JFP **18** (4): 503–551, 2008. © 2007 Cambridge University Press
doi:10.1017/S0956796807006624 First published online 8 November 2007 Printed in the United Kingdom

Safety of Nöcker's strictness analysis

MANFRED SCHMIDT-SCHAUSS and DAVID SABEL
Institut für Informatik, Johann Wolfgang Goethe-Universität, Postfach 11 19 32,
D-60054 Frankfurt, Germany
(*e-mail:* schauss@ki.informatik.uni-frankfurt.de)

MARKO SCHÜTZ
University of Puerto Rico at Mayagüez, Department of Mathematical Sciences, Mayagüez, PR 00681

Abstract

This paper proves correctness of Nöcker's method of strictness analysis, implemented in the Clean compiler, which is an effective way for strictness analysis in lazy functional languages based on their operational semantics. We improve upon the work Clark, Hankin and Hunt did on the correctness of the abstract reduction rules in two aspects. Our correctness proof is based on a functional core language and a contextual semantics, thus proving a wider range of strictness-based optimizations as correct, and our method fully considers the cycle detection rules, which contribute to the strength of Nöcker's strictness analysis.

Our algorithm SAL is a reformulation of Nöcker's strictness analysis algorithm in a functional core language LR. This is a higher order call-by-need lambda calculus with `case`, constructors, `letrec`, and `seq`, which is extended during strictness analysis by set constants like `Top` or *Inf*, denoting sets of expressions, which indicate different evaluation demands. It is also possible to define new set constants by recursive equations with a greatest fixpoint semantics. The operational semantics of LR is a small-step semantics. Equality of expressions is defined by a contextual semantics that observes termination of expressions. Basically, SAL is a nontermination checker. The proof of its correctness and hence of Nöcker's strictness analysis is based mainly on an exact analysis of the lengths of evaluations, i.e., normal-order reduction sequences to WHNF. The main measure being the number of "essential" reductions in evaluations.

Our tools and results provide new insights into call-by-need lambda calculi, the role of sharing in functional programming languages, and into strictness analysis in general. The correctness result provides a foundation for Nöcker's strictness analysis in Clean, and also for its use in Haskell.

Contents

0 Introduction

0.1 General Introduction and Motivation

Strictness analysis is an essential phase when compiling programs in lazy functional languages such as Haskell (Peyton Jones 2003) and Clean Plasmeijer & van Eekelen (2003). Many optimizations become possible only with the information gained in strictness analysis. Among the applications are: optimizations by rearranging the evaluation sequence, proving preconditions for correct application of program transformations (see Santos (1995); Peyton Jones & Santos (1994)), and detecting possibilities for conservative parallel evaluation.

There are different methods to perform the strictness analysis, e.g. ad hoc strictness computation integrated with optimizations in compilation schemes, strictness analysis based on denotational semantics and abstract interpretation, use of type systems, and strictness analysis based on operational semantics.

A very effective way for strictness analysis in functional languages is algorithms based on abstract reduction, i.e. on the operational semantics. Nöcker's strictness analysis for Clean (see Nöcker (1990); Nöcker (1993)) is a prominent example.

This paper contributes to increase the applicability and trustworthiness of Nöcker's method and to provide foundations for its application, e.g. in Haskell.

In their paper (Clark *et al.* 2000) Clark, Hankin and Hunt show the correctness of the part of the algorithm that pushes the abstract values through the program using the operational semantics. However, this is only part of the correctness, since the cycle-detection rules are not considered, which account for much of the strength of Nöcker's algorithm. Moreover, the proof in (Clark *et al.* 2000) is limited in scope due to a restriction of the language and the semantics, which does not justify all useful program transformations, e.g. their method cannot justify the modification of previously defined supercombinators using strictness information.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Our paper contains a description of SAL, a reformulation of Nöcker's strictness analysis algorithm based on a functional core language LR, i.e., on an extended call-by-need lambda-calculus with constructors, `letrec`, `case` and `seq`, where the algorithm SAL uses abstract lambda terms including set constants such as $\top, \bot$, and, where other set constants can be defined by recursive equations. Graph reduction is modeled by `letrec`, which can also describe recursive definitions and can explicitly treat the sharing inherent in lazy functional languages. Our syntax is slightly different from Nöcker's, since we assume that all functions are defined in a (global) `letrec` environment including all relevant function definitions of the program; in addition our syntax can also express equality of (instances of) set constants by sharing (see Remark 3.35). An important improvement is that we use a contextual semantics, which equates expressions s, t, iff s and t exhibit the same termination behavior, if put in any program context. Though defined by operational means, the contextual semantics is superior to semantics based on conversion or on infinite normal forms or trees like Lévy–Longo trees (Lévy 1976) or Böhm trees (Barendregt 1984), and justifies a rich set of program transformations. The new result of this paper is a correctness proof of SAL w.r.t. our contextual semantics. It also contains a proof of correctness of the strictness optimization w.r.t. our contextual semantics. These results carry over to Haskell and Clean, since our semantics is correct for their respective semantics.

A foundational, though tedious, part of the proof of correctness of SAL is to show that the reduction rules used as transformations and a set of additional transformation rules (called extra transformations) do not change the contextual equivalence class of expressions, and to prove that there is a useful measure for closed terms that counts the number of "essential" reduction steps in an evaluation, and also to show that this measure is not influenced by the additional transformations. An intuitive reason for the existence of such a rather well-behaved length measure appears to be the exact treatment of sharing in our call-by-need lambda-calculus LR. We also show that the length measure is robust w.r.t. changing the order of reduction, e.g. using strictness of expressions, and also invariant w.r.t. simplification rules and rules that only rearrange the let-structure of expressions. The final induction in the correctness proof is based on the "essential" length of evaluations. The domains commonly used in the literature on denotational semantics do not provide such a measure, hence are not appropriate for the correctness proof.

Examples of Strictness Analysis

An expression `f` is called *strict* in argument i for arity n, iff the evaluation starting with `f` $t_1 \ldots t_{i-1} \ \bot \ t_{i+1} \ldots , t_n$ will never yield a weak head normal form by evaluation, where $\bot$ represents terms without WHNF, a more rigorous definition using contextual equivalence is given in Definition 2.2.

The first example is the combinator `K` with definition `K` $= \lambda x, y.x$, which is strict in its first argument (for arity 2). This will be detected by Nöcker's method as follows. With $\top$ representing every closed term, `K` $\bot \ \top$ (abstractly) reduces to $\bot$ indicating that `K` is indeed strict in its first argument.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

A nontrivial example (see also Example 3.40) is `length` with the following definition:

$$\begin{array}{ll} \texttt{length} = \texttt{letrec}\ len & = \lambda lst.\lambda a.\ \texttt{case}_{lst}\ lst \\ & \qquad\qquad (\texttt{Nil} \rightarrow a) \\ & \qquad\qquad (y : ys \rightarrow len\ ys\ (a+1)) \\ \qquad\quad \texttt{in}\ len. & \end{array}$$

Reducing (`length` $\top$ $\bot$) using the rules of the abstract calculus (using reductions, transformations, and case analysis) results either in $\bot$ or in an expression that is essentially the same as (`length` $\top$ $\bot$). Since the same expression is generated, and at least one (essential) normal order reduction was necessary, the strictness analysis algorithm concludes that the evaluation of the expression loops. That this constitutes a loop is not obvious, since $\top$ may stand for different terms. Summarizing, the answer will be that `length` is strict in its second argument.

Our proof of correctness of SAL justifies this reasoning by loop detection, even in connection with abstract set constants like $\top$ or *Inf*, whereas the proof in (Clark *et al.* 2000) cannot be used in this example.

0.2 Previous Work

In (Nöcker 1990; 1992; 1993) Nöcker described a strictness analysis based on abstracting the operational semantics of a nonstrict functional programming language. This strictness analysis is very appealing, both intuitively and pragmatically, but it has proven theoretically challenging.

The key concept is to add new names for abstract constants, such as $\bot$ for all terms without WHNF, or $\top$ for all expressions, and to add to the reduction rules from the term calculus appropriate (abstract) reduction rules capturing their semantics. This includes a case analysis, which can be modeled by unions or by nondeterministic choices, together with subsumption rules based on $\bot \leqslant t \leqslant \top$. This analysis was implemented at least twice: once by Nöcker in C for Concurrent Clean (Nöcker *et al.* 1991) and once by Schütz in Haskell (Schütz 1994). As of Concurrent Clean version 2.1 Nöcker's C-implementation is still in use in the compiler. The analysis is not very expensive to implement, runs quickly without large memory requirements, and obtains good results. It is able to find strictness information without being restricted to use finite domains, and it is also possible to exploit implementation details from several function definitions in one run.

Its drawback seems to be the slow progress in its theoretical foundation. Nöcker (Nöcker 1992) himself proved the correctness of the analysis for orthogonal term rewriting systems only. In (Schmidt-Schauß *et al.* 1995) we showed the correctness of the analysis for a supercombinator-based functional core language. In that exposition a treatment of sharing and `letrec` was missing. Then Clark, Hankin and Hunt (Clark *et al.* 2000) proved the correctness of a significant subset of the analysis, but did not consider the loop-detection rules. Since the loop-detection rules may well be the most important aspect of strictness analysis by abstract reduction, this paper provides a formal account of the analysis using a language with explicit sharing and proving correctness for all of the rules of Nöcker's algorithm.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

0.3 Overview of the Paper

This paper consists of three main parts and an appendix (Schmidt-Schauß *et al.* 2007), where almost all proofs in the first two parts on the calculus can be found in the appendix. The appendix is available on the home page of the *Journal of Functional Programming*.

The parts are as follows.

1. A description of the calculus LR and the normal order reduction (part 1).
2. A detailed analysis of the properties of contextual equivalence and of the length of evaluations (part 2).
3. A description of the strictness analysis algorithm, its data structures, a proof of its correctness, and some illustrating examples (part 3).

The first part (part 1) is concerned with describing the calculus for a call-by-need functional core language LR using sharing and with investigating equivalences and variants of lengths of evaluations. Set constants like $\top$ or *Inf* are permitted in the abstract language $LR_{\mathcal{U}}$.

The core language LR provides `letrec`, the usual primitives like a weakly typed `case`—in fact a `case` for every type—constructors, lambda abstraction, application, and `seq`. The core language and its analysis is partly borrowed from (Schmidt-Schauß 2003). The weak typing makes the language LR more similar in behavior to a typed functional programming language (see Example 1.12), and also permits optimizations of the strictness analysis that otherwise could not be justified.

The reduction rules for the language LR are defined for any matching subexpression. The normal order reduction is then defined as a specific strategy uniquely determining the next subexpression for reduction. We define contextual equivalence as usual where the only observation is successful termination of an expression's evaluation (also called convergence).

In the second part (part 2) we show that contextual equivalence is stable w.r.t. all reduction and transformation rules. In this, we employ a context lemma and the computing of overlappings of rules leading to so-called complete sets of commuting and forking diagrams, which are explained in the appendix, and which are an essential tool for proving contextual equivalence. The well-founded measure used in the final induction proof (Theorem 3.32) is the "essential" length of normal order reduction sequences. For technical reasons, we need to provide several measures each counting a specific set of reduction rules occurring in the normal order reduction sequence. We then study how these measures are affected by reduction steps and transformations.

A further base is a theorem on the correctness of copying parts of concrete terms. We conjecture that unrestricted copying is a correct program transformation, but were not able to prove it correct.[1]

[1] This was proved in the meantime, see reports on `www.ki.informatik.uni-frankfurt.de/papers/schauss`

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

In the third part (part 3) we define the algorithm SAL as a reformulation of Nöcker's algorithm. It may use previously computed strictness knowledge about functions and strictness of built-in functions. The main data structure of SAL is a directed graph with abstract expressions as nodes, where the directed edges correspond to abstract reductions, case analysis, or to cycle checks. An abstract expression is an expression where also set constants are permitted as abstractions. Set constants may be $\top$, the set of all closed expressions, or Inf, the set of all expressions evaluating to infinite lists or lists without tails. It is also possible to define new set constants by recursive equations. An abstract expression represents a set of terms from the concrete core language, which are called its concretizations. The definition of concretizations is complicated by the presence of set constants, since it consists of a combination of semantics (the concretizations of the set constants), and syntax. Fortunately, it is possible to prove a strong connection between the abstract reductions on abstract terms and the reduction lengths of the corresponding concretizations (Theorem 3.28). This leads to a concise representation of unions, and the elegance of the final proof indicates that a directed graph is appropriate to check Nöcker's termination conditions.

The conditions on successful termination of SAL give new insights into the nature of the algorithm. In fact SAL is a nontermination checker for an infinite set of expressions specified by an abstract expression. The proof justifies the intuition that certain reductions (normal order and reductions at strict positions) "make progress", whereas this is not the case for several other reductions and transformations.

Our approach makes a proof of the correctness of the strictness optimization possible and also yields some insight into the effects of strictness optimization in a call-by-need calculus (see Theorem 3.37 and Corollary 3.38), since we show that an evaluation exploiting strictness of functions does not improve the number of (essential) reductions; however, a compiler can produce optimized code for an abstract machine if strictness information is available.

Correctness of SAL is proved in Theorem 3.32 showing that SAL correctly detects nontermination provided SAL terminates. Corollaries 3.33 and 3.34 claim that strictness information is correctly detected on a custom-tailored input. We also explain some runs of SAL on examples, illustrating the different rules, and also a slight extension to Nöcker's formulation.

1 The Calculus LR

In this part we introduce the calculus LR, i.e. its syntax, the operational semantics, and the program equivalence based on contextual equivalence. In subsection 1.1, we argue, why we chose to use a `letrec`-calculi and discuss the relation between LR and nonstrict functional programming languages. In subsection 1.2, we introduce the syntax of LR, followed by subsection 1.3 where we define the normal order reduction for LR. Based on the notion of termination we introduce contextual equivalence in subsection 1.3. The part ends with a discussion (subsection 1.4) on semantical aspects w.r.t. the design of our calculus, e.g. why we add an operator `seq` for sequential evaluation.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

1.1 The Language and Equality: Design Decisions

Nöcker's description of his strictness analysis (Nöcker 1990) was based on a graphbased supercombinator language, which matched the Clean (core-)language (Plasmeijer & van Eekelen 2003). It is well known that such languages can also be represented in a lambda calculus extended with `letrec`. In order to use Nöcker's strictness analysis for Haskell, it is advantageous to show correctness within a language that is close to a core language of Haskell (see Peyton Jones & Marlow (2002)). This means that a correctness proof in our calculus can be transferred to Clean as well as Haskell.

There are semantic aspects to consider. Looking at papers dealing with `letrec`-calculi or cyclic lambda calculi, the semantics is often a conversion semantics (e.g. Ariola *et al.* (1995)), which is in general too weak to prove all the optimizations and transformations used in a compiler of a functional programming language. For example, the transformation from lambda calculus to a supercombinator language cannot be justified by conversion equality alone. It is also known that optimization using only conversion is insufficient for real-world optimizations (Sands *et al.* 2002) in certain lambda calculi. There are also other semantics such as defining equality by Böhm trees (Barendregt 1984), Lévy–Longo-trees, or by identical normal-forms w.r.t. an infinitary lambda-calculus (Kennaway *et al.* 1993). These semantics are weaker than semantics using contextual equivalences, i.e., they have fewer equations, the pair of terms $\lambda x.xx$ and $\lambda x.x(\lambda y.xy)$ is an example (see Dezani-Ciancaglini *et al.* (1999)). Also following (Plotkin 1975) and (Pitts 2002), we opt for the use of contextual equivalence (also called behavioral equivalence), which equates two expressions s, t, if they have the same termination behavior in all contexts; more rigorously: s, t are equal, if s and t are not distinguishable by substituting them in any context C and checking whether the normal order reduction in this calculus reduces $C[s]$ (or $C[t]$, respectively) to a weak-head normal form (WHNF).

The proof of a correctness result of a large part of the rules of Nöcker's strictness analysis in (Clark *et al.* 2000) is for a supercombinator language and w.r.t. a variant of a contextual semantics. A slight deficiency is that their contextual equality lacks "sufficiently many contexts", since it is defined for the available set of supercombinators and may change if a fresh supercombinator is introduced, since then more contexts are available. In a calculus with abstraction, in general, sufficiently many contexts can be formed, and contextual equivalence does not change by introducing fresh functions. Another deficiency of the contextual equivalence in (Clark *et al.* 2000) is that it is not applicable within definitions of supercombinators. Applying contextual equivalence within definitions of supercombinators means to apply it within abstractions, which in turn usually requires proving a context lemma, which is missing in (Clark *et al.* 2000), but provided in our paper. This means that they need further reasoning to show the correctness of optimization using strictness information.

As a summary, the combination of our `letrec`-case-constructor lambda calculus with contextual equivalence appears to be capable of achieving all the desired goals.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Unfortunately, the price is a higher complexity in writing down the rules of the calculus and to reason about the calculus.

1.2 Syntax and Reductions of the Functional Core Language LR

We define the calculus LR consisting of a language $\mathscr{L}(\text{LR})$ and its reduction rules, presented in this section, and the normal order reduction strategy and contextual equivalence, presented in section 1.3. If no confusion arises, we also speak of the language LR.

Our language, LR, the language of expressions (i.e., concrete terms), has the following syntax. There are finitely many constants, called constructors. The set of constructors is partitioned into (non-empty) types. For every type T, we denote the constructors as $c_{T,i}, i = 1, \ldots, |T|$. Every constructor has an arity $\text{ar}(c_{T,i}) \geqslant 0$.

The syntax for expressions E, case alternatives Alt and patterns Pat is as follows:

$$\begin{array}{rcl} E & ::= & V \mid (c\ E_1 \ldots E_{\text{ar}(c)}) \mid (\texttt{seq}\ E_1\ E_2) \mid (\texttt{case}_T\ E\ Alt_1 \ldots Alt_{|T|}) \mid (E_1\ E_2) \\ & & (\lambda\ V.E) \mid (\texttt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \texttt{in}\ E) \\ Alt & ::= & (Pat\ \rightarrow\ E) \\ Pat & ::= & (c\ V_1 \ldots V_{\text{ar}(c)}), \end{array}$$

where E, E_i are expressions, V, V_i are variables and where c denotes a constructor. Within each individual pattern, variables are not repeated. In a case-expression of the form $(\texttt{case}_T \ldots)$, for every constructor $c_{T,i}, i = 1, \ldots, |T|$ of type T, there is exactly one alternative with a pattern of the form $(c_{T,i}\ y_1 \ldots y_n)$, where $n = \text{ar}(c_{T,i})$. We assign the names *application*, *abstraction*, *constructor application*, seq-*expression*, case-*expression*, or letrec-*expression* to the expressions $(E_1\ E_2)$, $(\lambda V.E)$, $(c\ E_1 \ldots E_{\text{ar}(c)})$, $(\texttt{seq}\ E_1\ E_2)$, $(\texttt{case}_T\ E\ Alt_1 \ldots Alt_{|T|})$, $(\texttt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \texttt{in}\ E)$, respectively.

The constructs case, seq and the constructors $c_{T,i}$ can only occur in special syntactic constructions. Thus expressions where case, seq or a constructor is applied to the wrong number of arguments are not allowed.

The structure letrec obeys the following conditions. The variables V_i in the bindings are all distinct. We also assume that the bindings in letrec are commutative, i.e. letrecs with bindings interchanged are considered to be syntactically equivalent. letrec is recursive, i.e., the scope of x_j in $(\texttt{letrec}\ x_1 = E_1, \ldots, x_j = E_j, \ldots\ \texttt{in}\ E)$ is E and all expressions E_i. This fixes the notions of closed, open expressions and α-renamings. Free and bound variables in expressions are defined using the usual conventions. Variable binding primitives are λ, letrec, patterns, and the scope of variables bound in a letrec are all the expressions occurring in it. The set of free variables in an expression t is denoted as $FV(t)$. For simplicity, we use the distinct variable convention, i.e., all bound variables in expressions are assumed to be distinct, and free variables are distinct from bound variables. The reduction rules are assumed to implicitly rename bound variables in the result by α-renaming if necessary to obey this convention. Note that this is only necessary for the copy rule (cp) (see Figure 1). We follow the convention by omitting

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

parentheses in nested applications: $(s_1, \ldots, s_n)$ denotes $(\ldots(s_1\ s_2)\ldots s_n)$ provided s_1 is an expression. The set of closed LR expressions is denoted as LR^0.

