

# Lazy types and Program Analysis

Chris Hankin  
Department of Computing,  
Imperial College,  
LONDON SW7 2BZ, UK

Daniel Le Métayer  
INRIA/IRISA,  
Campus de Beaulieu,  
35042 RENNES CEDEX, FRANCE

September 5, 1994

## Abstract

Approaches to static analysis based on non-standard type systems have received considerable interest recently. Most work has concentrated on the relationship between such analyses and abstract interpretation. In this paper, we focus on the problem of producing efficient algorithms from such type-based analyses. The key idea is the notion of *lazy* types. We present the basic notions in the context of a higher-order strictness analysis of list-processing functions. We also sketch some recent work concerning a general framework for program analysis based on these ideas. We conclude with some experimental results.

## 1 Introduction

Two major formal frameworks have been proposed for static analysis of functional languages: abstract interpretation and type inference. A lot of work has been done to characterise formally the correctness and the power of abstract interpretation. However the development of algorithms has not kept pace with the theoretical developments. This is now a major barrier that is preventing the inclusion of the most advanced techniques in compilers. The most significant contributions for improving the efficiency of abstract interpretation include widening techniques [9, 14], chaotic iteration sequences [8, 34] (and the related minimal function graphs [25]), and *frontiers*-based algorithms [33, 21]. The latter has unacceptable performance for some commonly occurring higher-order programs. The first two are general approaches for accelerating convergence in fixed point computations.

In contrast to abstract interpretation, type inference systems are routinely implemented as part of production quality compilers. This has led some researchers to develop program analyses based on non-standard type inference. One of the earliest examples is Kuo and Mishra's strictness analysis [26]. A natural question arises concerning the relationship between this approach and abstract interpretation. Kuo and Mishra's system is strictly weaker than the standard approaches based on abstract interpretation but Jensen [23] has shown how it can be extended to regain this equivalence. The logic is not immediately suggestive of an algorithm; this is mainly because of the weakening rule which may be applied at arbitrary points in a derivation. In this paper we will redress this shortcoming of the type-based approach.

We will use strictness analysis as the case study but we show in section 7 that the techniques are more generally applicable. 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:

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

Rather than suspending the evaluation of each recursive call to *append* and returning the weak head normal form  $\mathbf{cons}(x, (\text{append } xs \ l))$ , 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, 28, 35, 36]. 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 [14, 21].

The main feature of our approach 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 [9]).

## 2 A strictness logic for the analysis of lists

We consider a strongly typed language,  $\Lambda_L$ , with terms defined by the following syntax:

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

The **case** operator is used in the translation of pattern matching. The third argument is the list parameter, the first argument is the result when the list is empty and the second argument is a binary function which is applied to the head and the tail of the list. For example, the *sum* function from the previous section is translated as:

$$\text{sum}(l) = \mathbf{fix}(\lambda s. \lambda l. \mathbf{case}(0, f, l)) \quad \text{where } f \ x \ xs = x + (\text{sum } xs)$$

The loss of accuracy that occurs without the **case** operator is discussed in [35].

Abstract interpretation represents the strictness properties of a function by an abstract function defined on boolean domains [31]. 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  $g : \mathbf{t} \rightarrow \mathbf{f} \rightarrow \mathbf{f}$ . Notice that **t** and **f** are now (non-standard) types. Conjunctive types are required to retain the power of abstract interpretation: a strict function like  $+$  must have type  $(\mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{f}) \wedge (\mathbf{t} \rightarrow \mathbf{f} \rightarrow \mathbf{f})$ . 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 [35]. We show that this extension can be generalised to domains of unbounded depth later. The four elements of the domain are  $\mathbf{f} \leq \infty \leq \mathbf{f}_\epsilon \leq \mathbf{t}$  where  $\infty$  represents infinite lists or lists ending with an undefined element and  $\mathbf{f}_\epsilon$  corresponds to finite lists whose elements may be undefined (plus the lists represented by  $\infty$ ).

The ordering on types is described in Fig. 1. Some occurrences of *t* are subscripted by a standard type because the set of constant types include in fact a collection of *t* and *f* [23] (one for each possible “arrow structure” of a standard type). These subscripts are often omitted because they can be inferred from the context. 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.  $\Gamma$  is an environment mapping variables to formulae (i.e. strictness types). In the rule **Cond-1**,  $\sigma$  represents the standard type of  $e_2$  (or  $e_3$ ). This system is an extension of [15, 23] and the soundness and completeness proofs of the logic (with respect to traditional abstract interpretation) follow straightforwardly from [24]. As an illustration, we show how the property,  $sum : \mathbf{f}_\epsilon \rightarrow \mathbf{f}$ , can be derived in this logic:

$$\begin{array}{c}
\text{Conj} \quad \frac{\frac{A \quad B}{[s : \mathbf{f}_\epsilon \rightarrow \mathbf{f}, l : \mathbf{f}_\epsilon] \vdash \lambda x. \lambda xs. x + (s \ xs) : \mathbf{t} \rightarrow \mathbf{f}_\epsilon \rightarrow \mathbf{f} \wedge \mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{f}}} \\
\text{Case - 3} \quad \frac{[s : \mathbf{f}_\epsilon \rightarrow \mathbf{f}, l : \mathbf{f}_\epsilon] \vdash \text{case}(0, \lambda x. \lambda xs. x + (s \ xs), l) : \mathbf{f}}{C} \\
\\
\vdots \\
\text{Abs} \quad \frac{\vdash (\lambda s. \lambda l. \text{case}(0, \lambda x. \lambda xs. x + (s \ xs), l)) : (\mathbf{f}_\epsilon \rightarrow \mathbf{f}) \rightarrow (\mathbf{f}_\epsilon \rightarrow \mathbf{f})}{\vdash \text{fix}(\lambda s. \lambda l. \text{case}(0, \lambda x. \lambda xs. x + (s \ xs), l)) : \mathbf{f}_\epsilon \rightarrow \mathbf{f}} \\
\text{Fix} \quad \frac{\vdash \text{fix}(\lambda s. \lambda l. \text{case}(0, \lambda x. \lambda xs. x + (s \ xs), l)) : \mathbf{f}_\epsilon \rightarrow \mathbf{f}}{\vdash sum : \mathbf{f}_\epsilon \rightarrow \mathbf{f}}
\end{array}$$

where A is:

$$\begin{array}{c}
\vdots \\
\frac{[s : \mathbf{f}_\epsilon \rightarrow \mathbf{f}, l : \mathbf{f}_\epsilon, x : \mathbf{t}, xs : \mathbf{f}_\epsilon] \vdash x + (s \ xs) : \mathbf{f}}{\vdots} \\
\text{Abs} \quad \frac{[s : \mathbf{f}_\epsilon \rightarrow \mathbf{f}, l : \mathbf{f}_\epsilon] \vdash \lambda x. \lambda xs. x + (s \ xs) : \mathbf{t} \rightarrow \mathbf{f}_\epsilon \rightarrow \mathbf{f}}{}
\end{array}$$

B is:

$$\begin{array}{c}
\vdots \\
\frac{[s : \mathbf{f}_\epsilon \rightarrow \mathbf{f}, l : \mathbf{f}_\epsilon, x : \mathbf{f}, xs : \mathbf{t}] \vdash x + (s \ xs) : \mathbf{f}}{\vdots} \\
\text{Abs} \quad \frac{[s : \mathbf{f}_\epsilon \rightarrow \mathbf{f}, l : \mathbf{f}_\epsilon] \vdash \lambda x. \lambda xs. x + (s \ xs) : \mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{f}}{}
\end{array}$$

and C is:

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

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.

$\mathbf{f} \leq \phi$	$\phi \leq \phi$	$\infty \leq \mathbf{f}_\epsilon$	$\phi \leq \mathbf{t}$	$\mathbf{t}_{\sigma \rightarrow \tau} \leq \mathbf{t}_\sigma \rightarrow \mathbf{t}_\tau$
$\frac{\phi \leq \psi, \psi \leq \chi}{\phi \leq \chi}$	$\frac{\phi \leq \psi_1, \phi \leq \psi_2}{\phi \leq \psi_1 \wedge \psi_2}$	$\phi \wedge \psi \leq \phi$	$\phi \wedge \psi \leq \psi$	
$\phi \rightarrow \psi_1 \wedge \phi \rightarrow \psi_2 \leq \phi \rightarrow (\psi_1 \wedge \psi_2)$	$\frac{\phi' \leq \phi, \psi \leq \psi'}{\phi \rightarrow \psi \leq \phi' \rightarrow \psi'}$			

Figure 1: The ordering on types

### 3 Lazy Types

There are two main reasons why it is difficult to produce an algorithm from the logic defined in Fig. 2:

- The rule **Weak** can be used at arbitrary points in a derivation.
- Some rules have multiple premises - this poses a problem of *strategy* when we sequentialise the derivation.

