxx), (xch), we obtain by induction that there is a reduction $t'_2 \xrightarrow{n,*} t'_1$, where either $t_1 \xrightarrow{rn} \cdot \xrightarrow{(cpxx \vee xch)^*} t'_1$, or $t_1 \xrightarrow{(cpxx \vee xch)^*} t'_1$. The diagram is:

$$\begin{tikzcd}
t_2 = R[s\ t] \arrow[rr, "rn"] \arrow[d, "n{,}*"] & & t'_2 \arrow[r, "rn{,}*"] \arrow[d, "n{,}*"] & t''_2 \arrow[d, "n{,}*"] \\
t_1 \arrow[r, "rn^{0 \vee 1}"] & \cdot \arrow[r, "iA{,}(xch \vee cpxx)^*"] & t'_1 \arrow[r, "(rn \vee xch \vee cpxx)^*"] & t''_1
\end{tikzcd}$$

   It is clear that t'_1 has no terminating normal-order reduction. Furthermore, it follows from the diagrams, that the length of $t'_2 \xrightarrow{n,*} t'_1$ is not greater than the length of $t_2 \xrightarrow{n} t_1$. If the (rn)-reduction is an (IOr), then it may be possible that the transformation fails. In this case, we simply modify the (rn)-reduction to use the same IO-pair as the normal order reduction. In this fail-case, the length of $t_2' \xrightarrow{n,*} t_1'$ is strictly smaller, and induction can be used.

The base case is that $n = 0$, hence the reduction $t_2 \xrightarrow{rn,*} t_2'' \xrightarrow{n,*} t_1''$ is the s-strict reduction, where t_1'' has no terminating normal order reduction.

So we have shown the first step in constructing an s-strict reduction to $\bot$. Induction on the number of (lbeta)-reductions with strictness-marked marked abstractions can be used to construct the this s-strict reduction, where instead of forking diagrams for two kinds of (rn)-reductions is required (see Lemma 20.22).

4. If there is an s-strict reduction to a term having no s-strict reductions to a WHNF, we have to argue that there is also a normal order reduction to a term without terminating normal-order reduction; i.e. to a term equivalent to $\bot$.

   Let the reduction be $t_0 \xrightarrow{str(s)} t_1$, and t_1 has no terminating s-strict reduction. From Proposition 21.10 and since s is strict it follows that t_1 cannot have a terminating normal-order reduction.

   If the s-strict reduction $t_0 \xrightarrow{str(t)} t_1$ consists only of normal order reductions, then we are ready. Hence the reduction is of the form $t_0 \xrightarrow{n,*} t_2 = R[s\ t] \xrightarrow{str(rn(s)),+} t_3 \xrightarrow{n,*} t_1 = \bot$.

   To simplify the following arguments, we assume that $t_2 \xrightarrow{str(rn(t)),+} t_3$ is an $rn(t)$-reduction.

   If t_3 is a rWHNF, then we can use Lemma 20.11, Theorem 16.1 and obtain a n-reduction to $\bot$. Hence we have only to consider the situation $t_0 \xrightarrow{n,*} t_2 \xrightarrow{rn,+} t_1 = \bot$.

   If there is no terminating normal order reduction for t_2, then we are ready. Hence we can select a terminating n-reduction of t_2.

   We use induction on the length of the rn-reduction to $\bot$, and then on the length of the selected n-reduction. The forking diagrams in Lemma 20.13 show that if no conflict occurs, the following diagram can be constructed:

$$\begin{tikzcd}
t_2 \arrow[rr, "{rn,*,h}"] \arrow[d, "{n,k}"] & & t_1 \arrow[d, "{n,\leq k}"] \\
t_2' \arrow[r, "{rn,*}"] & \cdot \arrow[r, "{iA,(xch \vee cpxx)^*}"] & t_1'
\end{tikzcd}$$

   If a conflict occurs, then we can use induction on the length of the rn-reduction.

   The term t_2' is a WHNF.

   If there are no rn-reductions necessary in the lower reduction, then we are ready, since t'_1 has no terminating normal-order reduction. In the reduction from t_2 to t'_2, there is an intermediate term such that t is in a reduction context, since s is strict and t_2 is of the form $R[s\ t]$. Hence there cannot be any $rn(t)$-reduction in the reduction from t'_2 to t'_1, since this could be transformed into the rn-reduction starting from t_2 leading to a rWHNF, which is already treated.

In the general case that the strict reduction $t_2 \xrightarrow{str(rn(t)),+} t_3$ is not $rn(t)$-reduction, but has subreductions, then we argue as follows:
Terminating subreductions can be transformed into rn-reductions of the higher level. For non-terminating ones we can use induction on the number of lbeta-reductions for abstractions that marked as strict. This is the the same diagram reasoning as for normal order.
Summarizing, we can construct a normal order reduction from t_2 to a term that has no further normal order reductions.

□

Remark 21.13. General Strictness Optimization
A more general strictness optimization would be to replace strict subterms $\lambda x.s$ in an environment, such that $C[\lambda x.s]$ is replaced by $C[\lambda x.(x\ \mathtt{seq}\ s)]$ where `seq` is definable as $\lambda x, y.\mathtt{case}\ x\ (Pat_1\ \rightarrow\ y)\ \ldots (Pat_N\ \rightarrow\ y)$. To show the correctness of this transformation helps to make the proof of the correctness of the strictness optimization incremental.

22 Parallel Evaluation

There are two basically different methods to parallelize evaluation:

- To analyze the program to detect expressions that can be evaluated in parallel, which is the same analysis as for strictness. This is an implicit parallelism.
- Annotations like `(parallel E1 E2)` which could mean to evaluate $E1, E2$ in parallel, the value being the value of $E2$. This is an explicit parallelism.