To abbreviate the notation, we will sometimes use ($\mathtt{case}_T$ E *alts*) instead of ($\mathtt{case}_T$ E $alt_1 \ldots alt_{|T|}$). Sometimes we abbreviate the notation of `letrec`-expression (`letrec` $x_1 = E_1, \ldots, x_n = E_n$ `in` E), as (`letrec` Env `in` E), where Env $\equiv$ $\{x_1 = E_1, \ldots, x_n = E_n\}$. This will also be used freely for parts of the bindings. The notation $\{x_{g(i)} = s_{h(i)}\}_{i=m}^{n}$ is used for the chain $x_{g(m)} = s_{h(m)}, x_{g(m+1)} = s_{h(m+1)}, \ldots, x_{g(n)} = s_{h(n)}$ of bindings where $g, h : \mathbb{N} \to \mathbb{N}$, e.g., $\{x_i = s_{i-1}\}_{i=m}^{n}$ means the bindings $x_m = s_{m-1}, x_{m+1} = s_m, \ldots x_n = s_{n-1}$. We assume that `letrec`-expressions have at least one binding. The set $\{x_1, \ldots, x_n\}$ of variables that are bound by the `letrec`-environment $\mathrm{Env} = \{x_1 = s_1, \ldots, x_n = s_n\}$ is denoted as $LV(\mathrm{Env})$. In examples we will use : as an infix binary list-constructor, and `Nil` as the constant constructor for lists. We will write $(c_i\ \overrightarrow{z})$ as shorthand for the constructor application $(c_i\ z_1\ \ldots\ z_{\mathrm{ar}(c_i)})$.

In the following we define different context classes and contexts. To visually distinguish context classes from individual contexts, we use different text styles.

Definition 1.1
The class $\mathscr{C}$ of all *contexts* is defined as the set of expressions C from LR, where the symbol $[\cdot]$, the *hole*, is a predefined context, treated as an atomic expression, such that $[\cdot]$ occurs exactly once in C.

Given a term t and a context C, we will write $C[t]$ for the expression constructed from C by plugging t into the hole, i.e, by replacing $[\cdot]$ in C by t, where this replacement is meant syntactically, i.e., a variable capture is permitted.

Definition 1.2
A *value* is either an abstraction, or a constructor application. We denote values by the letters v, w.

The reduction rules in Definition 1.3, i.e. in Figures 1 and 2 are defined more liberally than necessary for the normal order reduction, in order to permit an easy use as transformations.

Definition 1.3 (*reduction rules of the calculus* LR)
The (base) reduction rules for the calculus and language LR are defined in Figures 1 and 2, where the labels S, V are to be ignored. The reduction rules can be applied in any context. The union of (llet-in) and (llet-e) is called (llet), the union of (case-c), (case-in), (case-e) is called (case), the union of (seq-c), (seq-in), (seq-e) is called (seq), the union of (cp-in) and (cp-e) is called (cp), and the union of (llet), (lcase), (lapp), (lseq) is called (lll).

Reductions (and transformations) are denoted using an arrow with super and/or subscripts: e.g. $\xrightarrow{llet}$. To explicitly state the context in which a particular reduction is executed we annotate the reduction arrow with the context in which the reduction takes place. If no confusion arises, we omit the context at the arrow.

The *redex* of a reduction is the term as given on the left-side of a reduction rule. We will also speak of the *inner redex*, which is the modified `case`-expression for

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

(lbeta) $C[((\lambda x.s)^S\ r) \to C[(\texttt{letrec}\ x = r\ \texttt{in}\ s)]$

(cp-in) $(\texttt{letrec}\ x_1 = v^S, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \ \texttt{in}\ C[x_m^V])$
$\to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}\ \ \texttt{in}\ C[v])$
where v is an abstraction

(cp-e) $(\texttt{letrec}\ x_1 = v^S, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}, y = C[x_m^V]\ \texttt{in}\ r)$
$\to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^m, \mathit{Env}, y = C[v]\ \texttt{in}\ r)$
where v is an abstraction

(llet-in) $(\texttt{letrec}\ \mathit{Env}_1\ \texttt{in}\ (\texttt{letrec}\ \mathit{Env}_2\ \texttt{in}\ r)^S)$
$\to (\texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ r)$

(llet-e) $(\texttt{letrec}\ \mathit{Env}_1, x = (\texttt{letrec}\ \mathit{Env}_2\ \texttt{in}\ s_x)^S\ \texttt{in}\ r)$
$\to (\texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2, x = s_x\ \texttt{in}\ r)$

(lapp) $C[((\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t)^S\ s)] \to C[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (t\ s))]$

(lcase) $C[(\texttt{case}_T\ (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t)^S\ \mathit{alts})] \to C[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{case}_T\ t\ \mathit{alts}))]$

(seq-c) $C[(\texttt{seq}\ v^S\ t)] \to C[t]$ if v is a value

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

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

(lseq) $C[(\texttt{seq}\ (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s)^S\ t)] \to C[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (\texttt{seq}\ s\ t))]$

Fig. 1. Reduction rules, part a.

(case)-reductions, the modified seq-expression for (seq)-reductions, and the variable position which is replaced by a (cp). Otherwise it is the same as the redex.

Transitive closure of reductions is denoted by a $+$, reflexive transitive closure by a $*$. E.g. $\xrightarrow{*}$ is the reflexive, transitive closure of $\to$. If necessary, we attach more information to the arrow.

Note that the reduction rules generate only syntactically correct expressions, since reductions, transformations and contexts are appropriately defined.

Remark 1.4

The case-rule looks a bit more complex than necessary, but in subsection 1.4 we will argue that the contextual equivalence proof depends on this form. The rules for case are essentially two rules, one for $(\texttt{case}_T\ (c \ldots) \ldots)$ and one for $(\texttt{case}_T\ x\ \ldots)$. In Figure 2 all six variants are explicitly given. The variants for a constant constructor and a constructor of nonzero arity, and also the variants depending on whether the case-expression is in the environment of the letrec or not. The current definition of (case) also appears in FUNDIO (Schmidt-Schauß 2003).

1.3 Normal Order Reduction and Contextual Equivalence

We define and explain the final components of the calculus LR.

(case-c)	$C[(\mathtt{case}_T\ (c_i\ \overrightarrow{t})^S\ \ldots((c_i\ \overrightarrow{y}) \to t)\ldots)] \to C[(\mathtt{letrec}\ \{y_i = t_i\}_{i=1}^n\ \mathtt{in}\ t)]$
	where $n = \mathrm{ar}(c_i) \geq 1$
(case-c)	$C[(\mathtt{case}_T\ c_i^S\ \ldots\ (c_i \to t)\ldots)] \to C[t]$ if $\mathrm{ar}(c_i) = 0$
(case-in)	$\mathtt{letrec}\ x_1 = (c_i\ \overrightarrow{t})^S, \{x_i = x_{i-1}\}_{i=2}^m, Env$
	$\mathtt{in}\ C[\mathtt{case}_T\ x_m^V\ \ldots((c_i\ \overrightarrow{z})\ldots \to t)\ldots]$
	$\to \mathtt{letrec}\ x_1 = (c_i\ \overrightarrow{y}), \{y_i = t_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=2}^m, Env$
	$\mathtt{in}\ C[(\mathtt{letrec}\ \{z_i = y_i\}_{i=1}^n\ \mathtt{in}\ t)]$
	where $n = \mathrm{ar}(c_i) \geq 1$ and y_i are fresh variables
(case-in)	$\mathtt{letrec}\ x_1 = c_i^S, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[\mathtt{case}_T\ x_m^V\ \ldots\ (c_i \to t)\ldots]$
	$\to \mathtt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^m, Env\ \mathtt{in}\ C[t]$
	if $\mathrm{ar}(c_i) = 0$
(case-e)	$\mathtt{letrec}\ x_1 = (c_i\ \overrightarrow{t})^S, \{x_i = x_{i-1}\}_{i=2}^m,$
	$u = C[\mathtt{case}_T\ x_m^V\ \ldots((c_i\ \overrightarrow{z}) \to r_1)\ldots], Env$
	$\mathtt{in}\ r_2$
	$\to \mathtt{letrec}\ x_1 = (c_i\ \overrightarrow{y}), \{y_i = t_i\}_{i=1}^n, \{x_i = x_{i-1}\}_{i=2}^m,$
	$u = C[(\mathtt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \mathtt{in}\ r_1)], Env$
	$\mathtt{in}\ r_2$
	where $n = \mathrm{ar}(c_i) \geq 1$ and y_i are fresh variables
(case-e)	$\mathtt{letrec}\ x_1 = c_i^S, \{x_i = x_{i-1}\}_{i=2}^m, u = C[\mathtt{case}_T\ x_m^V\ \ldots\ (c_i \to r_1)\ldots], Env$
	$\mathtt{in}\ r_2$
	$\to \mathtt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^m \ldots, u = C[r_1], Env\ \mathtt{in}\ r_2$
	if $\mathrm{ar}(c_i) = 0$

Fig. 2. Reduction rules, part b.

Normal-Order Reduction

The normal-order reduction strategy of the calculus LR is a call-by-need strategy, which is a call-by-name strategy adapted to sharing. The following labeling algorithm will detect the position to which a reduction rule will be applied according to normal order. It uses the labels: S, T, V, W, where T means reduction of the top term, S means reduction of a subterm, and V, W mark already visited subexpressions, where W at a variable indicates that the variable must not be replaced by a (cp)-reduction. Note that the labeling algorithm does not look into S-labeled letrec-expressions. For a term s, the labeling algorithm starts with s^T, where no other subexpression in s is labeled and proceeds until no more labeling is possible or until a fail occurs.

The rules of the labeling algorithm are

$(\mathtt{letrec}\ \mathrm{Env}\ \mathtt{in}\ t)^T$	$\to$	$(\mathtt{letrec}\ \mathrm{Env}\ \mathtt{in}\ t^S)^V$
$(s\ t)^{S \vee T}$	$\to$	$(s^S\ t)^V$
$(\mathtt{seq}\ s\ t)^{S \vee T}$	$\to$	$(\mathtt{seq}\ s^S\ t)^V$
$(\mathtt{case}_T\ s\ alts)^{S \vee T}$	$\to$	$(\mathtt{case}_T\ s^S\ alts)^V$
$(\mathtt{letrec}\ x = s, \mathrm{Env}\ \mathtt{in}\ C[x^S])$	$\to$	$(\mathtt{letrec}\ x = s^S, \mathrm{Env}\ \mathtt{in}\ C[x^V])$
$(\mathtt{letrec}\ x = s, y = C[x^S], \mathrm{Env}\ \mathtt{in}\ t)$	$\to$	$(\mathtt{letrec}\ x = s^S, y = C[x^V], \mathrm{Env}\ \mathtt{in}\ t)$
		if $C[x] \neq x$
$(\mathtt{letrec}\ x = s, y = x^S, \mathrm{Env}\ \mathtt{in}\ t)$	$\to$	$(\mathtt{letrec}\ x = s^S, y = x^W, \mathrm{Env}\ \mathtt{in}\ t)$.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

The notation $S \vee T$ stands for S or T. If a rule tries to label a subexpression already labeled V or W, then a loop has been detected and the algorithm stops with fail. Otherwise, if the labeling algorithm terminates, since it is no longer possible to apply a rule, then we say the termination is successful, and a potential normal order redex is found, which can only be the direct superterm of the S-marked subexpression. It is possible that there is no normal order reduction: in this case either the evaluation is already finished, or it is a dynamically detected error (like a type-error), or it is prevented by the loop check above.

We define reduction contexts and weak reduction contexts:

Definition 1.5
A reduction context R is any context, such that its hole will be labeled with S or T by the labeling algorithm. A *weak-reduction context*, R^-, is a reduction context, where the hole is not within a `letrec`-expression.

In grammar notation for context classes, this could be written as

$$
\begin{array}{lll}
\mathcal{R}^- & ::= & [\cdot] \mid (\mathcal{R}^-\ E) \mid (\mathtt{case}_T\ \mathcal{R}^-\ alts) \mid (\mathtt{seq}\ \mathcal{R}^-\ E) \\
\mathcal{R} & ::= & \mathcal{R}^- \mid (\mathtt{letrec}\ \mathrm{Env}\ \mathtt{in}\ \mathcal{R}^-) \mid \\
 & & (\mathtt{letrec}\ x_1 = \mathcal{R}_1^-, \{x_i = \mathcal{R}_i^-[x_{i-1}]\}_{i=2}^{j}, \mathrm{Env}\ \mathtt{in}\ \mathcal{R}^-[x_j]) \\
 & & \quad \text{where } j \geqslant 1 \text{ and } \mathcal{R}^-, \mathcal{R}_i^-, i = 1, \dots, j \text{ are weak reduction contexts.}
\end{array}
$$

A *maximal reduction context* of an expression s is a reduction context R with $R[s'] = s$, such that the labeling algorithm applied to s will label the subexpression s' with S or T and will then stop with success.

For example, the maximal reduction context of $(\mathtt{letrec}\ x_2 = \lambda x.x, x_1 = x_2\ x_1\ \mathtt{in}\ x_1)$ is $(\mathtt{letrec}\ x_2 = [\cdot], x_1 = x_2\ x_1\ \mathtt{in}\ x_1)$, in contrast to the nonmaximal reduction context $(\mathtt{letrec}\ x_2 = \lambda x.x, x_1 = x_2\ x_1\ \mathtt{in}\ [\cdot])$.

Definition 1.6 (*normal-order reduction of* LR)
Let t be an expression. Then a single normal order reduction step $\xrightarrow{n}$ is defined by first applying the labeling algorithm to t, and if the labeling algorithm terminates successfully, then one of the rules in Figures 1 and 2 has to be applied, if possible, where the labels S, V must match the labels in the expression t.

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

The *inner normal-order redex* is the following V-labeled subterm in t: it is the modified `case`-expression for (case)-reductions, the modified `seq`-expression for (seq)-reductions, or the V-labeled variable position which is replaced by a (cp). Otherwise it is the same as the normal-order redex.

The normal-order reduction implies that `seq` behaves like a function strict (see Definition 2.2) in its first argument, and that the `case`-construct is strict in its first argument. That is, these rules can only be applied if the corresponding argument is a value or if the argument is a variable bound to a value.

We are interested in normal-order reduction sequences, i.e. $\xrightarrow{n,*}$-reductions, and mainly those that end with a generalized value, also called the weak-head normal form.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Definition 1.7
A *weak-head normal form (WHNF)* is one of the following cases:

- A value v.
- A term of the form ($\texttt{letrec}$ Env $\texttt{in}$ v), where v is a value.
- A term of the form ($\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}$, Env $\texttt{in}\ x_m$), where $v = (c\ \overrightarrow{t})$.

If the value v in the WHNF t is an abstraction, we call t a *functional WHNF (FWHNF)*, otherwise, if v is a constructor application, we call t a *constructor WHNF (CWHNF)*.

Lemma 1.8
For every term t, if t has a normal-order redex, then the normal-order redex, the inner normal-order redex and the normal-order reduction are unique.

Definition 1.9
A normal-order reduction sequence is called an *(normal-order) evaluation* if the last term is a WHNF. Otherwise, i.e. if the normal-order reduction sequence is nonterminating, or if the last term is not a WHNF, but has no normal-order reduction, then we say that it is a *failing* normal-order reduction sequence.

For a term t, we write $t\Downarrow$ iff there is an evaluation starting from t. We call this the *evaluation of* t and denote it as $nor(t)$. If $t\Downarrow$, we also say that t is *converging* (or *terminating*). Otherwise, if there is no evaluation of t, we write $t\Uparrow$. For a term t, we write $t\Uparrow\Uparrow$, if $t\Uparrow$ and if there is also no normal-order reduction sequence for t to a term of the form $R[x]$ where x is a free variable in $R[x]$, and R is a reduction context. A term t with $t\Uparrow\Uparrow$ is also called *bot-term*, and a specific representative is Ω, which can be defined as

$$\Omega := (\lambda z.(z\ z))\ (\lambda x.(x\ x)).$$

Note that there are useful open terms t that might not have an evaluation, e.g. x is such a term. Note also that there are (closed) terms t that are neither WHNFs nor have a normal order redex. For example, ($\texttt{case}_T$ $(\lambda x.x)$ *alts*) or $\texttt{((cons 1 2) 3)}$, where $\texttt{cons}$ is a constructor of arity 2. These terms are bot-terms and could be considered as violating-type conditions. Consider the closed "cyclic term" ($\texttt{letrec}\ x = x\ \texttt{in}\ x$). A reduction context for this term is ($\texttt{letrec}\ x = [\cdot]\ \texttt{in}\ x$). Obviously, there is no normal-order reduction defined for this term, hence also no evaluation of t.

As an example, we show the first normal-order reduction steps of an evaluation of $\Omega = (\lambda z.(z\ z))\,(\lambda x.(x\ x))$: $(\lambda z.(z\ z))\,(\lambda x.(x\ x)) \xrightarrow{n,lbeta} (\texttt{letrec}\ z = \lambda x.(x\ x)\ \texttt{in}\ (z\ z)) \xrightarrow{n,cp} (\texttt{letrec}\ z = \lambda x.(x\ x)\ \texttt{in}\ ((\lambda x'.(x'\ x'))\ z)) \xrightarrow{n,lbeta} (\texttt{letrec}\ z = \lambda x.(x\ x)\ \texttt{in}\ (\texttt{letrec}\ x_1 = z\ \texttt{in}\ (x_1\ x_1))) \xrightarrow{n,llet} (\texttt{letrec}\ z = \lambda x.(x\ x), x_1 = z\ \texttt{in}\ (x_1\ x_1)) \longrightarrow \cdots$.

Contextual Equivalence

The semantic foundation of our calculus LR is the equality of expressions defined by contextual equivalence. We define contextual equivalence w.r.t. evaluations.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Definition 1.10 (*contextual preorder and equivalence*)
Let s, t be terms. Then:

$$\begin{array}{lll} s \leqslant_c t & \text{iff} & \forall C[\cdot] : \ C[s]\Downarrow \Rightarrow C[t]\Downarrow \\ s \sim_c t & \text{iff} & s \leqslant_c t \wedge t \leqslant_c s. \end{array}$$

Note that we permit contexts $C[]$ such that $C[s]$ may be an open term. In appendix G we show that $s \leqslant_c t$ is equivalent to

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

By standard arguments, we see that $\leqslant_c$ is a precongruence and that $\sim_c$ is a congruence, where a *precongruence* $\leqslant$ is a preorder on expressions, such that $s \leqslant t \Rightarrow C[s] \leqslant C[t]$ for all contexts C, and a *congruence* is a precongruence that is also an equivalence relation.

1.4 Discussion

We will now discuss some of our design choices, i.e. why we chose to add an operator `seq` and weakly typed `case`-expressions to LR. We will also discuss, why we chose to use reduction rules that follow `letrec`-bindings, although this choice seems to be more complicated than obviously necessary.

Extensions: `seq`, *Constructors and* `case`

The language LR has a `seq`-primitive as an extension to `letrec`-calculi. In Clean, a function `seq` is definable, whereas in Haskell the `seq` is predefined. The `seq`-primitive is not redundant in the core language if it has our weakly typed `case`, as the following remark shows.

Remark 1.11
The operator `seq` cannot be simulated in the language LR_{noseq}, which denotes LR without the `seq`-construct and without the corresponding reductions. In the language LR_{noseq} it is not possible to simulate `seq`, i.e. there is no expression $t = \lambda x, y.t'$ that is equivalent to `seq`. Applying the expression t to the argument pairs ($\perp$, `True`), (`True`, `True`), and (`False`, `False`) shows that t must evaluate the first argument. Since normal order is unique, t must scrutinize x using a (`if` x `then`...). Now the argument pair (`Nil`, `Nil`) makes a difference, since t `Nil Nil` yields $\perp$, whereas `seq Nil Nil` results in `Nil`.

The following example shows that the addition of `seq` is not conservative for $\sim_c$ w.r.t. LR_{noseq} and LR, i.e., more expressions may be distinguished after the extension.

$$\begin{array}{lll} s & = & \lambda f.\texttt{if}\ (f\ \lambda x.\Omega)\ \texttt{then True else}\ \Omega \\ t & = & \lambda f.\texttt{if}\ (f\ \Omega)\ \texttt{then True else}\ \Omega. \end{array}$$

In LR_{noseq}, the expressions s, t cannot be distinguished, since in case f is a constant function, they behave equivalent. Otherwise, f evaluates its argument and the first usage must be an application to another subterm, which for both functions yields Ω as a result.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

If a seq is available, then we could use $f := \lambda x.(\texttt{seq}\ x\ \texttt{True})$, which distinguishes the two functions s, t.

We also added a (weak) typing to the case, which brings our language even closer to the typed languages Clean and Haskell.

Example 1.12
This example shows that the language LR is closer to a polymorphically typed language like Haskell and Clean than a functional core language without types. So assume, for this example, that there is a core language LR_{ut} with letrecs, constructors, and abstractions, where the weakly typed case is replaced by the following (untyped) case-construct: there are alternatives for every constructor, and also either a default alternative, or an alternative for abstractions. Note that seq can be defined in LR_{ut}.

