Strictness Analysis by Abstract Reduction in Orthogonal Constructor Systems - Extended Abstract -

(submitted for publication)

Eric Nöcker
University of Nijmegen*

April 1993

Abstract

The strictness analyser in the Concurrent Clean system is based on a technique called abstract reduction. In this paper, we give a theoretical foundation of abstract reduction. As basic model we will use orthogonal constructor term rewriting systems. We prove the basic properties of abstract reduction, and of the approximation techniques that can be applied.

1. Introduction

Strictness analysis is an important technique in the compilation of lazy functional languages. Most strictness analysis techniques are based on abstract interpretation frameworks (e.g. Mycroft [1981], Burn *et al.* [1985], Wadler & Hughes [1987]). However, the strictness analyser in the Concurrent Clean System (Concurrent Clean (Nöcker *et al.* [1991]) is a lazy functional language based on term graph rewriting (Barendregt *et al.* [1987])) is based on a technique called abstract reduction (Nöcker [1993]). Informally, this technique can be considered as a kind of *abstract* term graph rewriting. In contrast to methods based on abstract interpretation, an infinite domain of values is used. Recursion is not handled with fixed point techniques, but with reduction path analysis. In Nöcker [1993] an informal description is given. In the latter paper it is also shown that the results obtained are very good. This holds both for simple kinds of strictness analysis and for cases where higher order functions or complex data structures are involved. A short introduction to abstract reduction is given in the next section.

Until now there was no satisfying theoretical foundation of this technique. A first start was given by Eekelen *et al.* [1991]. In this paper a theory was developed based on a basic form of abstract interpretation, which used a complete but still infinite variant of the original domain (Goubault & Hankin [1991]). The results, however, were rather limited. The main problems are caused by the complexity of the domain, by the different forms of possible sharing, and by the functional reduction strategy.

In this paper, we will develop a framework that avoids these problems. We will describe abstract reduction for orthogonal term rewriting systems. To some extent, orthogonal term rewriting systems can be used for modeling the behaviour of functional languages. To be able to discuss a usable form of strictness, we will define a special notion of head normal forms and of neededness. We will prove the basic properties of abstract reduction and some approximation techniques.

1.1. Abstract reduction and reduction path analysis

First, we will give an informal and short overview of abstract reduction. A detailed description can be found in Nöcker [1993].

*Department of Computer Science, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, tel +31.80.652509, fax +31.80.652525, e-mail eric@cs.kun.nl.

Let Σ be the set of symbols defined by a functional program. The set $\Sigma^{\#}$ of abstract symbols is defined as:

$$\Sigma^{\#} = \{F^{\#} \mid F \in \Sigma\} \cup \{\mathsf{Top}, \mathsf{Bot}, \mathsf{Union}\}$$

In the sequel, we will omit the #'s. The objects in the abstract reduction framework are *graphs* built from the set of abstract symbols. Such graphs can be considered as representations of sets of concrete graphs. As usual, Top represents all expressions, whereas Bot represents non-termination. The Union is used to combine sets. By using Unions and cycles, sets can become very complex. In particular, it is possible to define some special abstract values with which strictness analysis for lists can be done. For example, inf: Cons Top inf represents all infinite list, and topmem: Union (Cons Top topmem) Nil represents all lists (with the colon symbol sharing of (sub)graphs is expressed).

The reduction process simply mimics concrete reduction as precise as possible. A possible definition is that an abstract reduction step should contain all concrete reduction steps. Let $\rightarrow_\alpha$ be reduction in the abstract domain, and γ be a concretisation function that relates abstract graphs and the sets of concrete graphs they represent. For such a reduction step it should hold that:

$$t_1 \rightarrow_\alpha t_2 \Rightarrow \{g \mid h \rightarrow g, h \in \gamma(t_1)\} \subseteq \gamma(t_2)$$

This requirement, however, is somewhat cumbersome. It is such precise that abstract reductions would very quickly lead to huge abstract terms (to include all the information) or to bad approximations (loosing too much information). In this paper we will formulate a better requirement, based on a notion of needed reductions.

Abstract reduction can be implemented by using the original rewrite rules, that is, no conversion or abstraction of these rules is necessary. A typical property of an abstract reduction sequence is that it is not necessarily terminating. (for example, if it precisely models a concrete reduction sequence). However, by analysing abstract reduction sequences (*reduction path analysis*) it is often possible to optimise or approximate the reductions. Currently, we consider the following three possibilities:

- Top introduction can be used to enforce termination of possibly infinite abstract reduction sequences. It states that at any point during the analysis any term can be replaced by Top. This is the 'worst case' approximation.
- Bottom introduction is used for handling recursive behaviour in reduction sequences. It states that if a value t reduces to a value in which t itself is contained on a needed position, t can equally well be replaced by Bot.
- Cycle introduction is also used for handling recursive behaviour in reduction sequences. It states that if a value t reduces to a value in which t itself is contained, t can equally well be replaced by a cyclic value. Thus, the difference with Bottom introduction is that the neededness condition is not fulfilled.

By way of illustration, we will give some simple examples of abstract reduction steps. Consider the following functions:

```
F (x)   → F (x)
G (x)   → A (G (x))
```

Then, we might have the following abstract reductions (suppose A is a constructor):

F (Top)	$\rightarrow_\alpha$ F (Top) $\rightarrow_\alpha$ Bot	(ordinary reduction, followed by Bot introduction)
F (Top)	$\rightarrow_\alpha$ Top	(Top introduction)
G (Top)	$\rightarrow_\alpha$ A (G (Top)) $\rightarrow_\alpha$ x: A (x)	(ordinary reduction, followed by cycle introduction)

1.2. Overview

The remainder of this paper is organised as follows. In section 2 we define the notions of constructor head normal form and $\perp$-reduction. They will be used for defining what strictness means. Also, these notions are related to a specific form of neededness. Section 3 treats approximations, with which we will describe abstract reduction itself. We prove some basic properties of abstract reduction and correctness of Bottom introduction. Section 4 gives a more operational interpretation of approximations. The technique of how to prove correctness of cycle introduction is summarised in section 5. It uses notions from the theory of transfinite reductions. Finally, section 6 concludes. We note that, because of space limitations, (almost) no proofs are given in this paper. A full formal treatment can be found in Nöcker [1992].

Basic Notions

We briefly introduce some basic notions and notations used in the remainder of this paper. We assume the reader is familiar with orthogonal term rewriting systems and the underlying theory (Klop [1992]).