Both methods are compatible with contextual equivalence of FUNDIO, if the explicit parallelism is encoded as a `seq`.
It appears to be wrong to view `(parallel E1 E2)` only as a hint to start evaluation of `E1`, since then there may be useless IO-requests, required by `E1`.
The modification of the strict evaluation strategy also provides a justification of correctness of parallel evaluation. The correctness is an issue, since other definitional variants of the semantics would make parallel evaluation incorrect. This enables in principle parallel processes that perform the reductions, including IO-calls, and which may be performed in different sequential orders. A formal treatment has to state the correctness of a parallel strategy as the correctness of a strategy that non-deterministically interleaves the different reductions. The strictness optimized strategy is an instance. This is not formally treated in this

paper, but the results for the correctness of the s-strict strategy show that we can be rather sure that there are no obstacles.

23 Weaker Definitions of Contextual Equivalence

Now we can prove the following lemma (see 6.6)

Lemma 23.1. *$s \leq_c t$ is equivalent to:*

$$\forall C[\cdot]\ C[s], C[t] \textit{ are closed} \Rightarrow \Big(\forall P : C[s]\Downarrow(P) \Rightarrow C[t]\Downarrow(P)\Big) \wedge C[t]\Uparrow \Rightarrow C[s]\Uparrow$$

Proof. Assume that

$$\forall C[\cdot]\ C[s], C[t] \text{ are closed} \Rightarrow \Big(\forall P : C[s]\Downarrow(P) \Rightarrow C[t]\Downarrow(P)\Big) \wedge C[t]\Uparrow \Rightarrow C[s]\Uparrow$$

holds, and let s, t be two terms, and C be a context, such that $C[s]$ contains free variables. Let P be an IO-multiset such that $C[s]\Downarrow(P)$. We have to show that $C[t]\Downarrow(P)$. Let C' be the context $(\lambda x_1, \ldots, x_n.C[])\ \bot \ldots \bot$, where x_i are all the free variables in $C[s], C[t]$. From $C[s]\Downarrow(P)$, we obtain that $((\lambda x_1, \ldots, x_n.C[s])\ \bot \ldots \bot)\Downarrow(P)$ since it is permitted to instantiate $\bot$. But then $C'[t]\Downarrow(P)$ holds, too. This means that $C[t]\Downarrow(P)$ also holds, since instantiating a WHNF gives a WHNF.
For divergence, similar arguments can be used. Instantiation of an error-term gives an error-term, and an infinite reduction does not change after instantiation. □

24 Behavioral Equivalence

This section gives instructive examples to demonstrate that a practically useful definition of behavioral approximation is far from being a simple extension of the deterministic variant of behavioral approximation.

Proposition 24.1. *There are closed s, t with $s \not\leq_c t$, but*

1. *$\forall P$: $s\Downarrow(P) \Rightarrow t\Downarrow(P)$*
2. *Neither s, t have a bot-reduction.*
3. *For all closed $r : s\ r \sim_c t\ r$*

Proof. Let a, b, c, d be different numbers and let $+$ be addition. Define the following expressions, where `choice` is defined as in Definition 6.14.

```
s := \x. choice (choice (a+c) (a+d)) (choice ( b+c) (b+d))
t := let y = choice a b in \x. y + (choice c d)
```

Apply both terms to an argument:

```
  s r = choice (choice (a+c) (a+d)) (choice ( b+c) (b+d))
  t r  = let y = choice a b in  y + choice c d
```

To see that $s\ r \sim_c t\ r$, we can test them in a reduction context: For every IO-multiset, we get the same results (up to some reductions), hence $s\ r \sim_c t\ r$. However, $s \not\sim_c t$:
Let $f =$ `\z . z 0 + z 0`, and check `f s` and `f t`.
`f s` reduces to

```
  choice (choice (a+c) (a+d)) (choice (b+c) (b+d))
    +   choice (choice (a+c) (a+d)) (choice (b+c) (b+d))
```

Note that `a+b+c+d` is a possible selection for the IO-multiset $\{(0,T),(0,T),(0,F),(0,F)\}$

```
f t = let z = s in z 0 + z 0
    = let z = (let y = choice a b in \x. y + choice c d) in z 0 + z 0
    = let y = choice a b in let z = \x. y + choice c d in z 0 + z 0
    = let y = choice a b in  (y + (choice c d)) + (y + (choice c d))
```

Now the possibility `a+b+c+d` is no longer available, and moreover, there is no reduction for the IO-multiset $\{(0,T),(0,T),(0,F),(0,F)\}$.

□

Proposition 24.2. *There are closed terms s, t with $s \sim_c t$, and*

1. $\forall P$: $s \Downarrow (P) \Leftrightarrow t \Downarrow (P)$
2. *Neither s nor t have a bot-reduction.*
3. *There is a WHNF s' of s, such that for all WHNFs t' of t: $s' \not\leq_c t'$.*

The terms are:

```
s = choice (\x. (choice (choice 1 bot) (choice 2 3)))
                 (\x. (choice (choice 4 5) (choice 5 6)))

t = choice (\x. (choice (choice 1 2) (choice 2 3)))
                 (\x. (choice (choice 4 bot) (choice 5 6)))
```

We provide only a sketch of proof for $s \sim_c t$: The expressions s, t have to be checked in reduction contexts. A critical one is $([]\ r)$. The terms $(s\ r)$ and $(t\ r)$ differ only by the way $\bot$ can be reached by reduction, hence $s \sim_c t$.
But if we select the WHNF $s' =$ `(\x. (choice (choice 4 5) (choice 5 6)))` of s, there is no WHNF t' of t with $s' \leq_c t'$, since for any closed r: $s'\ r$ has no bot-reduction, and it cannot be $\leq_c$ `(choice (choice 1 2) (choice 2 3)))`

□

25 An Abstract Machine for FUNDIO

25.1 An Abstract Machine for FUNDIO based on Sestoft's machine

This abstract machine is based on the abstract machines in [Ses97], and on the nondeterministic extensions in [MS99,Kut00].

