Math. Struct. in Comp. Science (2008), *vol.* 18, *pp.* 501–553. © 2008 Cambridge University Press
doi:10.1017/S0960129508006774 Printed in the United Kingdom

A call-by-need lambda calculus with locally bottom-avoiding choice: context lemma and correctness of transformations

DAVID SABEL and MANFRED SCHMIDT-SCHAUSS

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

Received 13 January 2006; revised 5 January 2007

We present a higher-order call-by-need lambda calculus enriched with constructors, `case` expressions, recursive `letrec` expressions, a `seq` operator for sequential evaluation and a non-deterministic operator `amb` that is locally bottom-avoiding. We use a small-step operational semantics in the form of a single-step rewriting system that defines a (non-deterministic) normal-order reduction. This strategy can be made fair by adding resources for book-keeping. As equational theory, we use contextual equivalence (that is, terms are equal if, when plugged into any program context, their termination behaviour is the same), in which we use a combination of may- and must-convergence, which is appropriate for non-deterministic computations. We show that we can drop the fairness condition for equational reasoning, since the valid equations with respect to normal-order reduction are the same as for fair normal-order reduction. We develop a number of proof tools for proving correctness of program transformations. In particular, we prove a context lemma for both may- and must- convergence that restricts the number of contexts that need to be examined for proving contextual equivalence. Combining this with so-called complete sets of commuting and forking diagrams, we show that all the deterministic reduction rules and some additional transformations preserve contextual equivalence. We also prove a standardisation theorem for fair normal-order reduction. The structure of the ordering $\leqslant_c$ is also analysed, and we show that Ω is not a least element and $\leqslant_c$ already implies contextual equivalence with respect to may-convergence.

1. Introduction

1.1. *Motivation*

Higher-order lambda calculi with non-deterministic operators have been investigated by several authors. In particular, a non-deterministic choice operator that chooses one of its arguments as result, but converges if one of its arguments is reducible to a value, is of relevance for modelling concurrent computation. It enables search algorithms to be expressed in a natural way (Henderson 1980; Bois *et al.* 2002), and allows the implementation of a `merge` operator for streams in event-driven systems such as, for example, graphical user interfaces (Hallgren and Carlsson 1995) or functional operating systems (Henderson 1982).

McCarthy's *amb* (McCarthy 1963) is such a non-deterministic operator. A typical implementation starts two concurrent (or parallel) processes, one for each of its arguments, and then chooses the first one that terminates. The operator *amb* is bottom avoiding: if $\perp$ is an expression that cannot converge and s is a value, the expressions $(amb\ s\ \perp)$ and $(amb\ \perp\ s)$ both evaluate to s. The bottom avoidance of *amb* is only *local* as its evaluation is independent of the surrounding context: for example, the expression[†] $(\texttt{if}\ (amb\ \texttt{True}\ \texttt{False})\ \texttt{then}\ \texttt{True}\ \texttt{else}\ \perp)$ is may-divergent.

There have been many investigations into the properties of higher-order calculi with amb using call-by-value, call-by-name or call-by-need evaluation strategies. The strategies can be distinguished by their internal treatment of function arguments that contain amb expressions. Call-by-name tends to copy arguments, even if an argument is, for example, of the form $(\texttt{amb}\ s\ t)$, and thus implements plural non-determinism (Søndergaard and Sestoft 1992), whereas call-by-value and call-by-need are more restrictive in copying arguments: only values, in particular abstractions, are allowed to be copied. Thus, these evaluation strategies implement singular non-determinism.

We will now provide some justification for the calculus we will investigate, which is fairly expressive, has a variety of constructs and many reductions, and is close to a real-world calculus. A locally bottom-avoiding choice combined with constructs for explicit sharing and sequential evaluation enables us to define many other non-deterministic operators within the language, for example, erratic choice, locally demonic choice (see Søndergaard and Sestoft (1992) for an overview of different non-deterministic operators) and a parallel operator that evaluates both of its arguments in parallel and returns both values as a pair – for example, Jones and Hudak (1993) uses such an operator. A further reason for the expressiveness is that we want the results to be immediately applicable to an implementation of amb in a variant of Haskell (Sabel 2003a). This means that the calculus must respect sharing and must be a call-by-need calculus that implements singular non-determinism, as already argued in Moran (1998). Another reason is that using a less expressive language to prove equalities of expressions is of limited value, since there is no guarantee that equations remain valid if the language is extended, since extensions of the language may increase the expressiveness of contexts and thus invalidate equalities. Also, the verification of the properties of programs and of compilers for a higher-order calculus with amb requires an exact semantics as a solid foundation, as well as methods for proving non-trivial properties.

For all these reasons, we investigate a higher-order lambda-calculus with an operator amb, (weakly) typed case, constructors, letrec and seq. The letrec expressions are used for explicit sharing of terms as well as for describing recursive definitions. The binary operator seq evaluates to its second argument if and only if its first argument converges, otherwise the whole seq expression diverges.

We will define a small-step operational semantics as a rewriting system on expressions together with a strategy. An unwinding mechanism determines all permitted subterms for the next reduction, which are called normal-order redexes. A normal-order reduction

[†] $\texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ t$ can be encoded in our calculus as $\texttt{case}_{Bool}\ b\ (\texttt{True} \rightarrow s)\ (\texttt{False} \rightarrow t)$.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

sequence is one that only reduces normal-order redexes. For amb-free expressions this will be deterministic, since normal-order redexes are unique, while for expressions containing occurrences of amb there may be several normal-order redexes. A single normal-order reduction sequence is a reduction that in every step non-deterministically chooses one of the concurrently possible subexpressions for reduction. The set of all normal-order reduction sequences also comprises reductions such that amb is not locally bottom-avoiding. We add resources in Definition 2.16 to normal-order reductions to single out the fair normal-order reductions that all have the property that amb is locally bottom-avoiding.

Using all normal-order reductions as a basis, we use as equational theory *contextual equivalence* (also known as observational equivalence), which equates two terms if both their termination and non-termination behaviours are the same in all program contexts. It is well known that only taking *may-convergence* into account is too weak for calculi with an amb operator (see, for example, Moran (1998)), so our equivalence will also test for *must-convergence*. Summarising, contextual equivalence $\sim_c$ is the symmetrisation of the contextual preorder $\leqslant_c$, where

$$s \leqslant_c t \text{ iff } (\forall C : C[s]{\downarrow} \implies C[t]{\downarrow} \text{ and } \forall C : C[s]{\Downarrow} \implies C[t]{\Downarrow})$$

with C denoting contexts and using $\downarrow$ and $\Downarrow$ for the predicates for may- and must-convergence, respectively. Note that our predicate for must-convergence is the converse of the one for may-divergence.

In Theorem 2.21 we show that the notions of may- and must-convergence are the same for both normal-order and fair normal-order reductions, which implies that the corresponding notions of contextual equivalence are identical. This will be a great simplification in the following, since normal-order reduction is sufficient for all arguments concerning equivalence and the correctness of program transformations. For a call-by-name calculus with amb the same coincidence was shown in Carayol *et al.* (2005).

In contrast to Hughes and Moran (1995) and Moran (1998), we only treat those divergences that are called *strong* in Carayol *et al.* (2005), using the distinction between strong and weak divergence introduced in Natarajan and Cleaveland (1995). A consequence of this is that a term that has an infinite reduction but never loses the ability to converge is not divergent.

Proving contextual equivalence directly seems to be very hard, since all program contexts have to be taken into account. Other methods, like bisimulation, for proving contextual equivalence have not been successful for call-by-need calculi with amb (see Moran (1998) for a discussion). Mann (2005a), Mann (2005b) and Mann and Schmidt-Schauss (2006) have shown that bisimulation can be used as a proof tool for a call-by-need calculus with non-recursive let and erratic choice, but there seems to be no obvious way to transfer this result to call-by-need calculi with recursive let, since their approach requires the elimination of let bindings, and the elimination method does not work for recursive lets (*cf.* Mann (2005a, Section 6.2)).

In this paper we will use the powerful technique of combining a context lemma for both may- and must-convergence with complete sets of forking and commuting diagrams to prove that the deterministic reductions of the calculus are correct program

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

transformations (Theorem 7.1), that is, their application preserves contextual equivalence. This is of importance in a compiler that uses reductions as optimisations, in particular, if partial evaluation is used. We will also show the correctness of some other program transformations that are used for optimisation in compilers of functional programming languages (see, for example, Santos (1995)).

An important result is the Standardisation Theorem (Theorem 7.13), which states that if there exists a sequence of transformations or reductions to a weak head normal form, then there is also a fair evaluation in normal order to a weak head normal form (see Corollary 7.14). The second part of the Standardisation Theorem states that if there exists a sequence of transformations inside surface contexts (that is, not within abstractions) resulting in a term that cannot converge, then fair normal-order reduction can also reduce to such a term. A consequence of the Standardisation Theorem is that must-convergence is preserved while applying program transformations inside surface contexts (see Proposition 7.15).

Using the Standardisation Theorem, we first show that all (closed) must-divergent terms are in the same equivalence class with respect to $\sim_c$. We then show a classical bottom-avoidance law of our amb operator (Proposition 8.17), that is, $\texttt{amb}\ \Omega\ t \sim_c t$ and $\texttt{amb}\ t\ \Omega \sim_c t$, where Ω is a term that cannot converge. As a final result, we show that contextual equivalence can be defined by just taking must-convergence into account (Corollary 8.22), that is, $s \sim_c t$ if and only if $\forall C : C[s]\Downarrow \Leftrightarrow C[t]\Downarrow$.

Our results show that it is possible to overcome some difficulties that Moran encountered in his thesis (Moran 1998). In particular, we were able to prove the full context lemma and show the correctness of several transformation rules for the full $\sim_c$-equivalence, where Moran had to confine himself mostly to may-convergence. To our knowledge, this is the first paper that proves the correctness of program transformations for a call-by-need calculus with amb including may- and must-convergence in the definition of correctness. Nevertheless, there still remains the open problem of exhibiting a bisimulation-based proof tool or even a bisimulation characterisation of contextual equivalence for call-by-need calculi with amb and recursive bindings.

An implementation of evaluation can be done by implementing normal-order reduction, taking care that only fair reductions are possible. Note that the unrestricted definition of normal-order reduction permits unfair reduction sequences: that is, the evaluation of an expression ($\texttt{amb}\ \bot\ \texttt{True}$) may reduce $\bot$ infinitely often while ignoring the fact that the second argument of amb is already a value. Fair reduction strategies can be achieved using Moran's approach (Moran 1998) to implement fair evaluation by annotating amb expressions with resources for both of its arguments, and decreasing the resource of an argument for every reduction step that is related to this argument. The (fair) scheduler increases the resources only if both resources are 0 and the increase is greater than 0 for both arguments.

Carayol *et al.* (2005) defines a fair operational semantics for its call-by-name calculus, which is free of resource annotations. We considered adopting this method for our investigation, but, unfortunately, it does not work properly for a calculus with shared bindings (see Section 2.6).

1.2. *Related work*

To our knowledge, the only papers addressing call-by-need calculi with locally bottom-avoiding choice are Hughes and Moran (1995) and Moran (1998). The work described in Moran (1998) is closely related to ours, since he also considers a call-by-need calculus with an *amb* operator. His syntax is similar to ours, though there are some small differences: in particular, he uses strict `let` expressions, where we use lazy `let` and a `seq` operator for implementing sequential evaluation, and we use (weakly) typed `case` expressions, while Moran (1998) uses an untyped `case`. Moran also uses contextual equivalence, but our equational theory differs from his, since his predicate for must-convergence captures weak divergences, and thus is not the same as our predicate (see Example 3.4).

Moran was not able to show correctness of program transformations with respect to contextual equivalence that takes may- and must-convergence into account. He only provides a context lemma (which is based on improvement theory (Moran and Sands 1999)) for the may-convergence part, but failed to prove a context lemma for must-convergence. This is unsatisfactory, since it is precisely the must-convergence that distinguishes an `amb` operator from erratic choice.

The technical advantage of our approach over Moran's is that we do not need an explicit heap, and that for large parts of the reasoning we can ignore the resource annotations of `amb`. This may be why we were able to prove a context lemma for both may- and must-convergence, as well as the correctness of several program transformations.

Our contextual preorder is similar to that of Carayol *et al.* (2005) for a call-by-name calculus with `amb`, since Carayol *et al.* (2005) also tests for strong divergences only. Call-by-name lambda calculi with `amb` operators are also treated in Hughes and Moran (1995), Lassen and Moran (1999), Moran (1998) and Lassen (2006), but, in the same way as Moran did in Moran (1998), with their call-by-need calculus, they test for weak divergences in their contextual equivalence.

There is other work on call-by-need calculi with other choice operators, especially erratic choice, and we can compare some of them with our approach. The calculi of Schmidt-Schauss *et al.* (2004) and Schmidt-Schauss (2003) are strongly related to our calculus, since both provide recursive `let` expressions and `case`, as well as constructors and unrestricted applications. Moran and Sands (1999) does not have the last of these, since they only allow variables as arguments. While Schmidt-Schauss *et al.* (2004) only uses (may-) convergence for the definition of contextual equivalence, Moran and Sands (1999) and Schmidt-Schauss (2003) also use predicates for divergence. Schmidt-Schauss (2003) uses a combination of contextual equivalence together with a trace semantics, where again only strong divergences are considered.

The proof technique of complete sets of commuting and forking diagrams was introduced by Kutzner and Schmidt-Schauss (1998) and Kutzner (2000) for a call-by-need lambda calculus with erratic choice and a non-recursive `let`. The same technique has also been used in Schmidt-Schauss (2003), Schmidt-Schauss *et al.* (2004) and Mann (2005a) for their call-by-need calculi with erratic choice. Kutzner (2000), Schmidt-Schauss (2003), Schmidt-Schauss *et al.* (2004) and Mann (2005a) use a normal-order reduction as small-step semantics – of these, Schmidt-Schauss *et al.* (2004) is the most similar to ours, while Moran and Sands (1999) uses an abstract machine semantics.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Diagram-based techniques for proving program equivalences have already been used in Machkasova and Turbak (2000) to prove meaning preservation in a call-by-value calculus with recursive definitions. Wells *et al.* (2003) provides an abstract framework for proving meaning preservation using diagrams. Unfortunately, for several reasons, the axioms of this framework do not apply in our calculus: one reason is that reduction in our calculus is non-deterministic, so evaluation does not preserve meaning in the sense of Wells *et al.* (2003).

Work on call-by-value calculi extended with bottom-avoiding choice has been reported in Lassen (1998), and Fernández and Khalil (2003) investigated interaction nets extended with an *amb* operator.

1.3. *Overview*

In Section 2 we introduce the $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$ calculus, define the convergence predicates and introduce a fair evaluation strategy. In Section 3 we define the contextual preorder and contextual equivalence, prove a context lemma, show some important properties of reduction rules that rearrange `letrec` environments, and, finally, introduce the notion of complete sets of commuting and forking diagrams. In Section 4 we prove the correctness of those reduction rules where correctness follows easily, define some general properties of a program transformation and show that their validity ensures the correctness of the transformation. Equipped with this proof tool, we go on in Sections 5 and 6 to prove the correctness of all defined deterministic reduction rules and of some additional program transformations. In Section 7 we prove the Standardisation Theorem. In Section 8 we show the equivalence of must-divergent terms and prove the bottom-avoidance law. We conclude Section 8 by proving some remarkable properties of the contextual preorder we use: in particular, we prove that the contextual preorder implies equivalence with respect to may-convergence. In the final section we give some conclusions and suggest some directions for further research.

2. The non-deterministic call-by-need calculus $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$

In this section we first introduce the syntax of the language $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$, then define the reduction rules and normal-order reduction. After presenting encodings of other parallel and non-deterministic operators, we define different predicates for convergence and divergence. The section ends by showing that the same predicates are induced if we use a fair reduction strategy.

2.1. *The syntax of the language*

The language $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$ is very similar to the abstract language used in Schmidt-Schauss *et al.* (2004), except that $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$ uses a bottom-avoiding choice operator `amb` while Schmidt-Schauss *et al.* (2004) uses erratic choice. The language of the non-deterministic call-by-need lambda calculus of Moran (1998) is also similar to ours, except that we use an operator

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

`seq` to provide sequential evaluation instead of strict `let` expressions, and our `case` expressions are weakly typed. Unlike the call-by-need calculus of Ariola *et al.* (1995), $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$ provides constructors, weakly typed `case` expressions and, of course, a non-deterministic `amb` operator. The language we use also has only small differences (aside from the `amb` operator) compared with the core language of Peyton Jones and Marlow (2002), which is used in the Glasgow Haskell Compiler.

The language $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$ has the following syntax. There is a finite set of constructors, which is partitioned into (non-empty) types. For every type T we use $c_{T,i}, i = 1, \dots, |T|$ to denote the constructors. Every constructor has an arity $\mathrm{ar}(c_{T,i}) \geqslant 0$.

The syntax for expressions E, case alternatives Alt and patterns Pat is defined by the following grammar:

$$
\begin{array}{lcll}
E & ::= & V & (\textit{variable}) \\
 & \mid & (c_{T,i}\ E_1 \dots E_{\mathrm{ar}(c_{T,i})}) & (\textit{constructor application}) \\
 & \mid & (\mathtt{seq}\ E_1\ E_2) & (\mathtt{seq}\ \textit{expression}) \\
 & \mid & (\mathtt{case}_T\ E\ Alt_1 \dots Alt_{|T|}) & (\mathtt{case}\ \textit{expression}) \\
 & \mid & (E_1\ E_2) & (\textit{application}) \\
 & \mid & (\mathtt{amb}\ E_1\ E_2) & (\mathtt{amb}\ \textit{expression}) \\
 & \mid & (\lambda V.E) & (\textit{abstraction}) \\
 & \mid & (\mathtt{letrec}\ V_1 = E_1, \dots V_n = E_n\ \mathtt{in}\ E) & (\mathtt{letrec}\ \textit{expression}) \\
 & & \quad \text{where } n \geqslant 1 & \\
Alt & ::= & (Pat\ \rightarrow\ E) & (\mathtt{case}\ \textit{alternative}) \\
Pat & ::= & (c_{T,i}\ V_1 \dots V_{\mathrm{ar}(c_{T,i})}) & (\textit{pattern})
\end{array}
$$

In addition to the above grammar, the following syntactic restrictions must hold for expressions:

- E, E_i are expressions and V, V_i are variables.
- Within a pattern, the variables $V_1 \dots V_{\mathrm{ar}(c_{T,i})}$ are pairwise disjoint.
- In a $\mathtt{case}_T$ expression, for every constructor $c_{T,i}, i = 1, \dots, |T|$, of type T, there is exactly one `case` alternative.
- The bindings of a `letrec` expression form a mapping from variable names to expressions, in particular, that means that the variables on the left-hand side of the bindings are all distinct and that the bindings of `letrec` expressions are commutative, that is, `letrec` expressions with permuted bindings are *syntactically equivalent*.
- `letrec` is recursive, that is, in ($\mathtt{letrec}\ x_1 = s_1, \dots, x_n = s_n\ \mathtt{in}\ t$), the scope of x_i, $1 \leqslant i \leqslant n$, is $s_1, \dots, s_n$ and t.
- We use the distinct variable convention, that is, all bound variables in expressions are assumed to be distinct, and free variables are distinct from bound variables. The reduction rules defined in later sections are assumed to rename bound variables implicitly in the result by α-renaming, if necessary, to obey this convention.

To abbreviate the notation, we will sometimes use:

- the word *term* synonymously for expressions;
- $(\mathtt{case}_T\ E\ alts)$ instead of $(\mathtt{case}_T\ E\ Alt_1 \dots Alt_{|T|})$;

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

- (`letrec` Env `in` E) instead of (`letrec` $x_1 = E_1, \dots x_n = E_n$ `in` E) – this will also be used freely for parts of the bindings;
- $(c_i\ \overrightarrow{s_i})$ instead of $(c_i\ s_1\ \dots\ s_{\mathrm{ar}(c_i)})$;
- $\{x_{f(i)} = s_{g(i)}\}_{i=j}^{n}$ for the chain $x_{f(j)} = s_{g(j)}, x_{f(j+1)} = s_{g(j+1)}, \dots, x_{f(n)} = s_{g(n)}$ of `letrec` bindings, where $f, g : \mathbb{N}_0 \to \mathbb{N}_0$;
- we assume application to be left-associative, that is, we write $(s_1\ s_2 \dots\ s_n)$ instead of $((s_1\ s_2)\ \dots\ s_n)$.

Since we already use $=$ as a symbol in the syntax of the language, we use $\equiv$ to denote syntactical equivalence of expressions.

Sometimes we will use tree addresses (*positions*) as strings of positive integers in expressions with their standard meaning. Then the *depth* of a subterm t' of t at position p is the length of the string p.

Definition 2.1. A *value* is either an abstraction or a constructor application.

In the following, we define different contexts, where we use different fonts for sets of contexts and individual contexts.

Definition 2.2 (Context). A context C is a term with exactly one hole, where the hole is not in the variable position of abstractions, in a pattern of a $\mathtt{case}_T$ alternative, or on the left-hand side of a `letrec` binding. We use $[\cdot]$ to denote the hole and $\mathcal{C}$ to denote the set of all contexts.

The *main depth* of a context C is the depth of the hole of the context C, that is, the length of the position of the hole. We use $C_{\#i}$ to denote a context of main depth i. Letting t be a term and C be a context, $C[t]$ is the result of replacing the hole of C with term t. Two contexts C_1, C_2 are called *disjoint* if and only if the positions p_1, p_2 of their respective holes are not prefixes of each other, that is, neither p_1 is a prefix of p_2 nor p_2 is a prefix of p_1.

Definition 2.3 (Reduction contexts). *Reduction* (respectively, *weak reduction*) *contexts*, the set of which is denoted $\mathcal{R}$ (respectively, $\mathcal{R}^-$), are defined as follows:

$$\begin{array}{lll}
\mathcal{R}^- & ::= & [\cdot] \mid (\mathcal{R}^-\ E) \mid (\mathtt{case}_T\ \mathcal{R}^-\ alts) \mid (\mathtt{seq}\ \mathcal{R}^-\ E) \mid (\mathtt{amb}\ \mathcal{R}^-\ E) \mid (\mathtt{amb}\ E\ \mathcal{R}^-) \\
\mathcal{R} & ::= & \mathcal{R}^- \mid (\mathtt{letrec}\ Env\ \mathtt{in}\ \mathcal{R}^-) \\
 & & \mid (\mathtt{letrec}\ x_1 = \mathcal{R}_1^-, x_2 = \mathcal{R}_2^-[x_1], \dots, x_j = \mathcal{R}_j^-[x_{j-1}], Env\ \mathtt{in}\ \mathcal{R}^-[x_j]) \\
 & & \quad \text{where } j \geqslant 1 \text{ and } \mathcal{R}^-, \mathcal{R}_i^-, i = 1, \dots, j \text{ are weak reduction contexts.}
\end{array}$$

For a term t with $t \equiv R^-[t_0]$ where R^- is a weak reduction context, we say R^- is *maximal* (*for* t) if there is no larger non-disjoint weak reduction context for t, that is, there is no weak reduction context R_1^- with $t \equiv R_1^-[t_1']$ where $t_1' \not\equiv t_0$ is a subterm of t_0.

For a term t with $t \equiv R[t_0]$, we say R is a *maximal reduction context (for t)* if and only if R is:

- a maximal weak reduction context; or
- of the form (`letrec` $x_1 = E_1, \dots, x_n = E_n$ `in` R^-) where R^- is a maximal weak reduction context and $t_0 \not\equiv x_j$ for all $j = 1, \dots, n$; or

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

— 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 $R_i^-, i = 1, \ldots, j$ are weak reduction contexts and R_1^- is a maximal weak reduction context for $R_1^-[t_0]$, and $t_0 \not\equiv y$ where y is a bound variable in t.

Our definition of a maximal reduction context differs from that used in Schmidt-Schauss *et al.* (2004) in that such a context is only 'maximal up to choice points'. As a consequence, the maximal reduction context for a term t is not necessarily unique.

Example 2.4. For $(\texttt{letrec}\ x_2 = \lambda x.x, x_1 = x_2\ x_1, x_3 = (\texttt{amb}\ (x_2\ x_1)\ y)\ \texttt{in}\ (\texttt{amb}\ x_1\ x_3))$ there exist the following maximal reduction contexts:

— $(\texttt{letrec}\ x_2 = [\cdot], x_1 = x_2\ x_1, x_3 = (\texttt{amb}\ (x_2\ x_1)\ y)\ \texttt{in}\ (\texttt{amb}\ x_1\ x_3))$;
— $(\texttt{letrec}\ x_2 = \lambda x.x, x_1 = x_2\ x_1, x_3 = (\texttt{amb}\ (x_2\ x_1)\ [\cdot])\ \texttt{in}\ (\texttt{amb}\ x_1\ x_3))$.

The first maximal reduction context can be calculated in two different ways depending on which argument is chosen for the amb expression in the in expression of the letrec.

2.2. *Reduction rules*

We define the reduction rules in a more general form than we will use later for the normal-order reduction. Thus the general rules can be used for partial evaluation and other compile-time optimisations.