A *Term Rewriting System* is a pair (Σ, R) of an *alphabet* Σ and a set of *rewrite rules* R. The alphabet Σ consists of a countable infinite set of variables $(x, y, z \ldots)$ and a non-empty set of *function symbols* $F, G \ldots$ each equipped with a fixed finite arity $n\ (\geq 0)$, being the number of arguments it can have. The set of terms $T(\Sigma)$ is built in the usual way. For a non-variable term $t = F(t_1, \ldots, t_n)$, F is called the *root symbol*, also written as $root(t)$. We write $t = s$ if t and s are syntactically equal. A *context* containing n *holes* is written as $C[\,,\ldots,\,]$. A term s is a *subterm* of a term t if $t = C[s]$, and a *proper* subterm if $s \not\equiv t$. If a term s appears k $(k \geq 0)$ times in a term t, we will also write $t = C[s^k]$. Subterms can be identified more precisely by *occurrences*. An occurrence is a sequence of positive integer, denoted by $u_1.u_2 \ldots u_n$, and where $\bullet$ is the empty sequence. For a term t, the subterm t/u is defined as follows:

$$\begin{array}{lcll} t/\bullet & = & t & \\ F(t_1, \ldots, t_n)/u_1.u & = & t_{u_1}/u & \text{if } 1 \leq u_1 \leq n \\ & = & \text{undefined} & \text{otherwise} \end{array}$$

The set of valid occurrences for a term t is denoted by $O(t)$. The *depth* of a subterm t/u in t is the length of the sequence u.

A *rewrite rule* is a pair (l, r) of terms with $l, r \in T(\Sigma)$, such that the left-hand side l is not a single variable and the variables of r are contained in l. A *function* F is the set of rewrite rules for which $root(l) = F$. We assume functions to be finite. A *substitution* σ is a map from $T(\Sigma)$ to $T(\Sigma)$ satisfying $\sigma(F(t_1, \ldots, t_n)) = F(\sigma(t_1), \ldots, \sigma(t_n))$. The set of rewrite rules R defines a reduction relation $\rightarrow_r$ on $T(\Sigma)$ as follows: $t \rightarrow_r s$ if there exist a rewrite rule (l, r), a substitution σ and a context $C[]$ such that $t = C[\sigma(l)]$ and $s = C[\sigma(r)]$. The term $\sigma(l)$ is called a *redex*.

The transitive reflexive closure of a reduction relation $\rightarrow$ is $\twoheadrightarrow$, the transitive closure $\rightarrow^+$. A sequence of exactly k reduction steps $a_0 \rightarrow a_1 \rightarrow \ldots \rightarrow a_k$ where $k > 0$, is denoted as $\rightarrow^k$. The union of two reduction relations is $\rightarrow_\alpha \cup \rightarrow_\beta$, or more shortly $\rightarrow_{\alpha\beta}$.

An important class of term rewriting systems are *orthogonal* term rewriting systems. They have many nice properties, of which confluence is the most important one. Two other important properties we will need, is that descendants of a redex (also called residuals) are redexes, and that a redex cannot contribute to another redex (informally, we say that a subterm *contributes* to a redex if it is matched against a non-variable part of the left-hand side of the rewrite rule). We refer to Klop [1992] for a more detailed discussion of these notions. From now on we will only consider orthogonal term rewrite systems.

2. Strictness, $\perp$-reduction and neededness

Clearly, a term t is in normal form if it contains no redexes. In term rewriting theory, properties related to normal forms are very important. However, in the context of functional languages and of strictness analysis, the notion of *head* normal forms is more useful.

2.1. DEFINITION. A term $t \in T(\Sigma)$ is in *head normal form (hnf)* if there exists no redex $s \in T(\Sigma)$ such that $t \twoheadrightarrow_r s$.

Why are we interested in head normal forms? Most reduction strategies applied in functional languages are based on reduction to head normal form. Normal form reduction is obtained by repeated application of head normal form reduction. As a consequence, head normal form strictness is much more important than normal form strictness in implementations of functional languages.

The second reason arises by strictness analysis itself. Suppose we have some kind of P-normal form (*pnf*) property (e.g. the head normal form property). Now, the *intention* of strictness is the following. A function F is p-strict in its i-th argument if, for any application of F, the i-th argument (or a descendant) must be reduced to pnf in order to obtain a pnf form for that application. Let $\perp$ be a new and special symbol, representing terms having no pnf. If F is p-strict then $F(\ldots,\perp,\ldots)$ has no pnf. What we want is to relate this property to a reduction relation $\rightarrow$, being an extension of $\rightarrow_r$. That is, if $F(\ldots,\perp,\ldots) \twoheadrightarrow \perp$ (for all possible other arguments), then F should be p-strict. In particular, we want that $t \rightarrow \perp$ if t cannot reduce to a pnf. Such a reduction relation is difficult to define for the normal form case (it is not confluent). Consider, for example, the rewrite rule:

$$\mathsf{G}\ (\mathsf{A}\ (\mathsf{x})) \quad \rightarrow \quad 3$$

$\mathsf{A}\ (\perp)$ has no normal form, thus $\mathsf{A}\ (\perp) \rightarrow \perp$. The term $\mathsf{G}\ (\mathsf{A}\ (\perp))$ can reduce to two different terms having no common reduct.

For head normal forms such a reduction relation is easy to define. However, the head normal form property is also problematic. In orthogonal term rewriting systems it is in general undecidable whether a term is in head normal form or not. Consider for example the single left-hand side of a function F:

$$\mathsf{F}\ (1) \quad \rightarrow \quad \ldots$$

We would like F to be strict in its argument, but it is not (for any term t with a head normal form not equal to 1, $\mathsf{F}\ (\mathsf{t})$ is in head normal form). It will be clear that a function can be strict only in some very specific cases. This might be solved by 'completing' the rewrite system, that is, by requiring that every term with as root a function symbol will eventually become a redex (either by restricting the kinds of rewrite systems, or by restricting the set of valid terms by some form of typing). Then, the head normal form property becomes decidable. In most functional languages such a strategy is applied. We will solve this by introducing a special variant of head normal forms.

2.2. DEFINITION. Within an alphabet Σ we will distinguish two kinds of symbols: a symbol is either a *constructor* or a *function symbol*. Let Σ_F be the set of function symbols. All other symbols are in Σ_C, the set of constructor symbols. A term rewriting system (Σ, R) is a *constructor system* if for all proper subterms s, not being a single variable, of all left-hand sides of R: $root(s) \in \Sigma_C$. A term t is in *constructor head normal form (chnf)* if $root(t) \in \Sigma_C$. A term t *has* a constructor head normal form if $t \twoheadrightarrow_r t_c$ and t_c is in constructor head normal form. Clearly, constructor head normal form implies head normal form.