$$\begin{array}{c}
\text{Conj} \quad \frac{\Gamma \vdash_T e : \psi_1 \quad \Gamma \vdash_T e : \psi_2}{\Gamma \vdash_T e : \psi_1 \wedge \psi_2} \qquad \text{Weak} \quad \frac{\Gamma \leq \Delta \quad \Delta \vdash_T e : \phi \quad \phi \leq \psi}{\Gamma \vdash_T e : \psi} \\
\\
\text{Var} \quad \Gamma[x \mapsto \phi] \vdash_T x : \phi \qquad \text{Abs} \quad \frac{\Gamma[x \mapsto \phi] \vdash_T e : \psi}{\Gamma \vdash_T \lambda x. e : (\phi \rightarrow \psi)} \qquad \text{Taut} \quad \Gamma \vdash_T c : \mathbf{t} \\
\\
\text{App} \quad \frac{\Gamma \vdash_T e_1 : (\phi \rightarrow \psi) \quad \Gamma \vdash_T e_2 : \phi}{\Gamma \vdash_T e_1 e_2 : \psi} \qquad \text{Fix} \quad \frac{\Gamma \vdash_T (\lambda g. e) : \phi \rightarrow \phi}{\Gamma \vdash_T \mathbf{fix}(\lambda g. e) : \phi} \\
\\
\text{Cond-1} \quad \frac{\Gamma \vdash_T e_1 : \mathbf{f}}{\Gamma \vdash_T \mathbf{cond}(e_1, e_2, e_3) : \mathbf{f}_\sigma} \qquad \text{Cond-2} \quad \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} \\
\\
\text{Hd} \quad \frac{\Gamma \vdash_T e : \mathbf{f}}{\Gamma \vdash_T \mathbf{hd}(e) : \mathbf{f}} \qquad \text{Tl-1} \quad \frac{\Gamma \vdash_T e : \mathbf{f}}{\Gamma \vdash_T \mathbf{tl}(e) : \mathbf{f}} \qquad \text{Tl-2} \quad \frac{\Gamma \vdash_T e : \infty}{\Gamma \vdash_T \mathbf{tl}(e) : \infty} \\
\\
\text{Cons-1} \quad \frac{\Gamma \vdash_T e_2 : \infty}{\Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \infty} \\
\\
\text{Cons-2} \quad \frac{\Gamma \vdash_T e_2 : \mathbf{f}_\infty}{\Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \mathbf{f}_\infty} \qquad \text{Cons-3} \quad \frac{\Gamma \vdash_T e_1 : \mathbf{f}}{\Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \mathbf{f}_\infty} \\
\\
\text{Case-1} \quad \frac{\Gamma \vdash_T e_3 : \mathbf{f}}{\Gamma \vdash_T \mathbf{case}(e_1, e_2, e_3) : \mathbf{f}} \\
\\
\text{Case-2} \quad \frac{\Gamma \vdash_T e_2 : \mathbf{t} \rightarrow \infty \rightarrow \phi \quad \Gamma \vdash_T e_3 : \infty}{\Gamma \vdash_T \mathbf{case}(e_1, e_2, e_3) : \phi} \\
\\
\text{Case-3} \quad \frac{\Gamma \vdash_T e_2 : \mathbf{t} \rightarrow \mathbf{f}_\infty \rightarrow \phi \wedge \mathbf{f} \rightarrow \mathbf{t} \rightarrow \phi \quad \Gamma \vdash_T e_3 : \mathbf{f}_\infty}{\Gamma \vdash_T \mathbf{case}(e_1, e_2, e_3) : \phi} \\
\\
\text{Case-4} \quad \frac{\Gamma \vdash_T e_1 : \phi \quad \Gamma \vdash_T e_2 : \mathbf{t} \rightarrow \mathbf{t} \rightarrow \phi}{\Gamma \vdash_T \mathbf{case}(e_1, e_2, e_3) : \phi} \\
\\
\text{Taut-hd} \quad \Gamma \vdash_T \mathbf{hd}(e) : \mathbf{t} \qquad \text{Taut-tl} \quad \Gamma \vdash_T \mathbf{tl}(e) : \mathbf{t} \\
\\
\text{Taut-cons} \quad \Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \mathbf{t}
\end{array}$$

Figure 2: The Strictness logic

As a first step to solve these problems, 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.

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

Figure 3: The language  $T_I$

We 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 3.1 (Most General Types)**

$$MGT(\Gamma, e) = \bigwedge \{ \sigma_i \in T_S \mid \Gamma \vdash_T e : \sigma_i \}$$

We show in [15] that the most general type of an expression is precisely the information returned by the standard abstract interpretation-based analysis. This suggests that abstract interpretation is sometimes inefficient just because it computes 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  $(\Gamma, e)$  stands for  $MGT(\Gamma, e)$ , the conjunction of all of the 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 3.2**

$$\begin{aligned}
& \text{Expand} : T_G \rightarrow T_I \\
& \text{Expand}(\mathbf{t}) = \mathbf{t} \qquad \text{Expand}(\mathbf{f}) = \mathbf{f} \\
& \text{Expand}(\infty) = \infty \qquad \text{Expand}(\mathbf{f}_\epsilon) = \mathbf{f}_\epsilon \\
& \text{Expand}(\sigma_1 \wedge \sigma_2) = \text{Expand}(\sigma_1) \wedge \text{Expand}(\sigma_2) \\
& \text{Expand}(\sigma_1 \rightarrow \sigma_2) = \text{Expand}(\sigma_1) \rightarrow \text{Expand}(\sigma_2) \\
& \text{Expand}((\Gamma, e)) = MGT(\text{Expand}(\Gamma), e)
\end{aligned}$$

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

**THEOREM 3.3 (Correctness)**

$$\Gamma \vdash_G e : \phi \implies \text{Expand}(\Gamma) \vdash_T e : \text{Expand}(\phi) \qquad \phi \in T_G$$

**THEOREM 3.4 (Completeness)**

$$\text{Expand}(\Gamma) \vdash_T e : \text{Expand}(\phi) \implies \Gamma \vdash_G e : \phi \quad \phi \in T'_S$$

**THEOREM 3.5 (Equivalence)**

$$\Gamma \vdash_T e : \phi \Leftrightarrow \Gamma \vdash_G e : \phi \quad \Gamma \in \text{Var} \rightarrow T_I, \quad e : \phi \in T_I$$

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 3.3 and 3.4.

**THEOREM 3.6**  $\sigma \leq_G \tau \Leftrightarrow \text{Expand}(\sigma) \leq \text{Expand}(\tau)$

**THEOREM 3.7**

$$\begin{aligned} \Gamma \vdash_G e : (\phi_1 \wedge \dots \wedge \phi_n) &\Leftrightarrow (\Gamma \vdash_G e : \phi_1) \text{ and } \dots \text{ and } (\Gamma \vdash_G e : \phi_n) \\ \Gamma \vdash_T e : (\phi_1 \wedge \dots \wedge \phi_n) &\Leftrightarrow (\Gamma \vdash_T e : \phi_1) \text{ and } \dots \text{ and } (\Gamma \vdash_T e : \phi_n) \end{aligned}$$

Theorem 3.6 can be proved by induction on the proof of the left hand side. Theorem 3.7 is shown by deriving a proof of the right hand side from a proof of the left hand side (it is quite straightforward). Theorem 3.7 allows us to prove Theorem 3.3 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, 29]. This property addresses the first problem identified above; now weakenings are applied at specific (rather than arbitrary) points in the proof. Then we use Theorems 3.6 and 3.7 and proceed by induction on  $e$  to prove completeness.

$nil \in env$	$\frac{\Gamma \in env \quad \sigma \in T_G}{\Gamma[x \mapsto \sigma] \in env}$	$\frac{\Gamma \in env \quad e \in \Lambda_L}{(\Gamma, e) \in T_G}$
$t, f, \infty, f_e \in T'_S$	$\frac{\sigma \in T_G \quad \psi \in T'_S}{\sigma \rightarrow \psi \in T'_S}$	$\frac{\phi_1 \in T'_S \dots \phi_n \in T'_S}{\phi_1 \wedge \dots \wedge \phi_n \in T_G}$

Figure 4: The language  $T_G$

## 4 The lazy types algorithm

This section presents our “lazy types” algorithm for proving properties in the logic defined in Fig. 5. Rather than introducing a new algorithm and proving its correctness in a second stage, we derive the algorithm from the logic by a succession of refinements in the style of [18]. As a first step towards an algorithm, we introduce a predicate  $M_1$  which includes an extra boolean argument capturing the idea of the result of a computation (a property in the logic will be either provable or not):

$$M_1 : env \rightarrow (\Lambda_L \times T_G) \rightarrow Bool \rightarrow o$$

where  $o$  is used to denote the types of propositions.  $M_1$  satisfies the following property:

$$M_1 \Gamma (e, \sigma) \text{ True} \Leftrightarrow \Gamma \vdash_S e : \sigma$$