Definition 2.5 (Reduction rules). The deterministic reduction rules of $\Lambda_{\texttt{amb}}^{\texttt{let}}$ are defined in Figure 1, the non-deterministic reduction rules are defined in Figure 2. We define the following unions of some reductions:

(amb-c)	:= (amb-l-c) ∪ (amb-r-c)	(lamb)	:= (lamb-l) ∪ (lamb-r)
(amb-in)	:= (amb-l-in) ∪ (amb-r-in)	(cp)	:= (cp-in) ∪ (cp-e)
(amb-e)	:= (amb-l-e) ∪ (amb-r-e)	(llet)	:= (llet-in) ∪ (llet-e)
(amb)	:= (amb-l) ∪ (amb-r)	(seq)	:= (seq-c) ∪ (seq-in) ∪ (seq-e)
(amb-l)	:= (amb-l-c) ∪ (amb-l-in) ∪ (amb-l-e)	(amb-r)	:= (amb-r-c) ∪ (amb-r-in) ∪ (amb-r-e)
(case)	:= (case-c) ∪ (case-in) ∪ (case-e)	(lll)	:= (llet) ∪ (lcase) ∪ (lapp) ∪ (lseq) ∪ (lamb).

Reductions are denoted using an arrow with superscripts: for example, $\xrightarrow{\text{llet}}$. In order to state explicitly the context in which a particular reduction is performed, we annotate the reduction arrow with the context in which the reduction takes place, though if no confusion can arise, we will omit the context from the arrow.

The *redex* of a reduction is the term as given on the left-hand side of a reduction rule. We will also speak of the *inner redex*, which is the modified case expression for (case) reductions, the modified seq expression for (seq) reductions, the modified amb expression for (amb) reductions and the variable position that is replaced by rule (cp). Otherwise, it is the same as the redex.

We use $+$ to denote the transitive closure of reductions, and $*$ for the reflexive transitive closure: for example, we use $\xrightarrow{\text{lll},*}$ for the reflexive transitive closure of $\xrightarrow{\text{lll}}$. We use upper-case words to denote (finite) sequences of reductions, for example, we write $\xrightarrow{RED}$ for the sequence $RED = \xrightarrow{\text{case}}\xrightarrow{\text{lll}}\xrightarrow{\text{lbeta}}$.

(lbeta) $((\lambda x.s)\ r) \rightarrow (\texttt{letrec}\ x = r\ \texttt{in}\ s)$

(cp-in) $(\texttt{letrec}\ x_1 = (\lambda x.s), \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[x_m])$
$\rightarrow (\texttt{letrec}\ x_1 = (\lambda x.s), \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[(\lambda x.s)])$

(cp-e) $(\texttt{letrec}\ x_1 = (\lambda x.s), \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[x_m]\ \texttt{in}\ r)$
$\rightarrow (\texttt{letrec}\ x_1 = (\lambda x.s), \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[(\lambda x.s)]\ \texttt{in}\ r)$

(llet-in) $(\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ r)) \rightarrow (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ r)$

(llet-e) $(\texttt{letrec}\ x_1 = s_1, \ldots, x_i = (\texttt{letrec}\ Env_2\ \texttt{in}\ s_i), \ldots, x_n = s_n\ \texttt{in}\ r)$
$\rightarrow (\texttt{letrec}\ x_1 = s_1, \ldots, x_i = s_i, \ldots, x_n = s_n, Env_2\ \texttt{in}\ r)$

(lapp) $((\texttt{letrec}\ Env\ \texttt{in}\ t)\ s) \rightarrow (\texttt{letrec}\ Env\ \texttt{in}\ (t\ s))$

(lcase) $(\texttt{case}_T\ (\texttt{letrec}\ Env\ \texttt{in}\ t)\ alts) \rightarrow (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{case}_T\ t\ alts))$

(lseq) $(\texttt{seq}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)\ t) \rightarrow (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{seq}\ s\ t))$

(lamb-l) $(\texttt{amb}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)\ t) \rightarrow (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{amb}\ s\ t))$

(lamb-r) $(\texttt{amb}\ s\ (\texttt{letrec}\ Env\ \texttt{in}\ t)) \rightarrow (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{amb}\ s\ t))$

(seq-c) $(\texttt{seq}\ v\ t) \rightarrow t,$ if v is a value

(seq-in) $(\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[(\texttt{seq}\ x_m\ t)])$
$\rightarrow (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[t]),$ if v is a value

(seq-e) $(\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[(\texttt{seq}\ x_m\ t)]\ \texttt{in}\ r)$
$\rightarrow (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[t]\ \texttt{in}\ r),$ if v is a value

(case-c) for the case $\mathrm{ar}(c_{T,i}) = n \geq 1$: Let y_i be fresh variables, then
$(\texttt{case}_T\ (c_{T,i}\ \overrightarrow{t_i})\ \ldots ((c_{T,i}\ \overrightarrow{y_i}) \rightarrow t)\ldots) \rightarrow (\texttt{letrec}\ \{y_i = t_i\}_{i=1}^{n}\ \texttt{in}\ t)$

(case-c) for the case $\mathrm{ar}(c_{T,i}) = 0 : (\texttt{case}_T\ c_{T,i}\ \ldots\ (c_{T,i} \rightarrow t)\ldots) \rightarrow t$

(case-in) for the case $\mathrm{ar}(c_{T,i}) = n \geq 1$: Let y_i be fresh variables, then
$\texttt{letrec}\ x_1 = (c_{T,i}\ \overrightarrow{t_i}), \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[\texttt{case}_T\ x_m\ \ldots\ (c_{T,i}\ \overrightarrow{z_i} \rightarrow t)\ \ldots]$
$\rightarrow \texttt{letrec}\ x_1 = (c_{T,i}\ \overrightarrow{y_i}), \{y_i = t_i\}_{i=1}^{n}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env$
$\texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^{n}\ \texttt{in}\ t)]$

(case-in) for the case $\mathrm{ar}(c_{T,i}) = 0$:
$\texttt{letrec}\ x_1 = c_{T.i}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[\texttt{case}_T\ x_m\ \ldots (c_{T,i} \rightarrow t)\ldots]$
$\rightarrow \texttt{letrec}\ x_1 = c_{T,i}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[t]$

(case-e) for the case $\mathrm{ar}(c_{T,i}) = n \geq 1$: Let y_i be fresh variables, then
$\texttt{letrec}\, x_1 = (c_{T,i}\ \overrightarrow{t_i}), \{x_i = x_{i-1}\}_{i=2}^{m}, Env, u = C[\texttt{case}_T\ x_m\ \ldots (c_{T,i}\ \overrightarrow{z_i} \rightarrow r_1)\ldots]$
$\texttt{in}\ r_2$
$\rightarrow \texttt{letrec}\ x_1 = (c_{T,i}\ \overrightarrow{y_i}), \{y_i = t_i\}_{i=1}^{n}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env,$
$u = C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^{n}\ \texttt{in}\ r_1)]$
$\texttt{in}\ r_2$

(case-e) for the case $\mathrm{ar}(c_{T,i}) = 0$:
$\texttt{letrec}\ x_1 = c_{T,i}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, u = C[\texttt{case}_T\ x_m\ \ldots\ (c_i \rightarrow r_1)\ldots]\ \texttt{in}\ r_2$
$\rightarrow \texttt{letrec}\ x_1 = c_{T,i}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, u = C[r_1]\ \texttt{in}\ r_2$

Fig. 1. Deterministic reduction rules of the calculus

We will now briefly compare our rules and the rules of the call-by-need calculus with recursion in Ariola *et al.* (1995, Section 7.2). The rule (lbeta) is the sharing-respecting variant of beta reduction, and is defined as rule (β_{need}) in Ariola *et al.* (1995). The rule

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

(amb-l-c) $(\texttt{amb}\ v\ s) \to v,$ if v is a value

(amb-r-c) $(\texttt{amb}\ s\ v) \to v,$ if v is a value

(amb-l-in) $(\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[(\texttt{amb}\ x_m\ s)])$
$\to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[x_m]),$ if v is a value

(amb-r-in) $(\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[(\texttt{amb}\ s\ x_m)])$
$\to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[x_m]),$ if v is a value

(amb-l-e) $(\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[(\texttt{amb}\ x_m\ t)]\ \texttt{in}\ r)$
$\to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[x_m]\ \texttt{in}\ r),$ if v is a value

(amb-r-e) $(\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[(\texttt{amb}\ t\ x_m)]\ \texttt{in}\ r)$
$\to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[x_m]\ \texttt{in}\ r),$ if v is a value

Fig. 2. Non-deterministic reduction rules of the calculus

(lll) arranges `letrec` environments, and is similar to the rules (lift), (assoc) and (assoc_i) of Ariola *et al.* (1995), though we have more rules, since we have the constructs `case`, `seq` and `amb`. The rule (cp) is analogous to the rules (deref) and (deref_i) of Ariola *et al.* (1995), except that we only allow abstractions to be copied, and do not copy variables. The consequence is that we need more variants for most of the reduction rules, since we explicitly follow the bindings during the reduction, instead of removing indirections. The reason for keeping indirections is technical: the reduction diagrams are easier to close and it seems easier to find measures for the induction in the correctness proofs. Nevertheless, we will show that the program transformations (abs), (cpx), (cpcx) and (gc) are correct (see Section 5), and thus the copying of variables and constructor applications are allowed optimisations. Another reason for having more rules than Ariola *et al.* (1995) is that our syntax has `case`, `seq` and `amb` expressions, which are not present in the call-by-need calculus of Ariola *et al.* (1995). The special variants of (case) for constants are necessary to ensure that we do not introduce empty `letrec` environments, and hence that the reduction rules generate syntactically correct expressions only.

2.3. *Normal-order reduction*

Let R be a maximal reduction context for a term t and $t \equiv R[s]$. The normal-order reduction applies a reduction rule of Definition 2.5 to s or to the direct superterm of s. To aid understanding, we will start by describing how a position of a normal-order redex can be reached using a non-deterministic unwinding algorithm $\mathcal{UW}$.

We let s be a term. If we have $s \equiv (\texttt{letrec}\ Env\ \texttt{in}\ s')$, we apply $\texttt{uw}$ to the pair $(s', (\texttt{letrec}\ Env\ \texttt{in}\ [\cdot]))$, otherwise we apply $\texttt{uw}$ to the pair $(s, [\cdot])$. To detect loops, the unwinding algorithm marks the bindings of a `letrec` expression when visiting them. We label the symbol $=$ with a dot, that is, a binding $x \overset{\bullet}{=} s$ is marked as 'visited' and the algorithm stops if it hits a visited binding. We assume that these markers are removed before the result of the algorithm is returned.

$$\begin{array}{ll}
\mathtt{uw}((s\ t),R) & \to \mathtt{uw}(s,R[([\cdot]\ t)]) \\
\mathtt{uw}((\mathtt{seq}\ s\ t),R) & \to \mathtt{uw}(s,R[(\mathtt{seq}\ [\cdot]\ t)]) \\
\mathtt{uw}((\mathtt{case}\ s\ alts),R) & \to \mathtt{uw}(s,R[(\mathtt{case}\ [\cdot]\ alts)]) \\
\mathtt{uw}((\mathtt{amb}\ s\ t),R) & \to \mathtt{uw}(s,R[(\mathtt{amb}\ [\cdot]\ t)])\ \textbf{or}\ \mathtt{uw}(t,R[(\mathtt{amb}\ s\ [\cdot])]) \\
\mathtt{uw}(x,(\mathtt{letrec}\ x=s,Env\ \mathtt{in}\ R^-)) & \to \mathtt{uw}(s,(\mathtt{letrec}\ x\doteq[\cdot],Env\ \mathtt{in}\ R^-[x])) \\
\mathtt{uw}(x,(\mathtt{letrec}\ y\doteq R^-,x=s,Env\ \mathtt{in}\ t)) & \to \mathtt{uw}(s,(\mathtt{letrec}\ y\doteq R^-[x],x\doteq[\cdot],Env\ \mathtt{in}\ t)) \\
\mathtt{uw}(s,R) & \to (s,R)\ \text{if no other rule is applicable.}
\end{array}$$

The algorithm starting with term s returns a pair (s',R) with $R[s']\equiv s$ and either R is a maximal reduction context for s or the algorithm stops because a cycle has been detected; in this case s' is a variable that is bound in R.

Since cycle detection is implemented, the algorithm always terminates, for example, for the term $(\mathtt{letrec}\ x=y,y=x\ \mathtt{in}\ x)$, the result is the pair $(x,(\mathtt{letrec}\ x=y,y=[\cdot]\ \mathtt{in}\ x))$:

$$\begin{array}{lll}
\mathtt{uw}(x,(\mathtt{letrec}\ x=y,y=x\ \mathtt{in}\ [\cdot])) & \to & \mathtt{uw}(y,(\mathtt{letrec}\ x\doteq[\cdot],y=x\ \mathtt{in}\ x)) \\
& \to & \mathtt{uw}(x,(\mathtt{letrec}\ x\doteq y,y\doteq[\cdot]\ \mathtt{in}\ x)) \\
& \to & \quad (x,(\mathtt{letrec}\ x=y,y=[\cdot]\ \mathtt{in}\ x)).
\end{array}$$

Definition 2.6. We say the unwinding algorithm *visits* a subterm during execution if there is a step, where the subterm is the first argument of the pair to which uw is applied, or if the subterm is the whole term.

Lemma 2.7. During evaluation the unwinding algorithm only visits subterms that are in a reduction context. If $s\equiv R[s']$, there exists an execution (by making the correct decision if the algorithm crosses an amb expression) that visits s'.

We now define the normal-order reduction. We apply a reduction rule by using a maximal reduction context for the term that should be reduced. It may be the case that no normal-order reduction is possible for the result (s,R) of the unwinding algorithm. For example, this happens if the first argument of a case expression has the wrong type, if a free variable occurs inside the maximal reduction context or if the unwinding algorithm stops because it has detected a loop.

Definition 2.8 (Normal-order reduction). Let t be an expression and R be a maximal reduction context for t, that is, $t\equiv R[t']$ for some t'. The normal-order reduction $\xrightarrow{\mathbf{no}}$ is defined by one of the following cases:

— If t' is a letrec expression $(\mathtt{letrec}\ Env_1\ \mathtt{in}\ t'')$ and $R\not\equiv[\cdot]$, there are the following cases, where R_0 is a reduction context:

1 $R\equiv R_0[(\mathtt{seq}\ [\cdot]\ r)]$.
Reduce $(\mathtt{seq}\ t'\ r)$ using rule (lseq).

2 $R\equiv R_0[([\cdot]\ r)]$.
Reduce $(t'\ r)$ using rule (lapp).

3 $R\equiv R_0[(\mathtt{case}_T\ [\cdot]\ alts)]$.
Reduce $(\mathtt{case}_T\ t'\ alts)$ using rule (lcase).

4 $R \equiv R_0[(\texttt{amb}\ [\cdot]\ s)]$.
Reduce $(\texttt{amb}\ t'\ s)$ using rule (lamb-l).

5 $R \equiv R_0[(\texttt{amb}\ s\ [\cdot])]$.
Reduce $(\texttt{amb}\ s\ t')$ using rule (lamb-r).

6 $R \equiv (\texttt{letrec}\ Env_2\ \texttt{in}\ [\cdot])$.
Reduce t using rule (llet-in) to give $(\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ t'')$.

7 $R \equiv (\texttt{letrec}\ x = [\cdot], Env_2\ \texttt{in}\ t''')$.
Reduce t using (llet-e) to give $(\texttt{letrec}\ x = t'', Env_1, Env_2\ \texttt{in}\ t''')$.

— If t' is a value, there are the following cases:

8 $R \equiv R_0[\texttt{case}_T\ [\cdot]\ \ldots]$, $t' \equiv (c_T\ \ldots)$, that is, the top constructor of t' belongs to type T.
Apply (case-c) to $(\texttt{case}_T\ t'\ \ldots)$. Note that this covers two cases of the reduction (case-c).

9 $R \equiv \texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[\texttt{case}_T\ x_m\ (c_{T,j}\ \overrightarrow{y_i} \to r)\ alts]$ and $t' \equiv (c_{T,j}\ \overrightarrow{t_i})$.
Apply (case-in) to give

$$\begin{array}{l}\texttt{letrec}\ x_1 = (c_{T,j}\ \overrightarrow{z_i}), \{x_i = x_{i-1}\}_{i=2}^{m}, \{z_i = t_i\}_{i=1}^{n}, Env \\ \texttt{in}\ R_0^-[(\texttt{letrec}\ \{y_i = z_i\}_{i=1}^{n}\ \texttt{in}\ r)].\end{array}$$

10 $R \equiv \texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[\texttt{case}_T\ x_m\ (c_{T,j} \to r)\ alts]$ and $t' \equiv c_{T,j}$.
Apply (case-in) to give

$$\texttt{letrec}\ x_1 = c_{T,j}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[r].$$

11 $R \equiv \texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = R_0^-[\texttt{case}_T\ x_m\ (c_{T,j}\ \overrightarrow{y_i} \to r)\ alts]$ $\texttt{in}\, r'$ and $t' \equiv (c_{T,j}\ \overrightarrow{t_i})$ and y is in a reduction context.
Apply (case-e) to give

$$\begin{array}{ll}\texttt{letrec} & x_1 = (c_{T,j}\ \overrightarrow{z_i}), \{x_i = x_{i-1}\}_{i=2}^{m}, \{z_i = t_i\}_{i=1}^{n}, Env, \\ & y = R_0^-[(\texttt{letrec}\ \{y_i = z_i\}_{i=1}^{n}\ \texttt{in}\ r)] \\ \texttt{in}\ r'. & \end{array}$$

12 $R \equiv \texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = R_0^-[\texttt{case}_T\ x_m\ (c_{T,j} \to r)\ alts]\ \texttt{in}\ r'$ and $t' \equiv c_{T,j}$ and y is in a reduction context.
Apply (case-e) to give

$$\texttt{letrec}\ x_1 = c_{T,j}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = R_0^-[r]\ \texttt{in}\ r'.$$

13 $R \equiv R_0[([\cdot]\ s)]$ where R_0 is a reduction context and t' is an abstraction.
Apply (lbeta) to $(t'\ s)$.

14 $R \equiv (\texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[x_m])$ where R_0^- is a weak reduction context and t' is an abstraction.
Apply (cp-in) and copy t' to the indicated position to give

$$(\texttt{letrec}\ x_1 = t', \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[t']).$$

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

15 $R \equiv (\texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = R_0^-[x_m]\ \texttt{in}\ r)$ where R_0^- is a weak reduction context, y is in a reduction context and t' is an abstraction.
Apply (cp-e) to give

$$(\texttt{letrec}\ x_1 = t', \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = R_0^-[t']\ \texttt{in}\ r).$$

16 $R \equiv R_0[(\texttt{seq}\ [\cdot]\ r)]$.
Apply (seq-c) to $(\texttt{seq}\ t'\ r)$ to give r.

17 $R \equiv (\texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[(\texttt{seq}\ x_m\ r)])$ and t' is a constructor application.
Apply (seq-in) to give

$$(\texttt{letrec}\ x_1 = t', \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[r]).$$

18 $R \equiv (\texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = R_0^-[(\texttt{seq}\ x_m\ r)]\ \texttt{in}\ r')$ where y is in a reduction context and t' is a constructor application.
Apply (seq-e) to give

$$(\texttt{letrec}\ x_1 = t', \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = R_0^-[r]\ \texttt{in}\ r').$$

19 $R \equiv R_0[(\texttt{amb}\ [\cdot]\ r)]$.
Apply (amb-l-c) to $(\texttt{amb}\ t'\ r)$.

20 $R \equiv R_0[(\texttt{amb}\ r\ [\cdot])]$.
Apply (amb-r-c) to $(\texttt{amb}\ r\ t')$.

21 $R \equiv (\texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[(\texttt{amb}\ x_m\ r)])$ and t' is a constructor application.
Apply (amb-l-in) to give

$$(\texttt{letrec}\ x_1 = t', \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[x_m]).$$

22 $R \equiv (\texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[(\texttt{amb}\ r\ x_m)])$ and t' is a constructor application.
Apply (amb-r-in) to give

$$(\texttt{letrec}\ x_1 = t', \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ R_0^-[x_m]).$$

23 $R \equiv (\texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = R_0^-[(\texttt{amb}\ x_m\ r)]\ \texttt{in}\ r')$ where y is in a reduction context and t' is a constructor application.
Apply (amb-l-e) to give

$$(\texttt{letrec}\ x_1 = t', \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = R_0^-[x_m]\ \texttt{in}\ r').$$

24 $R \equiv (\texttt{letrec}\ x_1 = [\cdot], \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = R_0^-[(\texttt{amb}\ r\ x_m)]\ \texttt{in}\ r')$ where y is in a reduction context and t' is a constructor application.
Apply (amb-r-e) to give

$$(\texttt{letrec}\ x_1 = t', \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = R_0^-[x_m]\ \texttt{in}\ r').$$

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

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

$$\begin{array}{lcl}
\texttt{par} & \equiv & \lambda x.\lambda y.\texttt{amb}\ (\texttt{seq}\ x\ y)\ (\texttt{seq}\ y\ y) \\
\texttt{spar} & \equiv & \lambda x.\lambda y.\texttt{amb}\ (\texttt{seq}\ x\ (\texttt{seq}\ y\ (\texttt{Pair}\ x\ y)))\ (\texttt{seq}\ y\ (\texttt{seq}\ x\ (\texttt{Pair}\ x\ y))) \\
\texttt{dchoice} & \equiv & \lambda x.\lambda y.\texttt{amb}\ (\texttt{seq}\ x\ (\texttt{seq}\ y\ x))\ (\texttt{seq}\ y\ (\texttt{seq}\ x\ y)) \\
\texttt{choice} & \equiv & \lambda x.\lambda y.((\texttt{amb}\ (\lambda z_1.x)\ (\lambda z_2.y))\ \texttt{True}) \\
\texttt{or} & \equiv & \lambda x.\lambda y.(\texttt{amb}\ (\texttt{if}\ x\ \texttt{then}\ \texttt{True}\ \texttt{else}\ y)\ (\texttt{if}\ y\ \texttt{then}\ \texttt{True}\ \texttt{else}\ x)) \\
\texttt{merge} & \equiv & \texttt{letrec}\ m = \lambda xs.\lambda ys.\texttt{amb}\ (\texttt{case}_{List}\ xs\ ([] \to ys)\ (z : zs \to z : m\ zs\ ys)) \\
 & & \qquad\qquad\qquad\qquad\quad (\texttt{case}_{List}\ ys\ ([] \to xs)\ (z : zs \to z : m\ xs\ zs)) \\
 & & \texttt{in}\ m
\end{array}$$

Fig. 3. Encoding of operators

Note that there are 24 cases for normal-order reduction and 25 reduction rules, however, every rule occurs exactly once. The reason for the difference of one is that two (case-c) reductions correspond to one case in the definition of normal-order reduction.

Some of our proofs will use induction on specific lengths of sequences of normal-order reductions, which are defined as follows.

Definition 2.9. We use $\texttt{rl}(RED)$ to denote the number of reductions of a finite sequence RED consisting of normal-order reductions. We use $\texttt{rl}_{(\setminus a)}(RED)$ to denote the number of non-a reductions in RED where a is a specific reduction.

Example 2.10. Let $RED = \xrightarrow{no,\mathsf{seq}}\xrightarrow{no,\mathsf{lapp}}\xrightarrow{no,\mathsf{lbeta}}\xrightarrow{no,\mathsf{llet}}$. Then $\texttt{rl}(RED) = 4$, and, for example, $\texttt{rl}_{(\setminus \mathsf{lll})}(RED)$ is the number of non-(lll) reductions in RED, that is, we have $\texttt{rl}_{(\setminus \mathsf{lll})}(RED) = 2$.

2.4. *Encoding of non-deterministic and parallel operators*

Figure 3 shows the encoding of other non-deterministic or parallel operators within our language. The operator `par` activates the concurrent evaluation of its first argument, but has the value of its second argument (Glasgow parallel Haskell has such an operator, see, for example, Trinder *et al.* (1998)). The operator `spar` evaluates both arguments in parallel and returns the pair of values (for example, this is the `par` operator suggested in Jones and Hudak (1993)). The locally demonic `dchoice` non-deterministically chooses one of its arguments if and only if both arguments converge. Erratic `choice` non-deterministically chooses one if its arguments before evaluating the arguments. The parallel `or` is non-strict in both of its arguments, that is, if one of the arguments evaluates to `True`, then the `or` expression evaluates to `True`. The `merge` operator implements bottom-avoiding merge of two lists.

2.5. *Convergence and divergence*

In this section we define the predicates for may- and must-convergence. We postpone the discussion of the meaningfulness of our definitions to Section 3 (see Examples 3.4 and 3.5) where we will introduce our equational theory.

The notion of a weak head normal form will be required.

Definition 2.11. An expression t is a *weak head normal form* (WHNF) if:

— t is a value; or
— t is of the form ($\texttt{letrec}\ Env\ \texttt{in}\ v$), where v is a value;
— or t is of the form

$$(\texttt{letrec}\ x_1 = c_{T,i}\ t_1 \ldots t_{\mathrm{ar}(c_{T,i})}, x_2 = x_1, \ldots x_m = x_{m-1}, Env\ \texttt{in}\ x_m).$$

Lemma 2.12. A WHNF has no normal-order reduction.

The next two definitions introduce predicates for may- and must-convergence and may- and must-divergence.

Definition 2.13 (May- and must-convergence). For a term t, we write $t\downarrow$ if and only if there exists a sequence of normal-order reductions starting from t that ends in a WHNF, that is,

$$t\downarrow := \exists s : (t \xrightarrow{\textbf{no},*} s \wedge s \text{ is a WHNF}).$$

If $t\downarrow$, we say that t *may-converges*. We use $\mathscr{CON}(t)$ to denote the set of finite sequences of normal-order reductions of an expression t ending in a WHNF, that is,

$$\begin{aligned}\mathscr{CON}(t) := \{RED \mid\ & t \xrightarrow{RED} s,\\ & s \text{ is a WHNF},\\ & RED \text{ contains only normal-order reductions}\}.\end{aligned}$$

We allow finite sequences of normal-order reductions to be empty, that is, if t is a WHNF, then $\mathscr{CON}(t)$ contains an empty reduction sequence.

