Lazy type inference for the strictness analysis of lists

Chris Hankin[1] and Daniel Le Métayer[2]

[1] Department of Computing, Imperial College, LONDON SW7 2BZ, UK
[2] INRIA/IRISA, Campus de Beaulieu, 35042 RENNES CEDEX, FRANCE

Abstract. We present a type inference system for the strictness analysis of lists and we show that it can be used as the basis for an efficient algorithm. The algorithm is as accurate as the usual abstract interpretation technique. One distinctive advantage of this approach is that it is not necessary to impose an abstract domain of a particular depth prior to the analysis: the lazy type algorithm will instead explore the part of a potentially infinite domain required to prove the strictness property.

1 Introduction

Simple strictness analysis returns information about the fact that the result of a function application is undefined when some of the arguments are undefined. This information can be used in a compiler for a lazy functional language because the argument of a strict function can be evaluated (up to weak head normal form) and passed by value. However a more sophisticated property might be useful in the presence of lists or other recursive data structures which are pervasive in functional programs. For example, consider the following program:

$$\begin{aligned}
&\mathit{sum}\ \mathbf{nil} &&= 0\\
&\mathit{sum}\ \mathbf{cons}(x, xs) &&= x + (\mathit{sum}\ xs)\\
&\mathit{append}\ \mathbf{nil}\ l &&= l\\
&\mathit{append}\ \mathbf{cons}(x, xs)\ l &&= \mathbf{cons}(x, (\mathit{append}\ xs\ l))\\
&H\ l_1\ l_2 &&= \mathit{sum}(\mathit{append}\ l_1\ l_2)
\end{aligned}$$

Rather than suspending the evaluation of each recursive call to *append* and returning the weak head normal form $\mathbf{cons}(x, (\mathit{append}\ xs\ l))$, we may want to compute directly the normal form of the argument to *sum* in H because the whole list will be needed. There have been a number of proposals to extend strictness analysis to recursively defined data structures [4, 21, 25, 26]. These have led to sophisticated analyses but two aspects of the problem have received little attention until recently: (1) the integration of the results of the analysis into a real compiler, (2) the efficiency of the algorithm implementing the analysis. The first issue has been tackled recently both from an experimental point of view [11, 16] and from a theoretical point of view [5, 8]. We are concerned with the second issue in this paper. The abstract interpretation and the projections approaches have led to the construction of analyses based on rich domains which make them intractable even for some simple examples. Techniques striving for a better representation of the domains do not really solve the problem [12, 17].

This observation has motivated some researchers, [2, 18, 19, 20], to develop non-standard type inference systems for strictness analysis. Kuo and Mishra, [20], proposed a type inference system for strictness information; they developed a sound and complete inference algorithm but did not show the correctness of the inference system (with respect to a standard semantics). In [21] it is shown how their type inference system can be extended to a form of full strictness for lists and to the 4-point domain of Wadler [25].

The other authors, [2, 18, 19], have developed sound and complete inference systems but have not given much attention to algorithms. In [13] we demonstrated a technique for deriving efficient static analysis algorithms from type inference systems. The basis for this work was Jensen's conjunctive strictness logic [18]; we used techniques similar to [14, 15] to refine the logic into an algorithm.

In this paper, we follow the approach taken in [13] to construct an efficient algorithm for the analysis of lists. The algorithm is both correct and complete with respect to the usual abstract interpretation approach. So it does not incur the loss of accuracy of previous type inference systems for the analysis of lists [21]. The core of the algorithm is the notion of lazy types (or lazily evaluated types) which allows us to compute only the information required to answer a particular question about the strictness of a function. One significant advantage of the approach is that it extends naturally to domains of any depth and domains are only explored at the particular depth required for the original question. In other words, we do not have to choose a particular domain before the analysis as is usually done for abstract interpretation (except when widening operators are used as in [7]).

We first describe an extension of Jensen's strictness logic [18] to include an analogue of Wadler's 4-point domain [25] (Section 2). In Section 3, we introduce the notion of lazy types for lists and we present the corresponding system. We state the correctness and completeness properties with respect to the original system and we proceed in Section 4 to define the lazy type inference algorithm. The algorithm can in fact be derived in the same way as in [13] and the correctness proof follows for the same reasons. Section 5 is an example of the functioning of the algorithm. We show in Section 6 that the type system and algorithm can be extended to domains of unbounded depth and we present an example showing that the algorithm naturally explores the depth of the domain required by a particular question. Related work is discussed in Section 7.

2 A strictness logic for the analysis of lists

We consider a strongly typed language, Λ_L, with terms defined by the following syntax:

$$\begin{aligned} e = {} & x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \mathbf{fix}(\lambda g.e) \mid \mathbf{cond}(e_1, e_2, e_3) \mid \\ & \mathbf{nil} \mid \mathbf{cons}(e_1, e_2) \mid \mathbf{hd}(e) \mid \mathbf{tl}(e) \mid \mathbf{case}(e_1, e_2, e_3) \end{aligned}$$

The **case** operator is used in the translation of pattern matching. For example, the *sum* function from the previous section is translated as:

$$sum(l) \;=\; \mathbf{case}(0, f, l) \qquad where \quad f\ x\ xs \;=\; x + (sum\ xs)$$

The loss af accuracy that occurs without the **case** operator is discussed in [25].

Abstract interpretation represents the strictness properties of a function by an abstract function defined on boolean domains [23]. For instance $g_{abs}\ \mathbf{t}\ \mathbf{f} \;=\; \mathbf{f}$ means that g is undefined if its second argument is undefined. In terms of types, this property is represented by $g : \mathbf{t} \to \mathbf{f} \to \mathbf{f}$. Notice that $\mathbf{t}$ and $\mathbf{f}$ are now (non-standard) types. Conjunctive types are required to retain the power of abstract interpretation: a strict function like $+$ must have type $(\mathbf{f} \to \mathbf{t} \to \mathbf{f}) \wedge (\mathbf{t} \to \mathbf{f} \to \mathbf{f})$. Let us now turn to the types used for the representation of properties of lists. As a first stage, we consider the extension of the boolean domain to Wadler's 4-point domain [25]. We show that this extension can be generalised to domains of unbounded depth in Section 6. The four elements of the domain are $\mathbf{f} \leq \infty \leq \mathbf{f}_\in \leq \mathbf{t}$ where ∞ represents infinite lists or lists ending with an undefined element and $\mathbf{f}_\in$ corresponds to finite lists whose elements may be undefined (plus the lists represented by ∞).