We postpone the treatment of recursion and come back to it at the end of the section. We take as an illustration the rules for conjunction, constants, and application:

$$\begin{array}{c}
\mathbf{f} \leq_G \phi \quad \phi \leq_G \phi \quad \infty \leq_G \mathbf{f}_\infty \quad \phi \leq_G \mathbf{t} \\
\\
\frac{\phi_1 \rightarrow \dots \rightarrow \phi_n \rightarrow \phi \leq_G \psi_1 \rightarrow \dots \rightarrow \psi_n \rightarrow \mathbf{t}}{\forall j \in [1, m], \exists i \in [1, n] \phi_i \leq_G \psi_j} \quad \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} \quad (\phi \neq (\Gamma', e')) \quad \frac{\phi' \leq_G \phi, \psi \leq_G \psi'}{\phi \rightarrow \psi \leq_G \phi' \rightarrow \psi'} \\
\text{Conj} \quad \frac{\Gamma \vdash_G e : \psi_1 \quad \Gamma \vdash_G e : \psi_2}{\Gamma \vdash_G e : \psi_1 \wedge \psi_2} \quad \text{Var} \quad \frac{\psi_1 \leq_G \psi_2}{\Gamma[x \mapsto \psi_1] \vdash_G x : \psi_2} \\
\text{Abs} \quad \frac{\Gamma[x \mapsto \phi] \vdash_G e : \psi}{\Gamma \vdash_G \lambda x. e : (\phi \rightarrow \psi)} \quad \text{Taut} \quad \Gamma \vdash_G c : \mathbf{t} \\
\text{App} \quad \frac{\Gamma \vdash_G e_1 : ((\Gamma, e_2) \rightarrow \psi)}{\Gamma \vdash_G e_1 e_2 : \psi} \\
\text{Fix} \quad \frac{\Gamma \vdash_G (\lambda g. e) : (\bigwedge_{i=1}^n \phi_i \rightarrow \phi_1) \wedge \dots \wedge (\bigwedge_{i=1}^n \phi_i \rightarrow \phi_n)}{\Gamma \vdash_G \text{fix}(\lambda g. e) : \phi_k \quad (k \in [1, n])} \\
\text{Cond-1} \quad \frac{\Gamma \vdash_G e_1 : \mathbf{f}}{\Gamma \vdash_G \text{cond}(e_1, e_2, e_3) : \phi} \quad \text{Cond-2} \quad \frac{\Gamma \vdash_G e_2 : \phi \quad \Gamma \vdash_G e_3 : \phi}{\Gamma \vdash_G \text{cond}(e_1, e_2, e_3) : \phi} \\
\text{Hd} \quad \frac{\Gamma \vdash_G e : \mathbf{f}}{\Gamma \vdash_G \text{hd}(e) : \phi} \quad \text{Tl-1} \quad \frac{\Gamma \vdash_G e : \mathbf{f}}{\Gamma \vdash_G \text{tl}(e) : \mathbf{f}} \quad \text{Tl-2} \quad \frac{\Gamma \vdash_G e : \infty}{\Gamma \vdash_G \text{tl}(e) : \infty} \\
\text{Tl-3} \quad \frac{\Gamma \vdash_G e : \infty}{\Gamma \vdash_G \text{tl}(e) : \mathbf{f}_\infty} \quad \text{Cons-1} \quad \frac{\Gamma \vdash_G e_2 : \infty}{\Gamma \vdash_G \text{cons}(e_1, e_2) : \infty} \\
\text{Cons-2} \quad \frac{\Gamma \vdash_G e_2 : \mathbf{f}_\infty}{\Gamma \vdash_G \text{cons}(e_1, e_2) : \mathbf{f}_\infty} \quad \text{Cons-3} \quad \frac{\Gamma \vdash_G e_1 : \mathbf{f}}{\Gamma \vdash_G \text{cons}(e_1, e_2) : \mathbf{f}_\infty} \\
\text{Case-1} \quad \frac{\Gamma \vdash_G e_3 : \mathbf{f}}{\Gamma \vdash_G \text{case}(e_1, e_2, e_3) : \phi} \\
\text{Case-2} \quad \frac{\Gamma \vdash_G e_2 : \mathbf{t} \rightarrow \infty \rightarrow \phi \quad \Gamma \vdash_G e_3 : \infty}{\Gamma \vdash_G \text{case}(e_1, e_2, e_3) : \phi} \\
\text{Case-3} \quad \frac{\Gamma \vdash_G e_2 : \mathbf{t} \rightarrow \mathbf{f}_\infty \rightarrow \phi \wedge \mathbf{f} \rightarrow \mathbf{t} \rightarrow \phi \quad \Gamma \vdash_G e_3 : \mathbf{f}_\infty}{\Gamma \vdash_G \text{case}(e_1, e_2, e_3) : \phi} \\
\text{Case-4} \quad \frac{\Gamma \vdash_G e_1 : \phi \quad \Gamma \vdash_G e_2 : \mathbf{t} \rightarrow \mathbf{t} \rightarrow \phi}{\Gamma \vdash_G \text{case}(e_1, e_2, e_3) : \phi} \\
\text{Taut-hd} \quad \Gamma \vdash_G \text{hd}(e) : \mathbf{t} \quad \text{Taut-tl} \quad \Gamma \vdash_G \text{tl}(e) : \mathbf{t} \\
\text{Taut-cons} \quad \Gamma \vdash_G \text{cons}(e_1, e_2) : \mathbf{t}
\end{array}$$

Figure 5: The Lazy Types system

$$\frac{M_1 \quad \Gamma \quad (e, \psi_1) \quad S_1 \quad M_1 \quad \Gamma \quad (e, \psi_2) \quad S_2}{M_1 \quad \Gamma \quad (e, \psi_1 \wedge \psi_2) \quad And(S_1, S_2)}$$

$$M_1 \quad \Gamma \quad (c, \mathbf{t}) \quad \mathbf{True}$$

$$M_1 \quad \Gamma \quad (c, \mathbf{f}) \quad \mathbf{False}$$

$$\frac{M_1 \quad \Gamma \quad (e_1, (\Gamma, e_2) \rightarrow \psi) \quad S}{M_1 \quad \Gamma(e_1 e_2, \psi) \quad S}$$

The second problem mentioned earlier is the occurrence of multi-premise rules. We define predicate  $M_2$  which captures the notion of a succession of proofs in the original logic:

$$M_2 : list(env) \rightarrow list(\Lambda_L \times T_G) \rightarrow list(Bool) \rightarrow \circ$$

such that:

$$M_2 \quad \overline{\Gamma} \quad \overline{(e, \sigma)} \quad \overline{S} \quad \Leftrightarrow \quad \forall i. M_1 \quad \Gamma_i \quad (e_i, \sigma_i) \quad S_i$$

$M_2$  is defined as follows for conjunction, constants and application:

$$\frac{M_2 \quad \Gamma : \Gamma : E \quad (e, \psi_1) : (e, \psi_2) : C \quad S_1 : S_2 : S}{M_2 \quad \Gamma : E \quad (e, \psi_1 \wedge \psi_2) : C \quad And(S_1, S_2) : S}$$

$$\frac{M_2 \quad E \quad C \quad S}{M_2 \quad \Gamma : E \quad (c, \mathbf{t}) : C \quad True : S}$$

$$\frac{M_2 \quad E \quad C \quad S}{M_2 \quad \Gamma : E \quad (c, \mathbf{f}) : C \quad False : S}$$

$$\frac{M_2 \quad \Gamma : E \quad (e_1, (\Gamma, e_2) \rightarrow \psi) : C \quad S_1 : S}{M_2 \quad \Gamma : E \quad (e_1 e_2, \psi) : C \quad S_1 : S}$$

where  $E, C, S$  represent the remaining lists of environments, expressions and types, respectively. We also need an axiom for the terminal case:

$$M_2 \quad nil \quad nil \quad nil$$

In this system a new environment is created for each instruction (subexpression) in the code. This is not very sensible and the next transformation replaces the list of environments by a single environment.

$$M_3 : env \rightarrow list(\Lambda_L \times T_G) \rightarrow list(Bool) \rightarrow \circ$$

$$M_3 \quad \Gamma \quad (e, \sigma) : nil \quad S : nil \quad \Leftrightarrow \quad M_2 \quad \Gamma : nil \quad (e, \sigma) : nil \quad S : nil$$

$M_3$  is derived from  $M_2$  in a straightforward way:

$$\frac{M_3 \quad \Gamma \quad (e, \psi_1) : (e, \psi_2) : C \quad S_1 : S_2 : S}{M_3 \quad \Gamma \quad (e, \psi_1 \wedge \psi_2) : C \quad And(S_1, S_2) : S}$$



$$\frac{M_3 \quad \Gamma \quad C \quad S}{M_3 \quad \Gamma \quad (c, \mathbf{t}) : C \quad True : S}$$

$$\frac{M_3 \quad \Gamma \quad C \quad S}{M_3 \quad \Gamma \quad (c, \mathbf{f}) : C \quad False : S}$$

$$\frac{M_3 \quad \Gamma \quad (e_1, (\Gamma, e_2) \rightarrow \psi) : C \quad S_1 : S}{M_3 \quad \Gamma \quad (e_1 e_2, \psi) : C \quad S_1 : S}$$

We now consider  $M_3$  as a model for a potential abstract machine. The third argument to  $M_3$  is a stack of results. Each rule can be read as a rewrite rule or a transition where the conclusion is the left hand side and the (single) premise is the right hand side. The only reason why  $M_3$  still does not behave like an abstract machine is the fact that the system does not exhibit a tail recursive behaviour. For instance, in the rule for conjunction, the *And* operation has to be applied to the result of the “rewriting” at the top of the stack. To solve this problem we introduce an extra argument  $R$  which is not modified in the rules and is ultimately instantiated with the result of the computation.

$$M_4 : env \rightarrow (list(\Lambda_L \times T_G) + (Bool \times Bool \rightarrow Bool) \rightarrow list(Bool) \rightarrow Bool \rightarrow \circ$$

$$M_4 \quad \Gamma \quad (e, \sigma) : nil \quad S : nil \quad S \Leftrightarrow M_3 \quad \Gamma \quad (e, \sigma) : nil \quad S : nil$$