Definition 25.1. *The language that is suitable for the abstract machine has the following restrictions:*

- *Arguments of applications must be variables: I.e. only* $(t\ x)$ *is admissible*
- *The arguments in constructor applications must be variables: I.e. only* $(c\ x_1\ \ldots\ x_n)$ *is admissible*

Proposition 17.5 shows that it is a correct program transformation to use the reverse of (ucp) as follows:

$$\begin{array}{ll}
(s\ t) & \rightarrow (\texttt{letrec}\ x = t\ \texttt{in}\ (s\ x)) \\
(c\ t_1\ \ldots\ t_n) & \rightarrow (\texttt{letrec}\ x_1 = t_1, \ldots, x_n = t_n\ \texttt{in}\ (c\ x_1\ \ldots\ x_n)) \\
 & \quad \text{where } x, x_i \text{ are new variables.}
\end{array}$$

A variant of an abstract machine derived from Sestoft's machine is as follows:

Definition 25.2. *The abstract machine for FUNDIO:*
It has as state a triple $\langle \Gamma, t, S\rangle$*, where* Γ *is a multiset of (recursive) bindings* $x_i = t_i$*,* t *is an expression,* S *is a stack, which may contain variables* x*, update markers* $\#x$*, a complete set of alternatives, or the constant* `IO`*.*
The transition rules of the machine are:

$$\begin{array}{lr}
(\mathit{Lookup}) & \langle \Gamma\{x = t_x\}, x, S\rangle \rightarrow \langle \Gamma\ , t_x, \#x : S\rangle \\
(\mathit{Update}1) & \langle \Gamma, r, \#x : S\rangle \rightarrow \langle \Gamma\{x = r\}, r, S\rangle \\
 & \textit{if } r \textit{ is a constructor application} \\
(\mathit{Update}2) & \langle \Gamma, r, \#x : S\rangle \rightarrow \langle \Gamma\{x = r'\}, r, S\rangle \\
 & \textit{if } r \textit{ is an abstraction; where } r' \textit{ is a } \alpha\textit{-renamed copy of } r \\
(\mathit{Unwind}) & \langle \Gamma, t\ x, S\rangle \rightarrow \langle \Gamma, t, x : S\rangle \\
(\mathit{Share}) & \langle \Gamma, \lambda x.t, y : S\rangle \rightarrow \langle \Gamma, (\texttt{letrec}\ x = y\ \texttt{in}\ t), S\rangle \\
(\mathit{Case}) & \langle \Gamma, \texttt{case}\ t\ \mathit{alts}, S\rangle \rightarrow \langle \Gamma, t, \mathit{alts} : S\rangle \\
(\mathit{Branch}1) & \langle \Gamma, c\ x_1 \ldots x_n, \{(c\ y_1 \ldots y_n \rightarrow t) \ldots\} : S\rangle \rightarrow \langle \Gamma, (\texttt{letrec}\ \overrightarrow{y_i = x_i}\ \texttt{in}\ t), S\rangle \\
(\mathit{Branch}2) & \langle \Gamma, \lambda x.r, \{(\texttt{lambda} \rightarrow t) \ldots\} : S\rangle \rightarrow \langle \Gamma, t, S\rangle \\
(\mathit{Letrec}) & \langle \Gamma, (\texttt{letrec}\ \overrightarrow{x = t}\ \texttt{in}\ s), S\rangle \rightarrow \langle \Gamma\{\overrightarrow{x = t}\}, s, S\rangle \\
(\mathit{UnwindIO}) & \langle \Gamma, \texttt{IO}\ t, S\rangle \rightarrow \langle \Gamma, t, \texttt{IO} : S\rangle \\
(\mathit{IO}) & \langle \Gamma, c, \texttt{IO} : S\rangle \rightarrow \langle \Gamma, d, S\rangle \\
 & \textit{where } c, d \textit{ are constants}
\end{array}$$

Evaluation of a closed term t *starts with* $\langle \{\ \}, t, \emptyset\rangle$*.*
The machine stops successfully, if $\langle \Gamma, r, \emptyset\rangle$ *is reached, and* r *is a value. The IO-multiset of the whole evaluation is defined as the multiset of the used IO-pairs.*

The machine has several possibilities for a fail with error

- If $\langle \Gamma, x, S\rangle$ is reached, but x is not bound in the environment. This may happen if the evaluation is looping.
- if $\langle \Gamma, r, S\rangle$ is reached, r is a constructor application that is not a constant, and the top of the stack does not consist of alternatives nor an update marker nor the constant `IO`. This is usually a typing error.
- if $\langle \Gamma, r, \texttt{IO} : S\rangle$ is reached, and r is a value, but not a constant. This is a typing error.

Remark 25.3.

- (Share) is named (Subst) in Moran's paper. It uses `letrec`, whereas Moran implements (betavar) (see [MS99]).
- (Branch) differs from Moran's (branch) in the point that the generation of a `letrec` is done instead of substitution.

The FUNDIO-abstract machine could be optimized using replacement of variables by variables. i.e. use the rule (Betavar) in the rules (Share) and (Branch). This however, would require a more involved correctness proof of the machine.

An optimization for this machine may be to modify the inputted term using lambda-lifting and then a reverse (ucp) to avoid nested lambdas, since these may be unnecessarily copied. The same holds for alternatives of a case.

25.2 Correctness of an Eager-Copy-Strategy

To show that the abstract machine makes a correct evaluation, we require the correctness of the eager-copy-strategy.