The ordering on types is described in Fig. 1. We define $=$ as the equivalence induced by the ordering on types: $\sigma = \tau \Leftrightarrow \sigma \leq \tau$ and $\tau \leq \sigma$. The type inference system is shown in Fig. 2. Γ is an environment mapping variables to formulae (i.e. strictness types). In the rule **Cond-1**, σ represents the standard type of e_2 (or e_3). This system is an extension of [13, 18] and the soundness and completeness proofs of the logic (with respect to traditional abstract interpretation) follow straightforwardly from [19]. As an illustration, we show how the property, $sum : \mathbf{f}_\in \to \mathbf{f}$, can be derived in this logic:

$$
\begin{array}{ll}
\mathbf{Conj} & \dfrac{A \qquad B}{[s:\mathbf{f}_\in \to \mathbf{f}, l:\mathbf{f}_\in] \vdash \lambda x.\lambda xs.x+(s\ xs) : \mathbf{t}\to\mathbf{f}_\in\to\mathbf{f} \wedge \mathbf{f}\to\mathbf{t}\to\mathbf{f}} \quad C \\
\mathbf{Case-3} & \dfrac{}{[s:\mathbf{f}_\in \to \mathbf{f}, l:\mathbf{f}_\in] \vdash \mathbf{case}(0, \lambda x.\lambda xs.x+(s\ xs), l) : \mathbf{f}} \\
 & \qquad\qquad\vdots \\
\mathbf{Abs} & \dfrac{}{\vdash (\lambda s.\lambda l.\mathbf{case}(0, \lambda x.\lambda xs.x+(s\ xs), l)) : (\mathbf{f}_\in \to \mathbf{f}) \to (\mathbf{f}_\in \to \mathbf{f})} \\
\mathbf{Fix} & \dfrac{}{\vdash \mathbf{fix}(\lambda s.\lambda l.\mathbf{case}(0, \lambda x.\lambda xs.x+(s\ xs), l)) : \mathbf{f}_\in \to \mathbf{f}} \\
 & \dfrac{}{\vdash sum : \mathbf{f}_\in \to \mathbf{f}}
\end{array}
$$

where A is:

$$
\begin{array}{ll}
 & \dfrac{\vdots}{[s:\mathbf{f}_\in \to \mathbf{f}, l:\mathbf{f}_\in, x:\mathbf{t}, xs:\mathbf{f}_\in] \vdash x+(s\ xs) : \mathbf{f}} \\
 & \qquad\qquad\vdots \\
\mathbf{Abs} & \dfrac{}{[s:\mathbf{f}_\in \to \mathbf{f}, l:\mathbf{f}_\in] \vdash \lambda x.\lambda xs.x+(s\ xs) : \mathbf{t} \to \mathbf{f}_\in \to \mathbf{f}}
\end{array}
$$

B is:

$$
\begin{array}{ll}
 & \dfrac{\vdots}{[s:\mathbf{f}_\in \to \mathbf{f}, l:\mathbf{f}_\in, x:\mathbf{f}, xs:\mathbf{t}] \vdash x+(s\ xs) : \mathbf{f}} \\
 & \qquad\qquad\vdots \\
\mathbf{Abs} & \dfrac{}{[s:\mathbf{f}_\in \to \mathbf{f}, l:\mathbf{f}_\in] \vdash \lambda x.\lambda xs.x+(s\ xs) : \mathbf{f} \to \mathbf{t} \to \mathbf{f}}
\end{array}
$$

and C is:

$$\mathbf{Var}\ [s : \mathbf{f}_{\in} \rightarrow \mathbf{f}, l : \mathbf{f}_{\in}] \vdash l : \mathbf{f}_{\in}$$

Note that A and B make use of the implicit assumption about the type of $+$. Any environment is supposed to contain all the types of primitive operators.

$$\mathbf{f} \leq \phi \qquad \phi \leq \phi \qquad \infty \leq \mathbf{f}_{\in} \qquad \phi \leq \mathbf{t} \qquad \mathbf{t}_{\sigma \rightarrow \tau} \leq \mathbf{t}_{\sigma} \rightarrow \mathbf{t}_{\tau}$$

$$\frac{\phi \leq \psi, \psi \leq \chi}{\phi \leq \chi} \qquad \frac{\phi \leq \psi_1, \phi \leq \psi_2}{\phi \leq \psi_1 \wedge \psi_2} \qquad \phi \wedge \psi \leq \phi \qquad \phi \wedge \psi \leq \psi$$

$$\phi \rightarrow \psi_1 \wedge \phi \rightarrow \psi_2 \leq \phi \rightarrow (\psi_1 \wedge \psi_2) \qquad \frac{\phi' \leq \phi, \psi \leq \psi'}{\phi \rightarrow \psi \leq \phi' \rightarrow \psi'}$$

Fig. 1. The ordering on types

3 Lazy Types

We introduce a slightly restricted language of strictness formulae T_I (Fig. 3); this language is closely related to van Bakel's strict types [1]. Basically strict types do not allow intersections on the right hand side of an arrow. This restriction is convenient because it does not weaken the expressive power of the system and it makes type manipulation easier.

We then define the notion of *complete type*. The restriction to complete types allows us to avoid the use of weakening because a complete type contains (is the conjunction of) all of the elements greater than (or equal to) it.

Definition 1.

$$CT(\tau) = \bigwedge\{ct(\sigma) \mid \sigma \in Sup(\tau)\}$$

$$Sup(\sigma) = \{\sigma' \in T_S \mid \sigma' \geq \sigma\}$$

$$ct(\mathbf{t}) = \mathbf{t} \qquad ct(\mathbf{f}) = \mathbf{f} \qquad ct(\infty) = \infty \qquad ct(\mathbf{f}_{\in}) = \mathbf{f}_{\in}$$

$$\begin{aligned} ct(\sigma \wedge \tau) &= ct(\sigma) \wedge ct(\tau) \\ ct(\sigma \rightarrow \tau) &= CT(\sigma) \rightarrow ct(\tau) \end{aligned}$$

$Sup(\sigma)$ can be defined by induction on σ. Notice that CT can be extended to contexts in the obvious way.

Finally, we can define the notion of *most general type* of an expression (with respect to some context): it is the conjunction of all of the types possessed by the expression in the given environment.