For a term t, *must-convergence* is defined by

$$t\Downarrow := \forall s : (t \xrightarrow{\textbf{no},*} s \implies s\downarrow).$$

Definition 2.14 (May- and must-divergence). For a term t we write $t\Uparrow$ if and only if there does not exist a sequence of normal-order reductions starting with t that ends in a WHNF. Then we say t *must-diverges*, that is,

$$t\Uparrow := \forall s : (t \xrightarrow{\textbf{no},*} s \implies s \text{ is not a WHNF})$$

For a term t, we say t *may-diverges*, denoted $t\uparrow$, if and only if t may reduce to a term that must-diverges, that is,

$$t\uparrow := \exists s : (t \xrightarrow{\textbf{no},*} s \wedge s\Uparrow).$$

For a term, we define the set of all finite sequences of normal-order reductions that lead to a term that must-diverges as follows:

$$\mathscr{DIV}(t) := \{RED \mid t \xrightarrow{RED} s, s\Uparrow, RED \text{ contains only normal-order reductions}\}.$$

We allow these sequences to be empty, that is, if $t\Uparrow$, then $\mathscr{DIV}(t)$ contains an empty sequence.

The following lemma shows some relations between convergence and divergence.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Lemma 2.15. $(t{\Downarrow} \iff \neg(t{\uparrow}))$, $(t{\downarrow} \iff \neg(t{\Uparrow}))$, $(t{\Downarrow} \implies t{\downarrow})$ and $(t{\Uparrow} \implies t{\uparrow})$.

2.6. *Fair normal-order reduction*

Our normal-order reduction does not take fairness into account, that is, the set of all sequences of normal-order reductions for a given term t may include infinite sequences where a normal-order redex is infinitely often not chosen for reduction.

For example, we consider the term $t \equiv (\texttt{letrec}\ x = (\lambda y.y\ y)\ \texttt{in amb}\ (x\ x)\ \texttt{True})$. Then there exists the infinite sequence of normal-order reductions that repeats the four reductions $\xrightarrow{no,cp}\xrightarrow{no,lbeta}\xrightarrow{no,lamb\text{-}l}\xrightarrow{no,llet}$ infinitely often, that is, this sequence begins as follows:

$$\begin{array}{ll}
\multicolumn{2}{l}{(\texttt{letrec}\ x = (\lambda y.y\ y)\ \texttt{in amb}\ (x\ x)\ \texttt{True})} \\
\xrightarrow{no,cp} & (\texttt{letrec}\ x = (\lambda y.y\ y)\ \texttt{in amb}\ ((\lambda x_1.x_1\ x_1)\ x)\ \texttt{True}) \\
\xrightarrow{no,lbeta} & (\texttt{letrec}\ x = (\lambda y.y\ y)\ \texttt{in amb}\ ((\texttt{letrec}\ x_1 = x\ \texttt{in}\ (x_1\ x_1)))\ \texttt{True}) \\
\xrightarrow{no,lamb\text{-}l} & (\texttt{letrec}\ x = (\lambda y.y\ y)\ \texttt{in}\ (\texttt{letrec}\ x_1 = x\ \texttt{in amb}\ ((x_1\ x_1))\ \texttt{True})) \\
\xrightarrow{no,llet} & (\texttt{letrec}\ x = (\lambda y.y\ y), x_1 = x\ \texttt{in amb}\ (x_1\ x_1)\ \texttt{True}) \\
\xrightarrow{no,cp} & (\texttt{letrec}\ x = (\lambda y.y\ y), x_1 = x\ \texttt{in amb}\ ((\lambda x_2.x_2\ x_2)\ x_1)\ \texttt{True}).
\end{array}$$

This sequence is not fair, since the (amb-r-c) redex is always avoided by normal-order reduction. Nevertheless, t is must-convergent in our calculus. Hence, our notion of convergence already introduces a kind of fairness at the semantic level. A similar observation has already been made in Carayol *et al.* (2005) for a call-by-name calculus with amb.

This example also shows that without fair evaluation, our operator amb is not bottom-avoiding, since evaluation may forever reduce the must-divergent argument of the amb expression, although the other argument is a value. Moran has already remarked that fairness of the evaluator is *needed* to implement bottom-avoiding choice (Moran 1998). Hence, in this section we will define *fair normal-order* reduction such that these unfair sequences of reductions are forbidden.

The result of this section is that the notions of convergence and divergence using fair evaluation are the same as for our normal-order reduction. This is of great value, since we can use the normal-order reduction for reasoning, and all results are transferable to fair evaluation.

We considered two different approaches for implementing fair evaluation:

- the approach of Moran (1998), which uses resource annotations for every amb expression;
- the more elegant approach of Carayol *et al.* (2005), which defines a small-step semantics that allows the reduction of both arguments of an amb expression in parallel.

Unfortunately, the second approach does not work properly for calculi with shared bindings, since it is difficult to define a parallel reduction step if both redexes modify the same environment, for example, for the expression

$$(\texttt{letrec}\ x = c_T\ s_1 \ldots s_{\mathrm{ar}(c_T)}\ \texttt{in amb}\ (\texttt{case}_T\ x\ \ldots)\ (\texttt{case}_T\ x\ \ldots)).$$

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Thus, we chose to use the approach of Moran (1998) by annotating the `amb` expressions.

Definition 2.16. An *annotated variant* of a term s is s with all `amb` expressions annotated with a pair $\langle m,n\rangle$ of non-negative integers, which are denoted $\texttt{amb}_{\langle m,n\rangle}$. We use $ann_0(s)$ to denote the annotated variant of s with all the pairs being $\langle 0,0\rangle$. If s is an annotated variant of term t, we let $da(s) = t$.

Informally, an (inner) redex within the subterm s (t, respectively) of the expression $(\texttt{amb}_{\langle m,n\rangle}\ s\ t)$ can only be reduced if resource m (n, respectively) is non-zero. Any reduction 'inside' s decreases the annotation m by 1. Fairness emerges from the fact that resources can only be increased if both resources m and n are 0, and the increase for both resources must be strictly greater than 0.

We extend the notions of contexts and WHNFs to annotated variants as follows.

Definition 2.17. If C is an annotated variant of a term with a hole, then C is a context if and only if $da(C)$ is a context. An annotated variant s of a term is a WHNF if and only if $da(s)$ is a WHNF.

We now give a description of a non-deterministic unwinding algorithm $\mathcal{UWF}$ that leads to fair evaluation. The algorithm performs three tasks:

1 It finds a position where a normal-order reduction may be applied.
2 It decreases the annotations for the path that leads to this position.
3 If necessary, it performs scheduling by increasing the annotations.

Let s be an annotated variant of a term. If $s \equiv (\texttt{letrec}\ Env\ \texttt{in}\ s')$, apply `uwf` to the pair $(s', (\texttt{letrec}\ Env\ \texttt{in}\ [\cdot]))$, otherwise apply `uwf` to the pair $(s, [\cdot])$. In order to detect loops, the algorithm marks the visited `letrec` bindings (that is, with $\doteq$). We assume that these markers are removed before returning the result (s, R).

$$\begin{array}{ll}
\texttt{uwf}((s\ t), R) & \to \texttt{uwf}(s, R[([\cdot]\ t)]) \\
\texttt{uwf}((\texttt{seq}\ s\ t), R) & \to \texttt{uwf}(s, R[(\texttt{seq}\ [\cdot]\ t)]) \\
\texttt{uwf}((\texttt{case}\ s\ alts), R) & \to \texttt{uwf}(s, R[(\texttt{case}\ [\cdot]\ alts)]) \\
\texttt{uwf}((\texttt{amb}_{\langle m+1,n\rangle}\ s\ t), R) & \to \texttt{uwf}(s, R[(\texttt{amb}_{\langle m,n\rangle}\ [\cdot]\ t)]) \\
\texttt{uwf}((\texttt{amb}_{\langle m,n+1\rangle}\ s\ t), R) & \to \texttt{uwf}(t, R[(\texttt{amb}_{\langle m,n\rangle}\ s\ [\cdot])]) \\
\texttt{uwf}((\texttt{amb}_{\langle 0,0\rangle}\ s\ t), R) & \to \texttt{uwf}((\texttt{amb}_{\langle m,n\rangle}\ s\ t), R),\ \text{where } m,n > 0 \\
\texttt{uwf}(x, (\texttt{letrec}\ x = s, Env\ \texttt{in}\ R^-)) & \to \texttt{uwf}(s, (\texttt{letrec}\ x \doteq [\cdot], Env\ \texttt{in}\ R^-[x])) \\
\texttt{uwf}(x, (\texttt{letrec}\ y \doteq R^-, x = s, Env\ \texttt{in}\ t)) & \\
 & \to \texttt{uwf}(s, (\texttt{letrec}\ y \doteq R^-[x], x \doteq [\cdot], Env\ \texttt{in}\ t)) \\
\texttt{uwf}(s, R) & \to (s, R)\ \text{if no other rule is applicable.}
\end{array}$$

The unwinding algorithm $\mathcal{UWF}$ always terminates with a result (s, R).

Fair normal-order reduction $\xrightarrow{fno}$ on annotated variants consists of one or more executions of $\mathcal{UWF}$ followed by a normal-order reduction.

Definition 2.18 (Fair normal-order reduction). Let s be an annotated variant of a term. If s is a WHNF, then no fair normal-order reduction is possible. Otherwise, we perform the following steps:

1 Execute $\mathcal{UWF}$ starting with s. Let the result be (s'', R).
2 If no normal-order reduction is applicable to $R[s'']$ with maximal reduction context R, then go to Step 1 using the variant $R[s'']$ instead of s.
3 If a normal-order reduction is applicable to $R[s'']$ with maximal reduction context R, then apply this reduction where annotations are inherited like labels in labelled reduction (see Barendregt (1984)). Let the result be t.

Then $s \xrightarrow{\textbf{fno}} t$.

Note that when Step 2 matches, the new search of $\mathcal{UWF}$ starts with the result of the previous execution of $\mathcal{UWF}$. This is necessary to decrease the annotations for a subterm that cannot be reduced because it has a typing error (for example, $\texttt{case}_{List}$ $\texttt{True}$ ...) or it is a term with a black hole, for example, ($\texttt{letrec}$ $x = x$ $\texttt{in}$ x). Moran (1998) uses an additional reduction rule for these cases. We decrease the annotation by executing the unwinding algorithm again with another variant of the same term, where annotations are decreased.

Definition 2.19. Fair may- and must-convergence for annotated variants are defined by:

$$\begin{array}{lcl} t{\downarrow_F} & := & \exists s : (t \xrightarrow{\textbf{fno},*} s \wedge s \text{ is a WHNF}) \\ t{\Downarrow_F} & := & \forall s : (t \xrightarrow{\textbf{fno},*} s \implies s{\downarrow_F}). \end{array}$$

We use $\Uparrow_F$ ($\uparrow_F$, respectively) as the logical counterparts of $\downarrow_F$ ($\Downarrow_F$, respectively).

We can extend the notions of fair convergence and divergence to terms.

Definition 2.20. A term t *fair may-converges* (denoted $t{\downarrow_F}$) if and only if $ann_0(t){\downarrow_F}$. A term t *fair must-converges* (denoted $t{\Downarrow_F}$) if and only if $ann_0(t){\Downarrow_F}$. Again, we use $\uparrow_F$ and $\Uparrow_F$ for the logical counterparts of both convergence predicates.

Theorem 2.21. For all terms t, we have ($t{\downarrow}$ if and only if $t{\downarrow_F}$) and ($t{\Downarrow}$ if and only if $t{\Downarrow_F}$).

Proof. A complete proof can be found in Sabel and Schmidt-Schauss (2006). The idea of the proof is to show that every fair normal-order reduction for s is also a normal-order reduction for $da(s)$ and that for every sequence of normal-order reductions starting with s, there exists a sequence of fair normal-order reductions starting with $ann_0(s)$. □

3. Contextual equivalence and proof techniques

3.1. *Preorders for may- and must-convergence*

In this section we define a number of preorders resulting in a combined preorder that tests for may-convergence and must-convergence in all contexts. Contextual equivalence is then the symmetrisation of the combined preorder.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Definition 3.1. Let s, t be terms. We define the following relations:

$$
\begin{aligned}
s \leqslant_c^{\downarrow} t &\text{ if and only if } (\forall C \in \mathscr{C} : \ C[s]{\downarrow} \Rightarrow C[t]{\downarrow}) \\
s \leqslant_c^{\Downarrow} t &\text{ if and only if } (\forall C \in \mathscr{C} : \ C[s]{\Downarrow} \Rightarrow C[t]{\Downarrow}) \\
s \leqslant_c t &\text{ if and only if } s \leqslant_c^{\downarrow} t \wedge s \leqslant_c^{\Downarrow} t.
\end{aligned}
$$

The contextual equivalence is then defined by

$$s \sim_c t \text{ if and only if } s \leqslant_c t \wedge t \leqslant_c s.$$

Note that for all three preorders, $s, t, C[s]$ and $C[t]$ may be open terms.

Remark 3.2. It makes no difference if we replace the convergence predicates based on normal-order reduction with predicates based on fair normal-order reduction in the definitions of $\leqslant_c^{\downarrow}$ and $\leqslant_c^{\Downarrow}$, since Theorem 2.21 holds, that is,

$$s \leqslant_c t \text{ iff } (\forall C \in \mathscr{C} : \ C[s]{\downarrow_F} \Rightarrow C[t]{\downarrow_F}) \wedge (\forall C \in \mathscr{C} : \ C[s]{\Downarrow_F} \Rightarrow C[t]{\Downarrow_F}).$$

We choose to work with the predicates for normal-order reduction because this makes the reasoning considerably easier as we do not need to take care of resource annotations.

Remark 3.3. In Section 8 (Corollary 8.22) we show that contextual equivalence can be defined using must-convergence only, that is, $s \sim_c t$ iff $\forall C : C[s]{\Downarrow} \Leftrightarrow C[t]{\Downarrow}$.

However, even if we could prove this now, it would not simplify our proofs, since to prove $s \leqslant_c^{\Downarrow} t$, we always require as precondition that $t \leqslant_c^{\downarrow} s$ holds.

Our contextual equivalence is the same as Carayol *et al.* (2005) uses for its call-by-name calculus where so-called *weak* divergences are not considered. This is in contrast to Hughes and Moran (1995), Moran (1998) and Lassen (2006), where may-divergence holds for terms that have an infinite normal-order reduction but never lose the ability to converge. A consequence is that our equational theory is different from that of Moran (1998).

Example 3.4. The example of Carayol *et al.* (2005, page 453) is applicable to our calculus. Let the identity function **I**, a fixed-point operator **Y** and a must-divergent term Ω be defined by

$$
\begin{aligned}
\mathbf{I} &\equiv \lambda x.x \\
\mathbf{Y} &\equiv (\texttt{letrec } y = \lambda f.(f\ (y\ f)) \texttt{ in } y) \\
\Omega &\equiv (\texttt{letrec } x = x \texttt{ in } x).
\end{aligned}
$$

Then $\mathbf{I} \sim_c \mathbf{Y}\ (\lambda x.(\texttt{choice}\ x\ \mathbf{I})) \not\sim_c \texttt{choice}\ \Omega\ \mathbf{I}$. Now, consider a contextual equivalence $\sim_M$ that is the same as $\sim_c$ apart from the fact that a term t must-converges if and only if all sequences of normal-order reductions that start with t are finite and lead to a WHNF. The relation $\sim_M$ is analogous to the contextual equivalence used in Moran (1998). Hence, we have $\mathbf{I} \not\sim_M \mathbf{Y}\ (\lambda x.(\texttt{choice}\ x\ \mathbf{I})) \sim_M \texttt{choice}\ \Omega\ \mathbf{I}$.

Weak divergences are typical for some kinds of reactive systems, that is, those systems that run until they are successfully terminated by some user input (for example, a shutdown command for an operating system). Those programs should not be equivalent to

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

programs that loop forever without any chance to terminate, that is, programs that may *strongly* diverge.

Example 3.5. Assuming that integers and an operator + for addition of integers are implemented using Peano numbers, then, using the operator choice, we can define the following expression t that generates a natural number:

$$t \equiv (\texttt{letrec}\ gen = \lambda x.\texttt{choice}\ x\ (gen\ (x+1))\ \texttt{in}\ gen\ 1).$$

The expression t is must-convergent in our calculus, but may-divergent using Moran's semantics. This is because, if every choice chooses the second argument, the evaluation of t may not terminate (that is, t is weakly divergent). Using Moran's semantics, the equivalence $t \sim_M t'$ holds, where t' is defined by

$$t' \equiv (\texttt{letrec}\ gen = \lambda x.\texttt{choice}\ x\ (gen\ (x+1))\ \texttt{in}\ \texttt{choice}\ \Omega\ (gen\ 1)).$$

That is, if the equivalence of t and t' is used as a program transformation, the possibility of a strong divergence is introduced.

The operator choice can model the input of a user in such a way that if the user says 'Yes', it chooses the second argument of choice, otherwise it chooses the first argument. Using this view, t might be a small reactive system that implements a stopwatch. Obviously, t' no longer implements a stopwatch.

A well-known property (Lassen *et al.* 2005) for lambda calculi with locally bottom-avoiding choice also holds for $\Lambda^{\texttt{let}}_{\texttt{amb}}$.

Example 3.6. Ω is not least with respect to $\leqslant_c$. Consider $C \equiv (\texttt{amb}\ (\lambda x.\lambda y.x)\ [\cdot])\ \Omega)$. Then the term $C[\mathbf{I}]$ may-diverges but $C[\Omega]$ must-converges, hence $\Omega \not\leqslant_c \mathbf{I}$.

A *precongruence* $\preceq$ is a preorder on expressions such that $s \preceq t \Rightarrow C[s] \preceq C[t]$ for all contexts C. A *congruence* is a precongruence that is also an equivalence relation. The proof of the following proposition is standard.

Proposition 3.7. $\leqslant_c$ is a precongruence and $\sim_c$ is a congruence.

The following lemma will be used during the proofs of correctness of reductions. Using the contrapositive of the implication in the preorder for may-convergence and Lemma 2.15, we have the following lemma.

Lemma 3.8. Assuming s and t are terms, we have $s \leqslant_c^{\downarrow} t$ if and only if $\forall C \in \mathscr{C} : C[t]\Uparrow \implies C[s]\Uparrow$.

A main contribution of this paper is the proof that all deterministic reduction rules are correct program transformations.

Definition 3.9 (Correct program transformation). A binary relation P on terms is a *correct program transformation* if and only if $\forall$ terms $s, t : s\ P\ t \implies s \sim_c t$.

In the remaining parts of this section we develop some tools that will help us prove the correctness of program transformations.

3.2. *Context lemmas*

The goal of this section is to prove a 'context lemma' that enables us to prove contextual equivalence of two terms by just observing their termination behaviour in all *reduction contexts*, rather than in all contexts of set $\mathscr{C}$. Moran (1998) also provides a context lemma for his call-by-need calculus, but only for may-convergence. We obtained improved results by providing a context lemma for both may- and must-convergence.

The structure of this section is as follows. We first show some properties that will be necessary for proving context lemmas for may- and must-convergence, and then the section concludes with a context lemma for the combined precongruence.

In this section we will use *multicontexts*, that is, terms with several (or no) holes $\cdot_i$, where 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 placed into the holes $\cdot_i$, we use $C[s_1, \ldots, s_n]$ to denote the resulting term.

Lemma 3.10. Let C be a multicontext with n holes. If there are terms s_i with $i \in \{1, \ldots, n\}$ such that $C[s_1, \ldots, s_{i-1}, \cdot_i, s_{i+1}, \ldots, s_n]$ is a reduction context, then there exists a hole $\cdot_j$, such that for all terms $t_1, \ldots, t_n$, we have that $C[t_1, \ldots, t_{j-1}, \cdot_j, t_{j+1}, \ldots, t_n]$ is a reduction context.

Proof. We assume there is a multicontext C with n holes and there are terms $s_1, \ldots, s_n$ with $R_i \equiv C[s_1, \ldots, s_{i-1}, \cdot_i, s_{i+1}, \ldots, s_n]$ being a reduction context. Since R_i is a reduction context, there is an execution of the unwinding algorithm $\mathscr{UW}$ starting with $C[s_1, \ldots, s_n]$ that visits s_i (see Lemma 2.7). We fix this execution and apply the same execution to $C[\cdot_1, \ldots, \cdot_n]$ and stop when we arrive at a hole. Either the execution stops at hole $\cdot_i$ or earlier at some hole $\cdot_j$. Since the unwinding algorithm only visits positions in a reduction context, the claim follows. □

Note that the numbers i and j in the previous lemma are not necessarily identical. For example, for the multicontext ($\texttt{letrec}\ x = [\cdot_1], y = [\cdot_2]\ \texttt{in}\ y$) and the term $s_2 \equiv x$, the context ($\texttt{letrec}\ x = [\cdot_1], y = s_2\ \texttt{in}\ y$) is a reduction context. If we replace s_2 by another term t_2, for example, $t_2 \equiv y$, then ($\texttt{letrec}\ x = [\cdot_1], y = t_2\ \texttt{in}\ y$) is not a reduction context, but ($\texttt{letrec}\ x = t_1, y = [\cdot]\ \texttt{in}\ y$) is a reduction context for any term t_1.

Lemma 3.11. Let s, t be expressions and σ be a permutation on variables. Then:

— $(\forall R \in \mathscr{R} : R[s]{\downarrow} \implies R[t]{\downarrow}) \implies (\forall R \in \mathscr{R} : R[\sigma(s)]{\downarrow} \implies R[\sigma(t)]{\downarrow})$; and
— $(\forall R \in \mathscr{R} : R[s]{\uparrow} \implies R[t]{\uparrow}) \implies (\forall R \in \mathscr{R} : R[\sigma(s)]{\uparrow} \implies R[\sigma(t)]{\uparrow})$.

We now prove a lemma using multicontexts that is more general than needed, since the context lemma for may-convergence (Lemma 3.13) is a specialisation of the claim.

Lemma 3.12. For all $n \geqslant 0$, all multicontexts C with n holes and all expressions $s_1, ..., s_n$ and $t_1, ..., t_n$, we have, if for all $i = 1, \ldots, n$: $\forall R \in \mathscr{R} : (R[s_i]{\downarrow} \Rightarrow R[t_i]{\downarrow})$, then $C[s_1, \ldots, s_n]{\downarrow} \Rightarrow C[t_1, \ldots, t_n]{\downarrow}$.

Proof. The proof is by induction, where n, C, s_i, t_i for $i = 1, \ldots, n$ are given. The induction is on the measure (l, n), where:

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

- l is the length of the shortest reduction sequence in $\mathcal{CON}(C[s_1,\ldots,s_n])$.
- n is the number of holes in C.

We assume that the pairs are ordered lexicographically, thus this measure is well founded. The claim holds for $n = 0$, that is, all pairs $(l, 0)$, since if C has no holes, there is nothing to show.