The notion of constructor head normal form is more usable. Firstly, it is decidable whether a term is in constructor head normal form. Secondly, this property is more close to what we expect. For example, the above function F becomes strict in its argument. A disadvantage is that this notion of strictness makes sense only for constructor systems. Consider the rule:

$$\mathsf{F}\ (\mathsf{F}\ (1)) \quad \rightarrow \quad 3$$

Since $\mathsf{F}(1)$ has no chnf, we would like that $\mathsf{F}\ (\mathsf{F}\ (1)) \rightarrow \mathsf{F}\ (\perp)$. However, the resulting term has no chnf, whereas $\mathsf{F}\ (\mathsf{F}\ (1))$ has a chnf. This problem is caused by the fact that a term without chnf ($\mathsf{F}(1)$ in this case) can contribute to a redex. Clearly, this is not the case for constructor systems. In general, we can probably state that p-strictness is meaningful only in rewrite systems where each term without pnf cannot contribute to a redex.

There are more ways to define head normal form like properties (and related rewrite systems). In the following, we are only interested in constructor head normal forms, and thus we will only consider orthogonal constructor systems. However, the results can also be obtained for the head normal form property (for general orthogonal rewrite systems). In the remainder of this section, we will formalize the notion of $\perp$-reduction, and prove some useful properties. We note that we will use names of notions that are usually defined in a normal form context (e.g. neededness, normalising). In our context their meaning will be slightly different.

2.3. DEFINITION ($\perp$-terms and $\perp$-reduction). (i) Let $\perp$ be a new symbol, not contained in Σ. The set $T(\Sigma \cup \{\perp\})$, also denoted as $T_\perp$, is called the set of $\perp$-terms. The reduction relation $\rightarrow_r$ is extended to $\perp$-terms in the obvious way.

(ii) The partial ordering $\leq$ on $T_\perp$ is defined inductively as follows:

$$\begin{array}{lll} \perp & \leq & t \qquad\qquad\qquad \text{for all } t \in T_\perp. \\ F(t_1,\dots,t_n) & \leq & F(s_1,\dots,s_n) \quad (n \geq 0) \text{ if } t_i \leq s_i \text{ for all } i = 1\dots n. \end{array}$$

(iii) $\perp$-reduction, denoted by $\rightarrow_\perp$, is defined on $T_\perp$ as $C[t] \rightarrow_\perp C[\perp]$ if t has no constructor head normal form with respect to $\rightarrow_r$.

In general, it is undecidable whether a term has a chnf. However, in some cases it is safe to perform $\perp$-reductions. For example, if a term t has no redex, and has as root symbol a function symbol, then $t \rightarrow_\perp \perp$. Some simple examples of $\perp$-reductions are:

Rewrite rules:			Reductions:		
F (2, x)	$\rightarrow$	G (x)	F (3,4)	$\rightarrow_\perp$	$\perp$
G (1)	$\rightarrow$	G (1)	G (1)	$\rightarrow_\perp$	$\perp$
G (2)	$\rightarrow$	2	$\perp$	$\rightarrow_\perp$	$\perp$

2.4. Proposition. *$\rightarrow_{\perp r}$ is confluent.*

Proof. It is easy to show that $\rightarrow_\perp$ is confluent, and that $\twoheadrightarrow_\perp$ and $\twoheadrightarrow_r$ are commutative. Since also $\rightarrow_r$ is confluent, it follows by application of the Hindley-Rosen lemma (see, for example, Barendregt [1984]) that $\rightarrow_{\perp r}$ is confluent. □

In the following, we will abbreviate $\rightarrow_{\perp r}$ as $\rightarrow$. We can now state when a function is strict in an argument.

2.5. Definition. A function F is *strict* in its i-th argument if for all terms $t_1 \dots t_{i-1}$, $t_{i+1} \dots t_n \in T$:

$$F(t_1,\dots,t_{i-1},\perp,t_{i+1},\dots,t_n) \twoheadrightarrow \perp$$

Because of the confluence of $\rightarrow$, strictness is a transitivity property. Thus, let t be a term having no chnf, and F be a function that is strict in its i-th argument. Then: $F(t_1,\dots,t_{i-1},t,t_{i+1},\dots,t_n) \twoheadrightarrow \perp$.

There is a natural relation between strictness and neededness. Since we are interested in constructor head normal forms we will consider a form of neededness related to the constructor head normal form case.

2.6. Definition. (i) A redex Δ in a term $t = C[\Delta]$ is a *needed redex* if in every reduction sequence that leads to a chnf Δ or one of its residuals is reduced.

(ii) A subterm s in a term $t = C[s]$ is a *needed subterm* iff $C[\perp] \rightarrow_\perp \perp$.

(iii) The reduction relation $\rightarrow_n$ is defined as follows: $t \rightarrow_n t'$ if $t \twoheadrightarrow t'$ and at least one needed reduction is done.

If a term has no chnf all its redexes are needed. It is not difficult to prove that there is a direct relation between needed redexes and $\perp$-reduction. Let $t = C[\Delta]$ with Δ a redex. Then: Δ needed $\Leftrightarrow C[\perp] \rightarrow_\perp \perp$.

For the normal form case, it is a well-known fact that reduction of needed redexes leads to the normal form (if existing), i.e. this reduction strategy is *normalising*. A similar property can be derived for the chnf case:

2.7. Theorem. (i) *Every term t, not in chnf, contains a needed redex.*

(ii) *Repeated contraction of needed redexes is leads to a chnf (if existing), even if diluted with arbitrary reduction steps (i.e. is (hyper)normalising).*

Because of theorem 2.7, $\rightarrow_n$ is normalising (but not necessarily confluent: it might be possible to reduce a term to two different chnf's). We note that $\rightarrow_n^+ = \rightarrow_n$, that is, $\rightarrow_n$ is transitive. Our notion of neededness has a lot of nice properties.