Definition 2 (Most General Types).
$MGT(\Gamma, e) = CT(\bigwedge\{\sigma_i \in T_S \mid \Gamma \vdash_T e : \sigma_i\})$

$$\textbf{Conj}\ \frac{\Gamma \vdash_T e : \psi_1 \quad \Gamma \vdash_T e : \psi_2}{\Gamma \vdash_T e : \psi_1 \wedge \psi_2} \qquad \textbf{Weak}\ \frac{\Gamma \leq \Delta \quad \Delta \vdash_T e : \phi \quad \phi \leq \psi}{\Gamma \vdash_T e : \psi}$$

$$\textbf{Var}\ \ \Gamma[x \mapsto \phi] \vdash_T x : \phi \qquad \textbf{Abs}\ \frac{\Gamma[x \mapsto \phi] \vdash_T e : \psi}{\Gamma \vdash_T \lambda x.e : (\phi \to \psi)} \qquad \textbf{Taut}\ \ \Gamma \vdash_T c : \mathbf{t}$$

$$\textbf{App}\ \frac{\Gamma \vdash_T e_1 : (\phi \to \psi) \quad \Gamma \vdash_T e_2 : \phi}{\Gamma \vdash_T e_1 e_2 : \psi} \qquad \textbf{Fix}\ \frac{\Gamma \vdash_T (\lambda g.e) : \phi \to \phi}{\Gamma \vdash_T \mathbf{fix}(\lambda g.e) : \phi}$$

$$\textbf{Cond-1}\ \frac{\Gamma \vdash_T e_1 : \mathbf{f}}{\Gamma \vdash_T \mathbf{cond}(e_1, e_2, e_3) : \mathbf{f}_\sigma} \qquad \textbf{Cond-2}\ \frac{\Gamma \vdash_T e_2 : \phi \quad \Gamma \vdash_T e_3 : \phi}{\Gamma \vdash_T \mathbf{cond}(e_1, e_2, e_3) : \phi}$$

$$\textbf{Hd}\ \frac{\Gamma \vdash_T e : \mathbf{f}}{\Gamma \vdash_T \mathbf{hd}(e) : \mathbf{f}} \qquad \textbf{Tl-1}\ \frac{\Gamma \vdash_T e : \mathbf{f}}{\Gamma \vdash_T \mathbf{tl}(e) : \mathbf{f}} \qquad \textbf{Tl-2}\ \frac{\Gamma \vdash_T e : \infty}{\Gamma \vdash_T \mathbf{tl}(e) : \infty}$$

$$\textbf{Cons-1}\ \frac{\Gamma \vdash_T e_2 : \infty}{\Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \infty}$$

$$\textbf{Cons-2}\ \frac{\Gamma \vdash_T e_2 : \mathbf{f}_\in}{\Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \mathbf{f}_\in} \qquad \textbf{Cons-3}\ \frac{\Gamma \vdash_T e_1 : \mathbf{f}}{\Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \mathbf{f}_\in}$$

$$\textbf{Case-1}\ \frac{\Gamma \vdash_T e_3 : \mathbf{f}}{\Gamma \vdash_T \mathbf{case}(e_1, e_2, e_3) : \mathbf{f}}$$

$$\textbf{Case-2}\ \frac{\Gamma \vdash_T e_2 : \mathbf{t} \to \infty \to \phi \quad \Gamma \vdash_T e_3 : \infty}{\Gamma \vdash_T \mathbf{case}(e_1, e_2, e_3) : \phi}$$

$$\textbf{Case-3}\ \frac{\Gamma \vdash_T e_2 : \mathbf{t} \to \mathbf{f}_\in \to \phi\ \wedge\ \mathbf{f} \to \mathbf{t} \to \phi \quad \Gamma \vdash_T e_3 : \mathbf{f}_\in}{\Gamma \vdash_T \mathbf{case}(e_1, e_2, e_3) : \phi}$$

$$\textbf{Case-4}\ \frac{\Gamma \vdash_T e_1 : \phi \quad \Gamma \vdash_T e_2 : \mathbf{t} \to \mathbf{t} \to \phi}{\Gamma \vdash_T \mathbf{case}(e_1, e_2, e_3) : \phi}$$

$$\textbf{Taut-hd}\ \ \Gamma \vdash_T \mathbf{hd}(e) : \mathbf{t} \qquad \textbf{Taut-tl}\ \ \Gamma \vdash_T \mathbf{tl}(e) : \mathbf{t}$$

$$\textbf{Taut-cons}\ \ \Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \mathbf{t}$$

Fig. 2. The Strictness logic

We show in [13] that the most general type of an expression is precisely the information returned by the standard abstract interpretation-based analysis. This explains why abstract interpretation is sometimes inefficient because it may provide much more information than really required.

We take a different approach in this paper: rather than returning all possible information about the strictness of a function we compute only the information required to answer a particular question. This new philosophy naturally leads to a notion of lazy evaluation of types. The language of lazy types T_G is defined in Fig. 4. The ordering on types $\leq_G$ and the logic $\vdash_G$ are shown in Fig. 5.

The key idea is that an expression from the term language (with its environment) may appear as part of a type; this plays the rôle of a closure. More formally, a closure (Γ, e) stands for $MGT(\Gamma, e)$, the conjunction of all of the

$$\mathbf{t}, \mathbf{f}, \infty, \mathbf{f}_{\in} \in T_S \qquad \frac{\sigma \in T_I \quad \psi \in T_S}{\sigma \rightarrow \psi \in T_S} \qquad \frac{\phi_1 \in T_S \ldots \phi_n \in T_S}{\phi_1 \wedge \ldots \wedge \phi_n \in T_I}$$

Fig. 3. The language T_I

possible types of the term. This correspondence explains the new rules in the definition of $\leq_G$. Not surprisingly, the lazy evaluation of types is made explicit in the **App** rule: rather than deriving all possible types for e_2, we insert e_2 itself (with the current environment) into the type of e_1. The following definition establishes a correspondence between lazy types and ordinary types, the extension to environments is straightforward:

Definition 3.

$$Expand : T_G \rightarrow T_I$$

$$Expand(\mathbf{t}) = \mathbf{t} \qquad Expand(\mathbf{f}) = \mathbf{f}$$

$$Expand(\infty) = \infty \qquad Expand(\mathbf{f}_{\in}) = \mathbf{f}_{\in}$$

$$Expand(\sigma_1 \wedge \sigma_2) = Expand(\sigma_1) \wedge Expand(\sigma_2)$$

$$Expand(\sigma_1 \rightarrow \sigma_2) = Expand(\sigma_1) \rightarrow Expand(\sigma_2)$$

$$Expand((\Gamma, e)) = MGT(Expand(\Gamma), e)$$

We can now state the correctness and completeness of the lazy type system and the subsequent equivalence with the original system.

Theorem 4 (Correctness).

$$\Gamma \vdash_G e : \phi \Longrightarrow Expand(\Gamma) \vdash_T e : Expand(\phi) \qquad \phi \in T_G$$

Theorem 5 (Completeness).

$$Expand(\Gamma) \vdash_T e : Expand(\phi) \Longrightarrow \Gamma \vdash_G e : \phi \qquad \phi \in T'_S$$

Theorem 6 (Equivalence).

$$\Gamma \vdash_T e : \phi \Leftrightarrow \Gamma \vdash_G e : \phi \qquad \Gamma \in Var \rightarrow T_I, e : \phi \in T_I$$

First notice that we do not lose completeness by considering T_I types: it can be shown quite easily that any type is equivalent to a type in T_I. The following theorems are used in the proofs of theorems 4 and 5.