Now define the following functions:

$$\begin{aligned} s_0 &= \lambda f.\texttt{if}\ (f\ \texttt{True})\ \texttt{then}\ (\texttt{if}\ (f\ \texttt{Nil})\ \texttt{then}\ \Omega\ \texttt{else True})\ \texttt{else}\ \Omega \\ t_0 &= \lambda f.\texttt{if}\ (f\ \texttt{Nil})\ \texttt{then}\ (\texttt{if}\ (f\ \texttt{True})\ \texttt{then}\ \Omega\ \texttt{else False})\ \texttt{else}\ \Omega. \end{aligned}$$

We claim that s_0, t_0 cannot be distinguished in LR, since $(f\ \texttt{True})$ and $(f\ \texttt{Nil})$ cannot result in different (terminating) Boolean values. If a function f outputs different values for the inputs True and Nil, then there must be an evaluation of a case-expression scrutinizing the inputs. But then one of the results must be $\perp$, since $\texttt{case}_T$ is typed. This reasoning can be extended to show that $s_0 \sim_c t_0$. However, the expressions s_0, t_0 can be distinguished in LR_{ut}, since it is easy to define a function f as follows. The top level is a case having alternatives for all constructors, for True it yields True, and for Nil it yields False. Applying s_0, t_0 to the function f gives different results.

This means that the reason for the difference between the languages LR and LR_{ut} is only the restricted typing. The reason is that typing restricts the number of contexts in LR in contrast to LR_{ut}.

The typed case-primitive and the seq-primitive together have the same expressive power as an unrestricted case, however, contextual equality is closer to that in a typed lazy functional language.

The reduction rules of our calculus are similar to the rules in related call-by-need calculi. In (Ariola *et al.* 1995), rules for a let-calculus, for a calculus with constructors, and also for a calculus with a letrec are given, but rules for the combination of letrec and constructors are missing. In (Ariola & Arvind 1995) there are reduction rules for a calculus using letrec and supercombinators. The paper (Moran *et al.* 1999) describes a similar calculus, extended with a nondeterministic choice construct, but a slightly restricted language, where, e.g. only applications of the form $(t\ x)$ are permitted. They employ a contextual equality, which is adapted to the nondeterminism in their language. Contrary to Moran, Sands and Carlsson (Moran *et al.* 1999) in applications $(s\ t)$ of LR, we permit arguments t other than variables. The results in our paper show this restriction on the term structure to be irrelevant for equivalence and the main length measure. We cannot use the call-by-name variants of letrec-calculi, as e.g. (Ariola & Blom 2002), since

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

using these calculi, the reduction-length property would not hold, and since in the abstract language $\mathrm{LR}_{\mathscr{U}}$, the set constants have to be treated like nondeterministic choiceexpressions, i.e. should not be copied.

Reduction Rules

Our choice to use contextual equivalence comes with the obligation to show that all reduction rules retain contextual equivalence. Once the context lemma is proved, this becomes easier since only weaker contexts are to be considered. Our experience shows that correctness proofs for all transformations corresponding to reduction rules can be done using reduction diagrams. However, the correctness proofs of the transformations corresponding to the case reduction rules are a major hurdle; in fact, we did not find such a proof, if the case reductions are defined in the naive way, i.e. using the (abs)- and copy rule in combination with a case rule: $(\texttt{letrec}\ x = c\ \overrightarrow{t}, \mathrm{Env}\ \ \texttt{in case}\ x\ \ldots) \rightarrow (\texttt{letrec}\ x = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, \mathrm{Env}\ \ \texttt{in case}\ x\ \ldots) \rightarrow (\texttt{letrec}\ x = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, \mathrm{Env}\ \ \texttt{in case}\ (c\ \overrightarrow{y})\ \ldots)$.

If the case rule is defined as in our calculus (see Figure 2), i.e., constructor-expressions also can be seen from a case expression via chains of variable–variable bindings, then the diagram-chasing proofs are successful. We think that it would be possible to avoid the variable bindings $x = y$ in the rules, which would make the presentation of the rules a bit simpler. However, a price would have to be paid for this modification: the normal-order reduction had to include a rule for copying variables, which complicates the development of the correctness proofs of the reduction rules used as transformations, and also complicates the arguments on the lengths of evaluations. Based on these proofs of contextual equivalence for transformations, we think that it is possible to prove that simpler reduction rules would work also, e.g. for constructing an abstract machine for reduction, but this would not simplify, but increase the complexity of the treatment of the calculus in our paper.

2 Contextual Equivalence of LR

This part provides methods and tools used in the correctness proof. In subsection 2.1 we use the notion of contextual equivalence to define strictness of functions. In subsection 2.2 we develop the proof methods using a context lemma and overlap diagrams in order to show that $s \sim_c t$ for all reduction rules $s \rightarrow t$. A set of extra transformation rules used in SAL is defined and investigated in subsection 2.3. It is also necessary to analyze different kinds of length measures of a normal-order reduction, and to check every reduction and transformation whether the length remains equal or is decreased (or perhaps increased) after applying the transformation. We accomplish this analysis in subsection 2.4.

2.1 Surface Contexts and Strict Functions

To introduce the notion of strict subexpressions and in particular the extra transformation rules below, two kinds of surface contexts are introduced.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Definition 2.1 (*surface context classes*)
A *surface context* is a context where the hole is not contained in an abstraction. The class of surface contexts is denoted as $\mathcal{S}$.
An *application surface context* is a surface context where the hole is neither contained in an abstraction nor in an alternative of a case expression. A *weak-application surface context* is an application surface context where the hole is in addition not contained in a `letrec`-expression. The class of application surface contexts is denoted as $\mathcal{W}$, and the class of weak-application surface contexts as $\mathcal{W}^-$

For a context C, its *main depth* is defined as the depth of its hole. With $C_{(i)}$, we denote a context of main depth i. So $\mathcal{W}^-_{(1)}$ is the context class of *weak-application surface contexts of main depth 1*:

$$\begin{array}{rcl}\mathcal{W}^-_{(1)} & ::= & ([\cdot]\ E) \mid (E\ [\cdot]) \mid (c\ E_1 \dots E_{i-1}\ [\cdot]\ E_{i+1} \dots E_{\mathrm{ar}(c)}) \\ & & \mid (\mathtt{case}_T\ [\cdot]\ alts) \mid (\mathtt{seq}\ [\cdot]\ E) \mid (\mathtt{seq}\ E\ [\cdot]).\end{array}$$

Note that every reduction context is also a surface context and an application surface context, and that a weak-reduction context is also a weak-application surface context.

We define strictness of functions and expressions consistent with the notions from denotational semantics.

Definition 2.2
An expression s is *strict*, iff $(s\ \Omega) \sim_c \Omega$.

An expression s is *strict in the* ith *argument for arity* n, iff $1 \leqslant i \leqslant n$ and for all closed expressions $t_1, \dots, t_{i-1}, t_{i+1}, \dots t_n$: $(s\ t_1 \dots t_{i-1}\ \Omega\ t_{i+1} \dots t_n) \sim_c \Omega$.

Let s_0 be a subexpression of s not contained in another subexpression that is an abstraction; i.e. $s = S[s_0]$, where S is a surface context (see Definition 2.1). Then the expression s is *strict in the subexpression* s_0, iff for the term s' that is constructed from s by replacing s_0 by Ω, we have $s' \sim_c \Omega$. We also say s_0 *is a strict subexpression of* t. Here, we mean by subexpression also the position within the superterm.

Knowing strictness of functions and strict subexpressions of terms helps to rearrange evaluation and is thus of importance for optimizations and parallelization of nonstrict functional programs.

2.2 *Context Lemma and Properties of* LR*-Reductions*

Context Lemma

The context lemma restricts the criterion for contextual equivalence to reduction contexts. This restriction is of great value in proving the conservation of contextual equivalence by certain reductions, since there is no need to introduce parallel reductions like Barendregt's 1-reduction (Barendregt 1984). Its proof can be found in appendix A.

Lemma 2.3 (*context lemma*)
Let s, t be terms. If for all reduction contexts R: $(R[s]\Downarrow \Rightarrow R[t]\Downarrow)$, then $\forall C : (C[s]\Downarrow \Rightarrow C[t]\Downarrow)$; i.e. $s \leqslant_c t$.

In appendix G, it is argued that closing reduction contexts are sufficient in the context lemma.

Correctness of Transformations

We say that a transformation $\leadsto$ on terms is *correct*, if $s \leadsto t$ implies $s \sim_c t$ for all terms s, t. In the following, we will use the base reductions also as transformations (ignoring the labels S, V). In appendix B, we prove the following theorem on correctness of transformations.

Theorem 2.4
All the reductions (viewed as transformations) in the base calculus LR maintain contextual equivalence, i.e., whenever $t \xrightarrow{a} t'$, with $a \in \{\text{cp, lll, case, seq, lbeta}\}$, then $t \sim_c t'$.

Note that the correctness proof for $a \in \{\text{cp, lll, seq, lbeta}\}$ is straightforward using the context lemma and the diagrams, however, the correctness proof for (case) requires further tools, in particular, the extra transformations defined below in section 2.3

On (lll)-Transformations

The following lemma shows that `letrec`-environments in reduction contexts can immediately be moved to the top level environment.

Lemma 2.5
Let $t = (\texttt{letrec Env in } t')$ be an expression, and R be a reduction context. Then

1. If $R = (\texttt{letrec Env}_R \texttt{ in } R')$, where R' is a weak-reduction context, then $R[(\texttt{letrec Env in } t')] \xrightarrow{n,lll,+} (\texttt{letrec Env}_R, \text{Env} \texttt{ in } R'[t'])$.
2. If $R = (\texttt{letrec Env}_R, x = R' \texttt{ in } r)$, where R' is a weak-reduction context, then $R[(\texttt{letrec Env in } t')] \xrightarrow{n,lll,+} (\texttt{letrec Env}_R, \text{Env}, x = R'[t'] \texttt{ in } r)$, and $(\texttt{letrec Env}_R, \text{Env}, x = R'[\cdot] \texttt{ in } r)$ is a reduction context.
3. If R' is not a `letrec`-expression, i.e. R' is a weak-reduction context, then $R'[(\texttt{letrec Env in } t')] \xrightarrow{n,lll,*} (\texttt{letrec Env in } R'[t'])$, and $(\texttt{letrec Env in } R'[\cdot])$ is a reduction context.

Proof
This follows by induction on the main depth of the context R'. □

Definition 2.6
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 `lrdepth`(C) for all `letrec`-subexpressions s with $t \equiv C[s]$, where `lrdepth` is defined as follows:

$$\begin{array}{lcl} \texttt{lrdepth}([\cdot]) & = & 0 \\ \texttt{lrdepth}(C_{(1)}[C'[]]) & = & \begin{cases} 1 + \texttt{lrdepth}(C'[]) & \text{if } C_{(1)} \text{ is not a } \texttt{letrec} \\ \texttt{lrdepth}(C'[]) & \text{if } C_{(1)} \text{ is a } \texttt{letrec}. \end{cases} \end{array}$$

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

The following termination property of (lll) is required in later proofs.

Proposition 2.7
The transformation (lll) is terminating, i.e. there are no infinite transformation sequences consisting only of (lll) transformations.

Proof
This holds, since $t_1 \xrightarrow{lll} t_2$ implies $\mu_{lll}(t_1) > \mu_{lll}(t_2)$, and the ordering induced by the measure is well-founded. □

2.3 Extra Transformation Rules

We define further transformation rules that are useful as simplifications and are also a necessary tool for a correctness proof of the base reduction (case).

Definition 2.8
The extra transformation rules are defined in Figure 3. The union of (gc1) and (gc2) is called (gc), the union of (cpx-in) and (cpx-e) is called (cpx), the union of (cpcx-in) and (cpcx-e) is called (cpcx).
The transformation *(case-cx)* is like (case) with the difference, that if the involved constructor application is of the form $(c\ x_1 \ldots x_n)$, where x_i are variables, then the rule *(case-cx)* does not modify $(c\ x_1 \ldots x_n)$. The extra transformation rule (cpcxnoa) can be seen as an abbreviation of a (cpcx) with subsequent (cpx) and (gc)-transformations.

Note that the (useless) transformation $\texttt{letrec}\ x = x\ \texttt{in}\ t \rightarrow \texttt{letrec}\ x = x\ \texttt{in}\ t$ is not allowed as an instance of the (cpx)-rule. Note also that the transformation (lwas) includes the reductions (lapp), (lcase), (lseq).

Correctness of Extra Transformations

In appendix B we prove the following theorems in a series of lemmas.

Theorem 2.9
The transformations (ucp), (cpx), (cpax), (gc), (lwas), (cpcx), (abs), (abse), (xch), (cpcxnoa) and (case-cx) maintain contextual equivalence, i.e. whenever $t \xrightarrow{a} t'$, with $a \in \{\text{ucp}, \text{cpx}, \text{cpax}, \text{gc}, \text{lwas}, \text{cpcx}, \text{abs}, \text{abse}, \text{xch}, \text{cpcxnoa}, \text{case-cx}\}$, then $t \sim_c t'$.

Theorem 2.10 (*standardization*)
Let t be a term such that $t \xrightarrow{*} t'$, where t' is a WHNF, and the reduction steps are base reductions or extra transformations. Then $t\Downarrow$.

Proof
This follows from Theorems 2.4 and 2.9. □

A Convergent Rewrite System of Simplifications

Definition 2.11 (*simplifications*)
As *simplification rules* we will use (lwas), (llet), (gc), (cpax).

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

(gc1)	$(\mathtt{letrec}\ \{x_i = s_i\}_{i=1}^{n}, \mathit{Env}\ \mathtt{in}\ t) \to (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ t)$ if for all $i : x_i$ does not occur in Env nor in t
(gc2)	$(\mathtt{letrec}\ \{x_i = s_i\}_{i=1}^{n}\ \mathtt{in}\ t) \to t$ if for all $i : x_i$ does not occur in t
(cpx-in)	$(\mathtt{letrec}\ x = y, \mathit{Env}\ \mathtt{in}\ C[x])$ $\to (\mathtt{letrec}\ x = y, \mathit{Env}\ \mathtt{in}\ C[y])$ where y is a variable and $x \neq y$
(cpx-e)	$(\mathtt{letrec}\ x = y, z = C[x], \mathit{Env}\ \mathtt{in}\ t)$ $\to (\mathtt{letrec}\ x = y, z = C[y], \mathit{Env}\ \mathtt{in}\ t)$ where y is a variable and $x \neq y$
(cpax)	$(\mathtt{letrec}\ x = y, \mathit{Env}\ \mathtt{in}\ s)$ $\to (\mathtt{letrec}\ x = y, \mathit{Env}[y/x]\ \mathtt{in}\ s[y/x])$ where y is a variable, $x \neq y$ and $y \in FV(s, \mathit{Env})$
(cpcx-in)	$(\mathtt{letrec}\ x = c\ \overrightarrow{t}, \mathit{Env}\ \mathtt{in}\ C[x])$ $\to (\mathtt{letrec}\ x = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, \mathit{Env}\ \mathtt{in}\ C[c\ \overrightarrow{y}])$
(cpcx-e)	$(\mathtt{letrec}\ x = c\ \overrightarrow{t}, z = C[x], \mathit{Env}\ \mathtt{in}\ t)$ $\to (\mathtt{letrec}\ x = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, z = C[c\ \overrightarrow{y}], \mathit{Env}\ \mathtt{in}\ t)$
(abs)	$(\mathtt{letrec}\ x = c\ \overrightarrow{t}, \mathit{Env}\ \ \mathtt{in}\ s) \to (\mathtt{letrec}\ x = c\ \overrightarrow{x}, \{x_i = t_i\}_{i=1}^{\mathrm{ar}(c)}, \mathit{Env}\ \ \mathtt{in}\ s)$ where $\mathrm{ar}(c) \geq 1$
(abse)	$(c\ \overrightarrow{t}) \to (\mathtt{letrec}\ \{x_i = t_i\}_{i=1}^{\mathrm{ar}(c)}\ \mathtt{in}\ c\ \overrightarrow{x})$ where $\mathrm{ar}(c) \geq 1$
(xch)	$(\mathtt{letrec}\ x = t, y = x, \mathit{Env}\ \mathtt{in}\ r) \ \to\ (\mathtt{letrec}\ y = t, x = y, \mathit{Env}\ \mathtt{in}\ r)$
(ucp1)	$(\mathtt{letrec}\ \mathit{Env}, x = t\ \mathtt{in}\ S[x]) \to (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ S[t])$
(ucp2)	$(\mathtt{letrec}\ \mathit{Env}, x = t, y = S[x]\ \mathtt{in}\ r) \to (\mathtt{letrec}\ \mathit{Env}, y = S[t]\ \mathtt{in}\ r)$
(ucp3)	$(\mathtt{letrec}\ x = t\ \mathtt{in}\ S[x]) \to S[t]$ where in the (ucp)-rules, x has at most one occurrence in $S[x]$ and no occurrence in Env, t, r; and S is a surface context
(lwas)	$W^{-}_{(1)}[(\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ s)] \to (\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ W^{-}_{(1)}[s])$ where $W^{-}_{(1)}$ is a weak application surface context of main depth 1 (see Definition 2.1)
(cpcxnoa)	$(\mathtt{letrec}\ x = c\ x_1 \ldots x_m, \mathit{Env}\ \mathtt{in}\ C[x])$ $\to (\mathtt{letrec}\ x = c\ x_1 \ldots x_m, \mathit{Env}\ \mathtt{in}\ C[c\ x_1 \ldots x_m])$
(case-cx)	$(\mathtt{letrec}\ x = (c_{T,j}\ x_1 \ldots x_n), \mathit{Env}\ \mathtt{in}\ C[\mathtt{case}_T\ x\ ((c_{T,j}\ y_1 \ldots y_n) \to s)\ \mathit{alts}])$ $\to \mathtt{letrec}\ x = (c_{T,j}\ x_1 \ldots x_n), \mathit{Env}$ $\quad \mathtt{in}\ C[(\mathtt{letrec}\ y_1 = x_1, \ldots, y_n = x_n\ \mathtt{in}\ s)]$
(case-cx)	$\mathtt{letrec}\ x = (c_{T,j}\ x_1 \ldots x_n), \mathit{Env},$ $\quad y = C[\mathtt{case}_T\ x\ ((c_{T,j}\ y_1 \ldots y_n) \to s)\ \mathit{alts}]\ \mathtt{in}\ r$ $\to\ \mathtt{letrec}\ x = (c\ x_1 \ldots x_n), \mathit{Env},$ $\quad y = C[(\mathtt{letrec}\ y_1 = x_1, \ldots, y_n = x_n\ \mathtt{in}\ s)]\ \mathtt{in}\ r$
(case-cx)	like (case) in all other cases

Fig. 3. Extra transformation rules.

Note that the rule (lwas) includes (lseq), (lcase), (lapp), but not (llet). The simplification rules (lwas), (llet), (gc), (cpax) maintain contextual equivalence, which is proved in Theorems 2.4 and 2.9. For definitions of confluence and local confluence, see e.g. (Baader & Nipkow 1998). The reason for using (cpax) in the simplification rules is that it terminates, i.e. there are no infinite sequences consisting only of $\xrightarrow{cpax}$, in contrast to (cpx) (see subsection B.4).

In appendix F, we prove the following result,

Theorem 2.12
The set of transformations (lwas), (llet), (gc), (cpax) is confluent (up to α-renaming) and terminating. This defines a unique simplified normal form of every LR term, which can be computed by exhaustively applying simplification rules. The normal form has the following properties.

- There are no unnecessary bindings.
- The `letrec`-environments are joined at the top of the term, at the top in the body of abstractions, and at the top in the alternatives of cases.
- A normal order reduction of a normal form may only be a (case), (lbeta), (seq), or a (cp)-reduction.

2.4 Length of Normal Order Reduction Sequences

We develop properties of different lengths measures of normal-order reduction sequences, which enable the proof of correctness of SAL (see section 3.4). The measure $\mathrm{rl}\sharp\sharp(t)$ can be viewed as a distance of t from its WHNF. This measure is strictly decreased after normal-order (case), (lbeta), and (seq) reductions. It is robust w.r.t. all extra transformations and cannot be increased by applying base reductions as transformations.

Definition 2.13
Let t be a closed term with $t\Downarrow$,
then

1. if $\emptyset \neq M \subseteq \{\text{case,lbeta,seq,cp,lll}\}$, then $\mathrm{rl}_M(t)$ is defined to be the number of normal-order reductions $\xrightarrow{a}$ in $nor(t)$ with $a \in M$;
2. $\mathrm{rl}\sharp\sharp(t) := \mathrm{rl}_{\{\text{case,lbeta,seq}\}}(t)$;
3. $\mathrm{rl}\sharp(t) := \mathrm{rl}_{\{\text{case,lbeta,seq,cp}\}}(t)$;
4. $\mathrm{rl}\flat(t) := \mathrm{rl}_{\{lll\}}(t)$;
5. $\mathrm{rl}(t) := \mathrm{rl}_{\{\text{case,lbeta,seq,cp,lll}\}}(t)$.

The main measure in this paper will be $\mathrm{rl}\sharp\sharp(\cdot)$.
In the following, the specializations of (seq), (case), (cp) where the C-context mentioned in the corresponding rule definition in Figures 1 and 2 is restricted to a surface context are denoted as (seqS), (caseS), (cpS).

Theorem 2.14
Let t_1, s_1 be closed LR expressions with $t_1\Downarrow$ and $t_1 \longrightarrow s_1$ by a base reduction or an extra transformation. Then $s_1\Downarrow$ and the following holds.