We have the following rules for conjunction, constants and application:

$$\frac{M_4 \quad \Gamma \quad (e, \psi_1) : (e, \psi_2) : And : C \quad S \quad R}{M_4 \quad \Gamma \quad (e, \psi_1 \wedge \psi_2) : C \quad S \quad R}$$

$$\frac{M_4 \quad \Gamma \quad C \quad True : S \quad R}{M_4 \quad \Gamma \quad (c, \mathbf{t}) : C \quad S \quad R}$$

$$\frac{M_4 \quad \Gamma \quad C \quad False : S \quad R}{M_4 \quad \Gamma \quad (c, \mathbf{f}) : C \quad S \quad R}$$

$$\frac{M_4 \quad \Gamma \quad (e_1, (\Gamma, e_2) \rightarrow \psi) : C \quad S \quad R}{M_4 \quad \Gamma \quad (e_1 e_2, \psi) : C \quad S \quad R}$$

In addition we need a rule defining the behaviour of *And* and an axiom for the terminal case:

$$\frac{M_4 \quad \Gamma \quad C \quad (S_1 \text{ and } S_2) : S \quad R}{M_4 \quad \Gamma \quad And : C \quad S_1 : S_2 : S \quad R}$$

$$M_4 \quad \Gamma \quad nil \quad R : nil \quad R$$

The end result is that we now have an inference system which is an *Abstract Evaluation System* in the terminology of [18]. This means that we can alternatively present it as a rewriting system describing a machine. We just have to rewrite any rule:

$$\frac{M_4 \quad \Gamma' \quad C' \quad S' \quad R'}{M_4 \quad \Gamma \quad {}_g C \quad S \quad R}$$

as:

$$\langle \Gamma, C, S \rangle \triangleright \langle \Gamma', C', S' \rangle$$

Applying this technique to each rule in the lazy types system, and rearranging the order of the arguments, we get the rules for the algorithm defined in Fig. 6. The *Inf* operation computes the  $\leq$  predicate and is defined in Fig. 7; the *D* operation is used to clean up the environment.

Let us now turn to the implementation of the rule for fixed point. The typing of fixed points has to be an iterative process. Suppose that the goal is to prove that **fix**  $\lambda g.e$  has type  $\phi$ . The simplest sub-proof that would allow us to succeed would be one that proves  $\lambda g.e$  has type  $\phi \rightarrow \phi$ ; this in turn follows from a proof that  $e$  has type  $\phi$  under the assumption that  $g$  also has type  $\phi$ . Here there is a problem: the latter proof might fail because  $g$  is required to have a type  $\phi \wedge \psi$  in order to prove that  $e$  has type  $\phi$ . In other words, the assumption on  $g$  has to be strengthened. This justifies the rule for *Rec*( $g, \psi$ ) in Fig. 6. Section 5 contains an example illustrating the iteration involved in the treatment of recursion. Let us mention at this point that we do not consider embedded occurrences of **fix** here. The extension is straightforward but would introduce unnecessary complications in the presentation.

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

**THEOREM 4.1**

1.  $\langle S, \Gamma, (e, \phi) : C \rangle \triangleright_G^* \langle \text{True} : S, \Gamma, C \rangle \Leftrightarrow \Gamma \vdash_G e : \phi$
2.  $\langle S, \Gamma, (e, \phi) : C \rangle \triangleright_G^* \langle \text{False} : S, \Gamma, C \rangle \Leftrightarrow \neg(\Gamma \vdash_G e : \phi)$   
*if  $\Gamma$  and  $\phi$  do not contain any  $\mapsto_r$  assumption*

The proof of this theorem is simultaneous with the proof of the following result:

**THEOREM 4.2**

1.  $\langle S, \Gamma, \text{Inf}(\phi, \psi) : C \rangle \triangleright_G^* \langle \text{True} : S, \Gamma, C \rangle \Leftrightarrow \phi \leq_G \psi$
2.  $\langle S, \Gamma, \text{Inf}(\phi, \psi) : C \rangle \triangleright_G^* \langle \text{False} : S, \Gamma, C \rangle \Leftrightarrow \neg(\phi \leq_G \psi)$   
*if  $\Gamma$ ,  $\phi$  and  $\psi$  do not contain any  $\mapsto_r$  assumption*

The restrictions on the  $\mapsto_r$  assumptions just make the statement of the theorems simpler. A more general property holds in the presence of assumptions on the recursive function. 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. It is easy to show (by induction on the length of the proof) that the result is accurate when the iteration terminates with the *True* answer. The proof that the initial property cannot be satisfied if the answer is *False* is also made by induction on the length of the reduction. Termination is proved by showing that the rule for *Rec* is always applied in a context where  $\phi \wedge \psi <_G \phi$ .

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.<sup>10</sup>

$$\begin{array}{ll}
\langle S, E, (c, \mathbf{t}) : C \rangle & \triangleright_G \langle \text{True} : S, E, C \rangle \\
\langle S, E, (c, \mathbf{f}) : C \rangle & \triangleright_G \langle \text{False} : S, E, C \rangle \\
\\
\langle S, E, (e, \phi_1 \wedge \phi_2) : C \rangle & \triangleright_G \langle S, E, (e, \phi_1) : (e, \phi_2) : \text{And} : C \rangle \\
\langle S, E, (\lambda x. e, \sigma \rightarrow \tau) : C \rangle & \triangleright_G \langle S, (x : \sigma) : E, (e, \tau) : D(x) : C \rangle \\
\langle S, E, (e_1 e_2, \phi) : C \rangle & \triangleright_G \langle S, E, (e_1, (E, e_2) \rightarrow \phi) : C \rangle \\
\langle S, E[x \mapsto \phi], (x, \psi) : C \rangle & \triangleright_G \langle S, E[x \mapsto \phi], \text{Inf}(\phi, \psi) : C \rangle \\
\langle S, E, (\mathbf{cond}(e_1, e_2, e_3), \phi) : C \rangle & \triangleright_G \langle S, E, (e_1, \mathbf{f}) : (e_2, \phi) : (e_3, \phi) : \text{And} : \text{Or} : C \rangle \\
\langle S, (x : \sigma) : E, (D(x)) : C \rangle & \triangleright_G \langle S, E, C \rangle \\
\\
\langle S, E, (\mathbf{fix}(\lambda g. e), \phi) : C \rangle & \triangleright_G \langle S, (g :_r (e, \phi)) : E, (e, \phi) : D(g) : C \rangle \\
\langle S, E[g \mapsto_r (e, \phi)], (g, \psi) : C \rangle & \triangleright_G \langle S, E[g \mapsto_r (e, \phi)], \text{Inf}(\phi, \psi) : (\text{Rec}, g, \psi) : C \rangle \\
\langle \text{True} : S, E, (\text{Rec}, g, \psi) : C \rangle & \triangleright_G \langle \text{True} : S, E, C \rangle \\
\langle \text{False} : S, E[g \mapsto (e, \phi)], (\text{Rec}, g, \psi) : C \rangle & \triangleright_G \langle S, (g :_r (e, \phi \wedge \psi)) : E[g \mapsto (e, \phi)], (e, \psi) : D(g) : C \rangle \\
\\
\langle S, E, (\mathbf{hd}(e), \mathbf{t}) : C \rangle & \triangleright_G \langle \text{True} : S, E, C \rangle \\
\langle S, E, (\mathbf{hd}(e), \phi) : C \rangle & \triangleright_G \langle S, E, (e, \mathbf{f}) : C \rangle \\
\\
\langle S, E, (\mathbf{tl}(e), \mathbf{t}) : C \rangle & \triangleright_G \langle \text{True} : S, E, C \rangle \\
\langle S, E, (\mathbf{tl}(e), \mathbf{f}) : C \rangle & \triangleright_G \langle S, E, (e, \mathbf{f}) : C \rangle \\
\langle S, E, (\mathbf{tl}(e), \infty) : C \rangle & \triangleright_G \langle S, E, (e, \infty) : C \rangle \\
\langle S, E, (\mathbf{tl}(e), \mathbf{f}_\infty) : C \rangle & \triangleright_G \langle S, E, (e, \infty) : C \rangle \\
\\
\langle S, E, (\mathbf{cons}(e_1, e_2), \mathbf{t}) : C \rangle & \triangleright_G \langle \text{True} : S, E, C \rangle \\
\langle S, E, (\mathbf{cons}(e_1, e_2), \infty) : C \rangle & \triangleright_G \langle S, E, (e_2, \infty) : C \rangle \\
\langle S, E, (\mathbf{cons}(e_1, e_2), \mathbf{f}_\infty) : C \rangle & \triangleright_G \langle S, E, (e_1, \mathbf{f}) : (e_2, \mathbf{f}_\infty) : \text{Or} : C \rangle \\
\langle S, E, (\mathbf{cons}(e_1, e_2), \mathbf{f}) : C \rangle & \triangleright_G \langle \text{False} : S, E, C \rangle \\
\\
\langle S, E, (\mathbf{case}(e_1, e_2, e_3), \phi) : C \rangle & \triangleright_G \\
\langle S, E, (e_3, \mathbf{f}) : (e_2, \mathbf{t} \rightarrow \infty \rightarrow \phi) : (e_3, \infty) : \text{And} : (e_2, \mathbf{t} \rightarrow \mathbf{f}_\infty \rightarrow \phi \wedge \mathbf{f} \rightarrow \mathbf{t} \rightarrow \phi) : (e_3, \mathbf{f}_\infty) : \\
\text{And} : (e_1, \phi) : (e_2, \mathbf{t} \rightarrow \mathbf{t} \rightarrow \phi) : \text{And} : \text{Or} : \text{Or} : \text{Or} : C \rangle & \\
\\
\langle S_1 : S_2 : S, E, \text{Op} : C \rangle & \triangleright_G \langle (\text{Op } S_1 \ S_2) : S, E, C \rangle \\
& \text{Op} = \text{And} \quad \text{or} \quad \text{Op} = \text{Or}
\end{array}$$