2.8. Proposition (Some useful properties of needed subterms). *Let $t \in T_\perp$ be a term having a chnf.*

(i) *Each needed subterm of t, not being in chnf, contains a needed redex. The subterm has a chnf.*

(ii) *Reduction of needed redexes within a needed subterm of t leads to the chnf of the subterm, and, by repeated application, of t itself.*

(iii) *Each needed subterm of t, not being in chnf, has a descendant that is a needed redex.*
(iv) *A non-needed subterm cannot contain a needed subterm.*
(v) *Let $t = C[s]$ such that s is a non-needed subterm. Then $C[\perp]$ has a chnf.*
(vi) *t can be written as $t = C[s_1, \ldots, s_n]$ such that all subterms s_i are non-needed, and all subterms in C are needed. The term $C[\perp^n]$ has a chnf.*

Proposition 2.8.i and 2.8.ii state that if a needed redex (or subterm) has been found, this subterm can safely be reduced to chnf by looking for needed redexes (and subterms) within the subterm itself. Proposition 2.8.iv-vi state that we can remove non-needed parts of a term without influencing the chnf property of that term. The needed parts of a term are of special interest:

2.9. Definition. Let $t = C[s_1, \ldots, s_n]$ such that all subterms s_i are non-needed, and all subterms in C are needed. The *needed part* of t is $\mathcal{N}(t) = C[\perp^n]$.

3. Approximations

In this section we will introduce the notion of *approximations*. These approximations will be used for describing the basic machinery of the abstract reduction mechanism. To some extent, terms defined by an approximation (*abstract* terms) can be considered as representations of sets of elements of $T_\perp$ (*concrete* terms).

3.1. Definition (Approximations). An *approximation* $\mathcal{A}$ of a TRS (Σ, R) is a triple $\langle T_\alpha, \leq_\alpha, \rightarrow_\alpha \rangle$ containing:

(i) A set of terms $T_\alpha = T(\Sigma \cup \{\perp, \top\} \cup U)$ where $U = \{U_i \mid i \geq 1\}$. We suppose that the new symbols are not contained in Σ. A symbol U_i, called a *union symbol*, has arity i. For convenience, we will write $\langle t_1, \ldots, t_n \rangle$ instead of $U_n(t_1, \ldots, t_n)$.
(ii) A preordering $\leq_\alpha$ defined as follows:

$$
\begin{array}{lll l}
\perp & \leq_\alpha & s & \text{for all } s \in T_\alpha. \\
t & \leq_\alpha & \top & \text{for all } t \in T_\alpha. \\
F(t_1, \ldots, t_n) & \leq_\alpha & F(s_1, \ldots, s_n) & (n \geq 0) \text{ if } t_i \leq_\alpha s_i \text{ for all } i = 1 \ldots n. \\
\langle t_1, \ldots, t_n \rangle & \leq_\alpha & s & (n \geq 1) \text{ if } t_i \leq_\alpha s \text{ for all } i = 1 \ldots n. \\
t & \leq_\alpha & \langle s_1, \ldots, s_n \rangle & (n \geq 1) \text{ if } t \leq_\alpha s_i \text{ for some } i = 1 \ldots n.
\end{array}
$$

(iii) An abstract reduction relation $\rightarrow_\alpha$ defined as follows (let $t_\alpha, t'_\alpha \in T_\alpha$ and $t, t' \in T_\perp$):

$$t_\alpha \rightarrow_\alpha t'_\alpha \Leftrightarrow \forall t \leq_\alpha t_\alpha \, [\exists t' \leq_\alpha t'_\alpha \, [t \rightarrow_n t']]$$

that is, $t_\alpha \rightarrow_\alpha t'_\alpha$ if the reduction "simulates" at least some needed reductions. A term t_α for which there is no reduction $t_\alpha \rightarrow_\alpha t'_\alpha$ is in chnf.

Remark. Clearly, $\leq$ from $T_\perp$ is included in $\leq_\alpha$. The ordering induces an equivalence relation over T_α. We write $t \equiv s$ if $t \leq_\alpha s$ and $s \leq_\alpha t$. For any term t, let $t^\equiv$ be the equivalence class belonging to t. The ordering $\leq_\alpha$ is extended to equivalence classes as follows: $t^\equiv \preceq s^\equiv \Leftrightarrow \exists t \in t^\equiv \, [\exists s \in s^\equiv \, [t \leq_\alpha s]]$. In the sequel, we will consider terms of T_α as representatives of their equivalence class, and use $\leq_\alpha$ instead of $\preceq$. Also, we will omit the subscript α if no confusion can occur. We note that $(T_\alpha^\equiv, \preceq)$ is a lattice but *not* a complete lattice (see Goubault & Hankin [1991]).

3.2. Lemma. (i) *For all t not in chnf we have $t \rightarrow_\alpha \top$ (Top-introduction).*
(ii) $\rightarrow_\alpha^+ = \rightarrow_\alpha$.
(iii) *Let $t = C[s]$ and $t' = C[s']$ such that $s \rightarrow_\alpha s'$. If $t' \rightarrow_\alpha t''$ then $t \rightarrow_\alpha t''$.*

The last three properties of α-reduction are important. The first one states that we can introduce approximations to ensure termination. The 'transitivity' property of the reduction will appear to be useful for many other properties. The last states that we can safely do additional reductions in subterms if necessary. This implies that sharing of subterms is safe, which is important for an implementation of abstract reduction.

We can now express the strictness test in the abstract world:

3.3. Proposition (Strictness). *Let* $F(\top,\ldots,\top,\perp_i,\top,\ldots,\top) \twoheadrightarrow_\alpha \perp$ *(where* i *denotes the* i*-th argument of* F*). Then the function* F *is strict in its* i*-th argument.*

As in $T_\perp$ we can define a notion of neededness. Informally, a subterm of $t_\alpha \in T_\alpha$ is needed if all its corresponding subterms in all terms t_α represents are needed. To formalise the notion of corresponding subterms, we need a precise relationship between subterms of an abstract term and subterms of the concrete terms it represents. This is done by the notion of *embeddings*. An embedding acts as a kind of homomorphism. Let $t \in T_\perp$ such that $t \le t_\alpha$. An embedding maps the structure of t on t_α. Each subterm is mapped on a corresponding subterm, not being a union. Given a subterm of t_α, is not necessarily the case that t has a corresponding subterm. Therefore, we will consider a *set* of needed subterms of t_α to define neededness.

3.4. Definition. Let $\varphi\colon O(t) \to O(t_\alpha)$ be an occurrence mapping. It is an *embedding* if for all $(u_1 \ldots u_m) \in O(t)$:

(i) $root(t/u_1 \ldots u_m) \le root(t_\alpha/\varphi(u_1 \ldots u_m))$ (where $\perp \le F \le \top$ for any symbol F)
(ii) $\varphi(u_1 \ldots u_m) = \varphi(u_1 \ldots u_{m-1})$ if $root(t_\alpha/\varphi(u_1 \ldots u_{m-1})) = \top$, and otherwise $\varphi(u_1 \ldots u_m) = \varphi(u_1 \ldots u_{m-1}).u_m.v_1 \ldots v_n$ such that $t_\alpha/\varphi(u_1 \ldots u_{m-1}).u_m.v_1 \ldots v_i$ is a union node (for all $i < n$).
(iii) $\varphi(\bullet) = v_1 \ldots v_n$ such that $t_\alpha/v_1 \ldots v_i$ is a union node (for $i < n$).