1. If $t_1 \xrightarrow{a} s_1$ with $a \in \{\text{case, seq, lbeta, cp}\}$, then $\mathrm{rl}(t_1) \geqslant \mathrm{rl}(s_1)$, $\mathrm{rl}\sharp(t_1) \geqslant \mathrm{rl}\sharp(s_1)$ and $\mathrm{rl}\sharp\sharp(t_1) \geqslant \mathrm{rl}\sharp\sharp(s_1)$.
2. If $t_1 \xrightarrow{\mathscr{S},a} s_1$ with $a \in \{\text{caseS, seqS, lbeta, cpS}\}$, then $\mathrm{rl}\sharp(t_1) \geqslant \mathrm{rl}\sharp(s_1) \geqslant \mathrm{rl}\sharp(t_1) - 1$ and $\mathrm{rl}\sharp\sharp(t_1) \geqslant \mathrm{rl}\sharp\sharp(s_1) \geqslant \mathrm{rl}\sharp\sharp(t_1) - 1$. For $a = \text{cpS}$, the equation $\mathrm{rl}\sharp\sharp(t_1) = \mathrm{rl}\sharp\sharp(s_1)$ holds.
3. If $t_1 \xrightarrow{a} s_1$ with $a \in \{\text{lll, gc}\}$, then $\mathrm{rl}(t_1) \geqslant \mathrm{rl}(s_1)$, $\mathrm{rl}\sharp(t_1) = \mathrm{rl}\sharp(s_1)$ and $\mathrm{rl}\sharp\sharp(t_1) = \mathrm{rl}\sharp\sharp(s_1)$. For $a = gc1$ in addition $\mathrm{rl}(t_1) = \mathrm{rl}(s_1)$ holds.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

4. If $t_1 \xrightarrow{a} s_1$ with $a \in \{\text{cpx}, \text{cpax}, \text{xch}, \text{cpcx}, \text{abs}\}$, then $\text{rl}(t_1) = \text{rl}(s_1)$, $\text{rl}\sharp(t_1) = \text{rl}\sharp(s_1)$ and $\text{rl}\sharp\sharp(t_1) = \text{rl}\sharp\sharp(s_1)$.
5. If $t_1 \xrightarrow{ucp} s_1$, then $\text{rl}(t_1) \geqslant \text{rl}(s_1)$, $\text{rl}\sharp(t_1) \geqslant \text{rl}\sharp(s_1)$ and $\text{rl}\sharp\sharp(t_1) = \text{rl}\sharp\sharp(s_1)$.
6. If $t_1 \xrightarrow{lwas} s_1$, then $\text{rl}\sharp\sharp(t_1) = \text{rl}\sharp\sharp(s_1)$.

Proof

The proofs are in appendix E. □

The next proposition shows that a single reduction step exploiting strictness will not increase the number of (case)-, (cp)-, (seq)-, and (lbeta)-reductions required to reach a WHNF. Its proof is done in the appendix, in subsection E.9.

Proposition 2.15

Let t_1, s_1 be closed concrete LR expressions with $t_1\Downarrow$ and $t_1 \xrightarrow{\mathscr{S},b} s_1$, where $b \in \{(\text{caseS}), (\text{seqS}), (\text{lbeta}), (\text{cpS})\}$, such that the inner redex t_0 of the reduction is a strict subterm of t_1, and $t_1 = S[t_0]$ for a surface context S.

Then $\text{rl}\sharp(t_1) = 1+\text{rl}\sharp(s_1)$. If $b \in \{(\text{caseS}), (\text{seqS}), (\text{lbeta})\}$, then $\text{rl}\sharp\sharp(t_1) = 1+\text{rl}\sharp\sharp(s_1)$ and if $b = (cp\mathscr{S})$, then $\text{rl}\sharp\sharp(t_1) = \text{rl}\sharp\sharp(s_1)$.

Local Evaluation and Deep Subterms

We introduce deep strict subterms, which are strict subterms that are evaluated by a normal-order reduction only if also another `seq`-expression, `case`-expression or application is reduced first, which is "above" the deep subterm. As a tool a relativized normal-order reduction, called local evaluation, is defined. Using this tool, we will show that deep subterms have a (local) evaluation length strictly smaller than that of the top term. This will enable proving correctness of SAL's subsume2-rule (see Definition 3.27 and Example 3.42).

Definition 2.16

Let $t = (\texttt{letrec Env in } t')$ be a (concrete) LR expression, and let $x \in \text{LV}(\text{Env})$. Then the *local evaluation* of x is defined as the reduction sequence of t, which corresponds to the evaluation of $(\texttt{letrec Env in } x)$, only considering reductions that make modifications in Env, i.e. a possibly occurring last (n,cp) that replaces x by an abstraction in the evaluation is omitted in the local evaluation.

If $(\texttt{letrec Env in } x)\Downarrow$, then the length corresponding to $\text{rl}\sharp(\cdot)$ of a local evaluation is denoted as $\text{rl}\sharp_{loc}(\texttt{letrec Env in } x)$.

Proposition 2.17

Let $t_1 = (\texttt{letrec Env in } t_1')$ be a closed (concrete) LR expression with $t_1\Downarrow$. Let $x \in \text{LV}(\text{Env})$ where the binding in Env is $x = t_x$, and t_x is a strict subexpression in t_1. Then $\text{rl}\sharp(t_1) \geqslant \text{rl}\sharp_{loc}(\texttt{letrec Env in } x)$ and $\text{rl}\sharp\sharp(t_1) \geqslant \text{rl}\sharp\sharp(\texttt{letrec Env in } x)$.

Proof

The proof is in appendix E.10. □

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Definition 2.18
Let $t = (\texttt{letrec}\ \mathrm{Env}, x = t_x, x_1 = t_1\ \texttt{in}\ x_1)$. The subterm t_x is called a *deep subterm* in t if t_1 is either an application, a seq-expression, or a case-expression.

The next proposition shows that for deep and strict subexpressions, the number $\mathrm{rl}\sharp\sharp(\cdot)$ of local evaluations is strictly less than the corresponding number for the top term. The proof is in appendix E.10.

Proposition 2.19
Let $t_1 = (\texttt{letrec}\ \mathrm{Env}, x = t_x, x_1 = t_1'\ \texttt{in}\ x_1)$ be a closed concrete LR-expression with $t_1\Downarrow$, such that t_x is a strict and deep subterm in t_1.

Then $\mathrm{rl}\sharp\sharp(t_1) > \mathrm{rl}\sharp\sharp(\texttt{letrec}\ \mathrm{Env}, x = t_x\ \texttt{in}\ x)$.

Concrete Subterms and Environments

The goal of this subsection is to show that open subterms within certain surface contexts can be closed by localizing the global environment, thereby copying it. The main argument, proved in appendix H, is that concrete subterms can be copied to positions within surface contexts, and that concrete parts of letrec-environments can be duplicated together with a renaming without consequences for the $\sim_c$-equivalence of the top-expression. We conjecture that copying concrete subterms can be done without restrictions, but did not find a proof. Note that copying subterms may change the length of the normal-order reduction and the corresponding measures.

In appendix H, we prove the following proposition.[2]

Proposition 2.20
Let $t = (\texttt{letrec}\ \mathrm{Env}, x = W[t']\ \texttt{in}\ r)$ be a closed expression, where W is a weak-application surface context. Then there exists a closed expression t'', such that $t \sim_c (\texttt{letrec}\ \mathrm{Env}, x = W[t'']\ \texttt{in}\ r)$.

The term t'' can be constructed as follows. Let $\mathrm{Env} = \{y_i = s_i\}_{i=1}^n$, and $t'' := (\texttt{letrec}\ \mathrm{Env}', x' = W'[(t''')]\ \texttt{in}\ t''')$ where Env' and t''' is Env and t', respectively, renamed by $\rho := \{x \mapsto x', y_i \mapsto y_i' \mid i = 1, \ldots, n\}$ and y_i' are fresh variables.
The analogous claim holds if $W[t']$ is the "in"-term.

Corollary 2.21
Let t be a closed expression which has a CWHNF. Then there is a constructor c and for every $j = 1, \ldots, \mathrm{ar}(c)$ there are closed terms t_j such that $t \sim_c (c\ t_1\ \ldots\ t_{\mathrm{ar}(c)})$.

Proof
Let t have a CWHNF t'. If t' is of the form $(\texttt{letrec}\ \mathrm{Env}\ \texttt{in}\ (c\ t_1\ \ldots, t_{\mathrm{ar}(c)}))$, then applying the extra transformation (ucp), which is correct (see Theorem 2.9) shows that $t \sim_c (\texttt{letrec}\ \mathrm{Env}, x = (c\ t_1\ \ldots, t_{\mathrm{ar}(c)})\ \texttt{in}\ x)$, where x has only the two indicated occurrences. Proposition 2.20 shows that there are closed expressions t_i' for $i = 1, \ldots, \mathrm{ar}(c)$ such that $t \sim_c (\texttt{letrec}\ \mathrm{Env}, x = (c\ t_1'\ \ldots, t_{\mathrm{ar}(c)}')\ \texttt{in}\ x)$. Using the correct transformations (ucp) and (gc), we see that $t \sim_c (c\ t_1'\ \ldots\ t_{\mathrm{ar}(c)}')$. □

[2] This was proved in the meantime in a more general way, see reports on www.ki.informatik.uni-frankfurt.de/papers/schauss

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

The following corollary on Ω in a strict position is intuitively clear; however, the proof is surprisingly difficult, and requires a proof that copying terms into surface contexts maintains contextual equivalence.

Corollary 2.22

Let t be a closed LR-expression. Then either $t \sim_c \Omega$, or t has an FWHNF, or $t \sim_c (c\ t_1 \dots t_{\mathrm{ar}(c)})$ for a constructor expression $(c\ t_1 \dots t_{\mathrm{ar}(c)})$.

Corollary 2.23

Let f be a closed function expression that is strict in its *ith* argument for arity n. Then for all (open) terms $t_1, \dots, t_n : (f\ t_1 \dots t_{i-1}\ \Omega\ t_{i+1} \dots, t_n) \sim_c \Omega$.

Proof

We use the context lemma to show the claim. Let R be a reduction context. Then we want to show that $R[(f\ t_1 \dots t_{i-1}\ \Omega\ t_{i+1} \dots, t_n)]\Downarrow \Leftrightarrow R[\Omega]\Downarrow$. Since the latter is shown to be $\sim_c \Omega$ in Corollary C.2, we have to show that $R[(f\ t_1 \dots t_{i-1}\Omega\ t_{i+1} \dots, t_n)]\Uparrow$. By appendix G it is sufficient to take closing reduction contexts R into account (see Proposition G.1). Then either R is a weak-reduction context and hence $t_1, \dots, t_n$ are closed or R can be represented by $(\mathtt{letrec}\ \mathrm{Env}_R, x = W[\cdot]\ \mathtt{in}\ r)$. The case that the hole is in the "in"-term can be derived from the latter case using the correctness of (ucp). Proposition 2.20 shows that there are closed terms t'_i for $i = 1, \dots, n$, such that $R[(f\ t_1 \dots t_{i-1}\ \Omega\ t_{i+1} \dots, t_n)] \sim_c R[(f\ t'_1 \dots t'_{i-1}\ \Omega\ t'_{i+1} \dots, t'_n)]$. Now strictness of f in the *ith* argument for arity n shows $R[(f\ t'_1 \dots t'_{i-1}\Omega\ t'_{i+1} \dots, t'_n)] \sim_c R[\Omega] \sim_c \Omega$. □

3 Strictness Analysis and its Safety

This part describes the strictness analysis algorithm SAL, which operates on abstract terms, shows its correctness based on the results in the previous part, and also illustrates SAL by several examples. The outline of this part is as follows. In subsection 3.1, we introduce abstract terms, i.e. expressions from LR extended with set constants and concretizations of abstract terms, which are expressions from LR. In subsection 3.2, we show that concretizations are retained by reduction and transformation on abstract terms. In subsection 3.3, we define the algorithm SAL and in subsection 3.4 its correctness is proved. Finally, in subsection 3.5, we illustrate the execution of the algorithm by examples.

3.1 Abstract Terms

Abstract terms are expressions from LR extended with set constants, defined below, used in the formulation of strictness properties and strictness analysis algorithms, like $\bot$, $\top$, *Inf* etc.

These constants were already used in (Nöcker 1992; 1993; van Eekelen *et al.* 1993; Schütz 2000). The notation for sets of terms was coined "demands" in (Schütz 2000). Note that we only use set constants, up to Fun, where the semantics are sets of expressions that are closed w.r.t. $\leqslant_c$.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Set Constants

Set constants are new symbols that have as semantics sets of (infinite) trees (over constructors and $\bot$), which is defined as a greatest fixpoint of recursion equations. The sets of trees then have as semantics sets of closed expressions, giving also a semantics to set constants in LR^0.

Let $\mathscr{U} = \{\bot, \mathrm{Fun}\} \mathbin{\dot\cup} \{U_1, \dots, U_K\}$ be a finite set of names of set constants. The set constants $U_i, i = 1, \dots, K$ are called *proper* set constants. For every proper set constant U_i, there is a defining rule

$$(Eq_i) : U_i = \{\bot\} \cup r_{i,1} \cup \cdots \cup r_{i,n_i},$$

where $r_{i,j}$ may be Fun or an expression $(c\ u'_1 \dots u'_{\mathrm{ar}(c)})$, where u'_j are proper set constants or $\bot$. With $\mathrm{rhs}_{Eq}(U_i)$ we denote the right-hand side of Eq_i.

The restriction excludes Fun to be u'_j in expressions $(c\ u'_1 \dots u'_{\mathrm{ar}(c)})$ on the right-hand side; however, it is possible to define a set constant $\mathrm{Fun}_\bot := \{\bot\} \cup \mathrm{Fun}$, which corresponds to a lifted Fun, and use it on other right-hand sides.

The set constants definitions will have a coinductive interpretation as trees, which can be generated using the following grammar:

$$VT ::= \mathrm{Fun} \mid \bot \mid c_i\ VT_1\ \cdots\ VT_{\mathrm{ar}(c_i)}$$

Let $\mathscr{T}_\infty$ be the set of all trees (including infinite trees) coinductively defined according to this grammar and $\mathscr{T}_*$ be the set of all (finite) trees inductively defined according to this grammar. The set $\mathscr{T} \subseteq \mathscr{T}_\infty$ is defined as the set of computable trees, i.e. $\mathscr{T} := \{T \mid t' \in \mathrm{LR}^0, \mathtt{RT}(t') = T\}$, where a representative tree $\mathtt{RT}(t)$ for every term $t \in \mathrm{LR}^0$ is the coinductively defined as:

$$\mathtt{RT}(t) = \begin{cases} \bot, & \text{if } t \sim_c \Omega \\ \mathrm{Fun}, & \text{if } t \sim_c t' \text{ and } t' \text{ is a closed FWHNF} \\ c\ T_1 \dots T_{\mathrm{ar}(c)} & \text{if } t \sim_c c\ t'_1 \dots t'_{\mathrm{ar}(c)} \text{ and } T_i = \mathtt{RT}(t'_i). \end{cases}$$

Note that this definition is well defined, since for every expression $t \in \mathrm{LR}^0$ exactly one of the following equivalences hold:

- $t \sim_c \Omega$, or
- $t \sim_c t'$ where t' is a closed FWHNF, or
- $t \sim_c c\ t'_1 \dots\ t'_{\mathrm{ar}(c)}$ for a constructor c, where the expressions t'_i are unique up to $\sim_c$.

The set $\mathscr{T}$ is admissible in the sense of (Schmidt-Schauß *et al.* 2005), i.e. $\mathscr{T}_* \subseteq \mathscr{T}$ and $\mathscr{T}$ is subtree-closed, which means that for every $T \in \mathscr{T}$, all its subtrees are also contained in $\mathscr{T}$.

A mapping ψ from proper set constants to $\mathscr{P}(\mathscr{T})$, where $\mathscr{P}$ denotes the powerset is called an *sc-interpretation*. For sc-interpretations ψ_1, ψ_2, we write $\psi_1 \leqslant \psi_2$, iff for all $i : \psi_1(U_i) \subseteq \psi_2(U_i)$. We define an extension ψ^e for sc-interpretations ψ as follows:

$$\begin{array}{lcl} \psi^e(\bot) & := & \{\bot\} \\ \psi^e(\mathrm{Fun}) & := & \{\mathrm{Fun}\} \\ \psi^e(c\ u_1\ \dots\ u_{\mathrm{ar}(c)}) & := & \{(c\ a_1 \dots a_{\mathrm{ar}(c)}) \mid a_i \in \psi(u_i)\} \\ \psi^e(r_1 \cup r_2) & := & \psi^e(r_1) \cup \psi^e(r_2). \end{array}$$

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

The equations Eq_i for the set constants define an operator Ψ on sc-interpretations as follows: $\Psi(\psi) := \psi^e \circ \mathrm{rhs}_{Eq}$. The operator Ψ is monotone and has a greatest fixed point ψ^*.

Remark 3.1
The greatest fixpoint ψ^* of Ψ can be computed as follows (see Schmidt-Schauß *et al.* (2005) for a proof of continuity). Let ψ_0 be the sc-interpretation with $\psi_0(U_i) = \mathscr{T}$ for $i = 1, \dots, K$. With $\psi_j := \Psi^j(\psi_0)$, the j-fold application of Ψ, for every $i = 1, \dots, K$, the equation $\psi^*(U_i) = \bigcap_j \psi_j(U_i)$ holds. This representation of the greatest fixpoint allows coinduction proofs in the style of induction proofs (Pitts 1994; Gordon 1994).

Definition 3.2
For every set constant $u \in \mathscr{U}$, we define a (tree) semantics $\gamma_{\mathscr{T}}(u) \subseteq \mathscr{T}$:

$$\gamma_{\mathscr{T}}(u) := \psi^*(u)(\text{for the greatest fixpoint } \psi^* \text{ of } \Psi).$$

Lemma 3.3
Let t_1, t_2 be closed LR expressions, which have CWHNFs. Then $t_1 \leqslant_c t_2$ iff there is a constructor c, and for $j = 1, \dots, \mathrm{ar}(c)$ there are closed terms $t_{1,j}, t_{2,j}$ such that $t_{1,j} \leqslant_c t_{2,j}$, $t_1 \sim_c (c\ t_{1,1}\ \dots\ t_{1,\mathrm{ar}(c)})$ and $t_2 \sim_c (c\ t_{2,1}\ \dots\ t_{2,\mathrm{ar}(c)})$.

Proof
The if-direction is obvious. To prove the other direction, let $t_1 \leqslant_c t_2$. Then Corollary 2.21 shows that there are closed expressions $t_{1,j}, t_{2,j}$ for $j = 1, \dots, \mathrm{ar}(c)$, such that $(c\ t_{1,1}\ \dots\ t_{1,\mathrm{ar}(c)}) \sim_c t_1 \leqslant_c t_2 \sim_c (c\ t_{2,1}\ \dots\ t_{2,\mathrm{ar}(c)})$. Using contexts $C_i := (\mathtt{case}_T\ [\cdot]\ (c\ x_1\ \dots x_{\mathrm{ar}(c)}) \to x_i\ \dots)$ for $i = 1, \dots, n$, it is easy to see that for $j = 1 \dots, \mathrm{ar}(c)$: $t_{1,j} \leqslant_c t_{2,j}$. □

Lemma 3.4
Let $s \leqslant_c t \in \mathrm{LR}_0$, $\mathtt{RT}(t) \in \gamma_{\mathscr{T}}(u)$, then $\mathtt{RT}(s) \in \gamma_{\mathscr{T}}(u)$.

Proof
We use Remark 3.1 and show by induction that for every i the claim holds for the sets $\psi_i(U_k)$ for all k. The base case is $\psi_0(U_k) = \mathscr{T}$, for all k. Since for every term $t \in \mathrm{LR}^0$ its representative tree $\mathtt{RT}(t)$ is included in $\mathscr{T}$ the claim holds.

Let i be a positive integer. We use as induction hypothesis that the claim holds for $\psi_{i-1}(U_k)$ for all k. Let t_1, t_2 be closed expressions with $t_1 \leqslant_c t_2$ and $\mathtt{RT}(t_2) \in \psi_i(U_k)$. If $t_1 \sim_c \Omega$, then $\mathtt{RT}(t_1) = \bot$ and the lemma holds since every defining equation contains a component $\bot$. If t_1 has an FWHNF, then t_2 must also have an FWHNF, and hence $\mathtt{RT}(t_2) = \mathrm{Fun}$ and thus Fun must be a component of the defining equation for U_k. Since iterations of Ψ cannot remove *Fun*, we have $\mathtt{RT}(t_1) \in \psi_i(U_k)$.