Theorem 7. $\sigma \leq_G \tau \Leftrightarrow Expand(\sigma) \leq Expand(\tau)$

Theorem 8.

$$\Gamma \vdash_G e : (\phi_1 \wedge \ldots \wedge \phi_n) \quad \Leftrightarrow (\Gamma \vdash_G e : \phi_1) \ \textbf{and} \ \ldots \ \textbf{and} \ (\Gamma \vdash_G e : \phi_n)$$

$$\Gamma \vdash_T e : (\phi_1 \wedge \ldots \wedge \phi_n) \quad \Leftrightarrow (\Gamma \vdash_T e : \phi_1) \ \textbf{and} \ \ldots \ \textbf{and} \ (\Gamma \vdash_T e : \phi_n)$$

Theorem 7 can be proved by induction on the proof of the left hand side. Theorem 8 is shown by deriving a proof of the right hand side from a proof of the left hand side (it is quite straightforward). Theorem 8 allows us to prove theorem 4 by induction on e. The proof of completeness is carried out in two stages. First we show that the weakening rule can be removed from $\vdash_T$ without changing the set of derivable types provided we add a form of weakening in the **Var** and **Fix** rules. A similar property has been proved for other type systems including a form of weakening [1, 22]. Then we use theorems 7 and 8 and proceed by induction on e to prove completeness.

$$nil \in env \qquad \frac{\Gamma \in env \quad \sigma \in T_G}{\Gamma[x \mapsto \sigma] \in env} \qquad \frac{\Gamma \in env \quad e \in exp}{(\Gamma, e) \in T_G}$$

$$\mathbf{t}, \mathbf{f}, \infty, \mathbf{f}_\in \in T'_S \qquad \frac{\sigma \in T_G \quad \psi \in T'_S}{\sigma \to \psi \in T'_S} \qquad \frac{\phi_1 \in T'_S \ldots \phi_n \in T'_S}{\phi_1 \wedge \ldots \wedge \phi_n \in T_G}$$

Fig. 4. The language T_G

4 The lazy types algorithm

In [13], we show how to derive an abstract machine from the basic lazy types system. A similar derivation from the system defined in Section 3 leads to the machine shown in Fig 6. The state of the machine is a triple specifying the current contents of the stack, environment and code. $Inf(\phi, \psi)$ computes $\phi \leq_G \psi$ as defined in Fig. 5 (Theorem 10). Notice that a stack element S_i is either a boolean value or a disjunction of types. $True$ (resp. $False$) is installed at the top of the stack if and only if the original property (of the form (e, ϕ)) in the code is (resp. is not) provable in $\vdash_G$. Values which are neither $True$ nor $False$ in the stack are disjunctions of T_G types $(\phi_1 \vee \ldots \vee \phi_n)$. The occurrence of such a value at the top of the stack means that the original property is true if (and only if) the recursive function currently being analysed possesses one of the ϕ_i types (in order to make the presentation simpler we do not consider embedded occurrences of **fix** here; the extension is straightforward). In order to prove that $\mathbf{fix}(\lambda g.e)$ has type ϕ we add the assumption $(g :_r \phi)$ in the environment and try to prove $e : \phi$. If the result is $True$ or $False$ then the case is settled. Otherwise a disjunction of conditions ϕ_i is returned and the algorithm iterates to try to show that one of them is satisfied (rule for $Iter$). Instruction Rec is used to remember that we were trying to prove a property on a recursively defined variable (denoted by $\mapsto_r$ in the environment); so if it fails we just return this property in the stack rather than $False$.

Primitives And and Or are extended in the obvious way to apply on types: their result is always supposed to be a disjunction of T_G types.

$$\mathbf{f} \leq_G \phi \qquad \phi \leq_G \phi \qquad \infty \leq_G \mathbf{f}_\in \qquad \phi \leq_G \mathbf{t}$$

$$\phi_1 \to \ldots \to \phi_n \to \phi \leq_G \psi_1 \to \ldots \to \psi_n \to \mathbf{t}$$

$$\frac{\forall j \in [1,m], \exists i \in [1,n] \phi_i \leq_G \psi_j}{\phi_1 \wedge \ldots \wedge \phi_n \leq_G \psi_1 \wedge \ldots \wedge \psi_m} \qquad \frac{\forall \phi.(\Gamma \vdash_G e : \phi) \Rightarrow \psi \leq_G \phi}{\psi \leq_G (\Gamma, e)}$$

$$\frac{\Gamma \vdash_G e : \phi}{(\Gamma, e) \leq_G \phi} \;\; (\phi \neq (\Gamma', e')) \qquad \frac{\phi' \leq_G \phi, \psi \leq_G \psi'}{\phi \to \psi \leq_G \phi' \to \psi'}$$

$$\textbf{Conj} \;\; \frac{\Gamma \vdash_G e : \psi_1 \quad \Gamma \vdash_G e : \psi_2}{\Gamma \vdash_G e : \psi_1 \wedge \psi_2} \qquad \textbf{Var} \;\; \frac{\psi_1 \leq_G \psi_2}{\Gamma[x \mapsto \psi_1] \vdash_G x : \psi_2}$$

$$\textbf{Abs} \;\; \frac{\Gamma[x \mapsto \phi] \vdash_G e : \psi}{\Gamma \vdash_G \lambda x.e : (\phi \to \psi)} \qquad \textbf{Taut} \;\; \Gamma \vdash_G c : \mathbf{t}$$

$$\textbf{App} \;\; \frac{\Gamma \vdash_G e_1 : ((\Gamma, e_2) \to \psi)}{\Gamma \vdash_G e_1 e_2 : \psi}$$

$$\textbf{Fix} \;\; \frac{\Gamma \vdash_G (\lambda g.e) : (\bigwedge_{i=1}^{n} \phi_i \to \phi_1) \wedge \ldots \wedge (\bigwedge_{i=1}^{n} \phi_i \to \phi_n)}{\Gamma \vdash_G \mathbf{fix}(\lambda g.e) : \phi_k \quad (k \in [1,n])}$$

$$\textbf{Cond-1} \;\; \frac{\Gamma \vdash_G e_1 : \mathbf{f}}{\Gamma \vdash_G \mathbf{cond}(e_1, e_2, e_3) : \phi} \qquad \textbf{Cond-2} \;\; \frac{\Gamma \vdash_G e_2 : \phi \quad \Gamma \vdash_G e_3 : \phi}{\Gamma \vdash_G \mathbf{cond}(e_1, e_2, e_3) : \phi}$$

$$\textbf{Hd} \;\; \frac{\Gamma \vdash_T e : \mathbf{f}}{\Gamma \vdash_T \mathbf{hd}(e) : \phi} \qquad \textbf{Tl-1} \;\; \frac{\Gamma \vdash_T e : \mathbf{f}}{\Gamma \vdash_T \mathbf{tl}(e) : \mathbf{f}} \qquad \textbf{Tl-2} \;\; \frac{\Gamma \vdash_T e : \infty}{\Gamma \vdash_T \mathbf{tl}(e) : \infty}$$