3.5. Definition (Neededness). Let $S = \{s_1,\ldots,s_n\}$ be a non-empty set of disjoint subterms of a term t_α, such that $s_i = t_\alpha/u_i$ for $1 \le i \le n$. S is *needed* if for all $t \in T_\perp$ there is a $t' \in T_\perp$ $(t, t' \le t_\alpha)$ such that $t \le t'$ and such that there is a corresponding needed subterm $s = t'/u$, and $u_i = \varphi(u)$ for some $1 \le i \le n$, and any embedding $\varphi\colon O(t') \to O(t_\alpha)$.

Suppose, for example, a term $\mathsf{t}_\alpha = \langle$+ (s,1),- (s',2)$\rangle$, where + and - are the usual strict integer operations. Then {s,s'} is a set of needed subterms, but also {+ (s,1),- (s',2)} is such a set. Any concrete term represented by t_α contains a needed subterm, corresponding to one of the elements of the set. The t' in the definition of neededness is necessary because not every term has a subterm directly mapping on one of the elements of the set (e.g. $\perp$ and the set {s,s'}). In such a case, the subterm of t' corresponds with a $\perp$ subterm of t.

3.6. Lemma. *Let* $t_\alpha \in T_\alpha$. *Let* $\{s_1,\ldots,s_n\}$ *be a set of needed subterms in* t_α *such that* $t_\alpha = C[s_1,\ldots,s_n]$.

(i) *Neededness in* T_α *is directly related to neededness in* $T_\perp$. *That is, for all* $t \in T_\perp$ *such that* $t \le C[\perp^n]$ *it holds that* $t \twoheadrightarrow_\perp \perp$ *(that is:* $C[\perp^n] \twoheadrightarrow_\alpha \perp$*).*
(ii) *Let* $s_i \twoheadrightarrow_\alpha s_i'$ *for* $1 \le i \le n$. *Then* $t_\alpha \twoheadrightarrow_\alpha C[s_1',\ldots,s_n']$.

The second part of the lemma states that reduction of a set of needed subterms is an α-reduction step. This property is very useful in an implementation of abstract reduction. We are now ready for the proof of the correctness of $\perp$-introduction, as described in the introduction.

3.7. Proposition (Bottom introduction). *Let* $t_\alpha \twoheadrightarrow_\alpha t_\alpha'$ *such that* t_α' *has a needed set of subterms* $\{s_1 \ldots s_k\}$. *Furthermore, let* $s_i \le t_\alpha$ *for* $1 \le i \le k$. *Then* $t_\alpha \twoheadrightarrow_\alpha \perp$.

Proof. We can construct a reduction sequence of needed reductions, leading to $\perp$ or to infinity, as follows. First we note that t_α' contains no chnfs. Let $t \in T_\perp$ such that $t \le t_\alpha$. Then there is a reduction sequence such that $t \twoheadrightarrow_n t'$ and $t' \le t_\alpha'$. If $t' = \perp$, we are ready. Otherwise, t' contains a subterm $s = t'/u$ that is needed, and such that $s \le t_\alpha'/\varphi(u) = t_\alpha'/u_i \le t_\alpha$. Note that s is not in chnf. The needed reduction sequence can be continued by reducing s to chnf (proposition 2.8). Since $s \le t_\alpha$, we can repeat the argument for s. Now we have a needed reduction sequence, which terminates with $\perp$, or is infinite. Since $\to_n$ is hypernormalising t cannot have a chnf. □

To apply $\perp$-introductions we need knowledge of neededness. There are various way to find needed subterms, the easiest being the use of functions of which strictness information (already) is available. Furthermore, we know that the root of a term always is needed:

3.8. Corollary. *Let* $t_\alpha \twoheadrightarrow_\alpha t_\alpha'$ *such that* $t_\alpha' \le t_\alpha$. *Then* $t_\alpha \twoheadrightarrow_\alpha \perp$.

4. τ-Reduction

In the previous section, we described only some (be it important) properties of abstract reduction. Now, we will take a more operational viewpoint. We start by defining τ-reduction. It describes how a term might rewrite, based on the possible rewritings in the orthogonal term rewriting system itself. Combining all possible τ-reduction steps for a term gives us certain of α-reduction steps for that term. Given a term not in chnf, we can derive α-reduction steps, but we cannot necessarily determine all, or the most optimal α-reduction steps. This form of τ-reduction can form the basis of a concrete implementation of abstract reduction. For an example of this, we refer to Nöcker [1993].

4.1. Definition (τ-reduction). (i) A term $t \in T_\alpha$ not in chnf is a τ-redex if there is a term $t' \in T_\perp$ such that $t' \leq t$ and t' is a redex with respect to $\rightarrow$ (thus note that $\top$ is no τ-redex).

(ii) Let $t' = \sigma(l)$, $t' \leq t$ for some term l and a substitution σ. The substitution σ is *maximal* if there is no substitution σ' such that $t' < \sigma'(l) \leq t$.

(iii) The reduction relation $\rightarrow_\tau$ over T_α is defined as follows (consider a term $C[t]$ with t a τ-redex):

   (a) $C[t] \rightarrow_\tau C[\sigma(r)]$ if there is a rewrite rule (l, r) such that $\sigma(l) \leq t$ and σ maximal.

   (b) $C[t] \rightarrow_\tau C[\perp]$ otherwise

Intuitively spoken, a term is a τ-redex if it 'matches' a left-hand side. Matching is the same as for ordinary term rewriting, except for the term $\top$ which matches every pattern, and for Unions. A substitution should be maximal in order to simulate as many as possible reductions. Maximal substitutions can easily be found by pattern matching. Consider as an example the rewrite rule:

$$\mathsf{F}\ (\mathsf{A}\ (\mathsf{x}), 1, 1) \quad \rightarrow \quad \mathsf{B}\ (\mathsf{x})$$

The term $\mathsf{F}\ (\top, \top, 1)$ matches the left-hand side, and will rewrite to $\mathsf{B}\ (\top)$. That is, the variable x is bound to $\top$. Note that the subterms 1 of the left-hand side correspond with two different terms in the redex. This means that the redex cannot always be written as a direct substitution of the left-hand side (i.e. there is no σ such that σ $(\mathsf{F}\ (\mathsf{A}\ (\mathsf{x}), 1, 1)) = \mathsf{F}\ (\top, \top, 1))$. The term $\mathsf{F}\ (\langle \mathsf{A}\ (2), \mathsf{A}\ (3)\rangle, 1, 1)$ will rewrite to $\mathsf{B}\ (\langle 2, 3\rangle)$, that is, x is bound to $\langle 2, 3\rangle$.