If t_1 has a CWHNF for constructor c, then by Lemma 3.3 there are closed terms $t_{1,j}, t_{2,j}$ with $t_1 \sim_c (c\ t_{1,1}\ \dots\ t_{1,\mathrm{arc}})$, $t_2 \sim_c (c\ t_{2,1}\ \dots\ t_{2,\mathrm{arc}})$ and $t_{1,j} \leqslant_c t_{2,j}$. Then $\mathtt{RT}(t_2) = (c\ T'_{2,1}\ \dots\ T'_{2,\mathrm{ar}(c)})$, where $\mathtt{RT}(t_{2,i}) = T'_{2,i}$. Since $\mathtt{RT}(t_2) \in \psi_i(U_k)$ in the defining equation for U_k, there is a component $(c\ u_{k,1}\ \dots u_{k,\mathrm{ar}(c)})$ on the right-hand side. The definition of ψ implies that $T'_{2,j} \in \psi_{i-1}(u_{k,j})$ for all j. Using the induction hypothesis and the fact that only proper set constants and $\bot$ are possible as $u_{k,j}$: $\mathtt{RT}(t_{1,j}) \in \psi_{i-1}(u_{k,j})$ for all j. This implies $\mathtt{RT}(t_1) \in \psi_i(U_k)$.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

We have shown that the claim holds for all k and all $i : \psi_i(U_k)$. Since $\gamma(U_k) = \bigcap_i \psi_i(U_k)$, we obtain the claim for $\gamma(U_k)$, too. □

We now define the term-semantics γ for set constants by mapping the trees to terms.

Definition 3.5
The mapping $[\![\cdot]\!] : \mathscr{T} \rightarrow \mathrm{LR}^0$ is defined as $[\![T]\!] := \{t \in \mathrm{LR}^0 \mid \mathtt{RT}(t) = T\}$. The extension of $[\![\cdot]\!]$ to sets $S \subseteq \mathscr{T}$ is defined as $\theta(S) := \bigcup_{s \in S} [\![s]\!]$.

For every set constant $u \in \mathscr{U}$: $\gamma(u) \subseteq \mathrm{LR}^0$ is defined as $\gamma(u) := \theta(\gamma_{\mathscr{T}}(u))$

Lemma 3.6
For all terms $t \in \mathrm{LR}^0$: $t \in [\![\mathtt{RT}(t)]\!]$.

Proof
This follows from the definition of $[\![\cdot]\!]$ □

Lemma 3.7
For every proper set constant u: $\gamma(u)$ is down-closed, i.e. $t_1 \leqslant_c t_2 \in \gamma(u) \Rightarrow t_1 \in \gamma(u)$.

Proof
Let $t_1 \leqslant_c t_2$ and $t_2 \in \gamma(u) = \theta(\gamma_{\mathscr{T}}(u)) = \bigcup\limits_{T \in \gamma_{\mathscr{T}}(u)} [\![T]\!]$, i.e. there is a tree T_2 with $T_2 \in \gamma_{\mathscr{T}}(u)$ and $t_2 \in [\![T_2]\!]$. From the definition of $[\![\cdot]\!]$, we have $\mathtt{RT}(t_2) = T_2$. Using Lemma 3.4, we have $\mathtt{RT}(t_1) \in \gamma_{\mathscr{T}}(u)$ and from Lemma 3.6 we obtain $t_1 \in [\![\mathtt{RT}(t_1)]\!]$ and hence $t_1 \in \theta(\gamma_{\mathscr{T}}(u)) = \gamma(u)$. □

In the following, we assume that the set constants $\top$ and *Inf* are proper set constants and defined with the defining equations

$$\begin{array}{lll} \top & := & \{\bot\} \cup \mathrm{Fun} \cup (c_1\ \top \cdots \top) \cup \cdots \cup (c_N\ \top\ \cdots\ \top) \\ & & \text{where } c_1, \ldots, c_N \text{ are all constructors.} \\ \mathit{Inf} & := & \{\bot\} \cup (\top : \mathit{Inf}) \\ & & \text{where `` :'' is the binary constructor for lists.} \end{array}$$

Lemma 3.8
The equation $\gamma(\top) = \mathrm{LR}^0$ holds.

Proof
Using Remark 3.1 and that $\mathscr{T}$ is subtree-closed, it easy to show that $\gamma_{\mathscr{T}}(\top) = \mathscr{T}$. The definition of $[\![\cdot]\!]$ then shows $\theta(\mathscr{T}) = \mathrm{LR}^0$. Hence, we have $\gamma(\top) = \mathrm{LR}^0$. □

Lemma 3.9

- Every expression $t \in \gamma(\top)$ is either in $\gamma(\bot)$ or $\gamma(\mathrm{Fun})$, or contextually equivalent to a term of the form $(c\ t_1 \ldots t_n)$ where c is a constructor and t_i are closed expressions.
- Every expression $t \in \mathit{Inf}$ is either in $\gamma(\bot)$ or is contextually equivalent to a term of the form $t_1 : t_2$, where t_1, t_2 are closed and $t_2 \in \gamma(\mathit{Inf})$.

Remark 3.10
The definition of abstract constants does not cover the nondown-closed demands in (Schütz 2000) like Fin, the abstract constant representing all finite lists.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Terms Including Set Constants: Abstract Terms

We define structured terms including set constants, which are used to represent sets of concrete expressions. Since we will only use very special terms including set constants, we will define these terms as our abstract terms language.

Definition 3.11
We extend the language LR by the set constants (as expressions) in $\mathcal{U}$ according to the following restrictions. A closed term t including set constants as expressions is called an *abstract term* if it is of the form $(\texttt{letrec}\ \mathrm{Env}_{ac}, \mathrm{Env}_{up}\ \texttt{in}\ r)$, where the environment for abstract (set) constants is of the form $\mathrm{Env}_{ac} = \{x_1 = u_1, \ldots , x_n = u_n\}$, where u_i are set constants with $u_i \neq \bot$ for all $i = 1, \ldots , n$, and where Env_{up} and r may only contain $\bot$ as set constant. The language $\mathrm{LR}_{\mathcal{U}}$ is the set of all abstract terms built with set constants from $\mathcal{U}$. The variables in $\mathrm{LV}(\mathrm{Env}_{ac})$ will be called *ac-variables*, the variables in $\mathrm{LV}(\mathrm{Env}_{up})$ are called *up-variables*.

The language LR will remain the foundation for formal proofs, thus it is not necessary to repeat all the definitions and results also for the abstract language. Nevertheless, we have to clarify what we mean if we apply LR-notions also to abstract expressions.

Definition 3.12

1. *Contexts:* For abstract terms, we also speak of contexts, reduction contexts, surface contexts, etc., where the definitions are the same as for the concrete language.
2. *Abstract deep and strict subterms:* These are defined as for concrete terms. The definition of the property "deep" is the same. To apply the definition of strict subterms we view the set constants in a term as atomic expressions.

Now we define the reductions on abstract expressions.

Definition 3.13

1. *Abstract reductions.*
   - *Base reductions.* These are also used for abstract terms (see Definition 1.3), where set constants are treated like (atomic) expressions.
   - The following new reduction rule, which is easily seen as correct on the basis of our results on LR will also be used on abstract expression:

$$(\text{seq-Fun})\ (\texttt{seq}\ x\ t) \to t \qquad \text{if } x \text{ is bound to Fun.}$$

   - *(Abstract) extra transformations.* For abstract terms, the transformations from Definition 2.8 that are used for concrete expressions are also defined, where set constants are treated like expressions.

   However, since only abstract terms are acceptable as result, there is the following obvious restriction: The non-$\bot$ set constants must not be copied, i.e., the transformation (ucp) is not allowed for the binding $x = u$, where u is a non-$\bot$ set constant.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

(cp-bot-in)	$(\texttt{letrec}\ x = \bot, Env\ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = \bot, Env\ \texttt{in}\ C[\bot])$
(cp-bot-e)	$(\texttt{letrec}\ x = \bot, Env, y = C[x]\ \texttt{in}\ t) \to (\texttt{letrec}\ x = \bot, Env, y = C[\bot]\ \texttt{in}\ t)$
(hole)	$(\texttt{letrec}\ x = x, Env\ \ \texttt{in}\ r) \to (\texttt{letrec}\ x = \bot, Env\ \texttt{in}\ r)$

All the reductions are permitted in any context.

Fig. 4. Abstract reduction rules for $\bot$.

2. *Abstract normal-order reduction and abstract evaluation.* This is also defined accordingly. The rule (seq-Fun) is also permitted in abstract normal order reductions. Note that an abstract normal order reduction sequence may get stuck if a set constant is in a reduction context.
3. *Abstract bot-reductions.* Subterms of the following form can be replaced by $\bot$ in any context:
   $(\bot\ t), (\texttt{seq}\ \bot\ t), ((c\ \overrightarrow{t})\ r), (x\ t)$ where x is bound to a constructor application $(c\ \overrightarrow{t})$; $(\texttt{case}_T\ t\ \ldots)$ if $t = \bot$, or if the case-expression is "untyped", i.e., t is an abstraction or it has a top-constructor that does not belong to type T; $(\texttt{case}_T\ x\ \ldots)$ if x is bound to an abstraction or to a term t that has a top-constructor that does not belong to type T, or x is bound to Fun; $(f\ t_1\ \ldots t_{i-1}\ \bot\ t_{i+1}\ \ldots\ t_n)$, if f is strict in its i^{th} argument for arity n.
   There are further abstract -bot-reduction rules defined in figure 4.
   For more discussion on bot-rules see Remark 3.39.

For an abstract term t, let simp(t) be the result of exhaustively applying simplification rules and abstract bot-reduction rules.

Proposition 3.14
The application of simplification and bot-reduction rules terminates.

Proof
The simplification rules terminate, and do not increase the size of a term, and the bot-reduction rules strictly reduce the size of a term. □

Later, for the construction of expressions for subcases, there will be the following kinds of modifications on abstract expressions. For the exact conditions, see Definition 3.27.

Definition 3.15
Let t be an abstract term.

1. *uu-modification.* A binding $x = u$ in the ac-part is modified into $x = u'$ where u' is another set constant.
2. *ucx-modification.* Let $u = \{\bot\} \cup r_1 \cup \ldots \cup r_k$ be the defining equation for a set constant u. Then an *ucx-modification* consists in replacing a binding $x = u$ in the ac-part by one of the following: $x = \bot$, or $x =$ Fun, if $r_j =$ Fun for some $j = 1, \ldots, n$, or $x = (c\ x_1 \ldots x_{\mathrm{ar}(c)})$, where $x_i, i = 1, \ldots, \mathrm{ar}(c)$ are new ac-variables, and bindings $x_i = u_i, i = 1, \ldots, \mathrm{ar}(c)$ are added, where u_i are set

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

constants. The condition is that the expression $(c\ x_1 \dots x_{\mathrm{ar}(c)})$ will correspond to some r_j in the right hand side of the definition of u. Note that the binding $x = (c\ x_1\ \dots\ x_{\mathrm{ar}(c)})$ is in the upper part after the modification. If there is a new binding $x_i = \bot$, then a subsequent simplification will remove this binding and replace all occurrences of x_i by Ω.

3. *Generalisation.* $(\texttt{letrec}\ \mathrm{Env}_{ac}, \mathrm{Env}_{up}\ \texttt{in}\ C[s]) \rightarrow (\texttt{letrec}\ x = \top; \mathrm{Env}_{ac}, \mathrm{Env}_{up}\ \texttt{in}\ C[x])$, where the new ac-environment is $\{x = \top\} \dot\cup \mathrm{Env}_{ac}$.

It is clear that reducing an abstract closed term according to Definition 3.13 or modifying it according to Definition 3.15 results in an abstract closed expression.

Concretizations of Abstract Terms

We define concretizations s of abstract terms t as (concrete) LR terms, where the relationship can be informally described as follows. The terms s, t are assumed to be letrec-expressions. After eliminating the ac-part of the environment and after replacing all occurrences of $\bot$ in t by Ω, the terms s, t must be syntactically identical. Moreover, for the bound terms in s, t corresponding to the free variables (which are the ac-variables), roughly, a $\leqslant_c$-relationship must hold.

For example, $(\texttt{letrec}\ y = 2, x = 1 : x\ \texttt{in}\ y : x)$ is a concretization of $(\texttt{letrec}\ \mathrm{top} = \top, \mathrm{inf} =, \mathit{Inf}\ \texttt{in}\ \mathrm{top} : \mathrm{inf})$, where the variable renaming is $\{x \rightarrow \mathrm{inf}, y \rightarrow \mathrm{top}\}$. The situation is in general a bit more complicated due to sharing in the concretization.

Note that for an abstract term t, not every closed expression s with $s \leqslant_c s'$, where s' is constructed from t by appropriately replacing the set-constants, will qualify as a concretization of t. The reason for the definition is that the reduction length must be reflected in the concretizations. This is captured in the upper part. The ac-part captures the condition that something must hold for all terms. Fortunately, this does not destroy our argumentation on the reduction length.

Definition 3.16 (*concretization*)
Let $t = (\texttt{letrec}\ \mathrm{Env}_{t,ac}, \mathrm{Env}_{t,up}\ \texttt{in}\ t')$ be a closed abstract term, where $\mathrm{Env}_{t,ac} = \{y_1 = u_1, \dots, y_k = u_k\}$ and where u_i are set constants.
Then the set of *concretizations* $\gamma(t)$ is defined as follows: Let $s = (\texttt{letrec}\ \mathrm{Env}_s\ \texttt{in}\ s')$ be a closed concrete term. Then $s \in \gamma(t)$ iff the following holds

- There is a split of the environment Env_s into three parts: a sharing part $\mathrm{Env}_{s,sh}$, a part corresponding to the ac-part: $\mathrm{Env}_{s,ac}$, and an upper part $\mathrm{Env}_{s,up}$, I.e., $\mathrm{Env}_s = \mathrm{Env}_{s,sh} \dot\cup \mathrm{Env}_{s,ac} \dot\cup \mathrm{Env}_{s,up}$ with $|\mathrm{LV}(\mathrm{Env}_{s,ac})| = k$, $FV(\mathrm{Env}_{s,sh} \dot\cup \mathrm{Env}_{s,ac}) = \emptyset$ and $FV(\mathrm{Env}_{s,up}, s') \subseteq \mathrm{LV}(\mathrm{Env}_{s,ac})$.
- The expressions $s_1 := (\texttt{letrec}\ \mathrm{Env}_{s,up}\ \texttt{in}\ s')$ and $t_1 := (\texttt{letrec}\ \mathrm{Env}_{t,up}\ \texttt{in}\ t')$ are equal up to a renaming of free variables, after all occurrences of $\bot$ in t_1 are replaced by Ω. I.e., for $\mathrm{LV}(\mathrm{Env}_{s,ac}) = \{x_1, \dots, x_k\}$, $\mathrm{LV}(t_1) = \{y_1, \dots, y_k\}$ appropriately ordered and for ρ defined by $\rho(x_i) = y_i$ for $i = 1, \dots, k$, the equation $\rho(s_1) = t_1[\Omega/\bot]$ holds.
- For every i: $(\texttt{letrec}\ \mathrm{Env}_{s,sh}, \mathrm{Env}_{s,ac}\ \texttt{in}\ x_i) \in \gamma(u_i)$.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Example 3.17
We want to show that $s \in \gamma(t)$, where $s = (\texttt{letrec}\ x_1 = r, x_2 = (\texttt{repeat}\ x_1)\ \texttt{in}\ x_2)$ and $t = (\texttt{letrec}\ \text{inf} = \mathit{Inf}\ \texttt{in}\ \text{inf})$ and r is a closed expression. For convenience, we omit the definition of repeat as $\texttt{repeat} = \lambda x.x : (\texttt{repeat}\ x)$ in the respective upper environments.

The environments are: $\text{Env}_{t,ac} = \{\text{inf} = \mathit{Inf}\}$, $\text{Env}_{s,ac} = \{x_2 = (\texttt{repeat}\ x_1)\}$, $\text{Env}_{s,sh} = \{x_1 = r\}$. We only have to check that $(\texttt{letrec}\ x_1 = r, x_2 = (\texttt{repeat}\ x_1)\ \texttt{in}\ x_2) \in \gamma(\mathit{Inf})$. The correctness of reduction, extra transformations and Proposition 2.20 yield

$$\begin{array}{ll} & (\texttt{letrec}\ x_1 = r, x_2 = (\texttt{repeat}\ x_1)\ \texttt{in}\ x_2) \\ \sim_c & (\texttt{letrec}\ x_1 = r, x_2 = ((\texttt{letrec}\ x = x_1\ \texttt{in}\ x : \texttt{repeat}\ x))\ \texttt{in}\ x_2) \\ \sim_c & (\texttt{letrec}\ x_1 = r, x = x_1, x_2 = (x : \texttt{repeat}\ x)\ \texttt{in}\ x_2) \\ \sim_c & (\texttt{letrec}\ x_1 = r, x_2 = x_1 : \texttt{repeat}\ x_1\ \texttt{in}\ x_2) \\ \sim_c & (\texttt{letrec}\ x_1 = r, x_3 = \texttt{repeat}\ x_1\ \texttt{in}\ x_1 : x_3) \\ \sim_c & r\ :\ (\texttt{letrec}\ x_1 = r, x_3 = \texttt{repeat}\ x_1\ \texttt{in}\ x_3). \end{array}$$

Now the membership check means to test $r \in \gamma(\top)$, which holds, and to check $(\texttt{letrec}\ x_1 = r, x_3 = \texttt{repeat}\ x_1\ \texttt{in}\ x_3) \in \gamma(\mathit{Inf})$, which can now be proved using coinduction.

Proposition 3.18
Let t be an abstract term, $s \in \gamma(t)$, $t \to t'$ by a bot-reduction as defined in Definition 3.13, and $s\Downarrow$. Then there exists an s' with $s' \in \gamma(t')$, $s'\Downarrow$ and $\text{rl}\sharp\sharp(s) = \text{rl}\sharp\sharp(s')$.

Proof
This follows from Proposition C.4 in the appendix, where for the application of the proposition, the set constant $\bot$ has to be replaced by the bot-term Ω. □

We define strict reductions for abstract terms.

Definition 3.19 (abstract sp-reduction)
Let $t = (\texttt{letrec}\ \text{Env}_{ac}, \text{Env}_{up}\ \texttt{in}\ t')$ be an abstract term. Then a subterm r in Env_{up} or t' is called *strict*, iff for all concretizations $s \in \gamma(t)$, $s[\Omega/r] \sim_c \Omega$. This is well-defined, since the position of r in s can be uniquely determined.

A reduction $t \to t'$, which is a (case), (seq), (lbeta), (cp), or (seq-Fun) reduction where the inner redex is a strict subexpression in a surface context, is called an *abstract strict position reduction (abstract sp-reduction).*

It is obvious that the following sufficient condition holds

Lemma 3.20
Let $t = (\texttt{letrec}\ \text{Env}_{ac}, \text{Env}_{up}\ \texttt{in}\ t')$ be an abstract term, let r be a subexpression in Env_{up} or t', and let t'', r'' be constructed from $(\texttt{letrec}\ \text{Env}_{up}\ \texttt{in}\ t')$ and r, respectively, by replacing all $\bot$-occurrences by Ω. If r'' is a strict subterm in t'', then r is strict in t.

In subsection 3.3 we will exhibit more sufficient conditions for strictness in abstract terms.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Subset Relationship for Abstract Terms

The subset relation w.r.t. concretization between two abstract terms is defined as follows.

Definition 3.21
Let t_1, t_2 be two closed abstract terms.
Then $t_1 \subseteq_\gamma t_2$ iff $\gamma(t_1) \subseteq \gamma(t_2)$.

A sufficient condition for $\subseteq_\gamma$ is given in the following lemma.

Lemma 3.22
Let $t_1 = (\texttt{letrec}\ \mathrm{Env}_1\ \texttt{in}\ t'_1)$ and $t_2 = (\texttt{letrec}\ \mathrm{Env}_2\ \texttt{in}\ t'_2)$ be two closed abstract terms. Then $t_1 \subseteq_\gamma t_2$ if the following holds.

- The environment Env_1 is split into $\mathrm{Env}_1 = \mathrm{Env}_{1,ac} \dot\cup \mathrm{Env}_{1,up}$, where $\mathrm{Env}_{1,ac}$ is the ac-part which is of the form $\{x_{1,1} = u_{1,1}, \dots, x_{1,k} = u_{1,k}\}$.
- The environment Env_2 is split into $\mathrm{Env}_2 = \mathrm{Env}_{2,ac} \dot\cup \mathrm{Env}_{2,up}$, where $\mathrm{Env}_{2,ac}$ is the ac-part, which is of the form $\{x_{2,1} = u_{2,1}, \dots, x_{2,k} = u_{2,k}\}$.
- For $r_1 = (\texttt{letrec}\ \mathrm{Env}_{1,up}\ \texttt{in}\ t'_1)$ and $r_2 = (\texttt{letrec}\ \mathrm{Env}_{2,up}\ \texttt{in}\ t'_2)$, the equation $\rho(r_1) = r_2$ must hold, where $\rho(x_{1,i}) = x_{2,i}$ for $i = 1, \dots, k$.
- For every i: $\gamma(u_{1,i}) \subseteq \gamma(u_{2,i})$.