Figure 6: The Lazy Types algorithm

$$\begin{array}{ll}
\langle S, E, \text{Inf}(\mathbf{f}, \phi) : C \rangle & \triangleright_G \langle \text{True} : S, E, C \rangle \\
\langle S, E, \text{Inf}(\phi, \phi) : C \rangle & \triangleright_G \langle \text{True} : S, E, C \rangle \\
\langle S, E, \text{Inf}(\infty, \mathbf{f}_{\in}) : C \rangle & \triangleright_G \langle \text{True} : S, E, C \rangle \\
\langle S, E, \text{Inf}(\phi, \mathbf{t}) : C \rangle & \triangleright_G \langle \text{True} : S, E, C \rangle \\
\langle S, E, \text{Inf}(\phi_1 \rightarrow \dots \rightarrow \phi_n \rightarrow \phi, \psi_1 \rightarrow \dots \rightarrow \psi_n \rightarrow \mathbf{t}) : C \rangle & \triangleright_G \langle \text{True} : S, E, C \rangle \\
\langle S, E, \text{Inf}(\phi_1 \wedge \dots \wedge \phi_n, \psi_1 \wedge \dots \wedge \psi_m) : C \rangle & \triangleright_G \\
\langle S, E, \text{Inf}(\phi_1, \psi_1) : \dots : \text{Inf}(\phi_n, \psi_1) : \text{Or} : \dots \text{Inf}(\phi_1, \psi_m) : \dots : \text{Inf}(\phi_n, \psi_m) : \text{Or} : \text{And} : C \rangle & \\
\langle S, E, \text{Inf}(\psi, (\Gamma, e)) : C \rangle & \triangleright_G \\
\langle S, \Gamma, (e, \phi_1) : \text{Cond}(\text{False}, \text{True}) : \text{Inf}(\psi, \phi_1) : \dots (e, \phi_k) : \text{Cond}(\text{False}, \text{True}) : \text{Inf}(\psi, \phi_k) : \text{And} : \text{Setenv}(E) : C \rangle & \\
& \text{with } \phi_1, \dots, \phi_k \text{ the } T_S \text{ types} \\
& \text{compatible with the standard type of } e. \\
\\
\langle S, E, \text{Inf}((\Gamma, e), \phi) : C \rangle & \triangleright_G \langle S, \Gamma, (e, \phi) : \text{Setenv}(E) : C \rangle \\
& \text{with } \phi \neq (\Gamma', e') \\
\\
\langle S, E, \text{Inf}(\phi \rightarrow \psi, \phi' \rightarrow \psi') : C \rangle & \triangleright_G \langle S, E, \text{Inf}(\phi', \phi) : \text{Inf}(\psi, \psi') : \text{And} : C \rangle \\
\langle S, E, \text{Setenv}(E') : C \rangle & \triangleright_G \langle S, E', C \rangle \\
\langle B_1 : S, E, \text{Cond}(B_1, B_2) : C_0 : C \rangle & \triangleright_G \langle B_2 : S, E, C \rangle \\
\langle B'_1 : S, E, \text{Cond}(B_1, B_2) : C_0 : C \rangle & \triangleright_G \langle S, E, C_0 : C \rangle \\
& \text{with } B_1 \neq B'_1
\end{array}$$

Figure 7: Implementation of Inf

These optimisations are easy to justify formally and improve the efficiency of the resultant algorithm considerably.

## 5 Examples

This section describes the lazy types algorithm at work on two examples. The first one illustrates the iterative process involved in the treatment of recursion and the second one involves higher-order functions and lists.

### 5.1 Recursion

The following function was used in [26] to demonstrate the limitations of a type system without conjunction.

$$\mathbf{fix}(\lambda g.(\lambda x.\lambda y.\lambda z.\mathbf{cond}(eq\ z\ 0)(+ x\ y)(g\ y\ x\ (-\ z\ 1))))$$

We show how the lazy type algorithm is able to derive that this function is strict in its first argument, so has type  $T_1 = \mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{t} \rightarrow \mathbf{f}$ . The derivation is shown below. This example illustrates the implementation of **fix**: first the assumption  $g :_r T_1$  is added to the environment and the property to prove is  $(E, T_1)$ . The assumption is not strong enough to prove the required property ( $Inf(T_1, T_2)$  fails with  $T_2 = \mathbf{t} \rightarrow \mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{f}$ ). So  $T_2$  is added to the current type of  $g$  in the environment. This is because it is necessary to prove that the function is strict in its second argument to show that it is strict in its first argument. The second iteration step succeeds in proving  $(E', T_2)$  from the assumption  $(g : (T_1 \wedge T_2))$  and the final result is *True* as expected.

We use the following notation:

$$G = \mathbf{fix}(\lambda g.(\lambda x.\lambda y.\lambda z.\mathbf{cond}(eq\ z\ 0)(+ x\ y)(g\ y\ x\ (-\ z\ 1))))$$

$$E = \mathbf{cond}(eq\ z\ 0)(+ x\ y)(g\ y\ x\ (-\ z\ 1))$$

$$E' = (\lambda x.\lambda y.\lambda z.E)$$

$$T_1 = (\mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{t} \rightarrow \mathbf{f})$$

$$T_2 = (\mathbf{t} \rightarrow \mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{f})$$

In the development of the examples we omit the use of *nil* at the end of lists (representing the environment, the stack or the code) for the sake of conciseness. No ambiguity arises from this abuse of notation.

We show how the property  $G : T_1$  is proved by the lazy types algorithm:

$$\langle nil, nil, (G, T_1) \rangle \triangleright_G^*$$

$$\langle nil, (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r (E', T_1)),$$

$$(E, \mathbf{f}) : D(z) : D(y) : D(x) : D(g) \rangle \triangleright_G^*$$

$$\langle nil, (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r (E', T_1)),$$

$$((eq\ z\ 0), \mathbf{f}) : ((+ x\ y), \mathbf{f}) : ((g\ y\ x\ (-\ z\ 1)), \mathbf{f}) : And : Or : D(z) : D(y) : D(x) : D(g) \rangle \triangleright_G^*$$

$$\langle True : False, (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r (E', T_1)),$$

$$((g\ y\ x\ (-\ z\ 1)), \mathbf{f}) : And : Or : D(z) : D(y) : D(x) : D(g) \rangle \triangleright_G^*$$

$$\begin{aligned}
& \langle \text{True} : \text{False}, (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r (E', T_1)), \\
& ((g, T_2) : \text{And} : \text{Or} : D(z) : D(y) : D(x) : D(g)) \rangle \quad \triangleright_G^* \\
& \langle \text{True} : \text{False}, (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r (E', T_1)), \\
& (\text{Inf}(T_1, T_2) : (\text{Rec}, g, T_2) : \text{And} : \text{Or} : D(z) : D(y) : D(x) : D(g)) \rangle \quad \triangleright_G^* \\
& \langle \text{False} : \text{True} : \text{False}, (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r (E', T_1)), \\
& (\text{Rec}, g, T_2) : \text{And} : \text{Or} : D(z) : D(y) : D(x) : D(g)) \rangle \quad \triangleright_G^* \\
& \langle \text{True} : \text{False}, (g :_r (E', T_1 \wedge T_2)) : (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r (E', T_1)), \\
& (E', T_2) : D(g) : \text{And} : \text{Or} : D(z) : D(y) : D(x) : D(g)) \rangle \quad \triangleright_G^* \\
& \langle \text{True} : \text{False}, (z : \mathbf{t}) : (y : \mathbf{f}) : (x : \mathbf{t}) : (g :_r (E', T_1 \wedge T_2)) : (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r (E', T_1)), \\
& (E, \mathbf{f}) : D(z) : D(y) : D(x) : D(g) : \text{And} : \text{Or} : D(z) : D(y) : D(x) : D(g)) \rangle \quad \triangleright_G^* \\
& \langle \text{True} : \text{False} : \text{True} : \text{False}, (z : \mathbf{t}) : (y : \mathbf{f}) : (x : \mathbf{t}) : (g :_r (E', T_1 \wedge T_2)) : (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r (E', T_1)), \\
& (g, T_1) : \text{And} : \text{Or} : D(z) : D(y) : D(x) : D(g) : \text{And} : \text{Or} : D(z) : D(y) : D(x) : D(g)) \rangle \quad \triangleright_G^* \\
& \langle \text{True} : \text{True} : \text{False} : \text{True} : \text{False}, (z : \mathbf{t}) : (y : \mathbf{f}) : (x : \mathbf{t}) : (g :_r (E', T_1 \wedge T_2)) : (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r (E', T_1)), \\
& \text{And} : \text{Or} : D(z) : D(y) : D(x) : D(g) : \text{And} : \text{Or} : D(z) : D(y) : D(x) : D(g)) \rangle \quad \triangleright_G^* \\
& \langle \text{True}, \text{nil}, \text{nil} \rangle
\end{aligned}$$