A τ-reduction in a term simulates all possible reductions of all corresponding subterms in the concrete terms (with respect to one rule). If a τ-redex has no maximal substitution, it reduces to $\perp$. A term can have (finitely) many τ-redexes. A term can even be a τ-redex for different rewrite rules. To simulate the necessary needed reductions it is sufficient to reduce all τ-redexes, and to take the union of all the results. This form of α-reduction is expressed in the following proposition which is easy to prove:

4.2. Proposition. *Consider a term $t \in T_\alpha$. Let t have n different τ-redexes Δ_i, such that $t = C_i[\Delta_i]$ and $C_i[\Delta_i] \rightarrow_\tau t_i$ for $i = 1 \ldots n$. Then $t \rightarrow_\alpha \langle t_1, \ldots, t_n\rangle$.*

The reduction $\rightarrow_\alpha$ as constructed by proposition 4.2 can be used as the basis for an implementation. Termination can be assured by Top-introduction (lemma 3.2.i). However, this reduction system is not efficient, and it is very difficult to decide what parts of a term have to be reduced.

The efficiency can be improved in various ways. It is possible to apply the properties of α-reduction as derived in the previous section. For example, it is possible to use strictness information derived earlier (lemma 3.6.ii). Also $\perp$-introduction can be applied if possible. However, it is also possible to optimise the basic form of τ-reduction by simulating less reductions. For example, it is sufficient to simulate reductions of outermost needed redexes only. Such extensions can become rather complex. Here, we note that also specific properties of the functional language can be taken into account. For example, typing or the reduction strategy can introduce additional sources for optimisations. For a detailed discussion of this topic we refer to Nöcker [1993].

Consider for an example of abstract reductions and τ-reduction the following functions:

$$\begin{array}{lll} \mathsf{F}\ (\mathsf{x},0) & \rightarrow & \mathsf{x} \\ \mathsf{F}\ (\mathsf{x},1) & \rightarrow & \mathsf{F}\ (\mathsf{x}, \mathsf{G}\ (\mathsf{x})) \\ & & \\ \mathsf{G}\ (\mathsf{x}) & \rightarrow & \ldots \end{array}$$

(for any right-hand side of G). It is easy to show that F is strict in its second argument (since there are no τ-reductions for F $(\top,\bot)$). However, F is also strict in its first argument. This can be derived by analysing abstract reductions of F $(\bot,\top)$:

$$\begin{array}{lll} \mathsf{F}\ (\bot,\top) & \rightarrow_\alpha & \langle \bot, \mathsf{F}\ (\bot, \mathsf{G}\ (\bot))\rangle \quad (\tau\text{-reduction}) \\ & \equiv & \mathsf{F}\ (\bot, \mathsf{G}\ (\bot)) \\ & \rightarrow_\alpha & \bot \quad\quad\quad\quad\quad\quad\quad (\bot\text{-introduction}) \end{array}$$

5. μ-Approximations

The framework developed until know is not powerful enough to describe cycle-introductions. In this section we give a short summary of how the framework can be extended. For a full treatment we refer to Nöcker [1992]. Consider as an introductory example the following rewrite rule:

$$\mathsf{F}\ (\mathsf{x}) \quad \rightarrow \quad \mathsf{A}\ (\mathsf{F}\ (\mathsf{x}))$$

where A is a constructor. We have the following reduction: F $(\top) \rightarrow_\alpha$ A (F $(\top)$). However, we would *like* to have a reduction F $(\top) \rightarrow_\alpha \mu$x.A (x), where the resulting term is a representation for some infinite term, or a cyclic term. However, the basic property for α-reduction doesn't hold, since any term resulting from a concrete reduction of F (t), will contain the symbol F.

This can be solved by introducing *strongly converging reduction sequences* (Kennaway *et al.* [1990]) in the world of the concrete terms. Informally, a strongly converging reduction sequence is a possibly infinite reduction sequence in which the depth of the contracted redexes tends to infinity (if the sequence is infinite). A strongly converging reduction sequence can lead to an infinite term. An important property is that any strongly convergent reduction sequence of ordinal length α can be compressed into a reduction sequence of length $\leq \omega$. We refer to Kennaway *et al.* [1990] for a detailed treatment of this topic. Many important properties for orthogonal constructor systems and for $\bot$-reduction extend verbatim to infinite terms and strongly converging reduction sequences.

By allowing *cyclic* abstract terms we can simulate some concrete strongly converging reduction sequences. More specifically, we use the following concrete reduction relation: $t \rightarrow_n^{\leq\omega} t'$ if t reduces by a strongly converging reduction sequence to t' and a finite initial segment of the reduction sequence contains a needed reduction.

5.1. Definition. Let $\mathcal{A}$ be an approximation. We can extend the approximation further to $\mathcal{A}_\mu$ as follows. An approximation $\mathcal{A}_\mu$ is a triple $\langle T_\alpha, \leq_\alpha, \rightarrow_\alpha\rangle$ containing:

(i) A set of terms T_α. Terms for this set are built as usual, but we add the rule: if $t_\alpha \in T_\alpha$ such that this term contains exactly k (≥ 0) occurrences of the free variable x, thus $t_\alpha = C[x^k]$, then $\mu x.C[x^k] \in T_\alpha$. Within the new term, called a *cyclic* term, x is bound. The definition of occurrences is extended with an extra rule. Let u be an occurrence, then $\mu x.C[x^k]/u = C[(\mu x.C[x^k])^k]/u$.