Definition 25.4. *The* eager-copy-strategy, (ec-strategy) *for evaluation is defined as follows:*
All normal order reductions are admissible with the following exceptions:

- *(case-in) and (case-e) are not allowed, i.e. case-reductions are only allowed if the case-expression is of form* (`case` v *alts*)*, where v is a value. Instead of a (case-e) or (case-in)-reduction, a (cpcx)-reduction is performed for the first variable in the chain. I.e., if*

$$\begin{array}{l} R[\ (\texttt{letrec}\ x_1 = (c\ t_1\ \ldots t_n), x_2 = x_1, \ldots, x_{m-1} = x_m, Env \\ \quad \texttt{in}\ R'[\texttt{case}\ x_m\ ((c\ z_1 \ldots z_n) \to s)\ alts])] \\ \to R[\ (\texttt{letrec}\ x_1 = (c\ y_1\ \ldots y_n), y_1 = t_1, \ldots, y_n = t_n, \\ \qquad x_2 = x_1, \ldots, x_{m-1} = x_m, Env \\ \quad \texttt{in}\ R'[(\texttt{letrec}\ z_1 = y_1, \ldots z_n = y_n\ \texttt{in}\ s)])] \end{array}$$

is the normal order reduction, and $m > 1$, then apply (cpcx) instead and obtain the following term:

$$\begin{array}{l} R[\ (\texttt{letrec}\ x_1 = (c\ y_1\ \ldots y_n), y_1 = t_1, \ldots, y_n = t_n, \\ \qquad x_2 = (c\ y_1\ \ldots y_n), \ldots, x_{m-1} = x_m, Env \\ \quad \texttt{in}\ R'[\texttt{case}\ x_m\ ((c\ z_1 \ldots z_n) \to s)\ alts])] \end{array}$$

If $m = 1$, obtain the following term:

$$\begin{array}{l} R[\ (\mathtt{letrec}\ x_1 = (c\ y_1\ \ldots y_n), y_1 = t_1, \ldots, y_n = t_n, Env \\ \quad \mathtt{in}\ R'[\mathtt{case}\ (c\ y_1\ \ldots y_n)\ ((c\ z_1 \ldots z_n) \to s)\ alts])] \end{array}$$

- *(cp)-reductions are only allowed, if the copy is direct, i.e. if the term is* $R[(\mathtt{letrec}\ x_1 = \lambda x.s, x_2 = x_1, \ldots, x_{m-1} = x_m, Env\ \mathtt{in}\ R'[x_m])]$*, then instead of copying into x_m, the copy is made into x_2: The result is* $R[(\mathtt{letrec}\ x_1 = \lambda x.s, x_2 = \lambda x.s, \ldots, x_{m-1} = x_m, Env\ \mathtt{in}\ R'[x_m])]$.

Note that the WHNFs reached by the ec-strategy are of the form $(\mathtt{letrec}\ Env\ \mathtt{in}\ v)$, where v is a value.

Proposition 25.5. *The ec-strategy is a correct strategy.*

Proof. For the proof the criterion in Lemma 19.6 is used.
Let $t{\downarrow_{ec}}(\overrightarrow{P})$. Then the reduction sequence is a mixture of normal-order, cp-reductions and (cpcx)-reductions. The (cp)-reductions are in fact (cpt)-reductions.
Then Lemmas 11.7 and 14.7 show that there is a also a terminating normal order reduction of t with the same IO-sequence.
Let $t{\downarrow}(\overrightarrow{P})$. We have to show that $t{\downarrow_{ec}}(\overrightarrow{P})$. We do this by induction on the sum of the number of (n, case)-reductions and (lbeta)-reductions; and then on the number of n-reductions of t with $\overrightarrow{P}$.
If the first n-reduction is not a (case-e) nor (case-in), then we can use induction, since the normal-order reduction is also a reduction according to the ec-strategy.
If the reduction is a (case-e) or (case-in), then the ec-strategy-reduction is a (cpcx). It is obvious, that then $t \xrightarrow{cpcx,+} t' \xrightarrow{case,n} t''$ is the start of the ec-strategy-reduction sequence. The Lemmas on forking diagrams of (cpcx), (cpx), (gc), (xch) (see Lemmas 14.5, 13.2 14.2, and 12.3) show that t'' has a normal order reduction to WHNF with the same $\overrightarrow{P}$, but with a smaller number of (case)-reductions and (lbeta)-reductions, Hence we can use induction.
If the reduction is a (cp)-reduction over several indirections, then the ec-strategy-reduction is a (cpt). Then either $t \xrightarrow{cpt,+} t'$ and t' is a WHNF, or $t \xrightarrow{cpt,+} t' \xrightarrow{lbeta} t''$, or $t \xrightarrow{cpt,+} t' \xrightarrow{case-c} t''$. In the latter case it follows from Lemma 11.3, that t'' has a smaller number of (case)-reductions and (lbeta)-reductions. Hence we can use induction. These are all cases.
We can use the same arguments for the equivalence of validity.
This proves correctness of the ec-strategy.

□

25.3 Correctness of the Abstract Machine

Theorem 25.6. *Let t be a closed term. Let t' be the transformed term according to definition 25.1. The machine stops with success for input t' and IO-sequence $\overrightarrow{P}$ iff t has a normal order reduction for $\overrightarrow{P}$.*

Proof. (Sketch) We show that the machine simulates the reduction according to the correct eager-copy-strategy (see Proposition 25.5). The environment of the machine indicates the bindings in the outermost `letrec`of the intermediate term in the normal order reduction.
There is a correspondence of the structure of the intermediate term ($\texttt{letrec}\ x_1 = s, x_2 = R_2[x_1], x_3 = R_2[x_2], \ldots, x_m = R_2[x_{m-1}], Env\ \texttt{in}\ R[x_m]$) to the update-markers on the stack. If the normal order is operating on s, then all the update-markers are $\#x_1, \#x_2, \#x_3, \ldots \#x_m$ on the stack are, perhaps interspersed with other entries. (lll)-reductions are done by the machine not one-by-one, but always several at once, if a `letrec`-environment is encountered it is always shifted to the top environment.
Using these observations, it is possible to make an induction to show the correspondence between a machine evaluation and an evaluation by the eager-copy-strategy.