Proof
The conditions can directly be matched with the conditions in Definition 3.16. □

We do not give an algorithm for detecting $\gamma(u_1) \subseteq \gamma(u_2)$ based on the defining rules, since this is beyond the scope of this paper, however, the relation is decidable. In (Schmidt-Schauß *et al.* 2005) it is shown that the decision problem is DEXPTIME-complete. In examples we will only use the relations $\bot \subseteq \gamma(u)$ and $\gamma(u) \subseteq \gamma(\top)$.

3.2 Inheritance of Concretizations

In this section, we show that for every concretization of an abstract term there exists a corresponding concretization after abstract reduction, abstract sp-reduction or applying some extra transformation.

Lemma 3.23
Let t be an abstract expression $s \in \gamma(t)$, $s\Downarrow$ and $t \to t'$, where the abstract reduction (see Definition 3.13) is within the upper part. Then there is some $s' \in \gamma(t')$, such that $s'\Downarrow, s \sim_c s'$ and $\mathrm{rl}\sharp\sharp(s) \geqslant \mathrm{rl}\sharp\sharp(s')$.

Proof
Let $s \in \gamma(t)$ with $s\Downarrow$. We analyze the following cases.

1. Let the reduction be an abstract base reduction, or a non-exceptional abstract extra transformation, i.e. not a (gc)-reduction removing bindings of set-constants. If the reduction is in the upper part of t, then the same reduction is possible on s giving s'. Since the respective effects of the reductions are the same, and since the expressions in s' at the position of the set constants are

the same as before the reduction, we see that $s' \in \gamma(t')$. Theorems 2.4 and 2.9 show that contextual equivalence holds, and Theorem 2.14 shows the claim on the reduction length.

2. Let the reduction be a (gc) that eliminates bindings of set constants. Then we cannot use (gc) on s in every case, hence other arguments are required. We show that $s \in \gamma(t')$. Let $t = (\texttt{letrec}\ \mathrm{Env}_{t,ac}, \mathrm{Env}_{t,up}\ \texttt{in}\ t_0), s = (\texttt{letrec}\ \mathrm{Env}_{s,sh}, \mathrm{Env}_{s,ac}, \mathrm{Env}_{s,up}\ \texttt{in}\ s_0)$, and let $\mathrm{Env}'_{t,ac}$ be the reduced environment. For convenience assume that $\mathrm{LV}(\mathrm{Env}_{t,ac}) = \mathrm{LV}(\mathrm{Env}_{s,ac})$ has the bindings $x_i = u_i$, for $i = 1, \ldots, n$, and the name correspondence is according to the definition of concretization. Let $\mathrm{Env}_{s,ac} = \mathrm{Env}'_{s,ac} \dot\cup \mathrm{Env}''_{s,ac}$ with $\mathrm{LV}(\mathrm{Env}'_{s,ac}) = \mathrm{LV}(\mathrm{Env}'_{t,ac})$, and $\mathrm{Env}'_{s,sh} = \mathrm{Env}_{s,sh} \dot\cup \mathrm{Env}''_{s,ac}$. The condition $FV(\mathrm{Env}_{s,up}, s_0) \subseteq \mathrm{LV}(\mathrm{Env}'_{s,ac})$ holds, since (gc) is applicable to t.
From $x_i \in \mathrm{LV}(\mathrm{Env}'_{s,ac})$ for all i, and from $s \in \gamma(t)$ we derive $(\texttt{letrec}\ \mathrm{Env}_{s,sh}, \mathrm{Env}'_{s,ac}, \mathrm{Env}''_{s,ac}\ \texttt{in}\ x_i) \in \gamma(u_i)$, hence $s \in \gamma(t')$.
3. Let the reduction be a (seq-Fun) reduction. It is sufficient to consider the case: $t := (\texttt{letrec}\ x = \mathrm{Fun}, \ldots\ \texttt{in}\ C[(\texttt{seq}\ x\ r)]) \to (\texttt{letrec}\ x = \mathrm{Fun}, \ldots\ \texttt{in}\ C[r])$. Let $s \in \gamma(t)$. Then $s = (\texttt{letrec}\ \mathrm{Env}_{s,sh}, \mathrm{Env}_{s,ac}, \mathrm{Env}_{s,up}\ \texttt{in}\ C[(\texttt{seq}\ x\ r)])$, where $x \in \mathrm{LV}(\mathrm{Env}_{s,ac})$, and $(\texttt{letrec}\ \mathrm{Env}_{s,sh}, \mathrm{Env}_{s,ac}\ \texttt{in}\ x) \in \gamma(\mathrm{Fun})$. The definition of $\gamma(\mathrm{Fun})$ implies that there is a reduction sequence of $(\texttt{letrec}\ \mathrm{Env}_{s,sh}, \mathrm{Env}_{s,ac}\ \texttt{in}\ x)$ to a FWHNF. The same reductions can be made on s giving $s'' = (\texttt{letrec}\ \mathrm{Env}''_{s,sh}, \mathrm{Env}''_{s,ac}, \mathrm{Env}_{s,up}\ \texttt{in}\ C[(\texttt{seq}\ x\ r)])$, such that x is bound to an abstraction in $\mathrm{Env}''_{s,ac}$. Moreover, $s \sim_c s''$, and $\mathrm{rl}\sharp\sharp(s) \geqslant \mathrm{rl}\sharp\sharp(s'')$. Now a (seq)-reduction is possible: $s'' = (\texttt{letrec}\ \ldots\ \texttt{in}\ C[(\texttt{seq}\ x\ r)]) \to (\texttt{letrec}\ \ldots\ \texttt{in}\ C[r]) =: s'$. We have $\mathrm{rl}\sharp\sharp(s'') \geqslant \mathrm{rl}\sharp\sharp(s')$. To check the conditions for $s' \in \gamma(t')$, the only missing part is $(\texttt{letrec}\ \mathrm{Env}''_{s,sh}, \mathrm{Env}''_{s,ac}\ \texttt{in}\ x_i) \in \gamma(u_i)$ for other set constants. This follows from Theorem 2.4, and since $\gamma(u_i)$ is closed w.r.t. $\sim_c$. We obtain $s' \in \gamma(t')$. □

Proposition 3.24
Let t be an abstract expression. If $t \to t'$ by an abstract sp-reduction (see Definition 3.19), but not a (cp), and $s \in \gamma(t)$, then there is an expression $s' \in \gamma(t')$ with $s \sim_c s'$ and $\mathrm{rl}\sharp\sharp(s) > \mathrm{rl}\sharp\sharp(s')$.

Proof
We have only to argue that there is an expression $s' \in \gamma(t')$ with $\mathrm{rl}\sharp\sharp(s) > \mathrm{rl}\sharp\sharp(s')$. Using Proposition 2.15 and since inner redexes are in surface contexts, the proof of Lemma 3.23 shows the claim for abstract sp-reductions that are not (cp) and not (seq-Fun) reductions. In the case of a (seq-Fun) reduction, we have to show that the redex $(\texttt{seq}\ x\ r)$ in s'' remains a strict subterm after the reduction sequence $s \xrightarrow{*} s''$. This holds, since there are no modifications in Env_{up} and $C[(\texttt{seq}\ x\ r)]$. Now Proposition 2.15 shows the claim on the length. □

Lemma 3.25
Let $t = (\texttt{letrec}\ \mathrm{Env}_{t,ac}, \mathrm{Env}_{t,up}\ \texttt{in}\ t_{in})$ be an abstract expression, $s \in \gamma(t)$ with $s\Downarrow$, let $x_1 = u_1, \ldots, x_N = u_N$ be the ac-bindings, $x_i = u_i$ be a fixed binding in $\mathrm{Env}_{t,ac}$ and let $u_i = \{\bot\} \cup r_1 \cup \ldots \cup r_k$ be the defining equation for u_i. Let t_j for $j = 1, \ldots, k$

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

be the ucx-modification (see Definition 3.15) according to r_j. Then there is some $j \in \{0, \ldots, k\}$ and an $s' \in \gamma(t_j)$, such that $s \sim_c s'$ and $\mathrm{rl}\sharp\sharp(s) \geqslant \mathrm{rl}\sharp\sharp(s')$.

Proof

Let $s \in \gamma(t)$. Then s can be represented as $s := (\texttt{letrec}\ \mathrm{Env}_{s,sh}, \mathrm{Env}_{s,ac}, \mathrm{Env}_{s,up}\ \texttt{in}\ s_{in})$. With $\bar{s} := (\texttt{letrec}\ \mathrm{Env}_{s,sh}, \mathrm{Env}_{s,ac}\ \texttt{in}\ x_i)$, we have in particular $\bar{s} \in \gamma(u_i)$.

If $\bar{s}\Uparrow$, then we let $j = 0, s' := s$, and we have $s' \in \gamma(t_0)$. Assume $\bar{s}\Downarrow$. Then there is a normal-order reduction of $\bar{s}$ to a WHNF. If it is a FWHNF, then the same reductions as for $\bar{s}\Downarrow$ can be made on s with the exception of the last (cp), resulting in $s' = (\texttt{letrec}\ \mathrm{Env}'_{s,sh}, \mathrm{Env}'_{s,ac}, \mathrm{Env}_{s,up}\ \texttt{in}\ s_{in})$. We have $s \sim_c s'$ and $\mathrm{rl}\sharp\sharp(s) \geqslant \mathrm{rl}\sharp\sharp(s')$ by Theorems 2.4 and 2.14. It is obvious that $r_1 = \mathrm{Fun}$, and furthermore $s' \in \gamma(t_1)$.

If the WHNF of $\bar{s}$ is a CWHNF, then let c be the corresponding constructor of the value v in the CWHNF. The normal order reduction reduces $\bar{s}$ to $\bar{s}'' := (\texttt{letrec}\ \mathrm{Env}'_{s,sh}, \mathrm{Env}'_{s,ac}\ \texttt{in}\ x_i)$, such that x_i is bound to an expression $(c\ a_1 \ldots a_{\mathrm{ar}(c)})$. The same transformations can be performed for s and produce an expression $s'' := (\texttt{letrec}\ \mathrm{Env}'_{s,sh}, \mathrm{Env}'_{s,ac}, \mathrm{Env}_{s,up}\ \texttt{in}\ s_{in})$ with $s \sim_c s''$ and $\mathrm{rl}\sharp\sharp(s) \geqslant \mathrm{rl}\sharp\sharp(s'')$ by Theorems 2.4 and 2.14. There is a reduction sequence $\bar{s}'' \xrightarrow{(cpcx,cpx,gc)^*} \bar{s}'$, where the environment contains the bindings $x_i = (c\ x_{i,1} \ldots x_{i,\mathrm{ar}(c)}), x_{i,1} = a_1, \ldots, x_{i,\mathrm{ar}(c)} = a_{i,\mathrm{ar}(c)}$. The same reductions performed for s'' yield: $s'' \xrightarrow{(cpcx,cpx,gc)^*} s'$, where $s \sim_c s'$ and $\mathrm{rl}\sharp\sharp(s'') = \mathrm{rl}\sharp\sharp(s')$ by Theorem 2.14.

It remains to show that $s' \in \gamma(t_j)$ for some j. For any ac-variable $x_h \neq x_i$, the membership $(\texttt{letrec}\ \mathrm{Env}'_{s,sh}, \mathrm{Env}'_{s,ac}\ \texttt{in}\ x_h) \in \gamma(u_h)$ holds also in s', since (cpcx), (cpx), (gc) are correct program transformations. Since $\gamma(u_i) = \{\bot\} \cup \bigcup_{h=1,\ldots,k} \psi^*(r_h)$, there is some j such that $\bar{s}' \in \{(c\ b_1 \ldots b_{\mathrm{ar}(c)}) \mid b_h \in \gamma(u_{i,h}), h = 1, \ldots, \mathrm{ar}(c)\}$. For the thus chosen t_j, we show that $s' \in \gamma(t_j)$:

The modifications of s to generate s' are in the environments $\mathrm{Env}_{s,sh}, \mathrm{Env}_{s,ac}$, with the exception of the bindings $x_i = (c\ x_{i,1} \ldots x_{i,\mathrm{ar}(c)}), x_{i,1} = a_1, \ldots, x_{i,\mathrm{ar}(c)} = a_{i,\mathrm{ar}(c)}$, where the first binding moves into the upper part, and the other bindings move into the ac-part of s'. This corresponds to the environment of t_j, hence the renaming condition is satisfied. The conditions $(\texttt{letrec}\ \mathrm{Env}'_{s,sh}, \mathrm{Env}'_{s,ac}\ \texttt{in}\ x_{i,h}) \in \gamma(u_{i,h})$ for $h = 1, \ldots, \mathrm{ar}(c)$ follow from Corollary 2.21, and since we have only used correct program transformations. The membership $(\texttt{letrec}\ \mathrm{Env}_{s,sh}, \mathrm{Env}_{s,ac}\ \texttt{in}\ x_h) \in \gamma(u_h)$ for $h \neq i$ follows since only correct program transformations are used. □

3.3 The Algorithm SAL

We present the algorithm SAL (strictness analyzer for a lazy functional language), which is a reformulation of the algorithm Nöcker implemented for Clean. The core is a method to detect nontermination of concretizations of abstract terms.

Intuitively, strictness of a function f is detected if the normal order reduction of $(f\ \bot)$ in the abstract language can only yield $\bot$ or nontermination. This may be represented by the set constant $\bot$, or by a proof that normal order reduction sequences will not successfully terminate. The calculus is also applicable for detecting more general forms of strictness. For example strictness in the *ith* argument of an abstraction f can be detected by feeding $(f\ \top\ \cdots \top\ \bot\ \top\ \cdots \top)$ into the analyzer. By providing other set constants apart from $\top$ and $\bot$, even more complicated analyzes are possible like a test for tail-strictness, or strictness under certain conditions.

Reduction of expressions ($\texttt{case}_T\ \top\ \ldots$) will require a case analysis, which is in (Nöcker 1993) as a propagation of unions, whereas our calculus uses the equivalent method of generating a directed graph, in which the union of cases is represented by forking.

Definition 3.26 (*SAL*)
The data structure for the algorithm SAL is a directed graph, where the nodes are labeled by abstract terms that are simplified. The edges may carry specific labels. The algorithm *SAL* starts with a directed graph consisting only of one node labeled with the simplified initial abstract term.

Given a directed graph D, a new directed graph D' is constructed by using some rule from definition 3.27 below. For every node added, we assume that the simplification rules (i.e. (lwas), (llet), (gc), (cpax)) and the bot-reduction rules (see Figure 4 and Definition 3.13) have been applied exhaustively.

The algorithm stops successfully, if all leaves in the graph are labeled with $\bot$, i.e. if every non-$\bot$ node has an outgoing edge.

Note that the rules only generate cycles in the graph with at least one edge in the cycle being labeled.

The labels at edges in the directed graph indicate that the reductions lengths of the concretizations along this edge can be strictly decreased.

Definition 3.27 (*Rules of SAL*)
The nondeterministic construction rules of SAL are given. The subsume-rules only add edges to the graph, whereas all other rules focus a given leaf L, add one or more new leaves $L_1, \ldots, L_n$, and add edges from L to every new leaf. According to the conditions given in the rules, a label may be added to the new edges.

nred Let a leaf L be labeled with t, and let $t \xrightarrow{a} t'$ be an abstract sp-reduction. Let $t'' := \text{simp}(t')$. Then generate a new node L' labeled with t'' and add a directed edge from L to L'. If a is (case), (lbeta), (seq), or (seq-Fun), then the edge is to be labeled with the kind of reduction.

ired Let a leaf L be labeled with t and let $t \xrightarrow{a} t'$ by a reduction that is not an abstract sp-reduction, which may be a (case), (seq), (lbeta), (cp), or (seq-Fun). Let $t'' := \text{simp}(t')$. Then generate a new node L' labeled with t'' and add an unlabeled directed edge from L to L'.

scsplit Let a leaf L be labeled with t, and assume t has an occurrence of a proper set constant u. Let the defining equation for the set constant u have the right hand side $\{\bot\} \cup r_1 \cup \ldots \cup r_k$. Then generate new expressions $t_j, j = 0, \ldots, k$ from t, as follows.

- For $j = 0$, let t_0 be generated from t by replacing u by $\bot$ in the ac-environment.
- If $r_j = \text{Fun}$, then t_j is generated by replacing u by Fun in the ac-environment.
- If $r_j = (c\ u_{j,1}\ \ldots\ u_{j,\text{ar}(c)})$, then let t_j be the expression generated from t by replacing u by $(c\ x_1 \ldots x_{\text{ar}(c_j)})$, where $x_i, i = 1, \ldots, \text{ar}(c_j)$ are new variables. For $x_i, i = 1, \ldots, \text{ar}(c)$ add $x_i = u_{j,i}$ to the ac-environment.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

For $j = 0, \dots, k$ first simplify t_j, i.e. let $t'_j := \text{simp}(t_j)$ and then construct new nodes L_j with label t'_j and add directed, unlabeled edges from L to L_j for all j.

subsume Let L be a leaf with term label $t_1 \neq \bot$, and let $N \neq L$ be a node with term label t_2. For the purposes of the subsume-rules let $\xleftrightarrow{\text{ucp,simp,*}}$ be a transformation consisting of, perhaps several, (ucp)-transformations, simplifications, or their reverses. If there are abstract terms t'_1, t'_2 with $t_1 \xleftrightarrow{\text{ucp,simp,*}} t'_1$, $t_2 \xleftrightarrow{\text{ucp,simp,*}} t'_2$ such that $t'_1 \subseteq_\gamma t'_2$ holds, then add a directed unlabeled edge from L to N under the following condition: After completion of this operation, the graph does not contain a cycle of unlabeled directed edges.

subsume2 Let L be a leaf with term label t_1 where $\bot \neq t_1$, and let N be a node with $N \neq L$ with term label t_2. Let there be abstract terms t'_1, t''_1, t'_2 with $t'_1 \subseteq_\gamma t'_2$, such that the following conditions hold:

1. $t_1 \xleftrightarrow{\text{ucp,simp,*}} t''_1 \equiv (\texttt{letrec}\ \text{Env}_1, x = t_x, x_1 = r_1\ \texttt{in}\ x_1)$, such that t_x is a strict and deep subterm in t''_1; this implies that the term r_1 is an application, a seq-expression or a case-expression; and
2. $(\texttt{letrec}\ \text{Env}, x = t_x\ \texttt{in}\ x) \xleftrightarrow{\text{ucp,simp,*}} t'_1$; and
3. $t_2 \xleftrightarrow{\text{ucp,simp,*}} t'_2 = (\texttt{letrec}\ \text{Env}_2, x = t_x\ \texttt{in}\ x)$.

Then add a directed edge from L to N labeled with (subsume2).

generalizeFun Let a leaf L be labeled with t, and let $t \equiv S[(x\ r)]$, where S is a surface context, and x is bound to Fun. Then apply the rule generalize such that $t' = S[\text{top}]$, where top is an ac-variable bound to $\top$.

generalize Given a leaf L with label t, construct a new term t' as follows: add a binding top $= \top$ to the top letrec-environment, where top is a new ac-variable. Select a subterm of t on a surface position and in the upper part of t and replace this subterm by the variable top. Add an unlabeled directed edge from L to the node L' labeled with t'.

Note that a subsume-edge may end in any node. It is not necessary that it is a predecessor of the leaf. Note also that in order to make $\subseteq_\gamma$ effective in the rule subsume, the criterion in Lemma 3.22 can be applied. The subsume rules can be made effective by bounding the number of applications of (ucp).

The usual strategy for the construction of the directed graph is to apply the following rules ordered by their priority.

- subsume-rules.
- nred, i.e. abstract sp-reduction.
- scsplit only if for the splitted set constant u, the corresponding variable x from the binding $x = u$ has a strict occurrences as a subterm in t.
- generalizeFun if the generalized application is a strict subterm.
- The other rules.

SAL has the following sources of nondeterminism.

- The different possibilities for applying reduction.
- The different possibilities for applying a split.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Correspondence Between Concrete and Abstract Terms

The following theorem may well be the central one in the correctness proof of SAL, though it is rather technical. It claims, given a terminating fixed concretization s of a node, that we will always find a directed edge to some successor node, such that this successor node has a concretization s' and the reduction measure does not increase, and if the edge is labeled, then we can find an s' with a strictly smaller measure.

Theorem 3.28
Let t be a closed abstract term, such that $s \in \gamma(t)$ and $s\Downarrow$.