(ii) An ordering relation $\leq_\alpha$. It is defined in two steps. First, define $\leq$ such that $t \leq t'$ is defined only if t contains no μ-terms. $\leq$ equals the previous $\leq_\alpha$ of definition 3.1 plus one new rule:

$$t \quad \leq \quad \mu x.C[x^k] \quad \text{if } t \leq C[(\mu x.C[x^k])^k]$$

Now we define $\leq_\alpha$ as:

$$t \leq_\alpha t' \quad \text{if for all } s \leq t \text{ it holds that } s \leq t'.$$

(iii) A reduction relation $\rightarrow_\alpha$. For $\rightarrow_\alpha$ it should hold that it simulates at least needed reductions, but takes account of converging infinite reductions. That is, let $t_\alpha \rightarrow_\alpha t'_\alpha$ and $t \in T_\bot$ such that $t \leq t_\alpha$. Then there is a $t' \in T_\bot^\infty$ such that $t' \leq t'_\alpha$ and $t \rightarrow_n^{\leq\omega} t'$.

Cyclic terms are somewhat peculiar. Let $t = \mu x.C[x^k]$. Then, we have $t \equiv \mu x.C[t^k]$. As follows from the definition of occurrences, a cyclic term has infinitely many valid occurrences. However, it is easy to see that a cyclic term only has a finite number of distinct subterms (two subterms might be syntactically unequal, but can be equivalent). This means that occurrence and subterm are no longer similar notions.

We note that the above notion of μ-terms is used more frequently for defining recursive structures. Usually, it is combined with a reduction relation $\mu x.C[x] \rightarrow C[\mu x.C[x]]$. However, we have *not* $\mu x.C[x^k] \rightarrow_\alpha C[(\mu x.C[x^k])^k]$. Instead, the recursive behaviour of μ is obtained via the $\leq_\alpha$ operator.

The properties described in section 3 also hold for extended approximations. We can now state cycle introduction:

5.2. PROPOSITION (Cycle introduction). *Let $t_\alpha \rightarrow_\alpha t'_\alpha$ such that t'_α has a non-empty set of disjoint subterms $\{s_1 \ldots s_k\}$ such that $s_i \leq t_\alpha$ for $1 \leq i \leq k$. Let $t'_\alpha = C_\alpha[s_1, \ldots, s_n]$. Then $t_\alpha \rightarrow_\alpha \mu x.C_\alpha[x^k]$ (where x is a fresh variable).*

PROOF. By construction of strongly converging reduction sequences, leading to terms contained in $\mu x.C_\alpha[x^k]$ (Nöcker [1992]). □

Cyclic terms and cycle-introduction can be used in various ways. An important application is that we can use cyclic terms to represent possibly infinite concrete terms. For example, μx.Cons ($\top$,x) can be used to represent all infinite lists. Such a term can also be created during abstract reductions by cycle-introduction (i.e. if a function creates an infinite list). Consider the following function:

Length (Cons (x,y))	$\rightarrow$	+ (1,Length (y))
Length (Nil)	$\rightarrow$	0

It is easy to show that Length is strict in its argument. However, it is also spine strict (the spine of the list can be reduced in advance). This can be derived by analysing reductions of Length (μx.Cons ($\top$,x)):

Length (μx.Cons ($\top$,x))	$\rightarrow_\alpha$	+ (1,Length (μx.Cons ($\top$,x)))	(τ-reduction)
	$\rightarrow_\alpha$	$\perp$	($\perp$-introduction)

6. Discussion

We have developed a new framework for analysing strictness in orthogonal constructor term rewriting systems. We have shown correctness of some important properties of abstract reduction and of some analysis techniques. In section 5 we discussed a basis for a possible implementation. A more detailed treatment of implementation aspects is beyond the scope of this paper; the usefulness of abstract reduction is extensively discussed in Nöcker [1993].

One might ask in which sense it differs from abstract interpretation frameworks. Also, it is interesting in what way the work can be extended to our original goal: strictness analysis in functional languages (and term graph rewrite systems).

6.1. Strictness analysis by abstract interpretation

Many theoretical results are achieved in the field of abstract interpretation (e.g. Mycroft [1981], Burn *et al.* [1985], Wadler & Hughes [1987], Sekar *et al.* [1990]). One of the most important differences between abstract reduction and abstract interpretation is the moment on which approximations are applied. In abstract interpretation approximations are necessary when functions are converted to their abstractions. In abstract reductions approximations might be necessary during the reduction process. As a consequence, it is easy to give examples where an abstract reduction strictness analyser gives better results than an abstract interpretation strictness analyser. The main reason is that is always possible (given an abstract interpretation) to force approximations that are not necessary for abstract reduction. For example:

Head (Cons (a,b))	$\rightarrow$	a
F (x)	$\rightarrow$	Head (Cons (x,Nil))

With abstract reduction, it is straightforward to show that F is strict in its argument (by reducing F ($\perp$)). Within the abstract interpretation context it is not usual to include an abstract value for Cons ($\perp$,Nil). On the other hand, it is also possible to give examples where abstract reduction finds less information.

For some part, the theory in this paper can be defined in terms of abstract interpretation. We can consider $T_\perp$ as concrete domain, and $T_\alpha^\equiv$ (or its μ-variant) as its abstraction. The two domains can be connected via a Galois connection (α, γ) defined as follows:

$$\begin{array}{llll} \alpha: \mathcal{P}T_\perp \rightarrow T_\alpha^\equiv & \text{where} & \alpha(S) & = \sqcup S \\ \gamma: T_\alpha^\equiv \rightarrow \mathcal{P}T_\perp & \text{where} & \gamma(t) & = \{s \in T_\perp \mid s \leq t\} \end{array}$$

where $\sqcup$ is the least upper bound operator. However, as we already mentioned, $T_\alpha^\equiv$ is not a complete lattice, and the least upper bound doesn't need to exist always. There are several ways to complete the lattice (Goubault & Hankin [1991]), but they decrease the power of the analysis.

However, it seems difficult to adopt the reduction mechanism as we have described. In standard abstract interpretations, concrete functions are converted to abstract ones. Of the abstract functions fixed points are determined. Both the conversion as the fixed point analysis are not possible for our technique. Of course, there are alternatives. The technique developed in Hankin [1991] for the abstract interpretation of term graph rewrite systems is based on an approximation of a collecting semantics. For concrete applications it seems to complicated. A related technique has been used in Eekelen *et al.* [1991]. It turned out that the obtained results are rather limited.

6.2. Term graph rewriting

In this paper, we have restricted ourselves to the world of terms. Originally, abstract reduction was described in the context of term graph rewriting, or more specifically, in the context of the language Concurrent Clean. If we consider Clean as a functional language (which to some extent is possible), we can describe the semantics by using a term rewriting model. Why then, are we interested in a term graph rewriting model? There are three reasons. First, in an implementation of abstract reduction, sharing can be used as an optimisation of term rewriting. This is similar to using graph rewriting as an implementation for term rewriting. Lemma 3.2.iii indicates that it is indeed possible to share computations. Of course, it is possible to represent the μ-terms as cycles.