Now let $(l, n) > (0, 0)$. For the induction step we assume that the claim holds for all n', C', s'_i, t'_i, $i = 1,\ldots,n'$ with $(l', n') < (l, n)$. Let us assume that the precondition holds, that is, that $\forall i : \forall R \in \mathcal{R} : (R[s_i]\downarrow \Rightarrow R[t_i]\downarrow)$. Let RED be a shortest reduction sequence in $\mathcal{CON}(C[s_1,\ldots,s_n])$ with $\mathtt{rl}(RED) = l$. We distinguish two cases:

- There is some index j such that $C[s_1,\ldots,s_{j-1},\cdot_j,s_{j+1},\ldots,s_n]$ is a reduction context. Lemma 3.10 implies that there is a hole $\cdot_i$ such that
$$R_1 \equiv C[s_1,\ldots,s_{i-1},\cdot_i,s_{i+1},\ldots,s_n]$$
and
$$R_2 \equiv C[t_1,\ldots,t_{i-1},\cdot_i,t_{i+1},\ldots,t_n]$$
are both reduction contexts. Let
$$C_1 \equiv C[\cdot_1,\ldots,\cdot_{i-1},s_i,\cdot_{i+1},\ldots,\cdot_n].$$
From
$$C[s_1,\ldots,s_n] \equiv C_1[s_1,\ldots,s_{i-1},s_{i+1},\ldots,s_n],$$
we have
$$RED \in \mathcal{CON}(C_1[s_1,\ldots,s_{i-1},s_{i+1},\ldots,s_n]).$$
Since C_1 has $n-1$ holes, we can use the induction hypothesis and derive
$$C_1[t_1,\ldots,t_{i-1},t_{i+1},\ldots,t_n]\downarrow,$$
that is,
$$C[t_1,\ldots,t_{i-1},s_i,t_{i+1},\ldots,t_n]\downarrow.$$
From this we have $R_2[s_i]\downarrow$. Using the precondition, we can then derive $R_2[t_i]\downarrow$, that is, $C[t_1,\ldots,t_n]\downarrow$.
- There is no index j, such that $C[s_1,\ldots,s_{j-1},\cdot_j,s_{j+1},\ldots,s_n]$ is a reduction context. If $l = 0$, then $C[s_1,\ldots,s_n]$ is a WHNF, and since no hole is in a reduction context, $C[t_1,\ldots,t_n]$ is also a WHNF, hence $C[t_1,\ldots,t_n]\downarrow$. If $l > 0$, the first normal-order reduction of RED can also be used for $C[t_1,\ldots,t_n]$, that is, the position of the redex and the inner redex are the same. This normal-order reduction can modify the context C, the number of occurrences of the terms s_i and the positions of the terms s_i. We now argue that the elimination, duplication or variable permutation for every s_i can also be applied to t_i. More formally, we will show that if $C[s_1,\ldots,s_n] \xrightarrow{\mathbf{no},a} C'[s'_1,\ldots,s'_m]$, there exists a variable permutation ρ such that for every $i = 1,\ldots,m$, there is some j such that $(s'_i, t'_i) \equiv (\rho(s_j), \rho(t_j))$. Moreover, $C[t_1,\ldots,t_n] \xrightarrow{\mathbf{no},a} C'[t'_1,\ldots,t'_m]$.
We need to consider the cases of Definition 2.8 and figure out how the terms s_i and t_i are modified by the reduction step – we will only discuss the interesting cases:

- If $\cdot_i$ is in an alternative of case, which is discarded by a (case) reduction, or $\cdot_i$ is in the argument of a seq or amb expression that is discarded by a (seq) or (amb) reduction, then s_i and t_i are both eliminated.
- If the normal-order reduction is not a (cp) reduction that copies a superterm of s_i or t_i, and s_i and t_i are not eliminated as mentioned in the previous item, then s_i and t_i can only change their respective position.
- If the normal-order reduction is a (cp) reduction that copies a superterm of s_i or t_i, then renamed copies $\rho_{s,i}(s_i)$ and $\rho_{t,i}(t_i)$ of s_i and t_i will occur, where $\rho_{s,i}, \rho_{t,i}$ are permutations on variables. Without loss of generality, for all i, we have $\rho_{s,i} = \rho_{t,i}$. Free variables of s_i or t_i can also be renamed in $\rho_{s,i}(s_i)$ and $\rho_{t,i}(t_i)$ if they are bound in the copied superterm. But with Lemma 3.11 we have that the precondition also holds for $\rho_{s,i}(s_i)$ and $\rho_{t,i}(t_i)$, that is, $\forall R \in \mathcal{R}$: $R[\rho_{s,i}(s_i)]\downarrow \implies R[\rho_{t,i}(t_i)]\downarrow$.

Now we can use the induction hypothesis. Since $C'[s'_1,\ldots,s'_m]$ has a terminating sequence of normal-order reductions of length $l-1$, we also have $C'[t'_1,\ldots,t'_m]\downarrow$. With $C[t_1,\ldots,t_n] \xrightarrow{\mathbf{no},a} C'[t'_1,\ldots,t'_m]$, we have $C[t_1,\ldots,t_n]\downarrow$. □

Lemma 3.13 (Context lemma for may-convergence). Let s, t be terms. If for all reduction contexts R we have $(R[s]\downarrow \Rightarrow R[t]\downarrow)$, then $\forall C : (C[s]\downarrow \Rightarrow C[t]\downarrow)$. That is, $s \leqslant_c^{\downarrow} t$.

Lemma 3.14 (Context lemma for must-convergence). Let s, t be terms. Then

$$((\forall R \in \mathcal{R} : R[s]\Downarrow \implies R[t]\Downarrow) \wedge (\forall R \in \mathcal{R} : R[s]\downarrow \implies R[t]\downarrow))$$
$$\implies$$
$$(\forall C : C[s]\Downarrow \implies C[t]\Downarrow).$$

Proof. We prove a more general claim using multicontexts and the contrapositive of the first of the inner implications. For all $n \geqslant 0$, all multicontexts C with n holes and all expressions $s_1,\ldots,s_n$ and $t_1,\ldots,t_n$, we have that if

$$((\forall R \in \mathcal{R} : R[t_i]\uparrow \implies R[s_i]\uparrow) \wedge (\forall R \in \mathcal{R} : R[s_i]\downarrow \implies R[t_i]\downarrow)),$$

then $C[t_1,\ldots,t_n]\uparrow \implies C[s_1,\ldots,s_n]\uparrow$.

The proof is by induction, where n, C, s_i, t_i for $i = 1,\ldots n$ are given. The induction is on the measure (l, n), where:

— l is the length of a shortest reduction sequence in $\mathcal{DIV}(C[t_1,\ldots,t_n])$.
— n is the number of holes in C.

The induction is analogous to the proof of Lemma 3.12. The precondition for may-convergence is necessary for the subcase in which C has no holes in a reduction context and $C[t_1,\ldots,t_n]\Uparrow$. □

Corollary 3.15. If $s \leqslant_c^{\downarrow} t$ and for all $R \in \mathcal{R}$ we have $R[t]\uparrow \implies R[s]\uparrow$, then $s \leqslant_c^{\Downarrow} t$.

By combining the context lemma for may-convergence (Lemma 3.13) and the context lemma for must-convergence (Lemma 3.14) we derive the following context lemma.

Lemma 3.16 (Context Lemma). Let s, t be terms. Then

$$((\forall R \in \mathcal{R} : R[s]\Downarrow \implies R[t]\Downarrow) \wedge (\forall R \in \mathcal{R} : R[s]\downarrow \implies R[t]\downarrow)) \implies s \leqslant_c t.$$

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

3.3. *Properties of the* (lll) *reduction*

The following lemma shows that `letrec`s in reduction contexts can be moved to the top-level environment by a sequence of normal-order reductions.

Lemma 3.17. The following hold:

1 For all terms of the form $(\texttt{letrec}\ Env_1\ \texttt{in}\ R_1^-[(\texttt{letrec}\ Env_2\ \texttt{in}\ t)])$ where R_1^- is a weak reduction context, there exists a sequence of normal-order (lll) reductions with

$$(\texttt{letrec}\ Env_1\ \texttt{in}\ R_1^-[(\texttt{letrec}\ Env_2\ \texttt{in}\ t)]) \xrightarrow{\mathbf{no},\mathsf{lll},+} (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ R_1^-[t]).$$

2 For all terms of the form

$$(\texttt{letrec}\ Env_1, x_1 = R_1^-[(\texttt{letrec}\ Env_2\ \texttt{in}\ t)], \{x_i = R_i^-[x_{i-1}]\}_{i=2}^m\ \texttt{in}\ R_{m+1}^-[x_m])$$

where R_j^-, $j = 1, \ldots, m+1$ are weak reduction contexts, there exists a sequence of normal-order (lll) reductions with

$$(\texttt{letrec}\ Env_1, x_1 = R_1^-[(\texttt{letrec}\ Env_2\ \texttt{in}\ t)], \{x_i = R_i^-[x_{i-1}]\}_{i=2}^m\ \texttt{in}\ R_{m+1}^-[x_m])$$
$$\xrightarrow{\mathbf{no},\mathsf{lll},+}$$
$$(\texttt{letrec}\ Env_1, Env_2, x_1 = R_1^-[t], \{x_i = R_i^-[x_{i-1}]\}_{i=2}^m\ \texttt{in}\ R_{m+1}^-[x_m]).$$

3 For all terms of the form $R_1^-[(\texttt{letrec}\ Env\ \texttt{in}\ t)]$ where R_1^- is a weak reduction context, there exists a sequence of normal-order (lll) reductions with

$$R_1^-[(\texttt{letrec}\ Env\ \texttt{in}\ t)] \xrightarrow{\mathbf{no},\mathsf{lll},*} (\texttt{letrec}\ Env\ \texttt{in}\ R_1^-[t]).$$

Proof. The statements follow by induction on the main depth of the context R_1^-. □

Another property of the (lll) reduction is that every reduction sequence consisting only of (lll) reductions must be finite.

Definition 3.18. For a given term t, the measure $\mu_{lll}(t)$ is a pair $(\mu_1(t), \mu_2(t))$, ordered lexicographically. The measure $\mu_1(t)$ is the number of `letrec` subexpressions in t, and $\mu_2(t)$ is the sum of $\texttt{lrdepth}(s,t)$ of all `letrec` subexpressions s of t, where `lrdepth` is defined as follows:

$$\begin{array}{lcll}
\texttt{lrdepth}(s,s) & = & 0 & \\
\texttt{lrdepth}(s, C_1[C_2[s]]) & = & \begin{cases} 1 + \texttt{lrdepth}(s, C_2[s]) & \text{if } C_1 \text{ is a context of main depth 1,} \\ & \text{and not a } \texttt{letrec} \\ \texttt{lrdepth}(s, C_2[s]) & \text{if } C_1 \text{ is a context of main depth 1,} \\ & \text{and it is a } \texttt{letrec}. \end{cases}
\end{array}$$

Example 3.19. Let $s \equiv (\texttt{letrec}\ x = ((\lambda y.y)\ (\texttt{letrec}\ z = \texttt{True}\ \texttt{in}\ z))\ \texttt{in}\ x)$. Then $\mu_{lll}(s) = (2,1)$ where

$$\begin{array}{l}
\texttt{lrdepth}(s, (\texttt{letrec}\ z = \texttt{True}\ \texttt{in}\ z)) \\
\qquad = \texttt{lrdepth}(((\lambda y.y)\ (\texttt{letrec}\ z = \texttt{True}\ \texttt{in}\ z)), (\texttt{letrec}\ z = \texttt{True}\ \texttt{in}\ z)) \\
\qquad = 1 + \texttt{lrdepth}((\texttt{letrec}\ z = \texttt{True}\ \texttt{in}\ z), (\texttt{letrec}\ z = \texttt{True}\ \texttt{in}\ z)) \\
\qquad = 1 + 0 \\
\qquad = 1.
\end{array}$$

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Since $s \xrightarrow{C,\mathsf{III}} t$ implies $\mu_{III}(s) > \mu_{III}(t)$, and $\forall\ t : \mu_{III}(t) \geqslant (0,0)$, the following termination property holds:

Proposition 3.20. The reduction (III) is terminating, that is, there are no infinite reductions sequences consisting only of (C, III) reductions.

3.4. *Complete sets of commuting and forking diagrams*

In order to prove the correctness of the reduction rules and of further program transformations, we introduce complete sets of commuting diagrams and complete sets of forking diagrams. They have already been successfully used in Kutzner (2000), Schmidt-Schauss (2003), Sabel (2003b), Schmidt-Schauss *et al.* (2004), Schmidt-Schauss *et al.* (2005) and Mann (2005a) as a proof tool for proving contextual equivalence of program transformations.

We start by defining so-called *internal* reductions.

Definition 3.21. Let s, t be terms, *red* be a reduction of Definition 2.5 and $\mathscr{X}$ be a set of contexts. A reduction $s \xrightarrow{X,red} t$ is called $\mathscr{X}$*-internal* if it is not a normal-order reduction for s, and $X \in \mathscr{X}$. We use $s \xrightarrow{i\mathscr{X},red} t$ to denote $\mathscr{X}$-internal reductions

A *reduction sequence* is of the form $t_1 \rightarrow \ldots \rightarrow t_n$, where t_i are terms and $t_i \rightarrow t_{i+1}$ is a reduction or some other program transformation. In the following we introduce transformations on reduction sequences using the notation

$$\xrightarrow{X,red} . \xrightarrow{\mathbf{no},a_1} \ldots \xrightarrow{\mathbf{no},a_k} \quad \rightsquigarrow \quad \xrightarrow{\mathbf{no},b_1} \ldots \xrightarrow{\mathbf{no},b_m} . \xrightarrow{X,red_1} \ldots.. \xrightarrow{X,red_h},$$

where $\xrightarrow{X,red}$ is a reduction inside a context of a specific set like $\mathscr{C}$ or an internal reduction inside such a set of contexts (for example, $i\mathscr{C}$).

Such a transformation rule *matches* a prefix of a reduction sequence RED if RED has a prefix: $s \xrightarrow{X,red} t_1 \xrightarrow{\mathbf{no},a_1} \ldots t_k \xrightarrow{\mathbf{no},a_k} t$. The transformation rule is *applicable* to the prefix of a reduction sequence RED with prefix $s \xrightarrow{X,red} t_1 \xrightarrow{\mathbf{no},a_1} \ldots t_k \xrightarrow{\mathbf{no},a_k} t$ if and only if the following holds:

$$\exists r_1, \ldots, r_{m+h-1} : s \xrightarrow{\mathbf{no},b_1} r_1 \ldots \xrightarrow{\mathbf{no},b_m} r_m \xrightarrow{X,red_1} r_{m+1} \ldots r_{m+h-1} \xrightarrow{X,red_h} t.$$

The transformation consists of replacing the prefix of RED with the result, that is, the new reduction sequence starts with $s \xrightarrow{\mathbf{no},b_1} r_1 \ldots \xrightarrow{\mathbf{no},b_m} r_m \xrightarrow{X,red_1} r_{m+1} \ldots \xrightarrow{X,red_h} t$.

Since we will use sets of transformation rules, it may be the case that there is a transformation rule in the set that matches a prefix of a reduction sequence, but it is not applicable as the right-hand side cannot be constructed. But in a complete set there is always at least one diagram that is applicable.

Definition 3.22 (Complete sets of commuting diagrams/forking diagrams). A *complete set of commuting diagrams for the reduction* $\xrightarrow{X,red}$ is a set of transformation rules on reduction

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

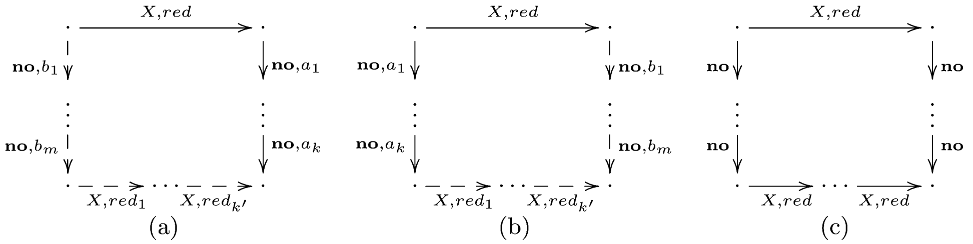

Fig. 4. Commuting (a) and forking diagrams (b) and their common representation (c)

sequences of the form

$$\xrightarrow{X,red} \cdot \xrightarrow{\mathbf{no},a_1} \dots \xrightarrow{\mathbf{no},a_k} \quad \rightsquigarrow \quad \xrightarrow{\mathbf{no},b_1} \dots \xrightarrow{\mathbf{no},b_m} \cdot \xrightarrow{X,red_1} \dots\dots \xrightarrow{X,red_{k'}},$$

depicted by a diagram of the form shown in Figure 4 (a), where $k, k' \geqslant 0, m \geqslant 1$, such that in every reduction sequence $t_0 \xrightarrow{X,red} t_1 \xrightarrow{\mathbf{no}} \dots \xrightarrow{\mathbf{no}} t_h$, where t_0 is not a WHNF, at least one of the transformation rules is applicable to a prefix of the sequence.

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

$$\xleftarrow{\mathbf{no},a_1} \dots \xleftarrow{\mathbf{no},a_k} \cdot \xrightarrow{X,red} \quad \rightsquigarrow \quad \xrightarrow{X,red_1} \dots\dots \xrightarrow{X,red_{k'}} \cdot \xleftarrow{\mathbf{no},b_1} \dots \xleftarrow{\mathbf{no},b_m},$$

depicted by a diagram of the form shown in Figure 4 (b), where $k, k' \geqslant 0, m \geqslant 1$, such that for every reduction sequence $t_h \xleftarrow{\mathbf{no}} \dots t_2 \xleftarrow{\mathbf{no}} t_1 \xrightarrow{X,red} t_0$, where t_1 is not a WHNF and $t_0 \not\equiv t_2$, at least one of the transformation rules from the set is applicable to a suffix of the sequence.

The two different kinds of diagrams are required for two different parts of the proof for the contextual equivalence of two terms. Commuting and forking diagrams often have a common representation (see Figure 4 (c)). We will give the diagrams only in the common representation if the commuting and forking diagrams can be read off obviously from it.

We abbreviate k reductions of type a with $\xrightarrow{a,k}$. As another notation, we use the $*$- and $+$-notation of regular expressions for the diagrams. The interpretation is an infinite set of diagrams constructed as follows. Repeat the following step as long as diagrams with reductions labelled with $*$ or $+$ exist:

> For a reduction $\xrightarrow{a,*}$ ($\xrightarrow{a,+}$, respectively) of a diagram, insert diagrams for all $i \in \mathbb{N}_0$ ($i \in \mathbb{N}$) with $\xrightarrow{a,*}$ ($\xrightarrow{a,+}$) replaced by $\xrightarrow{a,i}$ reductions into the set.

4. Correctness of (lbeta), (case-c), (seq-c) and the $\mathbb{CD}$-properties

In this section we use the context lemmas together with complete sets of commuting and forking diagrams to prove that (lbeta), (case-c) and (seq-c) are correct program transformations. The correctness proofs of the other reduction rules are more complex, hence in the second part of this section we will provide some general properties of a program transformation that ensure correctness of the transformation.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

4.1. *Correctness of* (lbeta), (case-c) *and* (seq-c)

Lemma 4.1. If $s \xrightarrow{red} t$ with $red \in \{\textsf{lbeta}, \textsf{case-c}, \textsf{seq-c}, \textsf{amb-c}, \textsf{lapp}, \textsf{lcase}, \textsf{lseq}, \textsf{lamb}\}$, then $t \leqslant_c^{\downarrow} s$.

Proof. Let $red \in \{\textsf{lbeta}, \textsf{case-c}, \textsf{seq-c}, \textsf{amb-c}, \textsf{lapp}, \textsf{lcase}, \textsf{lseq}, \textsf{lamb}\}$, $s \xrightarrow{red} t$ and $R[t]{\downarrow}$. Then there exists $RED \in \mathcal{CON}(R[t])$. Since every reduction *red* of the kind mentioned in the lemma inside a reduction context is a normal-order reduction, we have $R[s] \xrightarrow{\mathbf{no},a} R[t]$. By appending RED to $R[s] \xrightarrow{R,red} R[t]$, we have $R[s]{\downarrow}$. Hence, $\forall R \in \mathcal{R} : R[t]{\downarrow} \implies R[s]{\downarrow}$. The context lemma for may-convergence shows the claim. □

Since the defined normal-order reduction may reduce inside the arguments of `amb` expressions, the normal-order reduction is not unique. To treat this situation, it is not sufficient to use diagrams for $(\mathbf{no}, a)$, where a is a deterministic reduction with all other normal-order reductions. An adequate set of contexts for our proofs is the set of surface contexts.

Definition 4.2 (Surface context). *Surface contexts* are contexts where the hole is not in the body of an abstraction. We use $\mathcal{S}$ to denote the set of all surface contexts.

Note that every reduction context is also a surface context, that is, $\mathcal{R} \subset \mathcal{S}$.

Lemma 4.3. A complete set of forking diagrams for $\xrightarrow{\mathcal{S},red}$ with

$$red \in \{\textsf{lbeta}, \textsf{case-c}, \textsf{seq-c}\}$$

is

$$\begin{tikzcd} \cdot \arrow[r, "{\mathcal{S},red}"] \arrow[d, "{\mathbf{no},a}"'] & \cdot \arrow[d, dashed, "{\mathbf{no},a}"] \\ \cdot \arrow[r, dashed, "{\mathcal{S},red}"'] & \cdot \end{tikzcd} \qquad \begin{tikzcd} \cdot \arrow[r, "{\mathcal{S},red}"] \arrow[d, "{\mathbf{no},a}"'] & \cdot \arrow[dl, dashed, "{\mathbf{no},a}"] \\ \cdot & \end{tikzcd}$$

a arbitrary   for $a \in \{\textsf{case}, \textsf{seq}, \textsf{amb-l}, \textsf{amb-r}\}$

Proof. The claim follows by inspecting all cases where an $(\mathbf{no}, a)$ reduction overlaps with an (S, red) reduction with $red \in \{\textsf{lbeta}, \textsf{case-c}, \textsf{seq-c}\}$. The first case is when both reductions are performed independently, and hence can be commuted. The second diagram is applicable if the redex of *red* is inside an unused alternative of a `case` expression, inside the first argument of a `seq` expression or inside an argument of an `amb` expression that is discarded by an $(\mathbf{no}, \textsf{case})$, $(\mathbf{no}, \textsf{seq-c})$ or $(\mathbf{no}, \textsf{amb})$ reduction, respectively. □

Lemma 4.4. Let $red \in \{\textsf{lbeta}, \textsf{case-c}, \textsf{seq-c}\}$ and $s \xrightarrow{i\mathcal{S},red} t$. Then s is a WHNF if and only if t is a WHNF.

Lemma 4.5. Let $red \in \{\textsf{lbeta}, \textsf{case-c}, \textsf{seq-c}\}$ and $s \xrightarrow{red} t$. Then $s \leqslant_c^{\downarrow} t$

Proof. By using the context lemma for may-convergence, we need to show that if $s_0 \xrightarrow{red} t_0$, we have for all reduction contexts R that $R[s_0]{\downarrow} \implies R[t_0]{\downarrow}$. We will show the same statement for the larger set of all surface contexts. Let $s \equiv S[s_0]$, $t \equiv S[t_0]$ with

$s \xrightarrow{S,red} t$. Also, let $RED \in \mathcal{CON}(s)$. By induction on the length of RED, we show that $t{\downarrow}$. The base case is covered by Lemma 4.4. Let RED be of length $l > 0$. If the first reduction of RED is the same reduction as the (S, red) reduction, there is nothing to show. In all other cases, we can apply a diagram from the complete set of forking diagrams of Lemma 4.3 to a suffix of $\xleftarrow{RED} s \xrightarrow{S,red} t$. Let RED' be the suffix of RED of length $l - 1$. Then we have the following two possibilities:

$$\begin{tikzcd}
s \arrow[r, "\mathcal{S}{,}red"] \arrow[d, "\mathbf{no}{,}a"'] & t \arrow[d, dashed, "\mathbf{no}{,}a"] \\
s' \arrow[r, dashed, "\mathcal{S}{,}red"'] \arrow[d, "RED'"'] & t' \arrow[d, squiggly, "RED''"] \\
{} & {}
\end{tikzcd}$$

(1)

$$\begin{tikzcd}
s \arrow[r, "\mathcal{S}{,}red"] \arrow[d, "\mathbf{no}{,}a"'] & t \arrow[dl, dashed, "\mathbf{no}{,}a"] \\
s' \arrow[d, "RED'"'] & \\
{} &
\end{tikzcd}$$

(2)

(1) We use the induction hypothesis for RED', thus $t'{\downarrow}$. With $t \xrightarrow{\mathbf{no},a} t'$, we have $t{\downarrow}$.

(2) If the second diagram is applicable, we can append RED' to $t \xrightarrow{\mathbf{no},a} s'$, that is, we have $t{\downarrow}$. □

Lemma 4.6. If $s \xrightarrow{red} t$ with $red \in \{\mathsf{lbeta}, \mathsf{case\text{-}c}, \mathsf{seq\text{-}c}\}$, then $s \leqslant_c^{\Downarrow} t$.

Proof. We use Corollary 3.15. We have $s \leqslant_c^{\downarrow} t$ from Lemma 4.5. Let R be a reduction context, $R[s] \xrightarrow{R,red} R[t]$ and $R[t]{\uparrow}$. Since every reduction $red \in \{\mathsf{lbeta}, \mathsf{case\text{-}c}, \mathsf{seq\text{-}c}\}$ in a reduction context is also a normal-order reduction, we have $R[s]{\uparrow}$. □

Lemma 4.7. If $s \xrightarrow{red} t$ with $red \in \{\mathsf{lbeta}, \mathsf{case\text{-}c}, \mathsf{seq\text{-}c}\}$, then $t \leqslant_c^{\Downarrow} s$.

Proof. We use Corollary 3.15. From Lemma 4.1, we have $t \leqslant_c^{\downarrow} s$. It remains to show $\forall R \in \mathcal{R} : R[s]{\uparrow} \implies R[t]{\uparrow}$. We will show the statement for all surface contexts. Let $s_0 \equiv S[s]$, $t_0 = S[t]$, $s \xrightarrow{red} t$ and $s_0{\uparrow}$. We use induction on the length k of a sequence $RED \in \mathcal{DIV}(s_0)$. If $k = 0$, that is, $s_0{\Uparrow}$, then the claim follows from Lemma 4.1 using Lemma 3.8. Now let $k > 0$. Then the induction step is analogous to the proof of Lemma 4.5 using the forking diagrams.

Proposition 4.8. The reductions (lbeta), $(\mathsf{case\text{-}c})$ and $(\mathsf{seq\text{-}c})$ are correct program transformations.

Proof. The claim follows from Lemma 4.1, 4.5, 4.7 and 4.6. □

4.2. *Properties for correctness of program transformations*

The proofs of correctness of the remaining program transformations are more difficult. We postpone these proofs and provide here some general properties of program transformations and show that their validity guarantees the correctness of the transformation.

Definition 4.9. Let P be a program transformation. Then P *fulfils the* $\mathbb{CD}$*-properties* if all of the following properties hold for all expressions s, t with $s \xrightarrow{P} t$:

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

$\mathbb{C}_{\Rightarrow}(P)$:

$\forall S \in \mathcal{S}$: for all $RED_s \in \mathcal{CON}(S[s])$ there exists $RED_t \in \mathcal{CON}(S[t])$.

$\mathbb{C}_{\Leftarrow}(P)$:

$\forall S \in \mathcal{S}$: for all $RED_t \in \mathcal{CON}(S[t])$ there exists $RED_s \in \mathcal{CON}(S[s])$.