□

26 Applications

26.1 Application of the Results to the Core-Language of Haskell

26.1.1 The operators `seq` and `strict` It is clear that the Haskell-operators `seq` can be encoded in FUNDIO, where `s seq t` is transformed into $\texttt{case}\ s\ (Pat_1 \to t)\ \ldots (Pat_N \to t)$, since the pattern `lambda` is available. The operator `strict` is easily encoded using `seq`:

```
strict = \f x -> seq x (f x)
```

A hyperstrict evaluation can also be encoded as follows. We define a function `hyeval` s, which hyperstrictly evaluates its argument The definition is given recursively, but is easily transformed into a `letrec`-expression.

$$\begin{aligned} \texttt{hyeval} = \lambda x.\ &\texttt{case}\ x\ ((c_1\ x_{11} \ldots x_{1n_1}) \\ &\quad \to (\texttt{seq}(\texttt{hyeval}\ x_{11})\ (\texttt{seq}\ (\texttt{hyeval}\ x_{12}) \ldots \\ &\qquad (\texttt{seq}\ (\texttt{hyeval}\ x_{1n_1})\ x) \ldots)\ ))\ \ldots \end{aligned}$$

A hyperstrict normal form of t is a WHNF of the expression `(hyeval t)`. The application of `hyeval` to an argument permits parallel evaluation of the corresponding expressions.

26.2 The Relation between Haskell and FUNDIO

In order to apply the results for Haskell, we take for granted that Haskell expressions can be desugared into a core language. The missing parts we also consider are monadic IO, and the function `unsafePerformIO`. Usages of `unsafePerformIO` are translated into calls using FUNDIO's `IO`.
Two basic IO-functions in Haskell are, where we interpret `IO a` as of type `World -> (a,World)`.

```
getChar :: IO Char          --    World -> (Char,World)
putChar :: Char -> IO ()    --    Char -> (World -> ((),World))
```

The FUNDIO-simulation of `getChar` is a call to `IO` using a dummy-constant, say () as argument, and obtaining a character c as output. This corresponds to the Haskell call `(unsafePerformIO getChar)`. The Haskell call `(unsafePerformIO putChar)` is implemented as a call to `IO` using c as argument, and the result is (). The IO-pair of the first is $((), c)$, the IO-pair of the second is $(c, ())$.

```
-- FUNDIO:
getChar    =  \c -> IO (c~{\tt seq} ~())
putChar    =  \c -> (IO c)
```

26.3 Encoding Sequentialization of Actions

We present an encoding of programming method in FUNDIO to sequentialize the execution of actions, which is derived from monadic programming. Since FUNDIO is untyped, we fix actions to be strict abstractions. The intention is that actions take an argument, communicate with the outer world via `IO`, and then return a result.
In the following we use `case s of x -> t[x]` to abbreviate the case-expression

```
 case s of {p_1 -> t[p_1]; \ldots p_{N} -> t[P_N], lambda ->bot}
```

Instead of the monadic bind, we use the slightly different composition `>@>`, which is in our case almost a strict composition of actions (which are like unary functions).

```
return  x  =  case  x of r -> r
m >@>  k   =  \x -> case m x of r -> k r
```

The definition of `return` is the same as identity, but returns $\bot$ for abstractions. Comparing this with a monadic bind `>>=`, the operator `>@>` can be defined using `>>=` by

```
 m >@>  k   =  \x ->  m x >>= k
```

and bind itself can be encoded as

```
m >>=  k   =   case m of r -> k r
```

26.3.1 Checking the Monoid Laws

We check the monoid laws for the composition `>@>`.
The criterion is to check the following:
For abstractions s, t, check whether $\forall r : s\ r \sim_c t\ r$
We know that this criterion alone is insufficient to show contextual equivalence, but it is a critical test (see Proposition 24.1).
Verifying the laws is done under the following assumptions on actions:

1. actions m are closed abstractions.
2. the actions are strict in their argument
3. $m\ (\lambda y.s) \sim \perp$.

The first law is: `return >@> m` = `m`
We have to check `\a -> case (case a of r' -> r') of {r -> m r)}`. In the case that the argument a is an abstraction or bot, then the result is also bot.

```
      return  >@> m
   ~c \a -> case (case a of r' -> r')
             of  {r -> m r)}            Def. of >@>
   ~c \a -> case a of {r -> m r }       Def. of return, (case,ucp)
   ~c \a -> let r = a   in   m r)       (case,ucp)
   ~c \a ->  m a                        (case,ucp)
   ~c m                                 (η)
```

The second law is `m >@> return` = `m`
Again we assume that the arguments and results of actions are not $\perp$ nor abstractions. In these cases it is easy to see that both sides of the equation are $\perp$.

```
        m >@> return
     ~c \a -> case (m a) of r -> return r
     ~c \a -> case (m a) of r ->  r
     ~c m                                        (to be proved)
```

Now we check associativity of the strict composition:
`m1 >@>  (m2  >@> m3)` = `(m1 >@>  m2)  >@> m3`
First we compute the left hand side, under the assumption that all arguments and results are neither $\perp$ nor abstractions.

```
   \x -> case (m1 x) of (r1 -> (m2  >@> m3) r1)
~c \x1 -> case (m1 x1) of
            (r1 -> (\x2 -> case m2 x2 of (r2 -> m3 r2)) r1)
~c \x1 -> case (m1 x1) of
            (r1 -> (let x2 = r1 in  case m2 x2 of (r2 -> m3 r2)) )
~c \x1 -> case (m1 x1) of
            (r1 -> (case m2 r1 of (r2 -> m3 r2)) )                 ucp
```