The second point is that we can add some new properties by using sharing. Consider for example the rule:

$$\mathsf{F}\ (1,2) \rightarrow$$

Now consider the term $\mathsf{F}\ (\mathsf{x}{:}\top,\mathsf{x})$. That is, the arguments of F are shared. We can now consider this term either as representing all terms of the form $\mathsf{F}\ (\mathsf{x},\mathsf{y})$, or as representing all terms of the form $\mathsf{F}\ (\mathsf{x},\mathsf{x})$. In the second case the abstract term will not match the left-hand side of the rewrite rule. In other words, it is possible to keep track of all descendants of a certain term. This seems a nice optimisation, but it raises also some problems. For example, the term $\mu\mathsf{x}.\mathsf{Cons}\ (\top,\mathsf{x})$ no longer needs to represent all infinite lists (since all list elements are shared).

Lastly, it seems very well possible to develop a similar theory for term graph rewriting itself. Of course, the problems introduced by sharing and cycles have to be studied carefully. Also, the notion of transfinite reductions needs some reconsideration. In special the relation between infinite terms and cyclic graphs is of interest. This is certainly a topic for further research.

6.3. Functional languages

Most functional languages are similar to constructor systems, i.e. there is a clear distinction between constructor and function symbols. This is one of the aspects of the type system. Typing is used to reject all 'illegal' terms. This means, that, if we forget about curried applications for the moment, most head normal forms will be constructor head normal forms. If this is not the case (e.g. selecting the head from an empty list), a run-time error is generated. For such cases, the head normal form property corresponds with the constructor head normal form property.

Currying is easily expressible in orthogonal term rewriting systems by adding explicit rules for the 'apply' function. However, this function cannot be part of a constructor system (but yet of an orthogonal term rewriting system). Also, it is undecidable whether an application of apply is in head normal form. So, currying seems to be a problem for abstract reduction. However, in the case that the apply function is the only exception in the constructor system, we can make a decidable approximation of the head normal form property, a kind of weak head normal form (whnf) property (an apply term is in whnf if its first argument is in whnf or in chnf). Properties as derived for the chnf case, can also be derived for the whnf case. In general, we believe that abstract reduction is possible for any orthogonal term rewriting system for which a decidable (and useful) approximation of head normal forms can be given. Clearly, typing is a possible method.

Another important difference between functional languages and orthogonal term rewriting systems is the reduction strategy. The strategy determines what redex in a term is the next one going to be rewritten. In practice, a strategy determines evaluations of the arguments of a function application, if that application itself has to be reduced. In our framework there are several ways to take account of the reduction strategy. It would be nice if the starting point, orthogonal term rewriting systems, could be replaced by rewrite systems with a reduction strategy such that confluence is ensured. We believe that similar results can be obtained (but note that still some kind of head normal forms is needed). This is also a topic of further research. From an implementation viewpoint, a reduction strategy can be used to optimise abstract reduction. How this can be done for the functional strategy is described in Nöcker [1993].

Acknowledgements

Thanks are due to Sjaak Smetsers and Marko van Eekelen for some useful contributions, and for proof reading earlier versions of this paper.

7. References

Abramsky S., D. Gabbay and T. Maibaum, *Handbook of Logic in Computer Science*, volume I, Oxford University Press, 1992.

Barendregt H.P., *The Lambda Calculus: its syntax and semantics*, volume 103 of *Studies in Logic*, North-Holland, Amsterdam, 1984.

Barendregt H.P., M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep, Term Graph Reduction, in: *Proc. of Parallel Architectures and Languages Europe (PARLE)*, pages 141–158, Eindhoven, The Netherlands, Springer Verlag, LNCS 259 II, 1987.

Burn G.L., C.L. Hankin and S. Abramsky, The Theory of Strictness Analysis for Higher Order Functions, in: *Proc. of Workshop on Programs as Data Objects*, pages 42–62, DIKU, Denmark, Springer Verlag, LNCS 217, 1985.

Eekelen M.C.J.D. van, E. Goubault, C. Hankin and E. Nöcker, Abstract Reduction: Towards a Theory via Abstract Interpretation, in: *Proc. of Semagraph Symposium*, Nijmegen, The Netherlands, 1991, also in Sleep *et al.* [1993].

Goubault E. and C. Hankin, A Lattice for the Abstract Interpretation of Term Graph Rewrite Systems, in: *Proc. of Semagraph Symposium*, pages 263–273, Nijmegen, The Netherlands, 1991, also in Sleep *et al.* [1993].

Hankin C., Static Analysis of Term Graph Rewriting, in: *Proc. of Parallel Architectures and Languages Europe (PARLE'91)*, pages 367–385, Eindhoven, The Netherlands, Springer Verlag, LNCS 506, 1991.

Kennaway J.R., J.W. Klop, M.R. Sleep and F.J. de Vries, Transfinite Reductions in Orthogonal Term Rewriting Systems, Technical Report CS9041, CWI, 1990.

Klop J.W., *Term Rewriting Systems*, in: Abramsky *et al.* [1992].

Mycroft A., *Abstract interpretation and optimising transformations for applicative programs*, PhD thesis, University of Edinburgh, 1981.

Nöcker E., Strictness Analysis by Abstract Reduction in Orthogonal Term Rewriting Systems, Technical Report 92-31, University of Nijmegen, 1992.

Nöcker E., Strictness Analysis using Abstract Reduction, in: *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA'93)*, Kopenhagen, ACM Press, 1993.

Nöcker E., J.E.W. Smetsers, M.C.J.D. van Eekelen and M.J. Plasmeijer, Concurrent Clean, in: *Proc. of Parallel Architectures and Languages Europe (PARLE'91)*, pages 202–219, Eindhoven, The Netherlands, Springer Verlag, LNCS 505, 1991.

Sekar R.C., S. Pawagi and I.V. Ramarkrishnan, Small Domains Spell Fast Strictness Analysis, in: *Proc. of Symposium on Principles of Programming Languages (POPL)*, pages 169–183, San Francisco, USA, ACM Press, 1990.

Sleep M.R., M.J. Plasmeijer and M.C.J.D. van Eekelen, *Term Graph Rewriting, Theory and Practice*, John Wiley and Sons inc., 1993.

Wadler P. and R. J. M. Hughes, Projections for strictness analysis, in: *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*, pages 385–407, Portland, Oregon, USA, Springer Verlag, LNCS 274, 1987.