$\mathbb{D}_{\Rightarrow}(P)$:

$\forall S \in \mathcal{S}$: for all $RED_s \in \mathcal{DIV}(S[s])$ there exists $RED_t \in \mathcal{DIV}(S[t])$.

$\mathbb{D}_{\Leftarrow}(P)$:

$\forall S \in \mathcal{S}$: for all $RED_t \in \mathcal{DIV}(S[t])$ there exists $RED_s \in \mathcal{DIV}(S[s])$.

Theorem 4.10. If a program transformation fulfils the $\mathbb{CD}$-properties, it is correct.

Proof. Let P be a program transformation and s, t be expressions with $s \xrightarrow{P} t$. Then $\mathbb{C}_{\Rightarrow}(P)$ implies that $\forall S \in \mathcal{S} : S[s]\downarrow \implies S[t]\downarrow$. Using Lemma 2.15, it follows from $\mathbb{D}_{\Leftarrow}(P)$ that $\forall S \in \mathcal{S} : S[s]\Downarrow \implies S[t]\Downarrow$. Hence, the context lemma (Lemma 3.16) shows $s \leqslant_c t$. The relation $t \leqslant_c s$ follows analogously from $\mathbb{C}_{\Leftarrow}(P)$ and $\mathbb{D}_{\Rightarrow}(P)$. □

The next lemma is useful when proving properties $\mathbb{D}_{\Leftarrow}(P)$ and $\mathbb{D}_{\Rightarrow}(P)$, since their base cases (that is, RED_s and RED_t have length 0) follow directly from the validity of $\mathbb{C}_{\Leftarrow}(P)$ and $\mathbb{C}_{\Rightarrow}(P)$.

Lemma 4.11. Let s, t be expressions with $s \xrightarrow{P} t$, where P is a program transformation that fulfils the properties $\mathbb{C}_{\Rightarrow}(P)$ and $\mathbb{C}_{\Leftarrow}(P)$. Then $\forall C \in \mathcal{C}$: $C[s]\Uparrow$ if and only if $C[t]\Uparrow$.

Proof. Using the context lemma for may-convergence from $\mathbb{C}_{\Rightarrow}(P)$ and $\mathbb{C}_{\Leftarrow}(P)$, we have $s \leqslant_c^{\downarrow} t$ and $t \leqslant_c^{\downarrow} s$. With Lemma 3.8, the claim follows. □

The following lemma shows that for proving the properties $\mathbb{C}_{\Rightarrow}(P)$ and $\mathbb{D}_{\Rightarrow}(P)$ it is sufficient just to consider transformations inside surface contexts that are internal, that is, not a normal-order reduction.

Lemma 4.12. Let P be a program transformation. Then the following properties hold:

1 If for all expressions s, t with $s \xrightarrow{i\mathcal{S},P} t$ and for all $RED_s \in \mathcal{CON}(s)$ there exists $RED_t \in \mathcal{CON}(t)$, then $\mathbb{C}_{\Rightarrow}(P)$.

2 If for all expressions s, t with $s \xrightarrow{i\mathcal{S},P} t$ and for all $RED_s \in \mathcal{DIV}(s)$ there exists $RED_t \in \mathcal{DIV}(t)$, then $\mathbb{D}_{\Rightarrow}(P)$.

Proof. Let S be a surface context, and s, t be expressions with $s \xrightarrow{S,P} t$. It is sufficient to consider the case where $s \xrightarrow{S,P} t$ is a normal-order reduction. For every reduction sequence $RED_s \in \mathcal{CON}(s)$ ($RED_s \in \mathcal{DIV}(s)$, respectively) a reduction sequence $RED_t \in \mathcal{CON}(s)$ ($RED_t \in \mathcal{DIV}(t)$, respectively) can be constructed by appending RED_s to $s \xrightarrow{S,P} t$. □

Note that for program transformations that are never used as normal-order reductions (for example, the transformations in Section 5), every application inside a surface context is internal.

In the following sections we show that the reduction rules and further program transformations fulfil the $\mathbb{CD}$-properties and thus are correct.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

(gc1) $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t) \to t$
if $x_i, i = 1, \ldots, n$ does not occur free in t

(gc2) $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ t)$
$\to (\texttt{letrec}\ y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ t)$
if $x_i, i = 1, \ldots, n$ does not occur free in t nor in any t_j, for $j = 1, \ldots, m$

(cpx-in) $(\texttt{letrec}\ x = y, \mathit{Env}\ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = y, \mathit{Env}\ \texttt{in}\ C[y])$
if $x \not\equiv y$ and y is a variable

(cpx-e) $(\texttt{letrec}\ x = y, z = C[x], \mathit{Env}\ \texttt{in}\ s) \to (\texttt{letrec}\ x = y, z = C[y], \mathit{Env}\ \texttt{in}\ s)$
if $x \not\equiv y$ and y is a variable

(cpcx-in) $(\texttt{letrec}\ x = (c\ \overrightarrow{s_i}), \mathit{Env}\ \texttt{in}\ C[x])$
$\to (\texttt{letrec}\ x = (c\ \overrightarrow{y_i}), \{y_i = s_i\}_{i=1}^{\mathrm{ar}(c)}, \mathit{Env}\ \texttt{in}\ C[(c\ \overrightarrow{y_i})])$
where y_i are fresh variables

(cpcx-e) $(\texttt{letrec}\ x = (c\ \overrightarrow{s_i}), z = C[x], \mathit{Env}\ \texttt{in}\ r)$
$\to (\texttt{letrec}\ x = (c\ \overrightarrow{y_i}), \{y_i = s_i\}_{i=1}^{\mathrm{ar}(c)}, \mathit{Env}, z = C[(c\ \overrightarrow{y_i})]\ \texttt{in}\ r)$
where y_i are fresh variables

(abs) $(\texttt{letrec}\ x = (c\ \overrightarrow{s_i}), \mathit{Env}\ \texttt{in}\ r) \to (\texttt{letrec}\ x = (c\ \overrightarrow{y_i}), \{y_i = s_i\}_{i=1}^{\mathrm{ar}(c)}, \mathit{Env}\ \texttt{in}\ r)$
where y_i are fresh variables

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

Fig. 5. Additional transformation rules

We will now give a general description of the method we will use for proving the properties. Let P be a program transformation. Roughly speaking, the proofs of properties $\mathbb{C}_{\Rightarrow}(P)$ and $\mathbb{D}_{\Rightarrow}(P)$ are by induction on the length of the reduction sequence RED_s where the induction step uses the complete set of forking diagrams for P. Some inductions do not work using the length of RED_s only (this depends on the diagrams), so stronger claims with different measures are proved. The proofs of $\mathbb{C}_{\Leftarrow}(P)$ and $\mathbb{D}_{\Leftarrow}(P)$ are by induction on the length of RED_t, with specialised claims if this measure does not work. The induction step uses a complete set of commuting diagrams for P.

For a transformation P, we will tag a lemma corresponding to the property $\mathbb{C}_{\Rightarrow}(P)$, $\mathbb{C}_{\Leftarrow}(P)$, $\mathbb{D}_{\Leftarrow}(P)$ or $\mathbb{D}_{\Leftarrow}(P)$ if the validity of this property is a direct consequence of the lemma, that is, if we prove a stronger claim or the validity of the property follows from Lemma 4.12.

5. Additional correct program transformations

We now define some additional program transformations that will be necessary during the proofs of the correctness of the remaining reduction rules of $\Lambda^{\texttt{let}}_{\texttt{amb}}$ (see Section 6), and are also useful for compiler optimisations.

Definition 5.1. Some additional transformation rules are defined in Figure 5.

We define the following unions:

$$\begin{array}{ll} (\mathsf{gc}) \quad := (\mathsf{gc1}) \cup (\mathsf{gc2}) & (\mathsf{cpx}) := (\mathsf{cpx\text{-}in}) \cup (\mathsf{cpx\text{-}e}) \\ (\mathsf{cpcx}) := (\mathsf{cpcx\text{-}in}) \cup (\mathsf{cpcx\text{-}e}) & (\mathsf{opt}) \ := (\mathsf{gc}) \cup (\mathsf{cpx}) \cup (\mathsf{cpcx}) \cup (\mathsf{abs}) \cup (\mathsf{xch}). \end{array}$$

The transformation (gc) performs garbage collection by removing unnecessary bindings, (cpx) copies variables, (cpcx) abstracts a constructor application and then copies it, the rule (abs) abstracts a constructor application by sharing the arguments through new `letrec` bindings and the rule (xch) restructures two bindings in a `letrec` environment by reversing an indirection and the corresponding binding.

Lemma 5.2. A complete set of forking diagrams and a complete set of commuting diagrams for $(\mathscr{S}, \mathsf{opt})$ can be read off from the following diagrams:

$$\begin{tikzcd} \cdot \arrow[r, "{\mathscr{S},\mathsf{opt}}"] \arrow[d, "{\mathbf{no},a}"'] & \cdot \arrow[d, "{\mathbf{no},a}"] \\ \cdot \arrow[r, "{\mathscr{S},\mathsf{opt}}"'] & \cdot \end{tikzcd}$$

a arbitrary

$$\begin{tikzcd} \cdot \arrow[r, "{\mathscr{S},\mathsf{opt}}"] \arrow[d, "{\mathbf{no},a}"'] & \cdot \arrow[dl, "{\mathbf{no},a}"] \\ \cdot & \end{tikzcd}$$

for $a \in \{\mathsf{case}, \mathsf{seq}, \mathsf{amb\text{-}l}, \mathsf{amb\text{-}r}, \mathsf{cp}\}$

$$\begin{tikzcd} \cdot \arrow[r, "{\mathscr{S},\mathsf{opt}}"] \arrow[d, "{\mathbf{no},\mathsf{lll}}"'] & \cdot \\ \cdot \arrow[ur, "{\mathscr{S},\mathsf{opt}}"'] & \end{tikzcd}$$

$$\begin{tikzcd} \cdot \arrow[r, "{\mathscr{S},\mathsf{opt}}"] \arrow[d, "{\mathbf{no},a}"'] & \cdot \arrow[d, "{\mathbf{no},a}"] \\ \cdot \arrow[r, "{\mathscr{S},\mathsf{opt},+}"'] & \cdot \end{tikzcd}$$

for $a \in \{\mathsf{cp}, \mathsf{case}\}$

Proof. The claim follows by developing the diagrams for $(\mathscr{S}, \mathsf{gc})$, $(\mathscr{S}, \mathsf{cpx})$, $(\mathscr{S}, \mathsf{xch})$, $(\mathscr{S}, \mathsf{abs})$ and $(\mathscr{S}, \mathsf{cpcx})$, and then combining the diagrams. The detailed case analysis can be found in Sabel and Schmidt-Schauss (2006). □

Lemma 5.3. Let $s \xrightarrow{S,\mathsf{opt}} t$. Then the following properties hold:

— If s is a WHNF, then t is a WHNF.

— If t is a WHNF, then either s is a WHNF or $\xrightarrow{S,\mathsf{opt}}$ is a (S, gc) transformation and there is some WHNF s' with $s \xrightarrow{\mathbf{no},\mathsf{llet}} s'$.

Proof. The claim follows from the Definition 2.11. For (S, gc) there are three special cases:

— $s \equiv (\mathtt{letrec}\ Env\ \mathtt{in}\ s')$ where s' is a WHNF.

— $s \equiv (\mathtt{letrec}\ Env_2\ \mathtt{in}\ (\mathtt{letrec}\ Env\ \mathtt{in}\ s'))$ where $(\mathtt{letrec}\ Env_2\ \mathtt{in}\ s')$ is a WHNF.

— $s \equiv (\mathtt{letrec}\ Env_2, x = (\mathtt{letrec}\ Env\ \mathtt{in}\ r)\ \mathtt{in}\ s')$, where $(\mathtt{letrec}\ Env_2, x = r\ \mathtt{in}\ s')$ is a WHNF.

In all cases a single $(\mathbf{no}, \mathsf{llet})$ reduction applied to s leads to a WHNF. □

The claims that an application of (opt) inside surface contexts preserves may- and must-convergence is proved by proving a slightly stronger claim, in order to make the induction argument easier. These statements directly imply the validity of the $\mathbb{CD}$-properties, so it follows that (opt) is a correct program transformation.

Lemma 5.4 ($\mathbb{C}_{\Rightarrow}$(opt)). If $s \xrightarrow{S,\mathsf{opt}} t$, then for all $RED_s \in \mathscr{CON}(s)$ there exists $RED_t \in \mathscr{CON}(t)$ with $\mathtt{rl}(RED_t) \leqslant \mathtt{rl}(RED_s)$

Proof. Let $s \xrightarrow{S,\mathsf{opt}} t$ and $RED_s \in \mathscr{CON}(s)$ with length l. We use induction on l to show the existence of $RED_t \in \mathscr{CON}(t)$ with $\mathtt{rl}(RED_t) \leqslant l$. If $l = 0$, the claim follows from

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Lemma 5.3. If $l > 0$, we apply a forking diagram for (S, opt) from the complete set of Lemma 5.2 to the sequence $\xleftarrow{RED_s} s \xrightarrow{S,\mathsf{opt}} t$. With RED' being the suffix of RED_s of length $l-1$, we have the following cases:

$$\begin{tikzcd} s \arrow[r, "\mathscr{S}{,}\mathsf{opt}"] \arrow[d, "\mathbf{no}{,}a"'] & t \arrow[d, dashed, "\mathbf{no}{,}a"] \\ s' \arrow[r, dashed, "\mathscr{S}{,}\mathsf{opt}"'] \arrow[d, "RED'"'] & t' \arrow[d, squiggly, dashed, "RED'_t"] \\ {} & {} \end{tikzcd}$$

(1)

$$\begin{tikzcd} s \arrow[r, "\mathscr{S}{,}\mathsf{opt}"] \arrow[d, "\mathbf{no}{,}a"'] & t \arrow[dl, dashed, "\mathbf{no}{,}a"] \\ s' \arrow[d, "RED'"'] & {} \\ {} & {} \end{tikzcd}$$

(2)

$$\begin{tikzcd} s \arrow[r, "\mathscr{S}{,}\mathsf{opt}"] \arrow[d, "\mathbf{no}{,}\mathsf{lll}"'] & t \arrow[d, squiggly, dashed, "RED_t"] \\ s' \arrow[ur, dashed, "S{,}\mathsf{opt}"'] \arrow[d, "RED'"'] & {} \\ {} & {} \end{tikzcd}$$

(3)

$$\begin{tikzcd} s \arrow[r, "\mathscr{S}{,}\mathsf{opt}"] \arrow[d, "\mathbf{no}{,}a"'] & t \arrow[d, dashed, "\mathbf{no}{,}a"] \\ s' \arrow[r, dashed, "\mathscr{S}{,}\mathsf{opt}{,}+"'] \arrow[d, "RED'"'] & t' \arrow[d, squiggly, dashed, "RED'_t"] \\ {} & {} \end{tikzcd}$$

(4)

(1) We can apply the induction hypothesis to RED' and hence have $RED'_t \in \mathscr{CON}(t')$ with $\mathtt{rl}(RED'_t) \leqslant \mathtt{rl}(RED')$. By appending RED'_t to $t \xrightarrow{\mathbf{no},a} t'$, we have $RED_t \in \mathscr{CON}(t)$ with $\mathtt{rl}(RED_t) \leqslant \mathtt{rl}(RED_s)$.

(2) There exists $RED_t = \xrightarrow{\mathbf{no},a} . \xrightarrow{RED'}$ and $\mathtt{rl}(RED_t) = l$.

(3) By the induction hypothesis, we have $RED_t \in \mathscr{CON}(t)$ with $\mathtt{rl}(RED_t) \leqslant \mathtt{rl}(RED')$. With $\mathtt{rl}(RED') = \mathtt{rl}(RED_s) + 1$, the claim follows.

(4) We apply the induction hypothesis first for RED' and then for every derived normal-order reduction leading to $RED'_t \in \mathscr{CON}(t')$ with $\mathtt{rl}(RED'_t) \leqslant \mathtt{rl}(RED')$. By appending RED'_t to $t \xrightarrow{\mathbf{no},a} t'$, we have $RED_t \in \mathscr{CON}(t)$ with $\mathtt{rl}(RED_t) \leqslant l$. □

Lemma 5.5 ($\mathbb{C}_{\Leftarrow}(\mathsf{opt})$). If $s \xrightarrow{S,\mathsf{opt}} t$, then for all $RED_t \in \mathscr{CON}(t)$ there exists $RED_s \in \mathscr{CON}(s)$ with $\mathtt{rl}_{(\backslash\mathsf{lll})}(RED_s) \leqslant \mathtt{rl}_{(\backslash\mathsf{lll})}(RED_t)$.

Proof. We use induction on the following measure μ on reduction sequences $s \xrightarrow{S,\mathsf{opt}} t \xrightarrow{RED}$ with $\mu(s \xrightarrow{S,\mathsf{opt}} t \xrightarrow{RED}) = (\mathtt{rl}_{(\backslash\mathsf{lll})}(RED), \mu_{lll}(s))$ (the measure $\mu_{lll}(\cdot)$ was introduced in Definition 3.18). We assume the measure to be ordered lexicographically. If $\mu(s \xrightarrow{S,\mathsf{opt}} t \xrightarrow{RED_t}) = (0,(0,0))$, then $\mu_{lll}(s) = (0,0)$ implies $\mu_{lll}(t) = (0,0)$ since an (S, opt) transformation does not introduce new `letrec` expressions. Thus RED_t must be empty and t must be a WHNF. From Lemma 5.3, we have that either s is also a WHNF or $s \xrightarrow{\mathbf{no},\mathsf{lll}} s'$ where s' is a WHNF. In both cases we have a (possibly empty) $RED_s \in \mathscr{CON}(S)$ with $\mathtt{rl}_{(\backslash\mathsf{lll})}(RED_s) \leqslant \mathtt{rl}_{(\backslash\mathsf{lll})}(RED_t)$.

Now, let $\mu(s \xrightarrow{S,\mathsf{opt}} t \xrightarrow{RED_t}) = (l, m) > (0,(0,0))$. Without loss of generality, we assume that RED_t is non-empty, hence we can apply a commuting diagram from the complete set of Lemma 5.2. Let RED' be the suffix of RED_t of length $l-1$. We have the following cases:

$$\begin{tikzcd} s \arrow[r, "\mathscr{S}{,}\mathsf{opt}"] \arrow[d, dashed, "\mathbf{no}{,}a"'] & t \arrow[d, "\mathbf{no}{,}a"] \\ s' \arrow[r, dashed, "\mathscr{S}{,}\mathsf{opt}"'] \arrow[d, squiggly, dashed, "RED'_s"'] & t' \arrow[d, "RED'"] \\ {} & {} \end{tikzcd}$$

(1)

$$\begin{tikzcd} s \arrow[r, "\mathscr{S}{,}\mathsf{opt}"] \arrow[dr, dashed, "\mathbf{no}{,}a"'] & t \arrow[d, "\mathbf{no}{,}a"] \\ {} & t' \arrow[d, "RED'"'] \\ {} & {} \end{tikzcd}$$

(2)

$$\begin{tikzcd} s \arrow[r, "\mathscr{S}{,}\mathsf{opt}"] \arrow[d, dashed, "\mathbf{no}{,}\mathsf{lll}"'] & t \arrow[d, "RED_t"] \\ s' \arrow[ur, dashed, "S{,}\mathsf{opt}"'] \arrow[d, squiggly, dashed, "RED'_s"'] & {} \\ {} & {} \end{tikzcd}$$

(3)

$$\begin{tikzcd} s \arrow[r, "\mathscr{S}{,}\mathsf{opt}"] \arrow[d, dashed, "\mathbf{no}{,}a"'] & t \arrow[d, "\mathbf{no}{,}a"] \\ s' \arrow[r, dashed, "\mathscr{S}{,}\mathsf{opt}{,}+"'] \arrow[d, squiggly, dashed, "RED'_s"'] & t' \arrow[d, "RED'"] \\ {} & {} \end{tikzcd}$$

(4)

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

(1) We can apply the induction hypothesis to $s' \xrightarrow{S,\mathsf{opt}} t' \xrightarrow{RED'}$ since either $\mu_{lll}(s') < m$ if the $(\mathbf{no}, a)$ reduction is an (lll) reduction or $\mathtt{rl}_{(\backslash\mathsf{lll})}(RED') < \mathtt{rl}_{(\backslash\mathsf{lll})}(RED_t)$. Hence, we have $RED'_s \in \mathscr{CON}(s')$, and by appending RED'_s to $s \xrightarrow{\mathbf{no},a} s'$, we have $RED_s \in \mathscr{CON}(s)$ with $\mathtt{rl}_{(\backslash\mathsf{lll})}(RED_s) \leqslant \mathtt{rl}_{(\backslash\mathsf{lll})}(t)$.

(2) We have $RED_s = \xrightarrow{\mathbf{no},a} \xrightarrow{RED'}$ and $\mathtt{rl}_{(\backslash\mathsf{lll})}(RED_s) = \mathtt{rl}_{(\backslash\mathsf{lll})}(RED_t)$.

(3) Since $\mu_{lll}(s') < \mu_{lll}(s)$,we can apply the induction hypothesis to $s' \xrightarrow{S,\mathsf{opt}} t \xrightarrow{RED_t}$ and have $RED'_s \in \mathscr{CON}(s')$ with $\mathtt{rl}_{(\backslash\mathsf{lll})}(RED'_s) \leqslant \mathtt{rl}_{(\backslash\mathsf{lll})}(RED_t)$. By appending RED'_s to $s \xrightarrow{\mathbf{no},\mathsf{lll}} s'$, the claim follows.

(4) Since $\mathtt{rl}_{(\backslash\mathsf{lll})}(RED') < \mathtt{rl}_{(\backslash\mathsf{lll})}(RED_t)$, we can apply the induction hypothesis multiple times for every (S, opt) transformation leading to $RED'_s \in \mathscr{CON}(s')$ with $\mathtt{rl}_{(\backslash\mathsf{lll})}(RED'_s) \leqslant l-1$. By appending RED'_s to $s \xrightarrow{\mathbf{no},a} s'$, we have $RED_s \in \mathscr{CON}(s)$ with $\mathtt{rl}_{(\backslash\mathsf{lll})}(RED_s) \leqslant l$. □

Since we have shown $\mathbb{C}_{\Leftarrow}(\mathsf{opt})$ and $\mathbb{C}_{\Rightarrow}(\mathsf{opt})$, Lemma 4.11 can be applied.

Corollary 5.6. If $s \xrightarrow{S,\mathsf{opt}} t$, then $s\Uparrow$ if and only if $t\Uparrow$.

Lemma 5.7 ($\mathbb{D}_{\Rightarrow}(\mathsf{opt})$). If $s \xrightarrow{S,\mathsf{opt}} t$, then for all $RED_s \in \mathscr{DIV}(s)$ there exists $RED_t \in \mathscr{DIV}(t)$ with $\mathtt{rl}(RED_t) \leqslant \mathtt{rl}(RED_s)$

Proof. Let $s = S[s_0], t = S[t_0]$ with $s_0 \xrightarrow{\mathsf{opt}} t_0$, and let $RED_s \in \mathscr{DIV}(s)$ with $l = \mathtt{rl}(RED_s)$. We show by induction on l that there exists $RED_t \in \mathscr{DIV}(t)$ with $\mathtt{rl}(RED_t) \leqslant l$. For the base case, let RED_s be empty, that is, $s\Uparrow$. Then Corollary 5.6 shows the claim. The induction step uses the same arguments as the proof of Lemma 5.4. □

Lemma 5.8 ($\mathbb{D}_{\Leftarrow}(\mathsf{opt})$). If $s \xrightarrow{S,\mathsf{opt}} t$, then for all $RED_t \in \mathscr{DIV}(t)$ there exists $RED_s \in \mathscr{DIV}(s)$ with $\mathtt{rl}_{(\backslash\mathsf{lll})}(RED_s) \leqslant \mathtt{rl}_{(\backslash\mathsf{lll})}(RED_t)$.

Proof. The claim follows by induction on the measure μ on reduction sequences $s \xrightarrow{S,\mathsf{opt}} t \xrightarrow{RED}$ with $\mu(s \xrightarrow{S,\mathsf{opt}} t \xrightarrow{RED}) = (\mathtt{rl}_{(\backslash\mathsf{lll})}(RED), \mu_{lll}(s))$. Let the measure be ordered lexicographically. The base case is covered by Corollary 5.6. The induction step uses the same arguments as the proof of Lemma 5.5. □

The previous lemmas show that (opt) fulfils the $\mathbb{CD}$-properties, hence, with Theorem 4.10, the following proposition holds.

Proposition 5.9. (opt) is a correct program transformation.

6. Correctness of deterministic reduction rules

In this section we prove the correctness of the remaining reduction rules of $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$.

6.1. *Correctness of* (case)

We show the following proposition using the correctness of (opt) and (case-c).

Proposition 6.1. (case) is a correct program transformation, that is, if $s \xrightarrow{\mathsf{case}} t$, then $s \sim_c t$.