1. (nred) Let $t \xrightarrow{a} t'$ be an abstract sp-reduction with $a \in \{$(case), (seq), (lbeta), (cp), (seq-Fun)$\}$. Then there is a term $s' \in \gamma(t')$, such that $s \xrightarrow{a} s'$. If $a \in \{$(case), (seq), (lbeta), (seq-Fun)$\}$ then $\mathrm{rl}\sharp\sharp(s) > \mathrm{rl}\sharp\sharp(s')$ and if $a = (cp)$ then $\mathrm{rl}\sharp\sharp(s) \geqslant \mathrm{rl}\sharp\sharp(s')$.
2. (ired) Let $t \xrightarrow{a} t'$, where $\xrightarrow{a}$ is a base-reduction, an extra reduction, or a simplification. Then there is some $s' \in \gamma(t')$ with $s \sim_c s'$ and $\mathrm{rl}\sharp\sharp(s) \geqslant \mathrm{rl}\sharp\sharp(s')$.
3. Let the rule (scsplit) be applied to the term t resulting in the sons $t_0, t_1, \dots, t_k$. Then there exists a $j \in \{0, 1, \dots, k\}$, and an $s' \in \gamma(t_j)$ such that $s \sim_c s'$, and $\mathrm{rl}\sharp\sharp(s) \geqslant \mathrm{rl}\sharp\sharp(s')$.
4. (generalize) If t' is the result of a generalization (generalize or generalizeFun) applied to t, then there is a concretization $s' \in \gamma(t')$ with $s' \sim_c s$ and $\mathrm{rl}\sharp\sharp(s) = \mathrm{rl}\sharp\sharp(s')$.

Proof

1. This follows from Proposition 3.24 and Lemma 3.23.
2. This follows from Lemma 3.23 and Proposition 3.18.
3. In the case of an scsplit, this follows from Lemma 3.25.
4. Let t' be the result of a generalization applied to t. Let $s \in \gamma(t)$ where $s = (\texttt{letrec}\ \mathrm{Env}, y = S[s_0]\ \texttt{in}\ s_1)$ and $S[]$ indicates the position of the generalization (the other case can be treated similarly). Then a reverse (ucp) yields $s' = (\texttt{letrec}\ \mathrm{Env}, y = S[z], z = s_0\ \texttt{in}\ s_1)$ where $s \sim_c s'$. Generalization means for t to have a binding $z = top, top = \top$ in t'. Since $\top$ is maximal by Lemma 3.8 we obtain $s' \in \gamma(t')$. From Theorem 2.14 we derive $\mathrm{rl}\sharp\sharp(s) = \mathrm{rl}\sharp\sharp(s')$. □

Proposition 3.29
Let (N_1, N_2) be an edge introduced by one of the subsume-rules. Let t_1 be the term at N_1 and t_2 be the term at N_2. Let $s_1 \in \gamma(t_1)$ with $s_1\Downarrow$. Then the following holds.

1. If the edge is generated by the rule (subsume), then there some $s_2 \in \gamma(t_2)$ with $s_2\Downarrow$ and $\mathrm{rl}\sharp\sharp(s_1) = \mathrm{rl}\sharp\sharp(s_2)$.
2. If the edge is generated by the rule (subsume2), then there is a concretization $s_2 \in \gamma(t_2)$ with $s_2\Downarrow$ and $\mathrm{rl}\sharp\sharp(s_2) < \mathrm{rl}\sharp\sharp(s_1)$.

Proof
Assume the rule (subsume) has been applied. There is some $s_1' \in \gamma(t_1')$, where $s_1' \xleftrightarrow{\mathrm{ucp,simp,*}} s_1$ and $t_1 \xleftrightarrow{\mathrm{ucp,simp,*}} t_1'$. There is some $t_2' \xleftrightarrow{\mathrm{ucp,simp,*}} t_2$, with $t_1' \subseteq_\gamma t_2'$, hence

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

$s'_1 \in \gamma(t'_2)$. There is some $s_2 \in \gamma(t_2)$ with $s'_1 \xleftrightarrow{\text{ucp,simp},*} s_2$. Theorem 2.14 implies that $\text{rl}\sharp\sharp(s_1) = \text{rl}\sharp\sharp(s'_1) = \text{rl}\sharp\sharp(s_2)$.

Assume, the rule (subsume2) has been applied. From $s_1 \in \gamma(t_1)$, $s_1\Downarrow$ and $t_1 \xleftrightarrow{\text{ucp,simp},*} t''_1$, we obtain a concretization $s''_1 \in \gamma(t''_1)$ with $s''_1\Downarrow$ and $\text{rl}\sharp\sharp(s''_1) = \text{rl}\sharp\sharp(s_1)$ applying Theorem 2.14. Proposition 2.19 implies that by omitting the binding $x_1 = r_1$ in s''_1, we obtain a concretization $s'_1 \in \gamma(t'_1)$ with $s'_1\Downarrow$ and $\text{rl}\sharp\sharp(s'_1) < \text{rl}\sharp\sharp(s''_1)$. From $s'_1 \in \gamma(t'_2)$, $t'_1 \sqsubseteq_\gamma t'_2$ and $t'_2 \xleftrightarrow{ucp,simp,*} t_2$, we obtain that there is a concretization $s_2 \xleftrightarrow{ucp,simp,*} s'_1$ with $s_2 \in \gamma(t_2)$ and $\text{rl}\sharp\sharp(s'_1) = \text{rl}\sharp\sharp(s_2)$. □

Corollary 3.30

Let N, N' be two nodes in a graph generated by SAL, such that (N, N') is an edge. Let t, t' be the corresponding abstract terms. Let $s \in \gamma(t)$ with $s\Downarrow$ be a concretization.

1. If the edge is labeled, then there exists $s' \in \gamma(t')$ with $\text{rl}\sharp\sharp(s) > \text{rl}\sharp\sharp(s')$.
2. If the edge is not labeled and was generated using nred, ired, generalizeFun or generalize, then there is a concretization $s' \in \gamma(t')$ with $\text{rl}\sharp\sharp(s) \geqslant \text{rl}\sharp\sharp(s')$.
3. Let the sons $N_0, N_1 \dots, N_k$ be generated using scsplit. Then there is a N_j with term label t_j, and a concretization $s' \in \gamma(t_j)$ with $s \sim_c s'$ and $\text{rl}\sharp\sharp(s) \geqslant \text{rl}\sharp\sharp(s')$.

Exploiting Already Known Strictness Information

If we already have strictness information available. e.g. after several successful runs of the strictness analyzer SAL and using Corollaries 3.33 and 3.34, the steps nred and subsume2 can be made more effective and applicable in more situations:

Suppose there is already a finite family of finite sets of concrete closed expressions (functions) $SF^{n,i}$ for $i, n \in \mathbb{N}$ with $1 \leqslant i \leqslant n$, such that every expression $f \in SF^{n,i}$ is known to be strict in its i^{th} argument for arity n. These functions are assumed to be defined via a binding $x = f$ in the top level `letrec`, where the variable x is in the upper part.

Then the detection of strict positions can additionally use the rule: if $f\ t_1 \dots t_{i-1}\ t_i\ t_{i+1} \dots t_n$ is a strict subexpression and f is strict in its ith argument for arity n, then also t_i is strict subexpression, which is formulated in the following lemma. Note that this does not directly follow from the results of SAL, since the terms t_i may be open terms.

Proposition 3.31

If f is strict in its ith argument for arity n, and $t_0 := (f\ t_1 \dots t_{i-1}\ t_i\ t_{i+1} \dots t_n)$ is a strict subexpression of t in a surface context, then t_i is also a strict subexpression of t.

Proof

Follows from Corollary 2.23. □

The algorithm SAL has several places where it can exploit these kinds of computations: For the detection of abstract sp-reductions, and for the subsume2-rule.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

3.4 Correctness of Strictness Detection

Main Theorems

Theorem 3.32
Let t be a closed abstract term. If t leads to successful termination using SAL, then $s \in \gamma(t) \Rightarrow s\Uparrow$, i.e. $s \sim_c \Omega$.

Proof
Assume that there is a closed concrete term $s \in \gamma(t)$ with $s\Downarrow$. Theorem 3.28 and Corollary 3.30 show that for every node N: if t_N at N has a concretization s_N with WHNF, then there is a direct successor node N' with abstract term $t_{N'}$, $s_{N'} \in \gamma(t_{N'})$ and $\mathrm{rl}\sharp\sharp(s_N) \geqslant \mathrm{rl}\sharp\sharp(s_{N'})$. If the edge is labeled, then we have $\mathrm{rl}\sharp\sharp(s_N) > \mathrm{rl}\sharp\sharp(s_{N'})$ by Corollary 3.30 and Proposition 3.29. It is not possible that s has a successor in a leaf labeled $\bot$. Among the nodes that have a terminating concretization we select a node $N_{\min}$ with term label $t_{N,\min}$ that has the minimal length $\mathrm{rl}\sharp\sharp(s_{N,\min})$ of all terminating concretization $s_{N,\min} \in \gamma(t_{N,\min})$. Since $s_{N,\min}\Downarrow$, there is an outgoing edge of $N_{\min}$ to a node $N_{\min,2}$. Minimality shows that the corresponding edge cannot be labeled. The same holds for $N_{\min,2}$, such that we find a path $N_{\min}, N_{\min,2}, N_{\min,3}, \dots$ of nodes connected with unlabeled edges. However, since the graph is finite, this enforces a cycle with unlabeled edges, which does not exist due to the construction. This is a contradiction, and we have thus shown that for all $s \in \gamma(t) : s\Uparrow$. □

Corollary 3.33
Let f be a closed expression. If $f\ \bot$ leads to successful termination using SAL, then f is strict in its argument.

Proof
The term $s := (f\ \Omega)$ is a concretization of $(f\ \bot)$. Theorem 3.32 implies that $(f\ \Omega) \sim_c \Omega$. Hence by the definition of strictness, f is strict in its argument. □

Corollary 3.34
Let f be a closed LR-expression. If the term

$$\begin{aligned} t := \ & \texttt{letrec}\ top_1 = \top, \dots, top_n = \top \\ & \quad \texttt{in}\ (f\ top_1\ \dots\ top_{i-1}\ \bot\ top_{i+1}\ \dots\ top_n) \end{aligned}$$

leads to successful termination using SAL, then f is strict in its ith argument for arity n.

Proof
The definition of strictness requires that for every closed expression $t_j, j = 1, \dots, n$: $f\ t_1\ \dots t_{i-1}\ \Omega\ t_{i+1} \dots t_n \sim_c \Omega$. We have

$$\begin{aligned} & f\ t_1\ \dots t_{i-1}\ \Omega\ t_{i+1} \dots t_n \\ \sim_c\ & \texttt{letrec}\ x_1 = t_1, \dots, x_{i-1} = t_{i-1}, x_{i+1} = t_{i+1}, \dots, x_n = t_n \\ & \quad \texttt{in}\ f\ x_1\ \dots x_{i-1}\ \Omega\ x_{i+1} \dots x_n\ , \\ =:\ & s. \end{aligned}$$

This follows using correctness of (ucp) (see Theorem 2.9). We also have $s \in \gamma(t)$ by Lemma 3.8. Theorem 3.32 implies that $s \sim_c \Omega$. By the definition of strictness, we obtain that f is strict in its ith argument for arity n. □

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Remark 3.35
Discussion: A difference between Nöcker's subsumption rule and the subsumption rule in SAL is that in Nöcker's algorithm, it is always assumed that different occurrences of set constants are independent. Translated into SAL this means that ac-variables occur at most once in the upper part. This means that Nöcker's subsumption rule appears to be stronger, since set constants in expressions can be subsumed, regardless of sharing. However, this can easily be simulated by SAL by applying the rule generalize for equal $\top$'s, and by adding a rule similar to generalize that makes all the occurrences of ac-variables distinct.

Interestingly, adding a linearized subsumption rule to SAL would make SAL incorrect, an example can easily be constructed on the basis of Example 3.43. Presumably, a linearized subsumption rule could also be used in SAL if the subsumption is only permitted for ancestors w.r.t. abstract reduction. The argument for justifying this rule is that the graph can be further expanded such that the linearization will occur in a deeper part of the graph.

Correctness of Strictness Optimization

We show that strictness optimization in LR is correct.

Definition 3.36 (*strictness evaluation*)
Let t be an LR term. A reduction sequence is a *strictness evaluation*, iff for every reduction step $t \to t'$ the following holds: $t = S[s]$, $t' = S[s']$, where S is a surface context, s is a strict subexpression of t, $s \to s'$ is a base reduction, and the inner redex of the reduction is also a strict subexpression of t in a surface context.

Theorem 3.37 (*correctness of strictness evaluation*)
Let t be a closed expression. Then the following holds.

1. If t has an evaluation, then every strictness evaluation terminates.
2. If some strictness evaluation of t terminates with a WHNF, then $t\Downarrow$.

Proof
Theorem 2.14 shows that base reductions do not increase the measure $\mathrm{rl}\sharp\sharp(\cdot)$ during reduction. Proposition 2.7 shows that there is no infinite sequence consisting only of (lll)-reductions. Proposition 2.15 shows that a reduction that is not an (lll)- and not a (cp)-reduction, and where the inner redex is a strict subexpression, strictly decreases $\mathrm{rl}\sharp\sharp(\cdot)$. The missing argument is that there is no infinite reduction that consists only of (cp)- and (lll)-reductions, where the inner redex is in a surface context. This holds, since every such (cp)-reduction strictly decreases the number of occurrences of variables that are in a surface context, and (lll)-reductions do not change this measure. That $t\Downarrow$ holds, if there is some strict evaluation of t, follows from Theorem 2.4. □

Corollary 3.38
Let $f_1, \ldots, f_n$ be expressions, such that for $j = 1, \ldots, n$ the expression f_j is strict in its i_j^{th} argument for arity n_j. Let t be an expression with a subexpression $(f_j\ t_1 \ldots\ t_{n_j})$, that is itself strict in the top term t. Then it is permitted to first locally evaluate the argument t_{i_j}. Applying this strategy throughout the reduction will terminate iff $t\Downarrow$.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Proof

The claim follows from Lemma D.3 since the strict subterms remain strict ones after a base reduction, and thus the strict evaluation can proceed locally evaluating the respective argument, and other strict arguments of strict subexpressions $f_j\ t_1 \dots\ t_{n_j}$ also remain strict in the top term. □

As a remark on the lengths of the evaluation and strict evaluation, there is no essential difference: the number of (case)- and (lbeta)- reduction is the same. The optimization effect of strictness optimizations shows up only at the level of an abstract machine.

Remark 3.39

Note that the bot-reduction rules will also be necessary in SAL in our formulation, even if SAL starts with a (polymorphically) well-typed term. There are two different types of uses. The first one means eliminating in an expression (case_Tx...) the untyped constructor instantiations for x, which is done immediately, and without loss of efficiency, if x is in a reduction context. The other use is that after several steps, the term at the nodes may be no longer well typed. This happens, e.g., since constructors are not typed in our formulation and that in $\top : \top$ the second $\top$ might be instantiated with a nonlist.

3.5 Examples

We demonstrate the algorithm SAL for some nontrivial functions and show the generated directed graphs. However, for readability, the graphs differ from the SAL graphs in the following respects. The top letrec environment is not shown, instead the ac-variables are written using set-variables directly. Not all reductions are shown, e.g. sometimes (cp)-reductions are not shown. Case distinctions for $\top$ in the rule scsplit are only depicted for the list-constructors.

Example 3.40

We show in figure 5 that the tail-recursive length function (lenr) is strict in its second argument:

$$
\begin{array}{lll}
\texttt{letrec len} = \lambda lst.\texttt{lenr}\ lst\ 0 & & \\
\qquad \texttt{lenr} = \lambda lst.\lambda s. & \texttt{case}_{lst}\ lst & \\
 & (\texttt{Nil} \rightarrow s) & \\
 & (x : xs \rightarrow (\texttt{letrec}\ z = 1 + s\ \texttt{in lenr}\ xs\ z)) & \\
\texttt{in}\dots . & &
\end{array}
$$

This can be shown by running SAL on the expression (letrec $top = \top$ in (lenr $top\ \bot$)), including the definition of the functions in the environment. In the first diagram of Figure 5, we write $\top$ for a variable that is bound to $\top$.

Example 3.41

We show that the function lenr is tail-strict in the first argument in Figure 5: For the definition of *Inf* and the properties see the definition in subsection 3.1 and

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

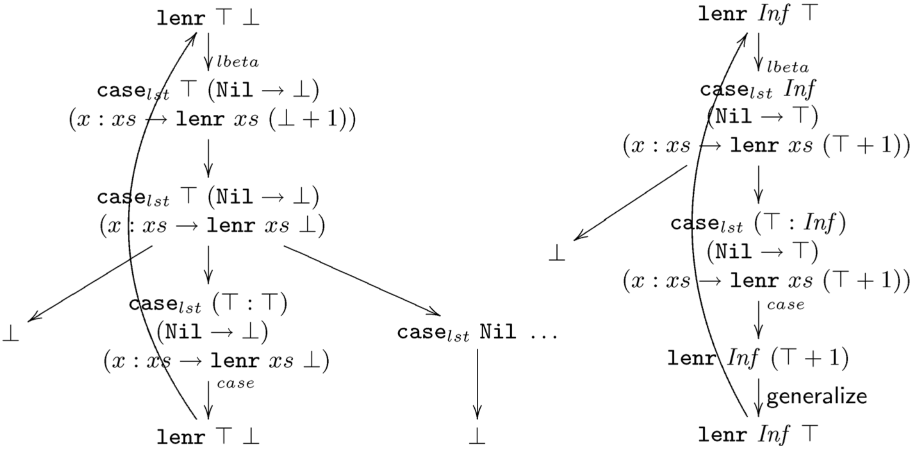


Fig. 5. SAL-graphs of Examples 3.40 and 3.41.

Lemma 3.8. Here we use the fact—which is, however, not proved in this paper—that a 2-ary function f is tail-strict in the first argument, if $(f\ \mathit{Inf}\ \top)$ has no terminating concretization.

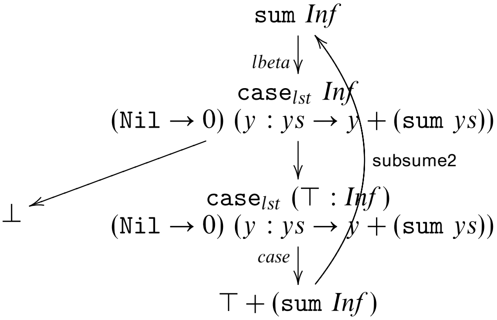
Example 3.42

We show that the function $\mathtt{sum}$ is tail-strict in its argument.

Let $\mathtt{sum}$ be defined in the environment by

$$\mathtt{sum} = \lambda xs.\mathtt{case}_{lst}\ xs\ (\mathtt{Nil} \to 0)\ (y : ys \to y + (\mathtt{sum}\ ys)).$$

The resulting graph is



Here the subterm criterion (subsume2) was used, and strictness of + in both arguments. Note that the simplification step in the subsume rule is used.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Example 3.43

We want to show that sharing of abstract constants is sometimes a (slight) improvement of SAL over Nöcker's method as described in (Nöcker 1993):

$$\begin{array}{lcl} f & = & \lambda x.\lambda z.g\ x\ x\ z \\ g & = & \lambda x.\lambda y.\lambda z.\ \texttt{if}\ x\ \texttt{then}\ (\texttt{if}\ y\ \texttt{then}\ z\ \texttt{else False}) \\ & & \qquad\qquad\quad \texttt{else}\ (\texttt{if}\ y\ \texttt{then False else}\ z). \end{array}$$

Checking whether f is strict in its second argument means to check ($\texttt{letrec}$ top $= \top, \ldots$ $\texttt{in}$ f top $\bot$).

Reducing this by (cp), (lbeta) yields ($\texttt{letrec}$ top $= \top, \ldots$ $\texttt{in}$ g top *top* $\bot$).

Now the effect is that the variable is the same, and $\top$ is not copied. The expression

$$\begin{array}{l} \texttt{if}\ \text{top}\ \texttt{then}\ (\texttt{if}\ \mathit{top}\ \texttt{then}\ \bot\ \texttt{else False}) \\ \qquad\qquad \texttt{else}\ (\texttt{if}\ \mathit{top}\ \texttt{then False else}\ \bot) \end{array}$$

yields for the $\texttt{True}$ case

$$\begin{array}{l} \texttt{if True then}\ (\texttt{if True then}\ \bot\ \texttt{else False}) \\ \qquad\qquad \texttt{else}\ (\texttt{if True then False else}\ \bot), \end{array}$$

which evaluates to $\bot$. The case $\texttt{False}$ yields also $\bot$. Hence SAL will detect strictness in the second argument of f.

As published, Nöcker's method copies the $\top$ and the information that it is the same variable is lost. Hence it is unable to detect this strictness.