$$\textbf{Tl-3} \;\; \frac{\Gamma \vdash_T e : \infty}{\Gamma \vdash_T \mathbf{tl}(e) : \mathbf{f}_\in} \qquad \textbf{Cons-1} \;\; \frac{\Gamma \vdash_T e_2 : \infty}{\Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \infty}$$

$$\textbf{Cons-2} \;\; \frac{\Gamma \vdash_T e_2 : \mathbf{f}_\in}{\Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \mathbf{f}_\in} \qquad \textbf{Cons-3} \;\; \frac{\Gamma \vdash_T e_1 : \mathbf{f}}{\Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \mathbf{f}_\in}$$

$$\textbf{Case-1} \;\; \frac{\Gamma \vdash_T e_3 : \mathbf{f}}{\Gamma \vdash_T \mathbf{case}(e_1, e_2, e_3) : \phi}$$

$$\textbf{Case-2} \;\; \frac{\Gamma \vdash_T e_2 : \mathbf{t} \to \infty \to \phi \quad \Gamma \vdash_T e_3 : \infty}{\Gamma \vdash_T \mathbf{case}(e_1, e_2, e_3) : \phi}$$

$$\textbf{Case-3} \;\; \frac{\Gamma \vdash_T e_2 : \mathbf{t} \to \mathbf{f}_\in \to \phi \;\wedge\; \mathbf{f} \to \mathbf{t} \to \phi \quad \Gamma \vdash_T e_3 : \mathbf{f}_\in}{\Gamma \vdash_T \mathbf{case}(e_1, e_2, e_3) : \phi}$$

$$\textbf{Case-4} \;\; \frac{\Gamma \vdash_T e_1 : \phi \quad \Gamma \vdash_T e_2 : \mathbf{t} \to \mathbf{t} \to \phi}{\Gamma \vdash_T \mathbf{case}(e_1, e_2, e_3) : \phi}$$

$$\textbf{Taut-hd} \;\; \Gamma \vdash_T \mathbf{hd}(e) : \mathbf{t} \qquad \textbf{Taut-tl} \;\; \Gamma \vdash_T \mathbf{tl}(e) : \mathbf{t}$$

$$\textbf{Taut-cons} \;\; \Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \mathbf{t}$$

Fig. 5. The Lazy Types system

$$\begin{aligned}
\langle S, E, (c, \mathbf{t}) : C\rangle &\rhd_G \langle True : S, E, C\rangle \\
\langle S, E, (c, \mathbf{f}) : C\rangle &\rhd_G \langle False : S, E, C\rangle
\end{aligned}$$

$$\langle S, E, (e, \phi_1 \wedge \phi_2) : C\rangle \rhd_G \langle S, E, (e, \phi_1) : (e, \phi_2) : And : C\rangle$$

$$\langle S, E, (\lambda x.e, \sigma \to \tau) : C\rangle \rhd_G \langle S, (x : \sigma) : E, (e, \tau) : D(x) : C\rangle$$

$$\langle S, E, (e_1 e_2, \phi) : C\rangle \rhd_G \langle S, E, (e_1, (E, e_2) \to \phi) : C\rangle$$

$$\langle S, E[x \mapsto \phi], (x, \psi) : C\rangle \rhd_G \langle S, E[x \mapsto \phi], Inf(\phi, \psi) : C\rangle$$

$$\langle S, E, (\mathbf{cond}(e_1, e_2, e_3), \phi) : C\rangle \rhd_G \langle S, E, (e_1, \mathbf{f}) : (e_2, \phi) : (e_3, \phi) : And : Or : C\rangle$$

$$\langle S, (x : \sigma) : E, (D(x)) : C\rangle \rhd_G \langle S, E, C\rangle$$

$$\begin{aligned}
\langle S, E, (\mathbf{fix}(\lambda g.e), \phi) : C\rangle &\rhd_G \langle S, (g :_r \phi) : E, (e, \phi) : Iter(g, e) : C\rangle \\
\langle S, E[g \mapsto_r \phi], (g, \psi) : C\rangle &\rhd_G \langle S, E[g \mapsto_r \phi], Inf(\phi, \psi) : (Rec, g, \psi) : C\rangle \\
\langle True : S, E, (Rec, g, \phi) : C\rangle &\rhd_G \langle True : S, E, C\rangle \\
\langle S_1 : S, E, (Rec, g, \phi) : C\rangle &\rhd_G \langle \phi : S, E, C\rangle \\
& \quad S_1 \neq True
\end{aligned}$$

$$\begin{aligned}
\langle S_1 : S, (g :_r \phi) : E, Iter(g, e) : C\rangle &\rhd_G \langle S_1 : S, E, C\rangle \\
& \quad S_1 = True \quad or \quad S_1 = False
\end{aligned}$$

$$\begin{aligned}
&\langle (\phi_1 \vee \ldots \vee \phi_n) : S, (g :_r \phi) : E, Iter(g, e) : C\rangle \rhd_G \\
&\qquad \langle S, E, (\mathbf{fix}(\lambda g.e), \phi_1) : (\mathbf{fix}(\lambda g.e), \phi_2) : Or : \ldots : Or : C\rangle
\end{aligned}$$

$$\begin{aligned}
\langle S, E, (\mathbf{hd}(e), \mathbf{t}) : C\rangle &\rhd_G \langle True : S, E, C\rangle \\
\langle S, E, (\mathbf{hd}(e), \phi) : C\rangle &\rhd_G \langle S, E, (e, \mathbf{f}) : C\rangle
\end{aligned}$$

$$\begin{aligned}
\langle S, E, (\mathbf{tl}(e), \mathbf{t}) : C\rangle &\rhd_G \langle True : S, E, C\rangle \\
\langle S, E, (\mathbf{tl}(e), \mathbf{f}) : C\rangle &\rhd_G \langle S, E, (e, \mathbf{f}) : C\rangle \\
\langle S, E, (\mathbf{tl}(e), \infty) : C\rangle &\rhd_G \langle S, E, (e, \infty) : C\rangle \\
\langle S, E, (\mathbf{tl}(e), \mathbf{f}_\in) : C\rangle &\rhd_G \langle S, E, (e, \infty) : C\rangle
\end{aligned}$$

$$\begin{aligned}
\langle S, E, (\mathbf{cons}(e_1, e_2), \mathbf{t}) : C\rangle &\rhd_G \langle True : S, E, C\rangle \\
\langle S, E, (\mathbf{cons}(e_1, e_2), \infty) : C\rangle &\rhd_G \langle S, E, (e_2, \infty) : C\rangle \\
\langle S, E, (\mathbf{cons}(e_1, e_2), \mathbf{f}_\in) : C\rangle &\rhd_G \langle S, E, (e_1, \mathbf{f}) : (e_2, \mathbf{f}_\in) : Or : C\rangle \\
\langle S, E, (\mathbf{cons}(e_1, e_2), \mathbf{f}) : C\rangle &\rhd_G \langle False : S, E, C\rangle
\end{aligned}$$