## 5.2 Higher-order and lists

We consider the following functions:

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

which were introduced in [21] 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:

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

Similarly *cat* should also be defined as a  $\lambda$ -abstraction. We use the following notation:

$$\phi = \mathbf{t} \rightarrow ((l : \mathbf{f}), \text{append}) \rightarrow \mathbf{f} \rightarrow \mathbf{f}$$

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

We describe some of the derivation steps of the lazy type algorithm to prove that *cat* has type  $\mathbf{f} \rightarrow \mathbf{f}$ .

$$\begin{aligned}
& \langle \text{nil}, \text{nil}, (\text{cat}, \mathbf{f} \rightarrow \mathbf{f}) \rangle \quad \triangleright_G \\
& \langle \text{nil}, (l : \mathbf{f}), (\text{foldr } \mathbf{nil} \ \text{append } l, \mathbf{f}) : D(l) \rangle \quad \triangleright_G \\
& \langle \text{nil}, (l : \mathbf{f}), (\text{foldr } \mathbf{nil} \ \text{append}, \mathbf{f} \rightarrow \mathbf{f}) : D(l) \rangle \quad \triangleright_G \\
& \langle \text{nil}, (l : \mathbf{f}), (\text{foldr } \mathbf{nil}, ((l : \mathbf{f}), \text{append}) \rightarrow \mathbf{f} \rightarrow \mathbf{f}) : D(l) \rangle \quad \triangleright_G \\
& \langle \text{nil}, (l : \mathbf{f}), (\text{foldr}, \mathbf{t} \rightarrow ((l : \mathbf{f}), \text{append}) \rightarrow \mathbf{f} \rightarrow \mathbf{f}) : D(l) \rangle \quad \triangleright_G^* \\
& \langle \text{nil}, (l : \mathbf{f}) : (g, ((l : \mathbf{f}), \text{append})) : (b : \mathbf{t}) : (f :_r (E, \phi)) : (l : \mathbf{f}), \\
& (\mathbf{case} \dots, \mathbf{f}) : D(l) : D(g) : D(b) : D(f) : D(l) \rangle \quad \triangleright_G^* \\
& \langle \text{nil}, \dots, (l, \mathbf{f}) : \dots : \text{Or} : D(l) : D(g) : D(b) : \dots \rangle \quad \triangleright_G^* \\
& \langle \text{True}, \dots, D(l) : D(g) : D(b) : \dots \rangle \quad \triangleright_G^* \\
& \langle \text{True}, (f :_r (E, \phi)) : (l : \mathbf{f}), D(f) : D(l) \rangle \quad \triangleright_G \\
& \langle \text{True}, (l : \mathbf{f}), D(l) \rangle \quad \triangleright_G \\
& \langle \text{True}, \text{nil}, \text{nil} \rangle
\end{aligned}$$

## 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 [35], 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

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

with:

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

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

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

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:

$$\begin{array}{ll} \text{Cons-2} & \frac{\Gamma \vdash_G e_2 : \sigma_\infty}{\Gamma \vdash_G \mathbf{cons}(e_1, e_2) : \sigma_\infty} \quad \text{Cons-3} \quad \frac{\Gamma \vdash_G e_1 : \sigma}{\Gamma \vdash_G \mathbf{cons}(e_1, e_2) : \sigma_\infty} \\ \text{Case-3} & \frac{\Gamma \vdash_G e_2 : \mathbf{t} \rightarrow \sigma_\infty \rightarrow \phi \wedge \sigma \rightarrow \mathbf{t} \rightarrow \phi \quad \Gamma \vdash_G e_3 : \sigma_\infty}{\Gamma \vdash_G \mathbf{case}(e_1, e_2, e_3) : \phi} \end{array}$$

and the ordering on types is extended with the rules:

$$\infty \leq \sigma_\infty \quad \frac{\sigma \leq \tau}{\sigma_\infty \leq \tau_\infty}$$

The extensions to the algorithm are not described here for the sake of brevity. 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 *Inf* in Fig. 7. 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  $\sigma$ , 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  $\Lambda_L$ :

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

We want to prove  $\mathit{cat} : \infty_\infty \rightarrow \infty$  which requires a proof of  $\mathit{foldr} : \mathbf{t} \rightarrow \mathit{append} \rightarrow \infty_\infty \rightarrow \infty$ , where *append* is used as a shorthand notation for  $(\mathit{nil}, \mathit{append})$ . We do not give all of the details of the derivation but rather focus on the main steps of the proof:

$$\begin{array}{c}
\vdots \\
A \quad B \\
\hline
\text{Conj} \quad \frac{\Gamma \vdash (\lambda x. \lambda xs. g \ x \ (f \ b \ g \ xs)) : (\mathbf{t} \rightarrow \infty_{\epsilon} \rightarrow \infty) \wedge (\infty \rightarrow \mathbf{t} \rightarrow \infty)}{\Gamma \vdash \text{case}(b, \lambda x \lambda xs. g \ x \ (f \ b \ g \ xs), l) : \infty} \quad C \\
\text{Case - 3} \\
\hline
\vdots \\
\text{Abs} \quad \frac{\vdash \lambda f. \lambda b. \lambda g. \lambda l. \text{case}(b, \lambda x \lambda xs. g \ x \ (f \ b \ g \ xs), l) : (\mathbf{t} \rightarrow \text{append} \rightarrow \infty_{\epsilon} \rightarrow \infty) \rightarrow (\mathbf{t} \rightarrow \text{append} \rightarrow \infty_{\epsilon} \rightarrow \infty)}{\vdash \text{fix}(\lambda f. \lambda b. \lambda g. \lambda l. \text{case}(b, \lambda x \lambda xs. g \ x \ (f \ b \ g \ xs), l)) : \mathbf{t} \rightarrow \text{append} \rightarrow \infty_{\epsilon} \rightarrow \infty} \\
\text{Fix} \\
\hline
\vdash \text{foldr} : \mathbf{t} \rightarrow \text{append} \rightarrow \infty_{\epsilon} \rightarrow \infty
\end{array}$$

where  $\Gamma$  is:  $[f : \mathbf{t} \rightarrow \text{append} \rightarrow \infty_{\epsilon} \rightarrow \infty, b : \mathbf{t}, g : \text{append}, l : \infty_{\epsilon}]$ .  
and  $A$  is:

$$\begin{array}{c}
\vdots \\
\Gamma'' \vdash f \ b \ g \ xs : \infty \\
\hline
(\Gamma'', f \ b \ g \ xs) \leq \infty \\
\hline
\Gamma' \vdash \lambda x. \lambda xs. \text{cons}(x, (\text{app } xs \ x_2)) : \mathbf{t} \rightarrow \mathbf{t} \rightarrow \infty \quad \Gamma' \vdash x_2 : \infty \\
\hline
\text{Case - 4} \quad \Gamma' \vdash \text{case}(x_2, \lambda x. \lambda xs. \text{cons}(x, (\text{app } xs \ x_2)), x_1) : \infty \\
\hline
\vdots \\
\text{App} \quad \frac{\Gamma'' \vdash g : \mathbf{t} \rightarrow (f \ b \ g \ xs) \rightarrow \infty}{\Gamma'' \vdash g \ x : (f \ b \ g \ xs) \rightarrow \infty} \\
\text{App} \quad \frac{\Gamma'' \vdash g \ x : (f \ b \ g \ xs) \rightarrow \infty}{\Gamma'' \vdash g \ x \ (f \ b \ g \ xs) : \infty} \\
\text{App} \\
\hline
\vdots \\
\text{Abs} \quad \frac{\Gamma \vdash (\lambda x. \lambda xs. g \ x \ (f \ b \ g \ xs)) : (\mathbf{t} \rightarrow \infty_{\epsilon} \rightarrow \infty)}{\Gamma \vdash (\lambda x. \lambda xs. g \ x \ (f \ b \ g \ xs)) : (\mathbf{t} \rightarrow \infty_{\epsilon} \rightarrow \infty)}
\end{array}$$

where

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

the proof tree for  $B$  is similarly constructed and  $C$  is  $\Gamma \vdash l : \infty_{\epsilon}$ . So the domain is explored up to depth 2 ( $D_2$ ). If we now ask the question  $\text{foldr} : \mathbf{t} \rightarrow \text{append} \rightarrow \mathbf{f}_{\epsilon} \rightarrow \infty$ , 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 PER's and binding time analysis

Strictness analysis was the original motivation for the study of lazy types but the techniques presented in this paper are more generally applicable. In [16] we propose a methodology for defining analyses based on these ideas. We just provide the main intuition here and we show how the framework can be specialised to PER models and binding time analysis.

We assume some sets of type constants  $B$  which are pre-ordered by  $\leq$  and type constructors including  $\wedge$  (intersection or conjunction) and  $\rightarrow$  (functions). The following definition allows us to formalise the notion of property over some standard domain of discourse  $D$ .

**DEFINITION 7.1** *A type structure  $\mathcal{M}$  is a tuple  $(X, \sqsubseteq, \sqcap, \Rightarrow, \text{norm})$ , where*

- $(X, \sqsubseteq)$  is a cpo of properties including interpretations for the constants.
- $\sqcap : X \times X \rightarrow X$  is the greatest lowest bound operation (used to interpret intersection).
- $\Rightarrow : X \times X \rightarrow X$  interprets  $\rightarrow$ .
- $\text{norm} : X \rightarrow \wp(D)$  maps any property to its underlying set of domain elements.