Proof. From Propositions 5.9 and 4.8, we have that (cpx), (cpcx) and (case-c) preserve contextual equivalence. Let $\{x_i = x_{i-1}\}_{i=1}^m$ be the chain used by a $(\mathscr{C},\mathsf{case\text{-}in})$ or $(\mathscr{C},\mathsf{case\text{-}e})$ reduction. Then every (case-in) reduction can be replaced by the sequence $\xrightarrow{\mathscr{C},\mathsf{cpx},m-1}\xrightarrow{\mathscr{C},\mathsf{cpcx}}\xrightarrow{\mathscr{C},\mathsf{case\text{-}c}}$:

$$\begin{aligned}
&\mathtt{letrec}\ x_1 = c\ \overrightarrow{t_i}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[\mathtt{case}_T\ x_m \ldots (c\ \overrightarrow{z_i} \to r)\ldots]\\
\xrightarrow{\mathsf{cpx},m-1}\ &\mathtt{letrec}\ x_1 = c\ \overrightarrow{t_i}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[\mathtt{case}_T\ x_1 \ldots (c\ \overrightarrow{z_i} \to r)\ldots]\\
\xrightarrow{\mathsf{cpcx}}\ &\mathtt{letrec}\ x_1 = c\ \overrightarrow{y_i}, \{y_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[\mathtt{case}_T\ (c\ \overrightarrow{y_i})\ \ldots]\\
\xrightarrow{\mathsf{case\text{-}c}}\ &\mathtt{letrec}\ x_1 = c\ \overrightarrow{y_i}, \{y_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[\mathtt{letrec}\ \{z_i = y_i\}_{i=1}^{\mathrm{ar}(c)}\ \mathtt{in}\ r].
\end{aligned}$$

Every $(\mathscr{C},\mathsf{case\text{-}e})$ reduction can also be replaced by the sequence $\xrightarrow{\mathscr{C},\mathsf{cpx},m-1}\xrightarrow{\mathscr{C},\mathsf{cpcx}}\xrightarrow{\mathscr{C},\mathsf{case\text{-}c}}$, where the transformation is analogous to the transformation for $(\mathscr{C},\mathsf{case\text{-}in})$. □

6.2. *Correctness of* (lll)

We develop complete sets of diagrams for (lll), and then prove correctness. By case analysis of the overlappings between (S,lll) and normal-order reductions, the following lemmas hold. Detailed proofs can be found in Sabel and Schmidt-Schauss (2006).

Lemma 6.2. A complete set of commuting diagrams for $(i\mathscr{S},\mathsf{lll})$ is

$$\begin{tikzcd}
\cdot \arrow[r, "i\mathscr{S}{,}\mathsf{lll}"] \arrow[d, dashed, "\mathbf{no}{,}a"'] & \cdot \arrow[d, "\mathbf{no}{,}a"] \\
\cdot \arrow[r, dashed, "iS{,}\mathsf{lll}"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "i\mathscr{S}{,}\mathsf{lll}"] \arrow[dr, dashed, "\mathbf{no}{,}a"'] & \cdot \arrow[d, "\mathbf{no}{,}a"] \\
 & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "i\mathscr{S}{,}\mathsf{llet}"] \arrow[dr, dashed, "\mathbf{no}{,}\mathsf{lll}{,}+"'] & \cdot \arrow[d, "\mathbf{no}{,}\mathsf{lll}"] \\
 & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "i\mathscr{S}{,}\mathsf{llet}"] \arrow[d, dashed, "\mathbf{no}{,}\mathsf{lll}"'] & \cdot \arrow[d, "\mathbf{no}{,}\mathsf{lll}"] \\
\cdot \arrow[r, dashed, "i\mathscr{S}{,}\mathsf{lll}{,}+"'] & \cdot
\end{tikzcd}$$

a arbitrary $\qquad a \in \{\mathsf{case},\mathsf{seq},\mathsf{amb\text{-}l},\mathsf{amb\text{-}r}\}$

Lemma 6.3. A complete set of forking diagrams for $(\mathscr{S},\mathsf{lll})$ is

$$\begin{tikzcd}
\cdot \arrow[r, "\mathscr{S}{,}\mathsf{lll}"] \arrow[d, "\mathbf{no}{,}a"'] & \cdot \arrow[d, dashed, "\mathbf{no}{,}a"] \\
\cdot \arrow[r, dashed, "\mathscr{S}{,}\mathsf{lll}"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathscr{S}{,}\mathsf{lll}"] \arrow[d, "\mathbf{no}{,}a"'] & \cdot \arrow[dl, dashed, "\mathbf{no}{,}a"] \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathscr{S}{,}\mathsf{lll}"] \arrow[d, "\mathbf{no}{,}\mathsf{lll}"'] & \cdot \arrow[d, dashed, "\mathbf{no}{,}\mathsf{lll}{,}+"] \\
\cdot \arrow[r, dashed, "\mathbf{no}{,}\mathsf{lll}{,}+"'] & \cdot
\end{tikzcd}$$

$$\begin{tikzcd}
\cdot \arrow[r, "\mathbf{no}{,}\mathsf{lamb}"] \arrow[dd, "\mathbf{no}{,}a"'] & \cdot \arrow[d, dashed, "\mathbf{no}{,}\mathsf{lll}{,}+"] \\
 & \cdot \arrow[d, dashed, "\mathbf{no}{,}a"] \\
\cdot \arrow[r, dashed, "\mathbf{no}{,}\mathsf{lll}{,}+"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathscr{S}{,}\mathsf{lamb}"] \arrow[dd, "\mathbf{no}{,}\mathsf{amb}"'] & \cdot \arrow[d, dashed, "\mathbf{no}{,}\mathsf{lll}{,}*"] \\
 & \cdot \arrow[d, dashed, "\mathbf{no}{,}\mathsf{amb}"] \\
\cdot \arrow[r, dashed, "\mathscr{S}{,}\mathsf{gc}^{-1}"'] & \cdot
\end{tikzcd}$$

where for the first diagram a is arbitrary, for the second $a \in \{\mathsf{case},\mathsf{seq},\mathsf{amb\text{-}l},\mathsf{amb\text{-}r}\}$ and for the fourth $a \in \{\mathsf{case},\mathsf{lbeta},\mathsf{cp},\mathsf{seq},\mathsf{amb\text{-}l},\ \mathsf{amb\text{-}r}\}$.

Lemma 6.4. If $s \xrightarrow{iS,\mathsf{lll}} t$, then s is a WHNF if and only if t is a WHNF.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Lemma 6.5 ($\mathbb{C}_{\Rightarrow}(\mathsf{lll})$). If $s \xrightarrow{S,\mathsf{lll}} t$, then for all $RED_s \in \mathscr{CON}(s)$ there exists $RED_t \in \mathscr{CON}(t)$ with $\mathtt{rl}_{(\backslash \mathsf{lll})}(RED_t) \leqslant \mathtt{rl}_{(\backslash \mathsf{lll})}(RED_s)$.

Proof. The proof is by induction on a measure μ on reduction sequences with

$$\mu(\xleftarrow{RED_s} s \xrightarrow{S,\mathsf{lll}} t) = (\mathtt{rl}_{(\backslash \mathsf{lll})}(RED_s), \mu_{lll}(s))$$

using Lemma 6.4 and Lemma 6.3. The complete proof can be found in Sabel and Schmidt-Schauss (2006). □

Lemma 6.6 ($\mathbb{C}_{\Leftarrow}(\mathsf{lll})$). If $s \xrightarrow{iS,\mathsf{lll}} t$, then for all $RED_t \in \mathscr{CON}(t)$ there exists $RED_s \in \mathscr{CON}(s)$ with $\mathtt{rl}_{(\backslash \mathsf{lll})}(RED_s) \leqslant \mathtt{rl}_{(\backslash \mathsf{lll})}(RED_t)$.

Proof. The claim follows by induction on a lexicographically ordered measure μ with

$$\mu(s \xrightarrow{iS,\mathsf{lll}} t \xrightarrow{RED_t} r) = (\mathtt{rl}_{(\backslash \mathsf{lll})}(RED_t), \mu_{lll}(s)),$$

where the base case uses Lemma 6.4 and the induction step uses the commuting diagrams for $(i\mathscr{S}, \mathsf{lll})$. The tedious, but straightforward proof can be found in Sabel and Schmidt-Schauss (2006). □

Corollary 6.7. If $s \xrightarrow{S,\mathsf{lll}} t$, then $s{\Uparrow}$ if and only if $t{\Uparrow}$.

Lemma 6.8 ($\mathbb{D}_{\Rightarrow}(\mathsf{lll})$). If $s \xrightarrow{S,\mathsf{lll}} t$, then for all $RED_s \in \mathscr{DIV}(s)$ there exists $RED_t \in \mathscr{DIV}(t)$ with $\mathtt{rl}_{(\backslash \mathsf{lll})}(RED_t) \leqslant \mathtt{rl}_{(\backslash \mathsf{lll})}(RED_s)$.

Proof. The claim follows by induction on a lexicographically ordered measure μ defined as $\mu(\xleftarrow{RED_s} s \xrightarrow{S,\mathsf{lll}} t) = (\mathtt{rl}_{(\backslash \mathsf{lll})}(RED_s), \mu_{lll}(s))$. The base case follows from Corollary 6.7, and the induction uses the forking diagrams for $(\mathscr{S}, \mathsf{lll})$. □

Lemma 6.9 ($\mathbb{D}_{\Leftarrow}(\mathsf{lll})$). If $s \xrightarrow{iS,\mathsf{lll}} t$, then for all $RED_t \in \mathscr{DIV}(t)$ there exists $RED_s \in \mathscr{DIV}(s)$ with $\mathtt{rl}_{(\backslash \mathsf{lll})}(RED_s) \leqslant \mathtt{rl}_{(\backslash \mathsf{lll})}(RED_t)$.

Proof. The claim follows by induction on the measure μ with

$$\mu(s \xrightarrow{iS,\mathsf{lll}} t \xrightarrow{RED_t} r) = (\mathtt{rl}_{(\backslash \mathsf{lll})}(RED_t), \mu_{lll}(s)),$$

where the base case is covered by Corollary 6.7 and the induction step uses the commuting diagrams for $(i\mathscr{S}, \mathsf{lll})$. □

Since we have shown the laws $\mathbb{C}_{\Rightarrow}(\mathsf{lll}), \mathbb{C}_{\Leftarrow}(\mathsf{lll}), \mathbb{D}_{\Rightarrow}(\mathsf{lll})$ and $\mathbb{D}_{\Leftarrow}(\mathsf{lll})$, the transformation (lll) fulfils the $\mathbb{CD}$-properties. Thus the following proposition holds.

Proposition 6.10. (lll) is a correct program transformation.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

6.3. *Correctness of* (seq)

Lemma 6.11. A complete set of forking diagrams for $(\mathcal{S}, \mathsf{seq})$ is

$$\begin{tikzcd}
\cdot \arrow[r, "\mathcal{S}{,}\mathsf{seq}"] \arrow[d, "\mathbf{no}{,}a"'] & \cdot \arrow[d, dashed, "\mathbf{no}{,}a"] \\
\cdot \arrow[r, dashed, "\mathcal{S}{,}\mathsf{seq}"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathcal{S}{,}\mathsf{seq}"] \arrow[d, "\mathbf{no}{,}a"'] & \cdot \arrow[dl, dashed, "\mathbf{no}{,}a"] \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[rr, "\mathcal{S}{,}\mathsf{seq}"] \arrow[d, "\mathbf{no}{,}\mathsf{cp}"'] & & \cdot \arrow[d, dashed, "\mathbf{no}{,}\mathsf{cp}"] \\
\cdot \arrow[r, dashed, "\mathcal{S}{,}\mathsf{seq}"'] & \cdot \arrow[r, dashed, "\mathcal{S}{,}\mathsf{seq}"'] & \cdot
\end{tikzcd}$$

a arbitrary for $a \in \{\mathsf{case}, \mathsf{seq}, \mathsf{amb\text{-}l}, \mathsf{amb\text{-}r}\}$

Proof. We have that:

— the reductions commute; or

— the (S, seq) is discarded by a normal-order reduction; or

— if the inner redex of the (S, seq) is in the body of an abstraction, which is copied by an $(\mathbf{no}, \mathsf{cp})$ reduction, then two (S, seq) reductions are necessary. □

Lemma 6.12. If $s \xrightarrow{iS,\mathsf{seq}} t$, then s is a WHNF if and only if t is a WHNF.

Lemma 6.13 ($\mathbb{C}_{\Rightarrow}(\mathsf{seq})$). If $s \xrightarrow{S,\mathsf{seq}} t$, then for all $RED_s \in \mathcal{CON}(s)$ there exists $RED_t \in \mathcal{CON}(t)$ with $\mathtt{rl}(RED_t) \leqslant \mathtt{rl}(RED_s)$.

Proof. Let $s \xrightarrow{S,\mathsf{seq}} t$ and $RED_s \in \mathcal{CON}(s)$ with $\mathtt{rl}(RED_s) = l$. We use induction on l.

If $l = 0$, then s is a WHNF and the claim follows from Lemma 6.12.

If $l > 0$, we let RED' be the suffix of RED_s of length $l - 1$. If the first reduction of RED_s is the same as the (S, seq) reduction, then $RED' \in \mathcal{CON}(t)$. Otherwise, we apply a forking diagram to a suffix of $\xleftarrow{RED_s} s \xrightarrow{S,\mathsf{seq}} t$ and have the cases:

$$\begin{tikzcd}
s \arrow[r, "\mathcal{S}{,}\mathsf{seq}"] \arrow[d, "\mathbf{no}{,}a"'] & t \arrow[d, dashed, "\mathbf{no}{,}a"] \\
s_1 \arrow[r, dashed, "\mathcal{S}{,}\mathsf{seq}"'] \arrow[d, "RED'"'] & t_1 \arrow[d, squiggly, "RED''"] \\
{} & {}
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathcal{S}{,}\mathsf{seq}"] \arrow[d, "\mathbf{no}{,}a"'] & t \arrow[dl, dashed, "\mathbf{no}{,}a"] \\
s_1 \arrow[d, "RED'"'] & \\
{} &
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[rr, "\mathcal{S}{,}\mathsf{seq}"] \arrow[d, "\mathbf{no}{,}\mathsf{cp}"'] & & t \arrow[d, dashed, "\mathbf{no}{,}\mathsf{cp}"] \\
s_1 \arrow[r, dashed, "\mathcal{S}{,}\mathsf{seq}"'] \arrow[d, "RED'"'] & s_1' \arrow[r, dashed, "\mathcal{S}{,}\mathsf{seq}"'] \arrow[d, squiggly, "RED''"] & t_1 \arrow[d, squiggly, "RED'''"] \\
{} & {} & {}
\end{tikzcd}$$

(1) (2) (3)

For case (1), we can apply the induction hypothesis to $\xleftarrow{RED'} s_1 \xrightarrow{S,\mathsf{seq}} t_1$. Case (2) is trivial. For case (3), we apply the induction hypothesis twice, that is, first to $\xleftarrow{RED'} s_1 \xrightarrow{S,\mathsf{seq}} s_1'$ and then to $\xleftarrow{RED''} s_1' \xrightarrow{S,\mathsf{seq}} t_1$. □

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Lemma 6.14. A complete set of commuting diagrams for $(i\mathscr{S},\mathsf{seq})$ is

$$\begin{tikzcd}
\cdot \arrow[r, "{i\mathscr{S},\mathsf{seq}}"] \arrow[d, dashed, "{\mathbf{no},a}"'] & \cdot \arrow[d, "{\mathbf{no},a}"] \\
\cdot \arrow[r, dashed, "{i\mathscr{S},\mathsf{seq}}"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "{i\mathscr{S},\mathsf{seq}}"] \arrow[dr, dashed, "{\mathbf{no},a}"'] & \cdot \arrow[d, "{\mathbf{no},a}"] \\
& \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[rr, "{i\mathscr{S},\mathsf{seq}}"] \arrow[d, dashed, "{\mathbf{no},\mathsf{cp}}"'] & & \cdot \arrow[d, "{\mathbf{no},\mathsf{cp}}"] \\
\cdot \arrow[r, dashed, "{i\mathscr{S},\mathsf{seq}}"'] & \cdot \arrow[r, dashed, "{i\mathscr{S},\mathsf{seq}}"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[rr, "{i\mathscr{S},\mathsf{seq}}"] \arrow[dr, dashed, "{\mathbf{no},a}"'] & & \cdot \arrow[dd, "{\mathbf{no},a}"] \\
& \cdot \arrow[dr, dashed, "{\mathbf{no},\mathsf{seq}}"'] & \\
& & \cdot
\end{tikzcd}$$

a arbitrary for $a \in \{\mathsf{case}, \mathsf{seq}, \mathsf{amb\text{-}l}, \mathsf{amb\text{-}r}\}$ a arbitrary

Proof. The first three diagrams describe the same cases as for the forking diagrams. The last diagram is applicable if the (iS,seq) reduction becomes normal order. □

Lemma 6.15 ($\mathbb{C}_{\Leftarrow}(\mathsf{seq})$). If $s \xrightarrow{iS,\mathsf{seq}} t$, then for every $RED_t \in \mathscr{CON}(t)$ there exists $RED_s \in \mathscr{CON}(s)$ with $\mathtt{rl}_{(\setminus\mathsf{seq})}(RED_s) \leqslant \mathtt{rl}_{(\setminus\mathsf{seq})}(RED_t)$.

Proof. Let $s \xrightarrow{iS,\mathsf{seq}} t$ and $RED_t \in \mathscr{CON}(t)$. We use induction on the measure μ ordered lexicographically with $\mu(RED) = (\mathtt{rl}_{(\setminus\mathsf{seq})}(RED), \mathtt{rl}(RED))$.

If $\mathtt{rl}(RED_t) = 0$, the claim follows from Lemma 6.12.

If $\mu(RED_t) = (l,m) \geqslant (0,1)$, we apply a commuting diagram from Lemma 6.14 to a prefix of $s \xrightarrow{iS,\mathsf{seq}} t \xrightarrow{RED_t}$. With RED' being the suffix of RED_t of length $(m-1)$, we have the cases:

$$\begin{tikzcd}
s \arrow[r, "{iS,\mathsf{seq}}"] \arrow[d, dashed, "{\mathbf{no},a}"'] & t \arrow[d, "{\mathbf{no},a}"] \\
s' \arrow[r, dashed, "{iS,\mathsf{seq}}"'] \arrow[d, squiggly, "RED''"'] & t' \arrow[d, "RED'"] \\
{} & {}
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "{iS,\mathsf{seq}}"] \arrow[dr, dashed, "{\mathbf{no},a}"'] & t \arrow[d, "{\mathbf{no},a}"] \\
& t' \arrow[d, "RED'"] \\
& {}
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[rr, "{iS,\mathsf{seq}}"] \arrow[d, dashed, "{\mathbf{no},\mathsf{cp}}"'] & & t \arrow[d, "{\mathbf{no},\mathsf{cp}}"] \\
s' \arrow[r, dashed, "{iS,\mathsf{seq}}"'] \arrow[d, squiggly, "RED''"'] & s'_1 \arrow[r, dashed, "{iS,\mathsf{seq}}"'] \arrow[d, squiggly, "RED''_1"'] & t' \arrow[d, "RED'"] \\
{} & {} & {}
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[rr, "{iS,\mathsf{seq}}"] \arrow[dr, dashed, "{\mathbf{no},a}"'] & & t \arrow[dd, "{\mathbf{no},a}"] \\
& s' \arrow[dr, dashed, "{\mathbf{no},\mathsf{seq}}"'] & \\
& & t' \arrow[d, "RED'"] \\
& & {}
\end{tikzcd}$$

(1) (2) (3) (4)

For case (1) we apply the induction hypothesis to $s' \xrightarrow{iS,\mathsf{seq}} t' \xrightarrow{RED'}$. Cases (2) and (4) are trivial. For case (3) we apply the induction hypothesis twice. □

Since $\mathbb{C}_{\Rightarrow}(\mathsf{seq})$ and $\mathbb{C}_{\Leftarrow}(\mathsf{seq})$ hold, the following corollary is true.

Corollary 6.16. If $s \xrightarrow{S,\mathsf{seq}} t$, then $s{\Uparrow}$ if and only if $t{\Uparrow}$.

Lemma 6.17 ($\mathbb{D}_{\Rightarrow}(\mathsf{seq})$). If $s \xrightarrow{S,\mathsf{seq}} t$, then for all $RED_s \in \mathscr{DIV}(s)$ there exists $RED_t \in \mathscr{DIV}(t)$ with $\mathtt{rl}(RED_t) \leqslant \mathtt{rl}(RED_s)$.

Proof. Let $s = S[s_0], t = S[t_0]$, $s_0 \xrightarrow{\mathsf{seq}} t_0$ and $RED_s \in \mathscr{DIV}(s)$ with $\mathtt{rl}(RED_s) = l$. The claim follows by induction on l, where the base case is covered by Corollary 6.16 and the induction step is analogous to the proof of Lemma 6.13. □

Lemma 6.18 ($\mathbb{D}_{\Leftarrow}(\mathsf{seq})$). If $s \xrightarrow{iS,\mathsf{seq}} t$, then for every $RED_t \in \mathscr{DIV}(t)$ there exists $RED_s \in \mathscr{DIV}(s)$ with $\mathtt{rl}_{(\setminus\mathsf{seq})}(RED_s) \leqslant \mathtt{rl}_{(\setminus\mathsf{seq})}(RED_t)$.

Proof. Let $s = S[s_0], t = S[t_0]$, $s_0 \xrightarrow{\mathsf{seq}} t_0$ and $RED_t \in \mathcal{DIV}(t)$. We use induction on the measure $\mu(RED_t)$ ordered lexicographically, where

$$\mu(RED) = (\mathtt{rl}_{(\backslash \mathsf{seq})}(RED), \mathtt{rl}(RED)).$$

If $\mathtt{rl}(RED_t) = 0$, the claim follows from Corollary 6.16. The induction step is analogous to the proof of Lemma 6.15. □

We have shown that (seq) fulfils the $\mathbb{CD}$-properties. Hence, we have the following proposition.

Proposition 6.19. (seq) is a correct program transformation.

6.4. *Correctness of* (cp)

Lemma 6.20. A complete set of forking diagrams for $(\mathcal{S}, \mathsf{cp})$ is

$$\begin{tikzcd}
\cdot \arrow[r, "{\mathcal{S},\mathsf{cp}}"] \arrow[d, "{\mathbf{no},a}"'] & \cdot \arrow[d, dashed, "{\mathbf{no},a}"] \\
\cdot \arrow[r, dashed, "{\mathcal{S},\mathsf{cp}}"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "{\mathcal{S},\mathsf{cp}}"] \arrow[d, "{\mathbf{no},a}"'] & \cdot \arrow[dl, dashed, "{\mathbf{no},a}"] \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[rr, "{\mathcal{S},\mathsf{cp}}"] \arrow[d, "{\mathbf{no},\mathsf{cp}}"'] & & \cdot \arrow[d, dashed, "{\mathbf{no},\mathsf{cp}}"] \\
\cdot \arrow[r, dashed, "{\mathcal{S},\mathsf{cp}}"'] & \cdot \arrow[r, dashed, "{\mathcal{S},\mathsf{cp}}"'] & \cdot
\end{tikzcd}$$

a arbitrary   for $a \in \{\mathsf{case}, \mathsf{seq}, \mathsf{amb\text{-}l}, \mathsf{amb\text{-}r}\}$

Proof. We have that:

- the reductions commute; or
- the redex of the target of the (S, cp) reduction is discarded by a normal-order reduction; or
- the (S, cp) reduction copies into the body of an abstraction that is copied by an $(\mathbf{no}, \mathsf{cp})$ reduction.

Lemma 6.21. If $s \xrightarrow{iS,\mathsf{cp}}$ t, then s is a WHNF if and only if t is a WHNF.

Lemma 6.22 $(\mathbb{C}_{\Rightarrow}(\mathsf{cp}))$**.** If $s \xrightarrow{S,\mathsf{cp}} t$, then for all $RED_s \in \mathcal{CON}(s)$ there exists $RED_t \in \mathcal{CON}(t)$ with $\mathtt{rl}(RED_t) \leqslant \mathtt{rl}(RED_s)$.

Proof. The proof is a copy of the proof of Lemma 6.13 using the complete set of forking diagrams for (S, cp) from Lemma 6.20 and using Lemma 6.21. □

For the other direction, we distinguish two kinds of (cp) reductions:

(cps) := the inner redex of the (cp) is of the form $S[x]$, that is, the target is inside a surface context.

(cpd) := the inner redex of the (cp) is of the form $C[\lambda z.C'[x]]$, that is, the target is inside the body of an abstraction.

Definition 6.23. Let s be a term. Then $\mu_{Sx}(s)$ is the number of occurrences of variables in s where the occurrence is inside a surface context.

Lemma 6.24. Every $(\mathcal{S}, \mathsf{cps})$ or $(\mathbf{no}, \mathsf{cp})$ reduction strictly reduces the measure μ_{Sx}. No $(\mathcal{S}, \mathsf{cpd})$ reduction changes the measure μ_{Sx}.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Analysing all overlappings of a (iS, cp) reduction with normal-order reductions shows the following lemma.