We compute the left hand side,

```
     (m1 >@>  m2)  >@> m3
  ~c \x1 -> case (m1 >@>  m2) x1 of (r2 ->  m3 r2)
  ~c \x1 -> case ((\x2 -> case (m1 x2) of
            (r1 ->   m2 r1)) x1 ) of (r2 ->  m3 r2)
  ~c \x1 -> case (let  x2 = x1 in  case (m1 x2) of
            (r1 ->   m2 r1)) of  (r2 ->  m3 r2)
  ~c \x1 -> case (case (m1 x1) of
            (r1 ->   m2 r1)) of  (r2 ->  m3 r2)          cpx, gc
  ~c \x1 ->  case (m1 x1) of
            (r1 ->   case (m2 r1) of   (r2 ->  m3 r2))   case-case
```

This argument uses the rule (case-case) for FUNDIO (see also [JS98]), which can be proved correct using the methods in this paper; for a worked-out proof see [Sab03].
We demonstrate the use of the encoding for two simple IO-functions, where we assume that the input and output are characters or the special constant (), and that `seq` defined as \ s t `->case` s $(Pat_1 \to t)$ $\ldots (Pat_N \to t)$, inlcuding the pattern `lambda`, where we use `seq` as binary infix symbol.

```
getChar    =  \c  ->  (IO (c seq ())
putChar    =  \c  ->  (IO c)
```

First we try `getChar >@>  getChar`

`\x  ->  case (getChar x) of (r ->  getChar r)`
$\sim_c$ `\x  ->  case (IO (x seq ())) of (r ->  (IO (r seq ())))` lbeta, gc

This means that the combined action does indeed two subsequent `getChar`'s , and also in the originally intended sequence.
Now we try to first get a character and put it back:
`getChar >@> putChar`

`\x  ->  case (getChar x) of (r ->  putChar r)`
$\sim_c$ `\x  ->  case (IO (seq x ())) of (r ->  IO r)` lbeta, gc

Example 26.1. The following example in [Jon01] shows that the replacement of equals for equals is no longer valid in Haskell according to the Haskell-semantics, where

```
\w -> case getChar w of
       (c,w1) -> case putChar c w1 of
                   (_,w2) -> putChar c w2
```

is transformed into

```
\w -> case getChar w of
       (c,w1) -> case putChar c w1 of
                   (_,w2) -> putChar (fst (getChar w)) w2
```

using different action combinators. This transformation is invalid, since the `getChar` is called twice after the transformation, whereas it was only called once before the transformation.
This cannot happen using FUNDIO-semantics, since copying the (`getChar w`)-call is not valid in FUNDIO.

The results in this paper allow to prove program transformations and compiler optimizations as correct or to detect incorrect w.r.t. the use of `unsafePerformIO` and the encoding of monadic IO. The results are consistent with the observations and proofs in [AS98].
A program transformation that is correct in FUNDIO is also correct in Haskell-core, and hence in Haskell.

If only a subset of Haskell-core (say the expressions that are type-correct in some sense) are considered, then there may be correct program transformations in Haskell-core that are not correct in FUNDIO. However, presumably, these are rare cases. Usually, the FUNDIO-counterexamples to equivalence can also be translated to Haskell-core.

26.4 Application to Strict Functional Programming Languages

26.4.1 Embedding a strict functional language in FUNDIO The language FUNDIO can also be used to justify optimizations and parallel evaluation in strict functional programming languages, like LISP, ML and Scheme, in particular it is also possible to justify rearrangements of input/output sequences. This is only possible, if the semantics of the strict language does not insist on the sequence of IOs. Of course, if sequentialization of I/O is enforced by data dependency, then this sequence cannot be changed.
The results of FUNDIO are applicable to strict functional programming languages as follows:
Let F_{strict} be a strict functional core language, the syntax being the same as the FUNDIO-syntax, however, the evaluation mechanism is such that

- every abstraction first evaluates the argument strictly before applying (lbeta)
- Every constructors is strict, i.e., evaluates the arguments before constructing the constructor application.
- Every `let` is strict, i.e. first all bindings are evaluated.

The effect is that expressions are hyperstrictly evaluated, since every application is strict.
Now we can also speak of a contextual preorder in F_{strict}.
Let $FUNDIO_{strict}$ by the sublanguage of FUNDIO generated from F_{strict} as follows. Given an expression r from F_{strict}, we have to ensure that `let`s are strict, that abstractions are strict and that constructors are strict. This is done by plugging in a `seq` at appropriate places:

1. To make every abstraction t strict, it has to be replaced by $(\lambda x.x\ \texttt{seq}\ (t\ x))$.

2. To make every constructor c strict, constructor applications $(c\ t_1 \ldots t_n)$ have to be replaced by an application $(c'\ t_1 \ldots t_n)$, where $c' = \lambda x_1, \ldots, x_n.x_1\ \texttt{seq}\ \ \ldots\ \texttt{seq}\ x_n\ \texttt{seq}\ (c\ x_1 \ldots x_n)$.
3. To make every `letrec` strict, every `letrec`-expression t has to be translated as a strict `letrec`:
   `letrec` $x_1 = t_1,\ \ldots, x_n = t_n$ `in` s
   is translated into
   t_1 `seq` t_2 `seq` t_n `seq` (`letrec` $x_1 = t_1,\ \ldots, x_n = t_n$ `in` s)
   This ensures that every binding will be evaluated.

Note that (lbeta) does only create new bindings that are already evaluated.

We have embedded F_{strict} into FUNDIO, with the following conclusions. The contextual equivalence w.r.t. full FUNDIO in FUNDIO_{strict} is more restrictive than in F_{strict}, but expressions that are equal w.r.t. FUNDIO are also equivalent in F_{strict}.

Example 26.2. In F_{strict} the two functions

```
p1 = \f -> bot
p2 = \f -> f bot
```

cannot be distinguished, since they can only be applied to abstractions that are strict. In FUNDIO, these are easily distinguished by applying them to $\lambda x.\mathtt{True}$.