$\sqsubseteq, \Rightarrow$  and *norm* must satisfy:

$f \in \text{norm}(x \Rightarrow y)$  if and only if  $\forall a. a \in \text{norm}(x)$  implies  $f a \in \text{norm}(y)$

$x \sqsubseteq y$  implies  $\text{norm}(x) \subseteq \text{norm}(y)$

Given a particular structure,  $\mathcal{M}$  and an interpretation of the type constants  $\mathcal{I} : B \rightarrow X$  we denote the interpretation of  $\sigma$  by  $\llbracket \sigma \rrbracket^{\mathcal{M}, \mathcal{I}}$  or just  $\llbracket \sigma \rrbracket$  if  $\mathcal{M}, \mathcal{I}$  is clear from the context.

**DEFINITION 7.2** *The structure is a model, if for all  $\phi$  and  $\psi$ :*

$$\phi \leq \psi \text{ implies } \llbracket \phi \rrbracket \sqsubseteq \llbracket \psi \rrbracket$$

There are a number of representations of properties which have been used in the literature. In each case there is usually a “natural” interpretation for the operators  $\sqcap, \Rightarrow$  and *norm* which, together with interpretations for constants, gives a type structure (see below). If we use one of these standard structures, Burn, [5], has shown that if the above implication holds for the type constants then it also holds for the derived types; this gives a “local” test to determine if a structure is a model. We choose here to illustrate type structures with the CPER (Complete Partial Equivalence Relations) model. A PER on a set  $D$  is a binary relation which is symmetric and transitive. A PER,  $P$ , is *strict* if:

$$- P -$$

and *inductive* if and only if whenever for all matching elements of the chains  $\{x_n\}_{n \in \omega}$  and  $\{y_n\}_{n \in \omega}$ ,  $x_i P y_i$ :

$$\bigsqcup_{n \in \omega} x_n P \bigsqcup_{n \in \omega} y_n$$

A complete PER is a strict and inductive PER. The motivation for using CPERs is that certain properties which cannot be represented by Scott-closed sets can be represented by CPERs. Hunt and Sands have used CPERs in binding time analysis [22].

Let  $\mathcal{CPER}(D)$  be the set of CPERs on  $D$ . We define the CPER structure as follows:

$$\mathcal{M}_{\text{cper}} = (\mathcal{CPER}(D), \sqsubseteq_{\text{cper}}, \sqcap_{\text{cper}}, \Rightarrow_{\text{cper}}, \text{norm}_{\text{cper}})$$

where

- $\sqsubseteq_{\text{cper}} = \subseteq$  (set inclusion)
- $\sqcap_{\text{cper}} = \cap$  (set intersection)
- $f (Q \Rightarrow_{\text{cper}} R) g \equiv [\forall q q'. q Q q' \Rightarrow (f q) R (g q')]$
- $\text{norm}_{\text{cper}}(P) = \{x \in D \mid x P x\}$

The requirements of Definition 7.1 are trivially satisfied. Let us note that the structure is not tied to one particular interpretation of constants. In particular, it can be used for strictness analysis as well as for binding time analysis. We illustrate the CPER structure with binding time analysis in the rest of this section.

First we introduce some notation.

**DEFINITION 7.3** *We use the following notation:*

- $d \models_{el} \phi \equiv d \in \text{norm}(\llbracket \phi \rrbracket)$
- $\rho \models_{el} \Gamma \equiv \forall x. \rho x \models_{el} \Gamma x$
- $\Gamma \models e : \phi \equiv \forall \rho \models_{el} \Gamma. \mathcal{S}[\llbracket e \rrbracket] \rho \models_{el} \phi$  where  $\mathcal{S}_\Gamma$  is the standard denotational semantics.

DEFINITION 7.4 *A rule for an  $n$ -ary constant:*

$$\mathbf{Const} \quad \frac{\Gamma \vdash_T e_1 : \phi_1 \dots \Gamma \vdash_T e_n : \phi_n}{\Gamma \vdash_T c \ e_1 \ \dots \ e_n : \phi}$$

is sound if, under the assumption that:

$$\Gamma \vdash_T e_i : \phi_i \text{ implies } \Gamma \models e_i : \phi_i$$

then  $\Gamma \models c \ e_1 \ \dots \ e_n : \phi$ .

In our earlier work, [16], we extend a result of Burn's, [5], and show that soundness of the constant rules ensures soundness of the logic. This gives a local correctness condition. In [16], there is an analogous result concerning soundness of the abstract machine.

Binding time analysis is an analysis which is used in partial evaluation systems to determine which parts of a program depend solely on values that are known at partial evaluation time (so-called “static” values); these parts of the program are candidates for specialisation. We first summarise the list of tasks identified in [16] in order to set up a correct instance of the generic analysis:

1. Define the list of constants of the language.
2. Define the list of type constants.
3. Provide a type structure and an interpretation for type constants. Show that the structure yields a model.
4. Define the type inference rules for the language constants and check the local correctness conditions.
5. Provide the rules stating the treatment of the constants by the abstract machine and check the correctness condition.

Let us now realise this programme for binding time analysis.

## 7.1 Constants of the language

For the sake of conciseness we just consider basic constants  $c$  and two functional constants:  $+$  and the conditional. Other operators would be treated in a similar way.

## 7.2 Type constants

There are two type constants *static* and *dynamic* with *static*  $\leq$  *dynamic*.

## 7.3 Type structure

We model constants as PERs in the following way:

$$\begin{aligned} I_{cper}(\textit{static}) &= \{(x, x) \mid x \in D\} \quad (= Id) \\ I_{cper}(\textit{dynamic}) &= \{(x, y) \mid x, y \in D\} \quad (= All) \end{aligned}$$

It is straightforward to verify that  $I_{cper}(\textit{static}) \subseteq I_{cper}(\textit{dynamic})$  and thus:

$$\phi \leq \psi \text{ implies } \llbracket \phi \rrbracket \sqsubseteq \llbracket \psi \rrbracket$$

and the structure is a model.

## 7.4 Type inference rules

All constants of base type are static, we thus have the following axiom:

$$\mathbf{Const} \quad \vdash_G c : \text{static}$$

which is sound since  $\text{norm}(\llbracket \text{static} \rrbracket)$  is  $D$ . The rule scheme for  $+$  and the conditional are respectively:

$$\begin{aligned} + \quad & \frac{\Gamma \vdash_G e_1 : \text{static} \quad \Gamma \vdash_G e_2 : \text{static}}{\Gamma \vdash_G +(e_1, e_2) : \text{static}} \\ & + \quad \frac{\Gamma \vdash_G e_1 : \phi \quad \Gamma \vdash_G e_2 : \psi}{\Gamma \vdash_G +(e_1, e_2) : \text{dynamic}} \\ \mathbf{Cond} \quad & \frac{\Gamma \vdash_G b : \text{static} \quad \Gamma \vdash_G e_1 : \phi \quad \Gamma \vdash_G e_2 : \phi}{\Gamma \vdash_G \mathbf{if } b \mathbf{ then } e_1 \mathbf{ else } e_2 : \phi} \\ \mathbf{Cond} \quad & \Gamma \vdash_G \mathbf{if } b \mathbf{ then } e_1 \mathbf{ else } e_2 : \text{dynamic} \end{aligned}$$

The correctness of the rule for  $+$  is obvious. We illustrate the correctness proof with the first rule for the conditional. By assumption we have:

$$\begin{aligned} \text{Expand}(\Gamma) & \models b : \text{static} \\ \text{Expand}(\Gamma) & \models e_1 : \text{Expand}(\phi) \\ \text{Expand}(\Gamma) & \models e_2 : \text{Expand}(\phi) \end{aligned}$$

Now if  $S\llbracket b \rrbracket\rho$  is  $-$  then  $S\llbracket \mathbf{if } b \mathbf{ then } e_1 \mathbf{ else } e_2 \rrbracket\rho = -$  and thus:

$$\text{Expand}(\Gamma) \models \mathbf{if } b \mathbf{ then } e_1 \mathbf{ else } e_2 : \text{Expand}(\phi)$$

since  $\text{Expand}(\phi)$  is a CPER. If  $S\llbracket b \rrbracket\rho \neq -$  then the soundness result is immediate.

## 7.5 Transition Rules

We add the new rules shown in Fig. 8.

$\langle S, E, \text{Inf}(\text{static}, \text{dynamic}) : C \rangle$	$\triangleright_G$	$\langle \text{True} : S, E, C \rangle$
$\langle S, E, (c, \text{static}) : C \rangle$	$\triangleright_G$	$\langle \text{True} : S, E, C \rangle$
$\langle S, E, (c, \text{dynamic}) : C \rangle$	$\triangleright_G$	$\langle \text{True} : S, E, C \rangle$
$\langle S, E, +(e_1, e_2), \text{static} \rangle : C$	$\triangleright_G$	
$\langle S, E, (e_1, \text{static}) : (e_2, \text{static}) : \text{And} : C \rangle$		
$\langle S, E, +(e_1, e_2), \text{dynamic} \rangle : C$	$\triangleright_G$	$\langle \text{True} : S, E, C \rangle$
$\langle S, E, (\mathbf{if } e_1 \mathbf{ then } e_2 \mathbf{ else } e_3, \phi) : C \rangle$	$\triangleright_G$	
$\langle S, E, (e_1, \text{static}) : (e_2, \phi) : (e_3, \phi) : \text{And} : \text{And} : C \rangle$		
$\langle S, E, (\mathbf{if } e_1 \mathbf{ then } e_2 \mathbf{ else } e_3, \text{dynamic}) : C \rangle$	$\triangleright_G$	$\langle \text{True} : S, E, C \rangle$

Figure 8: The new transitions for the binding time analysis algorithm

Since these rules are derived from the typing rules in a fairly direct manner, their correctness is immediate.