Lemma 6.25. A complete set of commuting diagrams for $(i\mathscr{S}, \mathsf{cp})$ is

$$\begin{tikzcd}
\cdot \arrow[r, "i\mathscr{S}{,}\mathsf{cp}"] \arrow[d, dashed, "\mathbf{no}{,}a"'] & \cdot \arrow[d, "\mathbf{no}{,}a"] \\
\cdot \arrow[r, dashed, "i\mathscr{S}{,}\mathsf{cp}"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[rr, "i\mathscr{S}{,}\mathsf{cps}"] \arrow[dr, dashed, "\mathbf{no}{,}a"'] & & \cdot \arrow[dd, "\mathbf{no}{,}a"] \\
& \cdot \arrow[dr, dashed, "\mathbf{no}{,}\mathsf{cp}"'] & \\
& & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "i\mathscr{S}{,}\mathsf{cp}"] \arrow[dr, dashed, "\mathbf{no}{,}a"'] & \cdot \arrow[d, "\mathbf{no}{,}a"] \\
& \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[rr, "i\mathscr{S}{,}\mathsf{cpd}"] \arrow[d, dashed, "\mathbf{no}{,}\mathsf{cp}"'] & & \cdot \arrow[d, "\mathbf{no}{,}\mathsf{cp}"] \\
\cdot \arrow[r, dashed, "i\mathscr{S}{,}\mathsf{cpd}"'] & \cdot \arrow[r, dashed, "i\mathscr{S}{,}\mathsf{cpd}"'] & \cdot
\end{tikzcd}$$

a arbitrary $\qquad$ a arbitrary $\qquad$ $a \in \{\mathsf{case}, \mathsf{seq}, \mathsf{amb\text{-}l}, \mathsf{amb\text{-}r}\}$

Lemma 6.26 ($\mathbb{C}_{\Leftarrow}(\mathsf{cp})$). If $s \xrightarrow{iS,\mathsf{cp}} t$, then for all $RED_t \in \mathscr{CON}(t)$ there exists $RED_s \in \mathscr{CON}(s)$ with $\mathtt{rl}_{(\setminus \mathsf{cp})}(RED_s) \leqslant \mathtt{rl}_{(\setminus \mathsf{cp})}(RED_t)$.

Proof. Let $s \xrightarrow{iS,\mathsf{cp}} t$ and $RED_t \in \mathscr{CON}(t)$. The claim follows by induction on the measure μ on reduction sequences with $\mu(s \xrightarrow{iS,\mathsf{cp}} t \xrightarrow{RED_t}) = (\mathtt{rl}_{(\setminus \mathsf{cp})}(RED_t), \mu_{Sx}(t))$.

If $\mu(s \xrightarrow{iS,\mathsf{cp}} t \xrightarrow{RED_t}) = (0,0)$, then from Lemma 6.24 we have that RED_t must be empty. Thus Lemma 6.21 shows the claim.

Now, let $\mu(s \xrightarrow{iS,\mathsf{cp}} t \xrightarrow{RED_t}) = (l,m) > (0,0)$. We apply a commuting diagram to a prefix of $s \xrightarrow{iS,\mathsf{cp}} t \xrightarrow{RED_t}$. With RED' being the suffix of RED_t where the first reduction is dropped. We have the following cases:

$$\begin{tikzcd}
s \arrow[r, "iS{,}\mathsf{cp}"] \arrow[d, dashed, "\mathbf{no}{,}a"'] & t \arrow[d, "\mathbf{no}{,}a"] \\
s' \arrow[r, dashed, "iS{,}\mathsf{cp}"'] \arrow[d, squiggly, "RED''"'] & t' \arrow[d, "RED'"] \\
{} & {}
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[rr, "iS{,}\mathsf{cp}"] \arrow[dr, dashed, "\mathbf{no}{,}a"'] & & t \arrow[dd, "\mathbf{no}{,}a"] \\
& s' \arrow[dr, dashed, "\mathbf{no}{,}\mathsf{cp}"'] & \\
& & t' \arrow[d, "RED'"] \\
& & {}
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "iS{,}\mathsf{cp}"] \arrow[dr, dashed, "\mathbf{no}{,}a"'] & t \arrow[d, "\mathbf{no}{,}a"] \\
& t' \arrow[d, "RED'"] \\
& {}
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[rr, "iS{,}\mathsf{cpd}"] \arrow[d, dashed, "\mathbf{no}{,}\mathsf{cpd}"'] & & t \arrow[d, "\mathbf{no}{,}\mathsf{cp}"] \\
s' \arrow[r, dashed, "iS{,}\mathsf{cpd}"'] \arrow[d, squiggly, "RED'''"'] & s'' \arrow[r, dashed, "iS{,}\mathsf{cpd}"'] \arrow[d, squiggly, "RED''"'] & t' \arrow[d, "RED'"] \\
{} & {} & {}
\end{tikzcd}$$

(1) $\qquad$ (2) $\qquad$ (3) $\qquad$ (4)

Cases (2) and (3) are trivial. In case (1) we apply the induction hypothesis to $s' \xrightarrow{iS,\mathsf{cp}} t' \xrightarrow{RED'}$. In case (4) the induction hypothesis is applied twice: first to $s'' \xrightarrow{iS,\mathsf{cpd}} t' \xrightarrow{RED'}$, and then to $s' \xrightarrow{iS,\mathsf{cpd}} s'' \xrightarrow{RED''}$. □

With Lemma 4.11, the following corollary is true.

Corollary 6.27. If $s \xrightarrow{S,\mathsf{cp}} t$, then $s\Uparrow$ if and only if $t\Uparrow$.

Lemma 6.28 ($\mathbb{D}_{\Rightarrow}(\mathsf{cp})$). If $s \xrightarrow{S,\mathsf{cp}} t$, then for all $RED_s \in \mathscr{DIV}(s)$ there exists $RED_t \in \mathscr{DIV}(t)$ with $\mathtt{rl}(RED_t) \leqslant \mathtt{rl}(RED_s)$.

Proof. The proof is the same as the proof of Lemma 6.17 using the complete set of forking diagrams for (S, cp) from Lemma 6.20, with Corollary 6.27 for the base case. □

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Lemma 6.29 ($\mathbb{D}_{\Leftarrow}$(cp)). If $s \xrightarrow{iS,\mathsf{cp}} t$, then for all $RED_t \in \mathcal{DIV}(t)$ there exists $RED_s \in \mathcal{DIV}(s)$ with $\mathtt{rl}_{(\backslash\mathsf{cp})}(RED_s) \leqslant \mathtt{rl}_{(\backslash\mathsf{cp})}(RED_t)$.

Proof. The proof is analogous to the proof of Lemma 6.26 using Corollary 6.27. □

We have shown that (cp) fulfils the $\mathbb{CD}$-properties. Hence, we have the following proposition.

Proposition 6.30. (cp) is a correct program transformation.

7. The Standardisation Theorem

We begin by summarising the results of the previous sections.

Theorem 7.1. All deterministic reductions of the calculus $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$ preserve contextual equivalence, that is, if $s \xrightarrow{a} t$ with $a \in \{\mathsf{lbeta}, \mathsf{lll}, \mathsf{case}, \mathsf{seq}, \mathsf{cp}\}$, then $s \sim_c t$.

Proof. The claim follows from Propositions 4.8, 6.10, 6.1, 6.19 and 6.30. □

We will now develop some properties of the reduction (amb) that will be required for the proof of the Standardisation Theorem (Theorem 7.13).

Lemma 7.2. If $s \xrightarrow{i\mathcal{S},\mathsf{amb}} t$, then s is a WHNF if and only if t is a WHNF.

The following lemma shows that it is sufficient to consider (amb-c) reductions.

Lemma 7.3. Let s, t be terms with $s \xrightarrow{\mathcal{C},\mathsf{amb\text{-}in}} t$ or $s \xrightarrow{\mathcal{C},\mathsf{amb\text{-}e}} t$. Then either

$$s \xrightarrow{\mathcal{C},\mathsf{cp}} \cdot \xrightarrow{\mathcal{C},\mathsf{amb\text{-}c}} \cdot \xleftarrow{\mathcal{C},\mathsf{cp}} t$$

or

$$s \xrightarrow{\mathcal{C},\mathsf{cpx},*} \cdot \xrightarrow{\mathcal{C},\mathsf{cpcx}} \cdot \xrightarrow{\mathcal{C},\mathsf{amb\text{-}c}} \cdot \xleftarrow{\mathcal{C},\mathsf{cpcx}} \cdot \xleftarrow{\mathcal{C},\mathsf{cpx},*} t.$$

Proof. The sequence of transformation copies the value into the argument of $\mathtt{amb}$, then performs (amb-c) and finally undoes the copying step. □

Lemma 7.4. If $s \xrightarrow{\mathsf{amb}} t$, then $t \leqslant_c^{\downarrow} s$.

Proof. The claim was proved for (amb-c) in Lemma 4.1. For (amb-in) or (amb-e), we replace the reduction using Lemma 7.3. Then Theorem 7.1, Proposition 5.9 and Lemma 4.1 show the claim. □

A consequence of the previous lemma is that the property $\mathbb{C}_{\Leftarrow}$(amb) also holds.

Remark 7.5. An (amb) reduction may introduce must-convergence, for example, consider the terms

$$s \equiv \mathsf{case}_{Bool}\ (\mathtt{amb\ True\ False})\ (\mathtt{True} \rightarrow \Omega)\ (\mathtt{False} \rightarrow \mathtt{False})$$

and

$$t \equiv \mathsf{case}_{Bool}\ \mathtt{False}\ (\mathtt{True} \rightarrow \Omega)\ (\mathtt{False} \rightarrow \mathtt{False})$$

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

such that $s \xrightarrow{\mathsf{amb}} t$. While $t\Downarrow$, s may reduce to Ω, that is, $\neg(s\Downarrow)$. Hence, (amb) is not a correct program transformation.

Lemma 7.6. A complete set of commuting diagrams and a complete set of forking diagrams for $(i\mathscr{S},\mathsf{amb\text{-}c})$ can be read off from the following diagrams:

$$\begin{tikzcd}
\cdot \arrow[r, "i\mathscr{S}{,}\mathsf{amb\text{-}c}"] \arrow[d, "\mathbf{no}{,}a"'] & \cdot \arrow[d, "\mathbf{no}{,}a"] \\
\cdot \arrow[r, "i\mathscr{S}{,}\mathsf{amb\text{-}c}"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "i\mathscr{S}{,}\mathsf{amb\text{-}c}"] \arrow[dr, "\mathbf{no}{,}a"'] & \cdot \arrow[d, "\mathbf{no}{,}a"] \\
 & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "i\mathscr{S}{,}\mathsf{amb\text{-}c}"] \arrow[d, "\mathbf{no}{,}a"'] & \cdot \arrow[d, "\mathbf{no}{,}a"] \\
\cdot \arrow[r, "\mathbf{no}{,}\mathsf{amb\text{-}c}"'] & \cdot
\end{tikzcd}$$

a arbitrary for $a \in \{\mathsf{case},\mathsf{seq},\mathsf{amb\text{-}l},\mathsf{amb\text{-}r}\}$ a arbitrary

Proof. The claim follows by case analysis. The reductions commute, or the redex of the $(S,\mathsf{amb\text{-}c})$ is discarded, or the internal $(\mathsf{amb\text{-}c})$ reduction becomes normal order. □

Remark 7.7. A complete set of forking diagrams for $(\mathscr{S},\mathsf{amb\text{-}c})$ does not exist. There are forks that cannot be closed, for example, $\mathtt{False} \xleftarrow{\mathbf{no},\mathsf{amb\text{-}l}} (\mathsf{amb}\ \mathtt{False}\ \mathtt{True}) \xrightarrow{\mathscr{S},\mathsf{amb\text{-}r}} \mathtt{True}$. Nevertheless, the following lemmas hold for $(\mathsf{amb\text{-}c})$ reductions within all surface contexts.

The following lemma will be used for the base case of the induction in the proof of Lemma 7.9. Note that we did not need such a lemma for the correct program transformations since the base cases were covered by property $\mathbb{C}_{\Leftarrow}(\cdot)$. As this property does not hold for (amb), we need the following specialised claim.

Lemma 7.8. If $s \xrightarrow{\mathscr{S},\mathsf{amb\text{-}c}} t$, then $s\Downarrow \implies t\downarrow$

Proof. Let $s \xrightarrow{\mathscr{S},\mathsf{amb\text{-}c}} t$. Then the claim follows by induction on the length of a sequence $RED \in \mathscr{CON}(s)$ by using Lemma 7.2 and the forking diagrams from Lemma 7.6. □

Lemma 7.9. If $s \xrightarrow{\mathscr{S},\mathsf{amb\text{-}c}} t$, then $t\uparrow \implies s\uparrow$.

Proof. Let $s \xrightarrow{S,\mathsf{amb\text{-}c}} t$. Then the claim can be shown by induction on the length of $RED \in \mathscr{DIV}(t)$ using Lemma 7.8 and the commuting diagrams for $(i\mathscr{S},\mathsf{amb\text{-}c})$. □

Analogously to (cps), let (ambs) be the reduction (amb) where the inner redex of $(\mathsf{amb\text{-}in})$ or $(\mathsf{amb\text{-}e})$ is inside a surface context.

Proposition 7.10. If $s \xrightarrow{S,\mathsf{ambs}} t$, then $t\uparrow \implies s\uparrow$

Proof. The claim follows from Lemmas 7.9 and 7.3 using Theorem 7.1 and Proposition 5.9. □

Corollary 7.11. The property $\mathbb{D}_{\Leftarrow}(\mathsf{amb})$ holds.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Example 7.12. There is a surprising counterexample showing that $s \xrightarrow{\mathscr{C},\mathsf{amb\text{-}c}} t$ and $t{\uparrow}$ do not imply $s{\uparrow}$.

$$\begin{array}{lcll} s & = & \texttt{letrec} & y = \lambda z.\texttt{amb True False} \\ & & & m = \lambda x.\texttt{if } (y\ 0) \texttt{ then } m\ x \texttt{ else True} \\ & & \texttt{in } m \texttt{ True} & \\ t & = & \texttt{letrec} & y = \lambda z.\texttt{True} \\ & & & m = \lambda x.\texttt{if } (y\ 0) \texttt{ then } m\ x \texttt{ else True} \\ & & \texttt{in } m \texttt{ True} & \end{array}$$

The definition ensures that $s \xrightarrow{\mathscr{C},\mathsf{amb\text{-}c}} t$. Inspecting the normal-order reduction, we see that $s{\Downarrow}$ but $t{\Uparrow}$. Consequences of this example are that Proposition 7.10 cannot be generalised from surface contexts to arbitrary contexts, and that the second part of the Standardisation Theorem (Theorem 7.13) cannot be generalised to arbitrary contexts.

We define the transformation (corr) as the union of those reductions and transformations that have been shown to be correct. The transformation (allr) is then the union of (ambs), (corr) and the inverse of (corr):

$$(\mathsf{corr}) \ := \ (\mathsf{lll}) \cup (\mathsf{lbeta}) \cup (\mathsf{seq}) \cup (\mathsf{case}) \cup (\mathsf{opt}) \qquad\qquad (\mathsf{allr}) \ := \ (\mathsf{corr}) \cup (\mathsf{corr})^{-1} \cup (\mathsf{ambs})$$

Now we formulate the Standardisation Theorem, which states that for every converging sequence consisting of all defined reductions and transformations there exists a normal-order reduction sequence that converges, and also that for every diverging sequence of reductions inside surface contexts there exists a normal-order reduction sequence that diverges.

Theorem 7.13 (Standardisation).

1 If t is a term with $t \xrightarrow{\mathscr{C},\mathsf{allr},*} t'$ where t' is a WHNF, then $t{\downarrow}$.

2 If t is a term with $t \xrightarrow{\mathscr{S},\mathsf{allr},*} t'$ where $t'{\Uparrow}$, then $t{\uparrow}$.

Proof.

1 Let $t \equiv t_0 \xrightarrow{\mathscr{C},red_1} t_1 \xrightarrow{\mathscr{C},red_2} \ldots \xrightarrow{\mathscr{C},red_{k-1}} t_k \equiv t'$ where t' is a WHNF. Using Theorem 7.1, Proposition 5.9 and Lemma 7.4, we have for every $t_i \xrightarrow{\mathscr{C},red_{i+1}} t_{i+1}$ that if $t_{i+1}{\downarrow}$, then $t_i{\downarrow}$. Using induction on k, we can then show $t_0{\downarrow}$.

2 Let $t \equiv t_0 \xrightarrow{\mathscr{S},red_1} t_1 \xrightarrow{\mathscr{S},red_2} \ldots \xrightarrow{\mathscr{S},red_{k-1}} t_k \equiv t'$ where $t'{\Uparrow}$. With Theorem 7.1 and Propositions 5.9 and 7.10, we have for every $t_i \xrightarrow{\mathscr{S},red_{i+1}} t_{i+1}$ that if $t_{i+1}{\uparrow}$ then $t_i{\uparrow}$. Using induction on k, we can then show $t_0{\uparrow}$. □

Since fair normal-order reduction induces the same notions of convergence and divergence as normal-order reduction, we can transfer the Standardisation Theorem to fair evaluation.

Corollary 7.14.

— If t is a term with $t \xrightarrow{\mathscr{C},\mathsf{allr},*} t'$ where t' is a WHNF, then $t{\downarrow}_F$.

— If t is a term with $t \xrightarrow{\mathscr{S},\mathsf{allr},*} t'$ where $t'{\Uparrow}$, then $t{\uparrow}_F$.

Note that Corollary 7.14 cannot be generalised to arbitrary contexts due to the counterexample of Example 7.12.

Using the second part of the Standardisation Theorem, we can prove the following proposition.

Proposition 7.15. Let s, t be expressions with $s{\Downarrow}$ ($s{\Downarrow}_F$, respectively) and $s \xrightarrow{\mathscr{S},\mathsf{allr},*} t$. Then $t{\Downarrow}$ ($t{\Downarrow}_F$, respectively).

Proof. Let $s \xrightarrow{\mathscr{S},\mathsf{allr},*} t$. We show that if $t{\uparrow}$, then $s{\uparrow}$. Since $t{\uparrow}$, there exists t' with $t \xrightarrow{\mathbf{no},*} t'$ and $t'{\Uparrow}$. The sequence $\xrightarrow{\mathscr{S},\mathsf{allr}*}\xrightarrow{\mathbf{no},*}$ is also a sequence of $(\mathscr{S}, \mathsf{allr})$ transformations. Hence, we can apply Theorem 7.13 and have $s{\uparrow}$. The claim for fair evaluation follows from Theorem 2.21. □

8. Some consequences of the Standardisation Theorem

In this section we will show that must-divergent expressions form an equivalence class with respect to $\sim_c$, prove a classical bottom-avoidance law and, finally, show that contextual equivalence can be defined by taking into account the must-convergence behaviour only.

8.1. *On the equivalence of* Ω*-terms*

Definition 8.1. Let s be an expression. If for all environments Env, $(\texttt{letrec}\ Env\ \texttt{in}\ s){\Uparrow}$, then s is called an Ω-term.

An example of such an expression is $(\texttt{letrec}\ y = \lambda z.z, x = x\ \texttt{in}\ x)$. The purpose of this section is to show that all Ω-terms are equal with respect to $\sim_c$. This result is easy to prove for calculi with erratic choice (see, for example, Schmidt-Schauss *et al.* (2004)) as in such calculi, for every reduction context R, the term $R[s]$, with s an Ω-term, is must-divergent. This does not hold for our calculi as the normal-order redexes are not unique and `amb` is bottom-avoiding: for example, for the reduction context $R \equiv (\texttt{amb}\ [\cdot]\ \texttt{True})$, the expression $R[s]$ may-converges for every expression s. Thus we will analyse the reduction behaviour of expressions of the form $S[s]$, where S is a surface context and s is an Ω-term. Then we will apply the context lemma.

In the following, we let S be a surface context and s be an Ω-term. Let RED be a normal-order reduction of $S[s]$, that is, $RED = S[s] = t_0 \xrightarrow{\mathbf{no}} t_1 \xrightarrow{\mathbf{no}} \dots \xrightarrow{\mathbf{no}} t_n$. We label subterms and bindings of the successor reducts t_i of $S[s]$ with the labels BM, BL, respectively, and we write B if we mean BL or BM. A subterm is a B-term, if it is within a term t of a BL-labelled binding $x = t$, or a subterm of the BM-labelled term.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Definition 8.2. A labelling is *admissible*, if the following properties hold:

- The label BL only occurs at bindings that are in a surface context, and the label BM labels at most one subterm in a surface context.
- The bound variables in bindings labelled BL only occur in B-labelled sub expressions.

We can reconstruct an S-part, a B-part and an E-part of every t_i as follows:

- The S-part of t_i can be obtained by replacing the BM-labelled subterm with the hole, and by removing all the BL-labelled bindings. Every `letrec` with empty binding is eliminated using (gc). We use $\mathrm{Rec}_S(t_i)$ to denote the resulting expression. It may be a surface context, or an expression.
- The B-part is obtained as ($\texttt{letrec}\ Env\ \texttt{in}\ r$), where Env consists exactly of all BL-labelled bindings, and r is the subexpression that is labelled BM. If there is no such expression, the B-part is undefined. We use $\mathrm{Rec}_B(t_i)$ to denote the resulting expression.
- The E-part is the set of bindings $x = t$ in a surface context of t_i that are not labelled BL. We use $\mathrm{Rec}_E(t_i)$ to denote the resulting environment.
- The EB-part $\mathrm{Rec}_{\mathrm{EB}}(t_i)$ is defined by ($\texttt{letrec}\ \mathrm{Rec}_E(t_i)\ \texttt{in}\ \mathrm{Rec}_B(t_i)$).

Note that the S- and EB-parts have the E-part in common. Note also that $\mathrm{Rec}_S(t_0) = S$ and $\mathrm{Rec}_B(t_0) = s$.

Now we define how the labelling is initially done, and how it is propagated in the reduction RED.

Definition 8.3. Initially, the labelling is $S[s^{\mathrm{BM}}]$. The labelling BM, initially and after every reduction step, is propagated according to the rule:

$$(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ r)^{\mathrm{BM}} \to (\texttt{letrec}\ x_1 =^{\mathrm{BL}} s_1, \ldots, x_n =^{\mathrm{BL}} s_n\ \texttt{in}\ r^{\mathrm{BM}})$$

For the normal-order reductions, the labelling is in most cases inherited using the rules of labelled reduction – the exceptions are as follows:

- (lbeta): $(s_1^{\mathrm{BM}}\ s_2)$ is never a normal-order redex, see Lemma 8.5.
- (case): $(\texttt{case}\ s_1^{\mathrm{BM}}\ alts)$ is never a normal-order redex, see Lemma 8.5. The same applies for $(\texttt{case}\ x\ alts)$, where x is bound (perhaps over a variable-chain) to the BM-labelled expression.
- (seq), (amb): no exception.
- (lll): Only (llet-e) has to be specified:

$$\begin{aligned}(\texttt{letrec}\ x =^{\mathrm{BL}} (\texttt{letrec}\ y_1 = r_1, \ldots, y_m = r_m\ \texttt{in}\ r_0), Env\ \texttt{in}\ u) \to \\ (\texttt{letrec}\ x =^{\mathrm{BL}} r_0, y_1 =^{\mathrm{BL}} r_1, \ldots, y_m =^{\mathrm{BL}} r_m, Env\ \texttt{in}\ u).\end{aligned}$$

- (cp): $(\texttt{letrec}\ x = s, \ldots C[x] \ldots) \to (\texttt{letrec}\ x = s, \ldots C[s])$. The cases $(\texttt{letrec}\ x = \lambda y.s, \ldots C[x^{\mathrm{BM}}] \ldots)$ and $(\texttt{letrec}\ x = (\lambda y.s)^{\mathrm{BM}}, \ldots C[x] \ldots)$ are impossible – see Lemma 8.5.

Definition 8.4. Given a sequence of normal-order reductions RED of $S[s]$, that is, $RED = S[s] = t_0 \xrightarrow{\mathbf{no}} t_1 \xrightarrow{\mathbf{no}} \ldots \xrightarrow{\mathbf{no}} t_n$, the following projected reduction sequences are defined:

- If $\mathrm{Rec}_S(t_i) \neq \mathrm{Rec}_S(t_{i+1})$, then $\mathrm{Rec}_S(t_i) \to \mathrm{Rec}_S(t_{i+1})$. We will show in Lemma 8.6 that these may be reductions (cp), (seq), (case), (amb), (lbeta),(lll) or (abs) in a surface context.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

— If $\mathrm{Rec}_{\mathrm{EB}}(t_i) \neq \mathrm{Rec}_{\mathrm{EB}}(t_{i+1})$, then $\mathrm{Rec}_{\mathrm{EB}}(t_i) \to \mathrm{Rec}_{\mathrm{EB}}(t_{i+1})$. We will show that these are normal-order reductions.

We use RED_S and RED_{EB}, respectively, to denote the derived sequences of reductions.

8.1.1. *Properties of B-labelled expressions* The construction of the induction proof on the length of the reduction sequence RED has to be done using several claims at once, since only a mutual induction of all the properties will be successful. So we prove three claims within one lemma.

Lemma 8.5. Let s be an Ω-term, S be a surface context and $RED = t_0 \xrightarrow{\mathbf{no}} t_1 \xrightarrow{\mathbf{no}} \ldots \xrightarrow{\mathbf{no}} t_n$ be a sequence of normal-order reductions of $S[s]$. Then the following hold:

1 The derived sequence of reductions RED_{EB} is a sequence of normal-order reductions of $\mathrm{Rec}_{\mathrm{EB}}(t_0)$. Moreover, the BM-labelled subexpression is never a value or bound to a value, and RED_{EB} never ends with a WHNF.
2 The labelling remains admissible after every normal-order reduction step.
3 The impossible cases of reductions in Definition 8.3 do not occur.

Proof. The base cases hold obviously. Now assume the lemma holds for the sequence of reductions until t_i. The cases where the normal-order reduction $t_i \xrightarrow{\mathbf{no}} t_{i+1}$ only modifies the S-part, but not the EB-part, or where the BM-labelled part is modified are trivial. In the other case, the unwinding algorithm first walks through the S-part, it may then visit the BM-labelled subterm, and then, finally, remains within the EB-part.