Though it appears that the sequence of evaluation is fixed in F_{strict}, this is not the case. The equivalence of expressions in F_{strict} is the essential invariant, not the sequence of evaluation. For example, the closed expression (s `seq` t) can also be evaluated by first evaluating t, then s, and then returning the value of t. If (s `seq` t) is open, or a subexpression in another expression, then there may be data dependencies, which prevent this evaluation.

26.4.2 A correct program transformations in strict functional languages

A FUNDIO-term t is in *hyperstrict normalform*, if it is an abstraction, or a constructor application, and all arguments are in hyperstrict normal form.

Proposition 26.3. *Let t be a FUNDIO-term in hyperstrict normal form. Then*

$$\begin{array}{ll}(\mathtt{letrec}\ x = t, Env\ \mathtt{in}\ C[x]) & \rightarrow (\mathtt{letrec}\ x = t, Env\ \mathtt{in}\ C[t])\\ (\mathtt{letrec}\ x = t, x' = C[x], Env\ \mathtt{in}\ t') & \rightarrow (\mathtt{letrec}\ x = t, x' = C[t], Env\ \mathtt{in}\ t')\end{array}$$

are correct program transformations.

Proof. The correctness of the rule follows from correctness of the rules (cpcx) in 14.7, (gc), and (cp). □

Hence it is also FUNDIO-correct to substitute hyperstrictly evaluated arguments into the body of functions, since they only consist of constructors and abstractions.

26.5 XML and XQuery

The query language XPath/XQuery to make queries to XML-documents is a functional language (see [w3-03a,w3-03b]), which is defined as a call-by-value language without side-effects. Adopting the FUNDIO-semantics would allow to also have a lazy functional language which also can perform updates and other side-effects.

The advantage of using a lazy variant of a functional language is that a lazy read of a file is easily possible.

26.6 A Warning for Practical Programming Languages

The contextual equivalence of FUNDIO does not guarantee the following: If there is a sequence of two IOs:

1. write 8 to external storage location a.
2. read the value from external storage location a.

and there is no data dependency between the two IO's, then FUNDIOs semantics may justify to parallelize the IOs or to perform them in the other order: first 2, then 1.

Hence a pragmatics would be to sequentialize IOs by data dependency or by a monadic method to sequentialize actions, if the programmer wants a fixed order for some reason.

27 Extensions and Parallel Evaluation

An extension to FUNDIO would be to allow a more general IO-interface that also accepts a list of arbitrary (but finite) length of constants, which are to be hyperstrictly evaluated to a lambda-free term. After this hyperstrict evaluation the IO-operation may start.

The only difference to FUNDIO appears to be that this would allow for an infinite number of different outputs and inputs. This is only a slight change in the contextual equivalence of expressions.

28 Conclusion and Future Work

We introduced the call-by-need lambda calculus FUNDIO for combining (IO) and lazy functional programming, gave a definition of contextual equality and argued that there is a rich equational theory. We also showed for a selection of program transformations that they are correct. The formalism has a high potential of serving as a theory of program transformations for strict and non-strict functional programming languages with direct-call IO, e.g. Haskell with `unsafePerformIO`, the Clean language, and also strict functional programming languages with a declarative IO.

Induction for recursive functions should be investigated, there appears to be no obstacle. It would also be a challenge to adapt the work on improvements [San98,MS99] to the FUNDIO semantics.

The generalization of a behavioral preorder $\leq_b$ has to be explored, i.e., an appropriate definition to be a found and then to be proved equivalent to $\leq_c$.

The reduction diagrams are verified by hand. We feel that it is necessary to apply the work in [Hub00] also to FUNDIO, which would then provide a mechanical check and test of the diagrams.

29 Acknowledgements

I would like to thank David Sabel and Matthias Mann for their constructive comments. I would particularly thank Arne Kutzner for his contributions to the abstract machine model for FUNDIO.

References

[ABB$^+$99] Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98: A non-strict, purely functional language, February 1999.

[AC79] Egidio Astesiano and Gerardo Costa. Sharing in nondeterminism. In *Proc. 6th ICALP 79*, pages 1–15, 1979.

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J Lévy. Explicit substitutions. *J. functional programming*, 4(1):375–416, 1991.

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

[AF97] Z.M. Ariola and M Felleisen. The call-by-need lambda calculus. *J. functional programming*, 7(3):265–301, 1997.

[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*, pages 233–246, San Francisco, California, 1995. ACM Press.

[AHH$^+$02] Elvira Albert, Michael Hanus, Frank Huch, Javier Oliver, and Germán Vidal. Operational semantics for lazy functional logic languages. In *ENTCS*, volume 76. Elsevier, 2002. Also available from `http://www.elsevier.nl/locate/entcs/volume76.html`.

[AS98] Zena M. Ariola and Amr Sabry. Correctness of monadic state: An imperative call-by-need calculus. In *POPL 98*, pages 62–74, 1998.

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

[Bro86] Manfred Broy. A theory for nondeterminism, parallelism, communication, and concurrency. *TCS*, 45:1–61, 1986.

[CG94] Roy L. Crole and Andrew D. Gordon. A sound metalogical semantics for input/output effects. In *Proceedings Computer Science Logic 94*, LNCS 933, pages 339–353. Springer-verlag, 1994.

[Cli82] William Clinger. Nondeterministic call by need is neither lazy nor by name. In *ACM Symp. on Lisp and Functional Programming*, pages 226–234. ACM, 1982.

[CMW96] John N. Crossley, Luis Mandel, and Martin Wirsing. First-order constrained lambda calculus. In *FroCos 1996*, pages 339–356, 1996.

[DCdP94] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Adolfo Piperno. Fully abstract semantics for concurrent lambda-calculus. *TACS*, pages 16–35, 1994.

[DCTU99] Mariangiola Dezani-Ciancaglini, Jerzy Tiuryn, and Pawel Urzyczyn. Discrimination by parallel observers: The algorithm. *Information and Computation*, 150(2):153–186, 1999.