$$\begin{aligned}
&\langle S, E, (\mathbf{case}(e_1, e_2, e_3), \phi) : C\rangle \rhd_G \\
&\langle S, E, (e_3, \mathbf{f}) : (e_2, \mathbf{t} \to \infty \to \phi) : (e_3, \infty) : And : (e_2, \mathbf{t} \to \mathbf{f}_\in \to \phi \wedge \mathbf{f} \to \mathbf{t} \to \phi) : (e_3, \mathbf{f}_\in) : \\
&And : (e_1, \phi) : (e_2, \mathbf{t} \to \mathbf{t} \to \phi) : And : Or : Or : Or : C\rangle
\end{aligned}$$

$$\begin{aligned}
\langle S_1 : S_2 : S, E, Op : C\rangle &\rhd_G \langle (Op\ S_1\ S_2) : S, E, C\rangle \\
& \quad Op = And \quad or \quad Op = Or
\end{aligned}$$

Fig. 6. The Lazy Types algorithm

The following theorem states the correctness of the lazy types algorithm.

Theorem 9.

1. $\langle S, \Gamma, (e, \phi) : C\rangle \rhd_G^* \ \langle True : S, \Gamma, C\rangle \Leftrightarrow \Gamma \vdash_G e : \phi$
2. $\langle S, \Gamma, (e, \phi) : C\rangle \rhd_G^* \ \langle False : S, \Gamma, C\rangle \Leftrightarrow \neg(\Gamma \vdash_G e : \phi)$

   if Γ *and* ϕ *do not contain any* $\mapsto_r$ *assumption*

The proof of this theorem is made hand in hand with the proof of the following result:

Theorem 10.

1. $\langle S, \Gamma, Inf(\phi, \psi) : C\rangle \rhd_G^* \ \langle True : S, \Gamma, C\rangle \Leftrightarrow \phi \leq_G \psi$
2. $\langle S, \Gamma, Inf(\phi, \psi) : C\rangle \rhd_G^* \ \langle False : S, \Gamma, C\rangle \Leftrightarrow \neg(\phi \leq_G \psi)$

   if Γ, ϕ *and* ψ *do not contain any* $\mapsto_r$ *assumption*

The most difficult part of the proof concerns the implementation of **fix**. We have two main facts to prove: (1) the iteration terminates and (2) the result is accurate. Termination is proved by showing that each type $\phi \wedge \phi_i$ satisfies $\phi \wedge \phi_i <_G \phi$. It is easy to show that the result is accurate when the iteration terminates with the $True$ answer. In order to show that the initial property cannot be satisfied if the answer is $False$, we prove that at least one of the ϕ_i types returned by the iteration step is a necessary condition to prove the original property (in other words, we do not "bypass" the least fixed point).

The algorithm described in this section can be optimised in several ways:

- The implementation of the conditional can avoid processing the second and third term when the first term has type **f**.
- In the same way, the implementation of the case operation can be considerably optimised if the first term has type **f**. More generally, And and Or can be modified in order to avoid the computation of their second argument when their first argument reduces respectively to $False$ and $True$.
- In the rule for application, when expression e_2 is a constant or a variable then its type (**t** for a constant, its type in the environment for a variable) can be inserted into the type of e_1 rather than passing the whole environment. Notice that this optimisation is common in the implementation of lazy languages.

These optimisations are easy to justify formally and improve the derivation considerably.

5 Example

We consider the following functions:

$$
\begin{aligned}
foldr\ b\ g\ \mathbf{nil} &= b \\
foldr\ b\ g\ \mathbf{cons}(x, xs) &= g\ x\ (foldr\ b\ g\ xs) \\
cat\ l &= foldr\ \mathbf{nil}\ append\ l
\end{aligned}
$$

which were introduced in [17] to demonstrate the inefficiency of traditional abstract interpretation. Notice that we have used pattern matching in the definition of $foldr$; this is for clarity - more properly it should have been defined as:

$$foldr \;=\; \mathbf{fix}(\lambda f.\lambda b.\lambda g.\lambda l.\mathbf{case}(b, \lambda x \lambda xs.g\ x\ (f\ b\ g\ xs), l))$$

Similarly cat should also be defined as a λ-abstraction.
Fig. 7 describes some of the derivation steps of the lazy type algorithm to prove that cat has type $\mathbf{f} \to \mathbf{f}$.

$$
\begin{array}{ll}
\langle Nil, Nil, (cat, \mathbf{f} \to \mathbf{f})\rangle & \rhd_G \\
\langle Nil, (l : \mathbf{f}), (foldr\ \mathbf{nil}\ append\ l, \mathbf{f}) : D(l)\rangle & \rhd_G \\
\langle Nil, (l : \mathbf{f}), (foldr\ \mathbf{nil}\ append, \mathbf{f} \to \mathbf{f}) : D(l)\rangle & \rhd_G \\
\langle Nil, (l : \mathbf{f}), (foldr\ \mathbf{nil}, ((l : \mathbf{f}), append) \to \mathbf{f} \to \mathbf{f}) : D(l)\rangle & \rhd_G \\
\langle Nil, (l : \mathbf{f}), (foldr, \mathbf{t} \to ((l : \mathbf{f}), append) \to \mathbf{f} \to \mathbf{f}) : D(l)\rangle & \rhd_G \\
\vdots & \\
\langle Nil, (l : \mathbf{f}) : (g, ((l : \mathbf{f}), append)) : (b : \mathbf{t}) : (f :_r \phi) : (l : \mathbf{f}), & \\
(\mathbf{case} \ldots, \mathbf{f}) : D(l) : D(g) : D(b) : Iter(f, \ldots) : D(l)\rangle & \rhd_G \\
\vdots & \\
\langle Nil, \ldots, (l, \mathbf{f}) : \ldots : Or : D(l) : D(g) : D(b) : \ldots\rangle & \rhd_G \\
\vdots & \\
\langle True, \ldots, D(l) : D(g) : D(b) : \ldots\rangle & \rhd_G \\
\vdots & \\
\langle True, (f :_r \phi) : (l : \mathbf{f}) : \ldots, Iter(f, \ldots) : D(l)\rangle & \rhd_G \\
\langle True, (l : \mathbf{f}), D(l)\rangle & \rhd_G \\
\langle True, Nil, Nil\rangle &
\end{array}
$$

where ϕ is $\mathbf{t} \to ((l : \mathbf{f}), append) \to \mathbf{f} \to \mathbf{f}$.