1 The same unwinding performance is valid for the term $\mathrm{Rec}_{\mathrm{EB}}(t_i)$, after visiting the BM-labelled subterm. We see that the reduction on $\mathrm{Rec}_{\mathrm{EB}}(t_i)$ either changes nothing, that is, $\mathrm{Rec}_{\mathrm{EB}}(t_i) = \mathrm{Rec}_{\mathrm{EB}}(t_{i+1})$ or is a normal-order reduction step of $\mathrm{Rec}_{\mathrm{EB}}(t_i)$ The BM-labelled term in t_{i+1} is neither a value nor bound to a value: since RED_{EB} is a sequence of normal-order reductions from 1 to $i+1$, we would otherwise obtain a WHNF $\mathrm{Rec}_{\mathrm{EB}}(t_{i+1})$ after the reduction step, which is impossible, since s is an Ω-term and $\mathrm{Rec}_S(t_0) = S$.
2 Since the BM-labelled expression is never a value or bound to a value, it is not possible to duplicate it, or to extract the direct subexpressions of the BM-labelled subexpression using (abs) or (case). The bound variables from BL-labelled bindings can only occur in B-subterms, also after a reduction step.
3 The impossible cases cannot happen as they correspond exactly to the case where $\mathrm{Rec}_{\mathrm{EB}}(t_{i+1})$ is a WHNF. □

8.1.2. *Equivalence of Ω-terms*

Lemma 8.6. The derived sequence of reductions RED_S consists of some reductions (cp), (seq), (case), (amb), (lbeta), (lll) or (abs) in a surface context. If t_n is a WHNF, then $\mathrm{Rec}_S(t_n)$ is a WHNF. If $\mathrm{Rec}_S(t_n)$ is a WHNF, then there is an $(\mathrm{lll},*)$ reduction of t_n to a WHNF.

Proof. The reductions may be completely in non-B-subexpressions. In this case the claim holds. The same holds if there are B-subexpressions that do not interfere with the

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

reduction. The exception is that a (case) reduction is performed in the B-part, and the effect in the E-part is like an (abs). Note that, in general, the reductions on $\mathrm{Rec}_S(t_i)$ are not forced to be normal-order reductions.

If t_n is a WHNF, the value subterm is not labelled BM, hence it is also in $\mathrm{Rec}_S(t_n)$, and thus it is also a WHNF. If $\mathrm{Rec}_S(t_n)$ is a WHNF, the only difference between it and a WHNF of t_n may be that some (llet-in) reductions are missing to reduce it to a WHNF. □

Theorem 8.7. Let s, s' be two Ω-terms. Then $s \sim_c s'$

Proof. We show that for every surface context S, we have $S[s]{\downarrow} \Rightarrow S[s']{\downarrow}$ and $S[s]{\uparrow} \Rightarrow S[s']{\uparrow}$. Since the other cases are symmetric, the context lemma implies that $s \sim_c s'$.

If $S[s]{\downarrow}$, let RED be some sequence of normal-order reductions of $S[s]$ ending in a WHNF. By Lemma 8.6, the surface reduction sequence RED_S yields a WHNF of $S = \mathrm{Rec}_S(t_0)$, hence the same reduction yields a WHNF of $S[s']$. Using the Standardisation Theorem yields $S[s']{\downarrow}$.

Now assume that $S[s]{\uparrow}$, but $S[s']{\Downarrow}$. Then let RED_1 be the sequence of normal-order reductions of $S[s]$ to a term s_0 with $s_0{\Uparrow}$. The sequence of reductions $RED_{1,S}$ reduces $\mathrm{Rec}_S(t_0)$ to a context S_0 that is not a WHNF. Note that $S_0 = \mathrm{Rec}_S(s_0)$. Let $r_0 := \mathrm{Rec}_B(s_0)$. Then $S_0[r_0] \xrightarrow{(\mathrm{lll}),*} s_0$, hence $S_0[r_0] \sim_c s_0$ by Proposition 6.10 and thus $S_0[r_0]{\Uparrow}$. Now we apply $RED_{1,S}$ to $S[s']$. If $RED_{1,S}$ eliminates the position of the hole of S, it is obvious that the result is a must-divergent expression since $s_0{\Uparrow}$, and we have reached a contradiction. Otherwise, we obtain a sequence of transformations $S[s'] \xrightarrow{\mathscr{S},\mathrm{allr},*} S_0[s']$ from $RED_{1,S}$. Using Proposition 7.15, we have $S_0[s']{\Downarrow}$. Now we use a fresh B-labelling for $S_0[s']$. There exists a term s'' with $S_0[s'] \xrightarrow{\mathbf{no},*} s''$ and s'' is a WHNF. Again using Lemma 8.6, we can construct RED_{2,S_0} that starts with $\mathrm{Rec}_S(S_0[s'])$ and ends in a surface context S' that is a WHNF. Now we apply RED_{2,S_0} to $S_0[r_0]$, leading to a WHNF, which contradicts $s_0{\Uparrow}$. Thus the assumption was wrong and $S[s']{\uparrow}$ holds. □

8.2. *Proving a bottom-avoidance law*

For reasoning we use specific Ω-terms. At the end of the section we will extend our results to all Ω-terms.

Definition 8.8 (Simple Ω-term). A term t is called a *simple* Ω*-term* in a context S, if either $t \equiv \Omega$ (see Example 3.4), or t is a variable x and S contains a `letrec`-binding $x = x$.

We define the bottom-avoidance law as the following program transformations:

(amb-l-o) $(\texttt{amb}\ s\ t) \rightarrow s$, if t is a simple Ω-term.
(amb-r-o) $(\texttt{amb}\ s\ t) \rightarrow t$, if s is a simple Ω-term.

Let (amb-o) be the union of (amb-l-o) and (amb-r-o). From now on we will extend the definition of forking and commuting diagrams by allowing the transformation $(\mathscr{S}, \mathrm{allr})$ instead of normal-order reductions in the existential quantified reductions on the

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

left- and right-hand sides of the diagrams. This is sufficient since Theorem 7.13 shows that for these cases a normal-order reduction also exists.

Lemma 8.9. A complete set of forking diagrams for $(\mathscr{S}, \textsf{amb-o})$ is

$$\begin{tikzcd}
\cdot \arrow[r, "{\mathscr{S},\textsf{amb-o}}"] \arrow[d, "{\mathbf{no},a}"'] & \cdot \arrow[d, dashed, "{\mathbf{no},a}"] \\
\cdot \arrow[r, dashed, "{\mathscr{S},\textsf{amb-o}}"'] & \cdot
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[r, "{\mathscr{S},\textsf{amb-o}}"] \arrow[d, "{\mathbf{no},a}"'] & \cdot \arrow[dl, dashed, "{\mathbf{no},a}"] \\
\cdot &
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[r, "{\mathscr{S},\textsf{amb-o}}"] \arrow[d, "{\mathbf{no},a}"'] & \cdot \\
\cdot \arrow[ur, dashed, "{\mathscr{S},\textsf{amb-o}}"'] &
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[r, "{\mathscr{S},\textsf{amb-o}}"] \arrow[d, "{\mathbf{no},a}"'] & \cdot \arrow[d, dashed, "{\mathscr{S},\textsf{gc}^{-1}}"] \\
\cdot \arrow[r, dashed, "{\mathscr{S},\textsf{amb-o}}"'] & \cdot
\end{tikzcd}$$

Proof. The claim follows by a case analysis (see Sabel and Schmidt-Schauss (2006)). □

Lemma 8.10. Let s, t be terms with $s \xrightarrow{iS,\textsf{amb-o}} t$. Then s is a WHNF if and only if t is a WHNF.

Lemma 8.11. $\mathbb{C}_{\Rightarrow}(\textsf{amb-o})$

Proof. By induction on the length l of a reduction sequence $RED_s \in \mathscr{CON}(s)$, we can show that there exists a sequence of $(\mathscr{S}, \textsf{allr})$ transformations starting with t that leads to a WHNF. The base case is covered by Lemma 8.10. The induction step uses the forking diagrams from Lemma 8.9. Finally, Theorem 7.13 (1) shows that $t\downarrow$, so there exists $RED_t \in \mathscr{CON}(t)$. □

Lemma 8.12. A complete set of commuting diagrams for $(i\mathscr{S}, \textsf{amb-o})$ is

$$\begin{tikzcd}
\cdot \arrow[r, "{\mathscr{S},\textsf{amb-o}}"] \arrow[d, dashed, "{\mathbf{no},a}"'] & \cdot \arrow[d, "{\mathbf{no},a}"] \\
\cdot \arrow[r, dashed, "{\mathscr{S},\textsf{amb-o}}"'] & \cdot
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[r, "{\mathscr{S},\textsf{amb-o}}"] \arrow[d, dashed, "{\mathscr{S},a}"'] & \cdot \arrow[d, "{\mathbf{no},a}"] \\
\cdot \arrow[r, dashed, "{\mathscr{S},\textsf{amb-o}}"'] & \cdot
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[r, "{\mathscr{S},\textsf{amb-o}}"] \arrow[dr, dashed, "{\mathbf{no},a}"'] & \cdot \arrow[d, "{\mathbf{no},a}"] \\
 & \cdot
\end{tikzcd}
\quad
\begin{tikzcd}
\cdot \arrow[r, "{\mathscr{S},\textsf{amb-o}}"] \arrow[d, dashed, "{\mathbf{no},\textsf{lamb}}"'] & \cdot \\
\cdot \arrow[ur, dashed, "{\mathscr{S},\textsf{amb-o}}"'] &
\end{tikzcd}$$

Proof. A detailed case analysis is given in Sabel and Schmidt-Schauss (2006). □

Lemma 8.13. $\mathbb{C}_{\Leftarrow}(\textsf{amb-o})$

Proof. Let $s_0 \xrightarrow{S,\textsf{amb-o}} t_0$ and $t_0\downarrow$. By induction on the measure (a, b) with $b = \mu_{lll}(s_0)$ and $a = \texttt{rl}(RED_t)$, ordered lexicographically, where $RED_t \in \mathscr{CON}(t_0)$, we can show that there exists a sequence of $(\mathscr{S}, \textsf{allr})$ transformations that leads from s_0 to a WHNF. The base case is covered by Lemma 8.10; the induction step uses the commuting diagrams from Lemma 8.12. Finally, Theorem 7.13 (1) shows the claim. □

Since $\mathbb{C}_{\Rightarrow}(\textsf{amb-o})$ and $\mathbb{C}_{\Leftarrow}(\textsf{amb-o})$ hold, we have the following corollary.

Corollary 8.14. If $s \xrightarrow{\mathscr{S},\textsf{amb-o}} t$ then $s\Uparrow$ if and only if $t\Uparrow$.

Lemma 8.15. $\mathbb{D}_{\Leftarrow}(\textsf{amb-o})$

Proof. Induction on the measure (a, b) where $b = \mu_{lll}(s)$ and $a = \texttt{rl}(RED_t)$ with $RED_t \in \mathscr{DIV}(t)$ shows that there exists a sequence of $(\mathscr{S}, \textsf{allr})$ transformations starting with s and ending in a term that must-diverges. The base case is covered by Corollary 8.14 and

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

the induction step uses the commuting diagrams from Lemma 8.12. Finally, Theorem 7.13 part 2 shows that $s\uparrow$, that is, there exists a reduction sequence $RED \in \mathcal{DIV}(s)$. □

Lemma 8.16. $\mathbb{D}_{\Rightarrow}(\textsf{amb-o})$

Proof. Induction on $\texttt{rl}(RED_s)$ with $RED_s \in \mathcal{DIV}(s)$ shows the existence of a sequence of $(\mathcal{S}, \textsf{allr})$ transformations from t to a term that must-diverges. The base case for this induction is covered by Corollary 8.14; the induction step uses the forking diagrams from Lemma 8.9. The last step uses Theorem 7.13 (2) to transform the sequence of $(\mathcal{S}, \textsf{allr})$ transformations into a normal-order reduction sequence $RED_t \in \mathcal{DIV}(t)$. □

Since $(\textsf{amb-o})$ fulfils the $\mathbb{CD}$-properties we have the following proposition.

Proposition 8.17. If $s \xrightarrow{\textsf{amb-o}} t$, then $s \sim_c t$.

Theorem 8.18. For all Ω-terms s and expressions t, we have

$$\texttt{amb}\ s\ t \sim_c t \sim_c \texttt{amb}\ t\ s.$$

Proof. Let s be an Ω-term. Then from Theorem 8.7, we have $s \sim_c \Omega$, since Ω is an Ω-term. The claim now follows as $\sim_c$ is a congruence and from Proposition 8.17. □

8.3. *On the relation between* $\leqslant_c^{\downarrow}$ *and* $\leqslant_c^{\Downarrow}$

A consequence of $\texttt{amb}$ being bottom-avoiding is that $s \leqslant_c^{\Downarrow} t$ implies $t \leqslant_c^{\downarrow} s$, which we will show by similar arguments to those given in Moran (1998) and Lassen (1998). Let the context BA be defined by $BA \equiv (\texttt{amb}\ \mathbf{I}\ (\texttt{seq}\ [\cdot]\ (\lambda x.\Omega)))\ \mathbf{I}$.

Lemma 8.19. $BA[s]\Downarrow$ if and only if $s\Uparrow$.

Proof. The claim follows by inspecting the possible normal-order reductions and using the Standardisation Theorem – details can be found in Sabel and Schmidt-Schauss (2006). □

Proposition 8.20. $\leqslant_c^{\Downarrow}\ \subseteq\ (\leqslant_c^{\downarrow})^{-1}$

Proof. Let s, t be arbitrary terms with $s \leqslant_c^{\Downarrow} t$. Then $\forall C \in \mathcal{C} : C[s]\Downarrow \implies C[t]\Downarrow$ and thus also $\forall C \in \mathcal{C} : BA[C[s]]\Downarrow \implies BA[C[t]]\Downarrow$. Using Lemma 8.19, this is equivalent to $\forall C \in \mathcal{C} : C[s]\Uparrow \implies C[t]\Uparrow$, and also $\forall C \in \mathcal{C} : C[t]\downarrow \implies C[s]\downarrow$, hence $t \leqslant_c^{\downarrow} s$. □

Let $\sim_c^{\downarrow}$ be the symmetrisation of may-convergence, that is, $s \sim_c^{\downarrow} t$ if and only if $s \leqslant_c^{\downarrow} t \wedge t \leqslant_c^{\downarrow} s$.

Corollary 8.21. If $s \leqslant_c t$ then $s \sim_c^{\downarrow} t$.

A consequence of Proposition 8.20 is that contextual equivalence can be defined using must-convergence only.

Corollary 8.22. $s \sim_c t$ if and only if $\forall C : C[s]\Downarrow \iff C[t]\Downarrow$

The remaining two propositions of this section show that $\leqslant_c$ is not an equivalence.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Proposition 8.23. Let s be an Ω-term and t be an arbitrary term. Then $s \leqslant_c^{\downarrow} t$.

Proof. This can be proved using the context lemma for may-convergence, the Standardisation Theorem and a complete set of forking diagrams for the transformation $s \to t$, and, finally, Theorem 8.7. A complete proof can be found in Sabel and Schmidt-Schauss (2006). □

Proposition 8.24. $\leqslant_c$ is not symmetric.

Proof. Let $s \equiv \texttt{choice}\ \Omega\ \mathbf{I}$ and $t \equiv \mathbf{I}$. From Proposition 8.23, we have $s \leqslant_c^{\downarrow} t$. For all surface contexts S, we can transform $S[s]$ into $S[t]$:

$$\begin{aligned} S[s] \equiv S[(\texttt{amb}\ (\lambda x.\Omega)\ (\lambda x.\mathbf{I}))\ \texttt{True}] &\xrightarrow{\mathcal{S},\mathsf{amb}} S[(\lambda x.\mathbf{I})\ \texttt{True}] \\ &\xrightarrow{\mathcal{S},\mathsf{lbeta}} S[(\texttt{letrec}\ x = \texttt{True in}\ \mathbf{I})] \\ &\xrightarrow{\mathcal{S},\mathsf{gc}} S[\mathbf{I}] \equiv S[t]. \end{aligned}$$

From the Standardisation Theorem, it follows that for all surface contexts $S[t]\Uparrow \implies S[s]\Uparrow$. Hence, using Corollary 3.15, we have $s \leqslant_c t$. Obviously, $s\Uparrow$ and $t\Downarrow$. Thus, the empty context shows that $t \not\leqslant_c^{\Downarrow} s$. □

9. Conclusions and directions for further research

We have presented an extended call-by-need lambda-calculus with a non-deterministic `amb` operator together with a fair small-step reduction semantics. The appropriate program equivalence is the contextual equivalence based on may- and must-termination. We have proved that all deterministic reduction rules and several additional program transformations preserve contextual equivalence, which permits useful program transformation, and, in particular, partial evaluation using deterministic reductions. This is clearly an improvement on the results in Moran's thesis, since we have also proved correctness with respect to must-convergence. The proof methodology consists of a context lemma, which restricts the number of contexts that need to be examined, and the computation of complete sets of commuting and forking diagrams for the reductions and transformations.

A remarkable result is that contextual preorder $\leqslant_c$ and equivalence $\sim_c$ can be defined by observing the must-convergent behaviour only. We have also shown that all must-divergent expressions are in the same equivalence class of $\sim_c$, and that must-divergent expressions are not least with respect to $\leqslant_c$.

Using the proof tools we have developed in this paper, a promising application would be to prove the correctness of further program transformations, for example, a rule for inlining expressions that are used only once, or for deterministic expressions (that is, ones that do not contain `amb` expressions and may need to satisfy some other conditions). Future research should also investigate more involved inductive proof rules like Bird's take-lemma. Our proof method using reduction diagrams may also be used to prove the correctness of the program transformations called 'global' in Santos (1995) if the analysed transformation is decomposed into a sequence of more local transformations. Another application area lies in proofs of the correctness of static analyses: for example, an analysis

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

of non-terminating terms and subterms. The results of Schmidt-Schauss *et al.* (2005), where we used this technique for a deterministic lambda calculus to prove the correctness of strictness analysis, look promising.

Another research direction would be to apply the methods of this paper (proving a context lemma, computing sets of reduction diagrams and using an unfair normal-order reduction to ease proofs) to other non-deterministic calculi. A further challenge would be to perform an investigation similar to that in Sabel (2003a), and thus to prove the correctness of program transformations used in Haskell (Peyton Jones 2003) compilers with respect to an extension with `amb`. After switching off incorrect transformations, the result would be a semantics preserving compiler for Haskell extended with `amb`.

Acknowledgements

We would like to thank the anonymous referees for their valuable comments and suggestions.

References

Ariola, Z., Felleisen, M., Maraist, J., Odersky, M. and Wadler, P. (1995) A call-by-need lambda calculus. In: *Proc. POPL '95, 22'nd Annual Symposium on Principles of Programming Languages, San Francisco, California*, ACM Press.

Barendregt, H. (1984) *The Lambda Calculus. Its Syntax and Semantics*, North-Holland.

Bois, A. R. D., Pointon, R. F., Loidl, H.-W. and Trinder, P. W. (2002) Implementing declarative parallel bottom-avoiding choice. In: *SBAC-PAD*, IEEE Computer Society 82–92.

Carayol, A., Hirschkoff, D. and Sangiorgi, D. (2005) On the representation of McCarthy's amb in the pi-calculus. *Theor. Comput. Sci.* **330** (3) 439–473.

Fernández, M. and Khalil, L. (2003) Interaction nets with McCarthy's amb: Properties and applications. *Nordic J. Comput.* **10** (2) 134–162.

Hallgren, T. and Carlsson, M. (1995) Programming with fudgets. In: Jeuring, J. and Meijer, E. (eds.) Advanced Functional Programming. *Springer-Verlag Lecture Notes in Computer Science* **925** 137–182.

Henderson, P. (1980) *Functional Programming – Application and Implementation*, Series in Computer Science, Prentice Hall International.

Henderson, P. (1982) Purely Functional Operating Systems. In: Darlington, J., Henderson, P. and Turner, D. A. (eds.) *Functional Programming and its Applications*, Cambridge University Press 177–192.

Hughes, J. and Moran, A. (1995) Making choices lazily. In: *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, ACM Press 108–119.

Jones, M. P. and Hudak, P. (1993) Implicit and explicit parallel programming in Haskell. Technical Report CT 06520-2158, Department of Computer Science, Yale University.

Kutzner, A. (2000) *Ein nichtdeterministischer call-by-need Lambda-Kalkül mit erratic choice: Operationale Semantik, Programmtransformationen und Anwendungen*, Dissertation, J. W. Goethe-Universität Frankfurt (in German).

Kutzner, A. and Schmidt-Schauss, M. (1998) A nondeterministic call-by-need lambda calculus. In: *International Conference on Functional Programming 1998*, ACM Press 324–335.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Lassen, S. B. (1998) *Relational Reasoning about Functions and Nondeterminism*, Ph.D. thesis, Department of Computer Science, University of Aarhus. (BRICS Dissertation Series DS-98-2.)

Lassen, S. B. (2006) Normal Form Simulation for McCarthy's Amb. In: Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI). *Electronic Notes in Theoretical Computer Science* **155** 445–465.

Lassen, S. B., Levy, P. B. and Panangaden, P. (2005) Divergence-least semantics of amb is Hoare. Short presentation at the APPSEM II workshop, Frauenchiemsee, Germany. (Available at `http://www.cs.bham.ac.uk/~pbl/papers/`.)

Lassen, S. B. and Moran, A. (1999) Unique fixed point induction for McCarthy's amb. In: Kutylowski, M., Pacholski, L. and Wierzbicki, T. (eds.) MFCS. *Springer-Verlag Lecture Notes in Computer Science* **1672** 198–208.

Machkasova, E. and Turbak, F. A. (2000) A calculus for link-time compilation. In: Smolka, G. (ed.) ESOP. *Springer-Verlag Lecture Notes in Computer Science* **1782** 260–274.

Mann, M. (2005a) *A Non-Deterministic Call-by-Need Lambda Calculus: Proving Similarity a Precongruence by an Extension of Howe's Method to Sharing*, Dissertation, J. W. Goethe-Universität, Frankfurt.

Mann, M. (2005b) Congruence of bisimulation in a non-deterministic call-by-need lambda calculus. *Electronic Notes in Theoretical Computer Science* **128** (1) 81–101.

Mann, M. and Schmidt-Schauss, M. (2006) How to prove similarity a precongruence in non-deterministic call-by-need lambda calculi. Frank Report 22, Institut für Informatik. J. W. Goethe-Universität Frankfurt.

McCarthy, J. (1963) A Basis for a Mathematical Theory of Computation. In: Braffort, P. and Hirschberg, D. (eds.) *Computer Programming and Formal Systems*, North-Holland 33–70.

Moran, A. and Sands, D. (1999) Improvement in a lazy context: an operational theory for call-by-need. In: *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM Press 43–56.

Moran, A. K. (1998) *Call-by-name, Call-by-need, and McCarthy's Amb*, Ph.D. thesis, Department of Computing Science, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden.

Natarajan, V. and Cleaveland, R. (1995) Divergence and fair testing. In: Fülöp, Z. and Gécseg, F. (eds.) ICALP. *Springer-Verlag Lecture Notes in Computer Science* **944** 648–659.

Peyton Jones, S. (ed.) (2003) *Haskell 98 language and libraries: the Revised Report*, Cambridge University Press. (Website: `www.haskell.org`.)

Peyton Jones, S. and Marlow, S. (2002) Secrets of the Glasgow Haskell Compiler inliner. *J. Funct. Programming* **12** (4+5) 393–434.

Sabel, D. (2003a) A guide through HasFuse. (Available at `http://www.ki.informatik.uni-frankfurt.de/research/diamond/hasfuse/`.)

Sabel, D. (2003b) Realising nondeterministic I/O in the Glasgow Haskell Compiler. Frank Report 17, Institut für Informatik, J. W. Goethe-Universität Frankfurt am Main.

Sabel, D. and Schmidt-Schauss, M. (2006) A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. Frank Report 24, Institut für Informatik, J. W. Goethe-Universität Frankfurt.

Santos, A. (1995) *Compilation by Transformation in Non-Strict Functional Languages*, Ph.D. thesis, Glasgow University, Department of Computing Science.

Schmidt-Schauss, M. (2003) FUNDIO: A Lambda-Calculus with a `letrec`, `case`, Constructors, and an IO-Interface: Approaching a Theory of `unsafePerformIO`. Frank Report 16, Institut für Informatik, J. W. Goethe-Universität Frankfurt.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press

Schmidt-Schauss, M., Schütz, M. and Sabel, D. (2004) On the safety of Nöcker's strictness analysis. Frank Report 19, Institut für Informatik, J. W. Goethe-Universität Frankfurt.

Schmidt-Schauss, M., Schütz, M. and Sabel, D. (2005) A complete proof of the safety of Nöcker's strictness analysis. Frank Report 20, Institut für Informatik. J. W. Goethe-Universität Frankfurt.

Søndergaard, H. and Sestoft, P. (1992) Non-determinism in functional languages. *Comput. J.* **35** (5) 514–523.

Trinder, P. W., Hammond, K., Loidl, H.-W. and Peyton Jones, S. L. (1998) Algorithm + Strategy = Parallelism. *J. Funct. Programming* **8** (1) 23–60.

Wells, J. B., Plump, D. and Kamareddine, F. (2003) Diagrams for meaning preservation. In Nieuwenhuis, R. (ed.) RTA. *Springer-Verlag Lecture Notes in Computer Science* **2706** 88–106.

https://doi.org/10.1017/S0960129508006774 Published online by Cambridge University Press