Example 3.44

Further set constants from Nöcker Nöcker (1990) can also be used in the analysis, slightly adapted:

$$\begin{array}{lcl} \text{Topmem} & = & \{\bot\} \cup \texttt{Nil} \cup (\top : \text{Topmem}) \\ \text{Botelem} & = & \{\bot\} \cup (\top : \text{Botelem}) \cup (\bot : \text{Topmem}). \end{array}$$

As a complicated example, we show how SAL shows that ($\texttt{reverse}$ ($\texttt{concat}$ Botelem)) is nonterminating, which indicates that in the expression ($\texttt{reverse}$ ($\texttt{concat}$ xs)), the elements of the list xs can be evaluated first. The definitions are

$$\begin{array}{lcl} \texttt{reverse} & = & \lambda xs.\texttt{rev}\ xs\ \texttt{Nil} \\ \texttt{rev} & = & \lambda xs.\lambda st.\texttt{case}_{lst}\ xs\ (\texttt{Nil} \to st)\ (y : ys \to \texttt{rev}\ ys\ (y : st)) \\ \texttt{concat} & = & \lambda xs.\texttt{foldr}\ (\texttt{++})\ \texttt{Nil}\ xs \\ \texttt{foldr} & = & \lambda f.\lambda e.\lambda xs.\texttt{case}_{lst}\ xs\ (\texttt{Nil} \to e)\ (y : ys \to f\ y\ (\texttt{foldr}\ f\ e\ ys)). \end{array}$$

We assume that we already know that the append-operator (++) is strict in its first argument. The presentation below will not show the $\texttt{letrec}$-structure, and the (++)-reductions will be done in one step. In step 2, there is a generalization, which has to be guessed. Also, the application of $\texttt{rev}$ with second argument $\top$, will be immediately followed by a generalization to deshare the two $\top$s for the Stack. The standard strategy will not find a subsumption possibility, since at the stack position, the successive expressions are: $\texttt{Nil}, \top : \texttt{Nil}, \top : \top : \texttt{Nil}$. One

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

possibility of a directed graph successfully generated by SAL in linear notation is as follows:

1. $\texttt{reverse}\ (\texttt{concat}\ \mathrm{Botelem})$

2. $\texttt{rev}\ (\texttt{concat}\ \mathrm{Botelem})\ \texttt{Nil}$

$\boxed{\text{Here we add a generalization for the stack}}$

3. $\texttt{rev}\ (\texttt{concat}\ \mathrm{Botelem})\ \top$

4. $(\texttt{case}_{lst}\ (\texttt{concat}\ \mathrm{Botelem})\ (\texttt{Nil} \to \top)\ (y : ys \to (\texttt{rev}\ ys\ (y : \top)))$

5. $(\texttt{case}_{lst}\ (\texttt{foldr}\ (\texttt{++})\ \texttt{Nil}\ \mathrm{Botelem})$
$\quad (\texttt{Nil} \to \top)\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

6. $(\texttt{case}_{lst}\ (\texttt{case}_{lst}\ \mathrm{Botelem}$
$\qquad (\texttt{Nil} \to \texttt{Nil})\ (y : ys \to y\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ \texttt{Nil}\ ys)))$
$\quad (\texttt{Nil} \to \top)\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

$\boxed{\text{Now Botelem will be splitted:}}$

7.A $\mathrm{simp}((\texttt{case}_{lst}\ (\texttt{case}_{lst}\ \bot\ \ldots)\ \ldots)) = \bot$

7.B.1 $(\texttt{case}_{lst}\ (\texttt{case}_{lst}\ (\top : \mathrm{Botelem})\ (\texttt{Nil} \to \texttt{Nil})\ \ldots)$

7.B.2 $(\texttt{case}_{lst}\ (\top\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ \texttt{Nil}\ \mathrm{Botelem}))$
$\quad (\texttt{Nil} \to \top)\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

$\boxed{\top \text{ will be splitted into } \bot, \texttt{Nil}, \top : \top \text{ omitting untyped cases}}$

7.B.2.A $\mathrm{simp}(\texttt{case}_{lst}\ \bot\ \ldots) = \bot$

7.B.2.B $(\texttt{case}_{lst}\ (\texttt{foldr}\ (\texttt{++})\ \texttt{Nil}\ \mathrm{Botelem})$
$\quad (\texttt{Nil} \to \top)\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

$\boxed{\text{Subsume-Link back to 5.}}$

7.B.2.C.1 $(\texttt{case}_{lst}\ (\top : (\top\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ \texttt{Nil}\ \mathrm{Botelem})))$
$\quad (\texttt{Nil} \to \top)\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

7.B.2.C.2 $(\texttt{rev}\ (\top\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ \texttt{Nil}\ \mathrm{Botelem}))\ (\top : \top))$

7.B.2.C.3 $(\texttt{case}_{lst}\ (\top\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ \texttt{Nil}\ \mathrm{Botelem}))$
$\quad (\texttt{Nil} \to (\top : \top))\ (y : ys \to (\texttt{rev}\ ys\ (y : \top : \top))))$

$\boxed{\text{Two generalisation steps for } \top : \top}$

7.B.2.C.4 $(\texttt{case}_{lst}\ (\top\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ \texttt{Nil}\ \mathrm{Botelem}))$
$\quad (\texttt{Nil} \to \top)\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

$\boxed{\text{Subsume-Link to 7.B.2}}$

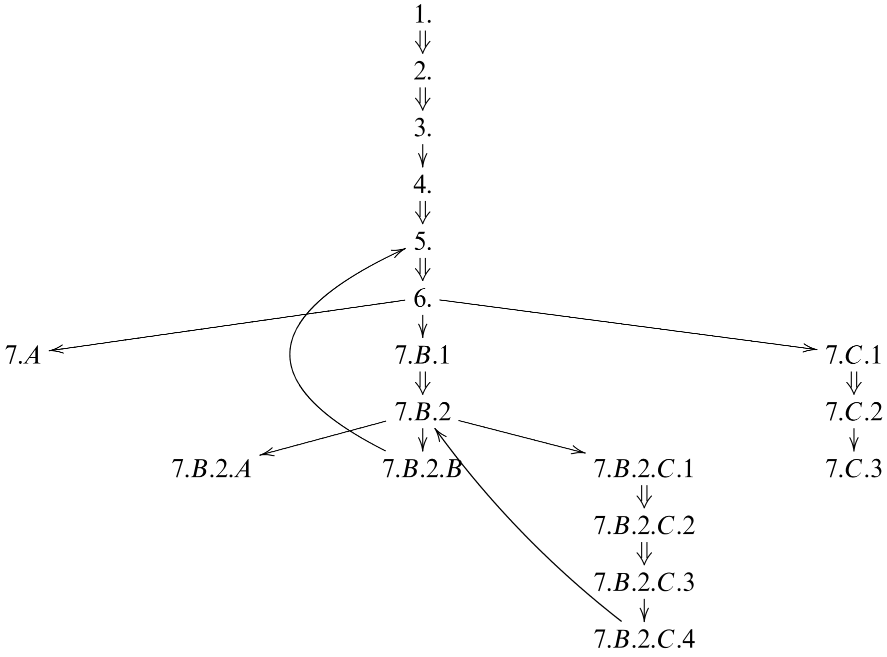
7.C.1 $(\texttt{case}_{lst}\ (\texttt{case}_{lst}\ (\bot : \mathrm{Topmem})$
$\qquad (\texttt{Nil} \to \texttt{Nil})\ (y : ys \to y\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ \texttt{Nil}\ ys)))$
$\quad (\texttt{Nil} \to \top)\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

7.C.2 $(\texttt{case}_{lst}\ (\bot\ \texttt{++}\ (\texttt{foldr}\ (\texttt{++})\ \texttt{Nil}\ \mathrm{Topmem}))$
$\quad (\texttt{Nil} \to \top)\ (y : ys \to (\texttt{rev}\ ys\ (y : \top))))$

7.C.3 $\boxed{\text{(since ++ is strict in its first argument)}}$
$\mathrm{simp}(\texttt{case}_{lst}\ \bot(\texttt{Nil} \to \top)\ (y : ys \to (\texttt{rev}\ ys\ (y : \top)))) = \bot$

The corresponding final graph looks as follows, where edges that contain labeled nred-steps are denoted with double arrows:

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press



4 Related Work

Strictness analysis has been approached from many different perspectives. These can roughly be characterized as based on abstract interpretation (e.g. Burn *et al.* (1985); Abramsky & Hankin (1987); Burn (1991); Cousot & Cousot (1977); Mycroft (1981); Wadler (1987)), projections (e.g. Wadler & Hughes (1987); Paterson (1996); Launchbury & Peyton Jones (1995)), nonstandard type systems (e.g. Kuo & Mishra (1989); Jensen (1998); Gasser *et al.* (1998); Coppo *et al.* (2002)) or abstract reduction (Nöcker 1992). For a detailed comparison of many of these approaches we refer to (Pape 1998, 2000).

Moran and Sands in (Moran & Sands 1999) developed the tool of improvement theory for the detailed analysis of reduction lengths in a lambda calculus with `letrec`, `case` and constructors. Unfortunately their methods and results could not be used here, since only certain essential normal order reduction steps are relevant and counting the number of `letrec`-shufflings is not appropriate for the proof of correctness of Nöcker's strictness analysis.

Contextual equivalence is also used to analyze equality and transformations in strict functional languages (Felleisen & Hieb 1992).

The methods in (Clark *et al.* 2000) (see also Ariola & Arvind (1995)) used a supercombinator calculus with `letrec` and a variant of modeling of case-constructor-primitives. The absence of abstractions allowed them to view expressions as graphs, and to arrive at a nice and easy-to-obtain confluence result for graph reduction. However, the papers of (Ariola & Klop 1997) and (Ariola & Blom 2002) provide examples showing that a lambda calculus using `letrec`, abstractions and

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

a copy reduction for expressions is in general not confluent, hence it is impossible to transfer the confluence result to our calculus. The paper (Ariola & Blom 2002) proposes to use a generalized confluence, called skew-confluence, but the relation of the equality defined by skew-confluence to contextual equivalence remains unclear, and skew-confluence appears to be weaker than our contextual equivalence.

4.1 Conclusion and Future Research

This paper defines SAL, a reconstruction of Nöcker's strictness analysis in a call-by-need lambda-calculus using `letrec`, `case`, and set constants equipped with a contextual semantics. It provides a correctness proof for all the essential steps of the algorithm including the loop-detection rules. It also provides a proof for correctness of strictness optimization following from the resulting strictness information w.r.t the contextual semantics. This is a foundation for using the strictness analysis in the lazy functional programming languages Haskell and Clean. We showed that rearranging evaluations or parallelizing them does not decrease the number of essential reductions, but improves determinacy of execution on an abstract machine. Our work also lays the foundation for potential refinements and extensions of the analysis and also for other analyzes.

Our proof is an improved and modified version of the proof in (Schmidt-Schauß *et al.* 2004), which used a nondeterministic lambda-calculus, where non-determinism was exploited for representing sets of expressions. However, the correctness proof in (Schmidt-Schauß *et al.* 2004) requires a conjecture on the relation between simulation and contextual preorder in this nondeterministic calculus. We believe that this conjecture holds, a support for this belief is a proof in (Mann 2004) that bisimulation implies contextual preorder for a nondeterministic lambda-calculus, which however uses a nonrecursive `let` and is constructor-free.

Extending the weakly typed lambda calculus by polymorphic types (e.g. as in (Pitts 2000)) and adapting and improving SAL, such that no untyped guesses have to be made and to show correctness of the adaptation, is left for future research. The gain of polymorphic typing would be that e.g. the treatment of Fun would be no longer necessary, as well as trying constructors for $\top$ in ($\texttt{case}_T\ \top\ \ldots$) which do not belong to the type T.

Acknowledgments

We thank the anonymous referees for their helpful remarks.

References

Abramsky, S. & Hankin, C. (1987) *Abstract Interpretation of Declarative Languages*. Chichester, UK: Ellis Horwood.

Ariola, Zena M. & Blom, S. (2002) Skew confluence and the lambda calculus with letrec. *Ann. Pure Appl. Logic*, **117**, 95–168.

Ariola, Zena M., & Klop, Jan Willem. (1997) Lambda calculus with explicit recursion. *Inf. Comput*., **139**(2), 154–233.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Ariola, Z. M., & Arvind. (1995) Properties of a first-order functional language with sharing. *Theor. Comput. Sci.*, **146**(3), 69–108.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M., & Wadler, P. (1995) A call-by-need lambda calculus. In *Principles of Programming Languages*. San Francisco, California: ACM Press, pp. 233–246.

Baader, Franz, & Nipkow, Tobias. (1998) *Term Rewriting and All That*. Cambridge: Cambridge University Press.

Barendregt, H. P. (1984) *The Lambda Calculus. Its Syntax and Semantics*. Amsterdam, New York: North-Holland.

Burn, G. L., Hankin, C. L., & Abramsky, S. (1985) The theory for strictness analysis for higher order functions. In *Programs as Data Structures*. Gazinger, H. & Jones, N. D. (eds) (*Lecture Notes in Computer Science*, vol. 217.) pp. 42–62. New York: Springer.

Burn, Geoffrey. (1991) *Lazy Functional Languages: Abstract Interpretation and Compilation*. London: Pitman.

Clark, David, Hankin, Chris, & Hunt, Sebastian. (2000) Safety of strictness analysis via term graph rewriting. *SAS 2000*, pp. 95–114.

Coppo, M., Damiani, F., & Giannini, P. (2002) Strictness, totality, and non-standard type inference. *Theor. Comput. sci.*, **272**(1-2), 69–112.

Cousot, Patrick, & Cousot, Radhia. (1977) Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*. New York: ACM Press.

Dezani-Ciancaglini, Mariangiola, Tiuryn, Jerzy, & Urzyczyn, Pawel. (1999) Discrimination by parallel observers: The algorithm. *Inf. Comput.*, **150**(2), 153–186.

Felleisen, Matthias, & Hieb, R. (1992) The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, **103**, 235–271.

Gasser, Kirsten Lackner Solberg, Nielson, Hanne Riis, & Nielson, Flemming. (1998) Strictness and totality analysis. *Sci. Comput. Prog.*, **31**(1), 113–145.

Gordon, Andrew D. (1994) A tutorial on coinduction and functional programming. In *Functional programming, Glasgow 1994*. (*Workshops in Computing*.) New York: Springer pp. 78–95.

Jensen, Thomas P. (1998) Inference of polymorphic and conditional strictness properties. In *Symposium on Principles of Programming Languages*. San Diego: ACM Press, pp. 209–221.

Kennaway, Richard, Klop, Jan Willem, Sleep, M. Ronan, & de Vries, Fer-Jan. (1993) Infinitary lambda calculi and böhm models. In *Proc. RTA 95*. LNCS, no. 914. Hisang, J. (ed), New York: Springer, pp. 257–270.

Kuo, Tsun-Ming, & Mishra, Prateek. (1989) Strictness analysis: A new perspective based on type inference. In *Functional Programming Languages and Computer Architecture*. San Diego: ACM Press, pp. 260–272.

Launchbury, John, & Peyton Jones, Simon. (1995) State in Haskell. *LISP Symb. Comput.*, **8**(2), 293–341.

Lévy, J.-J. (1976) An algebraic interpretation of the $\lambda\beta\kappa$-calculus and an application of a labelled λ-calculus. *Theor. Comput. Sci.*, **2**(1), 97–114.

Mann, Matthias. (2004) *Towards sharing in lazy computation systems*. Frank Report 18. Institut für Informatik, J. W. Goethe-Universität Frankfurt, Germany.

Moran, A. K. D., & Sands, D. (1999) Improvement in a lazy context: An operational theory for call-by-need. *POPL 1999*. New York: ACM Press, pp. 43–56.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Moran, Andrew K. D., Sands, David, & Carlsson, Magnus. (1999) Erratic fudgets: A semantic theory for an embedded coordination language. In *Coordination '99. Lecture Notes in Computer Science*, vol. 1594. New York: Springer, pp. 85–102.

Mycroft, Alan. (1981) *Abstract interpretation and optimising transformations for applicative programs*. Ph.D. thesis, University of Edinburgh.

Nöcker, E., Smetsers, J. E., van Eekelen, M., & Plasmeijer, M. J. (1991) Concurrent Clean. In *Proceedings of Parallel Architecture and Languages Europe (parle'91)*. New York: Springer Verlag, pp. 202–219.

Nöcker, Eric. (1990) *Strictness analysis using abstract reduction*. Technical Report 90-14. Department of Computer Science, University of Nijmegen.

Nöcker, Eric. (1992) *Strictness analysis by abstract reduction in orthogonal term rewriting systems*. Technical Report 92-31. University of Nijmegen, Department of Computer Science.

Nöcker, Eric. (1993) Strictness analysis using abstract reduction. In *Functional Programming Languages and Computer Architecture*. San Diego: ACM Press, pp. 255–265.

Pape, Dirk. (1998) Higher order demand propagation. In *Implementation of Functional Languages (IFL '98) London*. Hammond, K., Davie, A.J.T., & Clack, C. (eds), (*Lecture Notes in Computer Science*, vol. 1595.) New York: Springer. pp. 155–170.

Pape, Dirk. (2000) *Striktheitsanalysen funktionaler Sprachen*. Ph.D. thesis, Fachbereich Mathematik und Informatik, Freie Universität Berlin. (in German.)

Paterson, Ross. (1996) Compiling laziness using projections. In *Static Analysis Symposium*. (LNCS, vol. 1145). Aachen, Germany: Springer.

Peyton Jones, Simon. (2003) *Haskell 98 Language and Libraries*. Cambridge: Cambridge University Press. `www.haskell.org`.

Peyton Jones, Simon, & Marlow, Simon. (2002) Secrets of the Glasgow Haskell compiler inliner. *J. Funct. Prog.*, **12**(4&5), 393–434.

Peyton Jones, Simon L., & Santos, André. (1994) Compilation by transformation in the Glasgow Haskell Compiler. In *Functional programming, Glasgow 1994. Workshops in Computing*. New York: Springer, pp. 184–204.

Pitts, A. M. (1994) A coinduction principle for recursively defined domains. *Theor. Comput. Sci.*, **124**, 195–219.

Pitts, Andrew D. (2000) Parametric polymorphism and operational equivalence. *Math. Struct. Comput. Sci.*, **10**, 321–359.

Pitts, Andrew M. (2002) Operational semantics and program equivalence. *Applied Semantics*. (*Lecture Notes in Computer Science*, Vol. 2395. O'Donnell, J. T. (ed), New York: Springer pp. 378–412.

Plasmeijer, R., & van Eekelen, M. (2003) *The Concurrent Clean Language Report: Version 1.3 and 2.0*. Tech. rept. Dept. of Computer Science, University of Nijmegen. `http://www.cs.kun.nl/~clean/`.

Plotkin, Gordon D. (1975) Call-by-name, call-by-value, and the lambda-calculus. *Theor. Comput. Sci.*, **1**, 125–159.

Sands, D., Gustavsson, J., & Moran, A. (2002) Lambda calculi and linear speedups. *The Essence of Computation 2002*, New York, NY: Springer-Verlag, pp. 60–84.

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

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 am Main.

Schmidt-Schauß, M., Schütz, M. & Sabel, D. (2007) Appendix to "Safety of Nöcker's strictness analysis". *J. Funct. Programm.* Available at: http://www.cambridge.org/journals/JFP/.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press

Schmidt-Schauß, M., Panitz, S. E., & Schütz, M. (1995) Strictness analysis by abstract reduction using a tableau calculus. In *Proceedings of the Static Analysis Symposium*. Lecture Notes in Computer Science, no. 983. Springer-Verlag, pp. 348–365.

Schmidt-Schauß, M., Schütz, M., & Sabel, D. (2004) On the safety of Nöcker's strictness analysis. Tech. Rept. Frank-19. Institut für Informatik. J.W. Goethe-University.

Schmidt-Schauß, M., Schütz, M. & Sabel, D. (2005) Deciding subset relationship of co-inductively defined set constants. Tech. Rept. Frank-23. Institut für Informatik. J.W. Goethe-University. report revised in 2006; published as Schmidt-Schauß *et al.* (2007) in 2007.

Schmidt-Schauß, M., Sabel, D. & Schütz, M. (2007) Deciding inclusion of set constants over infinite nonstrict data structures. *Rairo-Theor. Inform. Appl.* **41**(2), 225–241.

Schütz, M. (1994) *Striktheits-Analyse mittels abstrakter Reduktion für den Sprachkern einer nicht-strikten funktionalen Programmiersprache*. Diploma thesis, Johann Wolfgang Goethe-Universität, Frankfurt.

Schütz, M. (2000) *Analysing Demand in Nonstrict Functional Programming Languages*. Dissertation, J.W. Goethe-Universität Frankfurt. Available at http://www.ki.informatik.uni-frankfurt.de/papers/marko.

van Eekelen, M., Goubault, E., Hankin, C. L. & Nöcker, E. (1993) Abstract reduction: Towards a theory via abstract interpretation. In *Term Graph Rewriting - Theory and Practice*, Sleep, M. R., Plasmeijer, M. J. & van Eekelen, M. C. J. D. (eds). Chichester: Wiley, chap. 9.

Wadler, P. (1987) Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In *Abstract interpretation of declarative languages*, Abramsky, S. & Hankin, C. (eds). Chichester: Ellis Horwood Limited, chap. 12.

Wadler, P. & Hughes, J. (1987) Projections for strictness analysis. In *Functional programming languages and computer architecture*. Lecture Notes in Computer Science, no. 274. Springer, pp. 385–407.

https://doi.org/10.1017/S0956796807006624 Published online by Cambridge University Press