[DP95] U. De'Liguoro and A. Piperno. Nondeterministic extensions of untyped λ-calculus. *Information and Computation*, 122:149–177, 1995.

[Fel78] Matthias Felleisen. *The calculi of Lambda-ν-cs Conversion: A syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Indiana UNiversity, 1978.

[FH92] Matthias Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *TCS*, 103:235–271, 1992.

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

[GP98] Andrew D. Gordon and Andrew D. Pitts. *Higher Order Operational Techniques in Semantics.* Cambridge University Press, 1998.

[HM92] J. Hughes and A. Moran. A semantics for locally bottom-avoiding choice. In *Proc. Glasgow functional programming workshop 1992*, Workshops in Computing. Springer-Verlag, 1992.

[HNSSH97] N.W.O. Hutchison, U. Neuhaus, M. Schmidt-Schauß, and C.V. Hall. Natural Expert: A commercial functional programming environment. *J. of Functional Programming*, 7(2):163–182, 1997.

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

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

[Hub00] Michael Huber. JONAH: Ein System zur Validierung von Reduktionsdiagrammen in nichtdeterministischen Lambda-Kalkülen mit let-Ausdrcken, letrec-Ausdrcken und Konstruktoren. Master's thesis, Fachbereich Informatik, J.W.Goethe-Universität Frankfurt, 2000. Diplomarbeit (Master thesis), in German.

[JME99] Simon Peyton Jones, Simon Marlow, and Conal Elliot. Stretching the storage manager: weak pointers and stable names in Haskell. In *IFL 99*, pages 37–58, 1999.

[Jon01] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Ralf Steinbruggen Tony Hoare, Manfred Broy, editor, *Engineering theories of software construction*, pages 47–96. IOS-Press, 2001. Presented at the 2000 Marktoberdorf Summer School.

[Jon03] Simon L. Peyton Jones. Wearing the hair shirt: a retrospective on haskell, 2003. Slides of invited talk at POPL'03.

[JS98] 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.

[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.

[Kut00] Arne Kutzner. *Ein nichtdeterministischer call-by-need Lambda-Kalkül mit erratic choice: Operationale Semantik, Programmtransformationen und Anwendungen.* Dissertation, J.W.Goethe-Universität Frankfurt, 2000. in german.

[Las98] Søren Bøgh Lassen. *Relational Reasoning about Functions and Nondeterminism.* PhD thesis, Faculty of Science, University of Aarhus, 1998.

[Liu93] F. Liu. Towards lazy evaluation, sharing and non-determinism in resolution based functional logic languages. In *PFPCA*, pages 201–209, 1993.

[LP00] Søren B. Lassen and Corin S. Pitcher. Similarity and bisimilarity for countable non-determinism and higher-order functions. *Electronic Notes in Theoretical Computer Science*, 10, 2000.

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

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

[MOW98] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *J. of Functional programming*, 8:275–317, 1998.

[MS94] Andy Mück and Thomas Streicher. A tiny constraint functional logic language and its continuation semantics. In *ESOP 1994*, pages 439–453, 1994.

[MS99] A.K.D. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *POPL 1999*, pages 43–56. ACM Press, 1999.

[MSC99] A.K.D. Moran, D. Sands, and M. Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, volume 1594 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[MST96] Ian Mason, Scott F. Smith, and Carolyn L. Talcott. From operational semantics to domain theory. *Information and Computation*, 128:26–47, 1996.

[MT91] Ian Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *Journal of functional programming*, 1(3):287–327, 1991.

[Nöc93] Eric Nöcker. Strictness analysis using abstract reduction. In *Functional Programming Languages and Computer Architecture*, pages 255–265. ACM Press, 1993.

[Ong93] C.-H. L. Ong. Non-determinism in a functional setting. In *Proc. 8th IEEE Symposium on Logic in Computer Science (LICS '93)*, pages 275–286. IEEE Computer Society Press, 1993.

[Pit97] Andrew D. Pitts. Operationally-based theories of program equivalence. In *Semantics and Logics of Computation.* Cambridge University Press, 1997.

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

[PJGF96] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *Proc. 23th Principles of Programming Languages, St. Petersburg Beach, Florida*, 1996.

[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.

[Plo76] G.D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5(3):452–487, 1976.

[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.

[PS98] A.M. Pitts and Ian Stark. *Operational Reasoning for functions with local state*, pages 227–273. Cambridge university press, 1998.

[Sab03] David Sabel. Realisierung der Ein-/Ausgabe in einem Compiler für Haskell bei Verwendung einer nichtdeterministischen Semantik. Master's thesis, Institut für Informatik, J.W.Goethe-Universität Frankfurt, 2003. Diplomarbeit, (to appear).

[San98] David Sands. *improvement theory and its applications*, pages 275–306. Cambridge university press, 1998.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *J. of functional programming*, 7(3):231–264, 1997.

[Smy78] M.B. Smyth. Power domains. *J. of Computer and System Sciences*, 16(3):23–35, 1978.

[SS92] H. Søndergard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.

[SSPS95] Manfred Schmidt-Schauß, Sven Eric Panitz, and Marko Schütz. Strictness analysis by abstract reduction using a tableau calculus. In *Proc. of the Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 348–365. Springer-Verlag, 1995.

[vEGHN93] M. van Eekelen, E. Goubault, C.L. Hankin, and E. Nöcker. Abstract reduction: Towards a theory via abstract interpretation. In M.R. Sleep, M.J. Plasmeijer, and M.C.J.D. van Eekelen, editors, *Term Graph Rewriting - Theory and Practice*, chapter 9. Wiley, Chichester, 1993.

[w3-03a] Xquery 1.0: An xml query language, 2003. URL = `http://www.w3.org/TR/xquery/`.

[w3-03b] Xquery 1.0 and xpath 2.0 formal semantics, 2003. URL = `http://www.w3.org/TR/xquery-semantics/`.

[Wad92] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[Wad95] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 24–52. Springer, 1995.

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