Fig. 7. cat has type $\mathbf{f} \to \mathbf{f}$

6 Generalisation to domains of any depth

The 4-point domain expresses information about lists with atomic elements. For example, it is not adequate for describing a property such as "this is a list containing lists whose one element is undefined". Following Wadler [25], we can in fact generalise the definition of 4-point domain from the 2-point domain to domains of any depth. Let

$$D_0 = \{\mathbf{t}, \mathbf{f}\}$$

with $\mathbf{f} \leq_0 \mathbf{t}$. Then

$$D_{i+1} = \{\mathbf{f}, \infty\} \cup \{x_\in \mid x \in D_i\}$$

with:

$$\mathbf{f} \leq_{i+1} \infty$$

$$\forall x_{\in} \in D_{i+1}.\ \infty \leq_{i+1} x_{\in}$$

$$\forall x_{\in},\ y_{\in} \in D_{i+1}.\ x \leq_i y \Leftrightarrow x_{\in} \leq_{i+1} y_{\in}$$

The following property shows that we can omit the subscript and write $\leq$ for $\leq_i$:

$$\forall x,\ y \in D_i \cap D_{i+1}.\ x \leq_i y \Leftrightarrow x \leq_{i+1} y$$

An interesting property of our type inference system (and algorithm) is that it can be generalised without further complication to domains of unbounded depth. The rules **Cons-2**, **Cons-3** and **Case-3** are generalised in the following way:

$$\textbf{Cons-2}\quad \frac{\Gamma \vdash_T e_2 : \sigma_{\in}}{\Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \sigma_{\in}} \qquad \textbf{Cons-3}\quad \frac{\Gamma \vdash_T e_1 : \sigma}{\Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \sigma_{\in}}$$

$$\textbf{Case-3}\quad \frac{\Gamma \vdash_T e_2 : \mathbf{t} \to \sigma_{\in} \to \phi\ \wedge\ \sigma \to \mathbf{t} \to \phi \qquad \Gamma \vdash_T e_3 : \sigma_{\in}}{\Gamma \vdash_T \mathbf{case}(e_1, e_2, e_3) : \phi}$$

and the ordering on types is extended with the rules:

$$\infty \leq \sigma_{\in} \qquad\qquad \frac{\sigma \leq \tau}{\sigma_{\in} \leq \tau_{\in}}$$

The extensions to the algorithm are not described here for the sake of briefness. The implementation of **Cons-2** and **Cons-3** is straightforward because all the free variables occurring in the premises appear in the conclusion. This is not the case for **Case-3** which requires an iteration very much like the rule for abstraction in Fig. 6. The iteration explores the domain starting with D_0 until the property is proven or the maximal depth corresponding to the type of the expression is reached. Several trivial optimisations can dramatically improve the algorithm at this stage. For instance e_3 will often be a variable whose type is defined in the environment (see example below) and can be used to make the appropriate choice of σ, thus avoiding the iteration mentioned above.

We continue the *foldr* example to show that our system (and algorithm) does not need a domain of fixed depth but rather explores the potentially infinite domain up to the depth required to answer a particular question. We first restate the definition of *append* as a term of Λ_L:

$$append\ =\ \mathbf{fix}(\lambda app.\lambda x_1.\lambda x_2.\mathbf{case}(x_2, \lambda x.\lambda xs.\mathbf{cons}(x, (app\ xs\ x_2)), x_1))$$

Assume that we want to prove $foldr : \mathbf{t} \to append \to \infty_{\in} \to \infty$, where *append* is used as a shorthand notation for $(\emptyset, append)$. We do not give all of the details of the derivation but rather focus on the main steps of the proof:

$$
\begin{array}{ll}
 & \vdots \\
 & A \qquad\qquad B \\
\textbf{Conj} & \overline{\Gamma \vdash (\lambda x.\lambda xs.g\ x\ (f\ b\ g\ xs)) : (\mathbf{t} \rightarrow \infty_{\in} \rightarrow \infty)\ \wedge\ (\infty \rightarrow \mathbf{t} \rightarrow \infty)} \quad C \\
\textbf{Case}-\textbf{3} & \overline{\Gamma \vdash \mathbf{case}(b, \lambda x \lambda xs.g\ x\ (f\ b\ g\ xs), l) : \infty} \\
 & \vdots \\
\textbf{Abs} & \overline{\vdash \lambda f.\lambda b.\lambda g.\lambda l.\mathbf{case}(b, \lambda x \lambda xs.g\ x\ (f\ b\ g\ xs), l) :} \\
 & (\mathbf{t} \rightarrow append \rightarrow \infty_{\in} \rightarrow \infty) \rightarrow (\mathbf{t} \rightarrow append \rightarrow \infty_{\in} \rightarrow \infty) \\
\textbf{Fix} & \overline{\vdash \mathbf{fix}(\lambda f.\lambda b.\lambda g.\lambda l.\mathbf{case}(b, \lambda x \lambda xs.g\ x\ (f\ b\ g\ xs), l)) : \mathbf{t} \rightarrow append \rightarrow \infty_{\in} \rightarrow \infty} \\
 & \overline{\vdash foldr : \mathbf{t} \rightarrow append \rightarrow \infty_{\in} \rightarrow \infty}
\end{array}
$$

where Γ is: $[f : \mathbf{t} \rightarrow append \rightarrow \infty_{\in} \rightarrow \infty, b : \mathbf{t}, g : append, l : \infty_{\in}]$. A is:

$$
\begin{array}{ll}
 & \dfrac{\vdots}{\Gamma' \vdash \lambda x.\lambda xs.\mathbf{cons}(x, (app\ xs\ x_2)) : \mathbf{t} \rightarrow \mathbf{t} \rightarrow \infty} \quad \dfrac{\dfrac{\dfrac{\vdots}{\Gamma'' \vdash f\ b\ g\ xs : \infty}}{(\Gamma'', f\ b\ g\ xs) \leq \infty}}{\Gamma' \vdash x_2 : \infty} \\
\textbf{Case}-\textbf{4} & \overline{\Gamma' \vdash \mathbf{case}(x_2, \lambda x.\lambda xs.\mathbf{cons}(x, (app\ xs\ x_2)), x_1) : \infty} \\
 & \vdots \\
\textbf{App} & \Gamma'' \vdash g : \mathbf{t} \rightarrow (f\ b\ g\ xs) \rightarrow \infty \\
\textbf{App} & \overline{\Gamma'' \vdash g\ x : (f\ b\ g\ xs) \rightarrow \infty} \\
\textbf{App} & \overline{\Gamma'' \vdash g\ x\ (f\ b\ g\ xs) : \infty} \\
 & \vdots \\
\textbf{Abs} & \overline{\Gamma \vdash (\lambda x.\lambda xs.g\ x\ (f\ b\ g\ xs)) : (\mathbf{t} \rightarrow \infty_{\in} \rightarrow \infty)}
\end{array}
$$

where

$$
\begin{aligned}
\Gamma' &= [app : (\mathbf{t} \rightarrow (\Gamma'', (f\ b\ g\ xs)) \rightarrow \infty), x_1 : \mathbf{t}, x_2 : (\Gamma'', (f\ b\ g\ xs))]\ :\ \Gamma'' \\
\Gamma'' &= [x : \mathbf{t}, xs : \infty_{\in}]\ :\ \Gamma
\end{aligned}
$$

the proof tree for B is similarly constructed and C is $\Gamma \vdash l : \infty_{\in}$. So the domain is explored up to depth 2 (D_2). If we now ask the question $foldr : \mathbf{t} \rightarrow append \rightarrow \mathbf{f}_{\in} \rightarrow \infty$, the domain is not explored further than depth 1, as the reader can easily verify (the structure of the proof is very similar to the previous one).

7 Conclusions

The problem of designing efficient algorithms for strictness analysis has received much attention recently and one current trend seems to revert from the usual "extensional" approach to more "intensional" or syntactic techniques [20, 21, 18, 6, 10, 24]. The key observation underlying these works is that the choice of representing abstract functions by functions can be disastrous in terms of efficiency and is not always justified in terms of accuracy. Some of these proposals trade a cheaper implementation against a loss of accuracy [20, 21]. In contrast, [10, 24] use extensional representations of functions to build very efficient algorithms without sacrificing accuracy. The analysis of [10] uses concrete data structures; these are special kinds of Scott domains whose elements can be seen as syntax trees. In [24] the analysis is expressed as a form of reduction of abstract graphs. An interesting avenue for further research would be to reexpress these analyses in terms of type inference as suggested here to prove their correctness and to be able to relate the techniques on a formal basis.

Wadler's domain construction does not readily generalise to other recursive data types. Recently Benton [3] has shown how to construct an abstract domain from any algebraic data type. It should be straightforward to extend our system (and algorithm) to incorporate such domains. Benton's construction leads to quite large domains; the size of the domains would make conventional abstract interpretation intractable and highlights the benefit of our approach which lazily explores the domain.

In his thesis Jensen, [19], has developed a more general logical treatment of recursive types. His approach involves two extensions to the logic; the first is to add disjunctions and the second extension involves adding modal operators for describing uniform properties of elements of recursive types. The extension of our techniques to these richer logics is an open research problem which we are currently investigating.

References

1. S. van Bakel, *Complete restrictions of the intersection type discipline*, Theoretical Computer Science, 102(1):135-163, 1992.
2. P. N. Benton, *Strictness logic and polymorphic invariance*, in *Proceedings of the 2nd Int. Symposium on Logical Foundations of Computer Science*, LNCS 620, Springer Verlag, 1992.
3. P. N. Benton, *Strictness Properties of Lazy Algebraic Datatypes*, in *Proceedings WSA'93*, LNCS 724, Springer Verlag, 1993.
4. G. L. Burn, *Evaluation Transformers - a model for the parallel evaluation of functional languages (extended abstract)*, in *Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture*, LNCS 274, Springer Verlag, 1987.
5. G. Burn and D. Le Métayer, *Proving the correctness of compiler optimisations based on strictness analysis*, in *Proceedings 5th int. Symp. on Programming Language Implementation and Logic Programming*, LNCS 714, Springer Verlag, 1993.
6. T.-R. Chuang and B. Goldberg, *A syntactic approach to fixed point computation on finite domains*, in *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, ACM Press, 1992.
7. P. Cousot and R. Cousot, *Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation*, in M. Bruynooghe and M. Wirsing (eds), *PLILP'92*, LNCS 631, Springer Verlag, 1992.
8. O. Danvy and J. Hatcliff, *CPS transformation after strictness analysis*, Technical Report, Kansas State University, to appear in ACM LOPLAS.
9. M. van Eekelen, E. Goubault, C. Hankin and E. Nøker, *Abstract reduction: a theory via abstract interpretation*, in R. Sleep et al (eds), *Term graph rewriting: theory and practice*, John Wiley & Sons Ltd, 1992.
10. A. Ferguson and R. J. M. Hughes, *Fast abstract interpretation using sequential algorithms*, in *Proceedings WSA'93*, LNCS 724, Springer Verlag, 1993.

11. S. Finne and G. Burn, *Assessing the evaluation transformer model of reduction on the spineless G-machine*, in *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture*, ACM Press, 1993, pp. 331-341.
12. C. L. Hankin and L. S. Hunt, *Approximate fixed points in abstract interpretation*, in B. Krieg-Brückner (ed), *Proceedings of the 4th European Symposium on Programming*, LNCS 582, Springer Verlag, 1992.
13. C. L. Hankin and D. Le Métayer, *Deriving algorithms from type inference systems: Application to strictness analysis*, to appear in *Proceedings of POPL'94*, ACM Press, 1994.
14. J. J. Hannan, *Investigating a proof-theoretic meta-language*, PhD thesis, University of Pennsylvania, DIKU Technical Report Nr 91/1, 1991.
15. J. Hannan and D. Miller, *From Operational Semantics to Abstract Machines*, Mathematical Structures in Computer Science, 2(4), 1992.
16. P. H. Hartel and K. G. Langendoen, *Benchmarking implementations of lazy functional languages*, in *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture*, ACM Press, 1993, pp. 341-350.
17. L. S. Hunt and C. L. Hankin, *Fixed Points and Frontiers: A New Perspective*, Journal of Functional Programming, **1**(1), 1991.
18. T. P. Jensen, *Strictness Analysis in Logical Form*, in J. Hughes (ed), *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS 523, Springer Verlag, 1991.
19. T. P. Jensen, *Abstract Interpretation in Logical Form*, PhD thesis, University of London, 1992. Also available as DIKU Technical Report 93/11.
20. T.-M. Kuo and P. Mishra, *Strictness analysis: a new perspective based on type inference*, in *Proceedings of the 4th ACM Conference on Functional Programming Languages and Computer Architecture*, ACM Press, 1989.
21. A. Leung and P. Mishra, *Reasoning about simple and exhaustive demand in higher-order lazy languages*, in *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS 523, Springer Verlag, 1991.
22. J. C. Mitchell, *Type inference with simple subtypes*, Journal of Functional Programming, **1**(3), 1991.
23. A. Mycroft, *Abstract Interpretation and Optimising Transformations for Applicative Programs*, PhD thesis, University of Edinburgh, December 1981.
24. E. Nöcker, *Strictness analysis using abstract reduction*, in *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture*, ACM Press, 1993.
25. P. Wadler, *Strictness Analysis on Non-flat Domains*, in S. Abramsky and C. L. Hankin (eds), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.
26. P. Wadler and J. Hughes, *Projections for Strictness Analysis*, in *Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture*, LNCS 274, Springer Verlag, 1987.