## 8 Experimental Results

The lazy types algorithm has been implemented (in CAML light) as an interpreter realising the abstract machine described in Fig. 6 and Fig. 7. We report here on experimental results. We use as a testbed two versions of a function concatenating lists of lists, the second one being defined in terms of continuations. These examples were provided by S. Hunt to illustrate the limitations of the frontiers optimisation [20, 21].

$foldr\ g\ \mathbf{nil}\ b$	$=\ b$
$foldr\ g\ \mathbf{cons}(x, xs)\ b$	$=\ g\ x\ (foldr\ g\ xs\ b)$
$append\ \mathbf{nil}\ l$	$=\ l$
$append\ \mathbf{cons}(x, xs)\ l$	$=\ \mathbf{cons}(x, (append\ xs\ l))$
$cat\ l$	$=\ foldr\ append\ l\ \mathbf{nil}$
$Cfoldr\ g\ \mathbf{nil}\ b\ C$	$=\ C\ b$
$Cfoldr\ g\ \mathbf{cons}(x, xs)\ b\ C$	$=\ Cfoldr\ g\ xs\ b\ (\lambda y. g\ x\ y\ C)$
$Cappend\ \mathbf{nil}\ l\ C$	$=\ C\ l$
$Cappend\ \mathbf{cons}(x, xs)\ l\ C$	$=\ Cappend\ xs\ l\ (\lambda y. C\ (\mathbf{cons}(x, y)))$
$Ccat\ l\ C$	$=\ Cfoldr\ Cappend\ l\ \mathbf{nil}\ C$
$K\ x\ y$	$=\ x$
$isnil\ \mathbf{nil}$	$=\ \mathbf{True}$
$isnil\ \mathbf{cons}(x, xs)$	$=\ \mathbf{False}$
$length\ \mathbf{nil}$	$=\ 0$
$length\ \mathbf{cons}(x, xs)$	$=\ 1 + (length\ xs)$
$sum\ \mathbf{nil}$	$=\ 0$
$sum\ \mathbf{cons}(x, xs)$	$=\ x + (length\ xs)$
$test_1\ l$	$=\ Ccat\ l\ (K\ 0)$
$test_2\ l$	$=\ Ccat\ l\ isnil$
$test_3\ l$	$=\ Ccat\ l\ length$
$test_4\ l$	$=\ Ccat\ l\ sum$

Figure 9: Testbed

The following table gathers some experimental results: we show for each property the answer provided by the algorithm (*True* or *False*) and the measured CPU execution time (the processor is a Sparc 2 IPX).

These results should be compared with other implementations. The contrast with frontiers based “optimisations” of abstract interpretation is striking: the analysis of [21] takes 30 minutes to process *cat* and does not terminate for examples involving *Ccat*. The basic reason is that abstract interpretation based analyses systematically compute all the properties satisfied by a function; when the function is higher-order this can involve a vast amount of information. It turns out that very often only a small part of this information is really necessary. It may be argued that for a fairer comparison we should add the execution times to compute the answers to all possible questions in the lazy type algorithm. Even so our algorithm performs much better (half a second for *cat*) than the frontiers based implementation; this is because it may not be necessary to compute total information about constituent higher-order functions.

Ferguson and Hughes, [12], report that the analysis of *cat* requires 5 seconds and the analysis of *Ccat* around 10 seconds. In comparison, our algorithm takes 0.5 second for *cat* and about 2

$cat : \mathbf{f}_\epsilon \rightarrow \mathbf{f}$	<i>False</i>	0.08 s
$cat : \infty \rightarrow \mathbf{f}$	<i>False</i>	0.17 s
$cat : \infty \rightarrow \infty$	<i>True</i>	0.02 s
$test_1 : \mathbf{f}_{\epsilon_\epsilon} \rightarrow \mathbf{f}$	<i>False</i>	0.2 s
$test_1 : \infty_\epsilon \rightarrow \mathbf{f}$	<i>True</i>	0.33 s
$test_2 : \mathbf{f}_{\epsilon_\epsilon} \rightarrow \mathbf{f}$	<i>False</i>	0.95 s
$test_2 : \infty_\epsilon \rightarrow \mathbf{f}$	<i>True</i>	3.97 s
$test_3 : \mathbf{f}_{\epsilon_\epsilon} \rightarrow \mathbf{f}$	<i>False</i>	2.9 s
$test_3 : \infty_\epsilon \rightarrow \mathbf{f}$	<i>True</i>	1.7 s
$test_4 : \mathbf{f}_{\epsilon_\epsilon} \rightarrow \mathbf{f}$	<i>True</i>	1.42 s
$test_4 : \infty_\epsilon \rightarrow \mathbf{f}$	<i>True</i>	0.37 s

Figure 10: Experimental results

seconds for *Ccat*. Furthermore their algorithm requires a huge amount of memory to execute. The reason seems to be that their analyser is based on a coding of abstract functions in terms of *concrete data structures* which take a lot of space.

The analyser described in [30] is an efficient implementation of abstract interpretation based on a representation of boolean functions as Typed Decision Graphs. It includes an implementation of the widening technique to accelerate fixed point iteration. The analysis of *cat* takes 4 seconds and the analysis of *Ccat* one hour.

It should be noted that our approach has the same worst-case complexity as these other approaches but the worst cases are different. We have not yet identified worst case examples for our approach.

## 9 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 [26, 28, 23, 7, 12, 32]. 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 [26, 28]. In contrast, [12, 32] use intensional representations of functions to build very efficient algorithms without sacrificing accuracy. The analysis of [12] uses concrete data structures; these are special kinds of Scott domains whose elements can be seen as syntax trees.

In [32] the analysis is expressed as a form of reduction of abstract graphs. As in our work, the abstract domain is infinite and computation is done lazily. There are important differences however. Their derivation strategy is even more lazy than ours in the following sense. Recasting their algorithm in terms of types, let us assume that in the course of trying to prove the property  $f : t_1 \rightarrow t_2 \rightarrow t_3$ , it turns out to be necessary to prove  $f : e_1 \rightarrow e_2 \rightarrow e_3$ . In the abstract graph reduction framework, the call to  $f$  is unfolded, which means, in terms of types, that we embark on a proof of  $f : e_1 \rightarrow e_2 \rightarrow e_3$  (except if  $f : e_1 \rightarrow e_2 \rightarrow e_3$  and  $f : t_1 \rightarrow t_2 \rightarrow t_3$  are syntactically equal) without any attempt to relate the types  $t_i$  and  $e_i$ . In contrast, the lazy type algorithm tries to prove  $t_1 \rightarrow t_2 \rightarrow t_3 \leq e_1 \rightarrow e_2 \rightarrow e_3$ , which means, in terms of graph reduction, that it may entail the evaluation of some of the arguments of the functions. The extremist view of laziness taken in abstract graph reduction has two consequences: on the plus side, it sometimes avoids the computation of information that would be computed by the lazy type system; the negative side is that it may entail more work in other cases and even non termination if some special measures are not taken. These extra measures can take the form of arbitrary cuts in the derivation (using empirical resource consumption criteria) incurring a loss of accuracy. A neededness analysis called reduction path analysis is also proposed in [32] to allow termination of the computation without throwing away too much information. Because of this parameterisable termination condition, it is difficult to formally qualify the power of abstract graph reduction. Another advantage of the lazy types approach is the fact that its correctness proof is much easier to establish (see [11] for an introduction to the complications involved by a formalisation of abstract graph reduction).

Another technique for improving the computation of fixed points is called *chaotic iteration*. It was introduced in [8] and extended to higher-order functional programs in [34]. The chaotic iteration starts with an initial set of arguments and each step computes a new version of the abstract function for some needed arguments. Several choices can be made for the selection of these arguments. The technique clearly bears some similarities with the analysis presented here: the initial set of arguments plays the rôle of the type in the initial query of the lazy types algorithm and the arguments selected at each step correspond to the types added to the current

assumption by the *Rec* instruction. The main departure of our algorithm is the lazy evaluation of types (as opposed to the eager evaluation of needed arguments in [34]). As an example the two algorithms exhibit different behaviours when applied to the following function:

$$\mathbf{fix}(\lambda f.(\lambda x.\lambda y.\lambda z.\mathbf{cond}(eq\ y\ 0)(+ y\ z)(f\ x\ z\ (f\ x\ z\ y))))$$

Assume that we want to decide whether this expression has type  $\mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{t} \rightarrow \mathbf{f}$ . Rephrased in terms of types, the chaotic iteration sequence described in [34] includes  $\mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{f} \rightarrow \mathbf{f}$  in the set of “needed” types. This type is not really required, it is called a *spurious element* in [34]. This element occurs because the chaotic iteration starts with the least abstract function in the domain (characterised by the type  $\mathbf{t} \rightarrow \mathbf{t} \rightarrow \mathbf{t} \rightarrow \mathbf{f}$ ). In contrast the lazy types algorithm returns *False* after the first iteration step. This can be seen as a difference in the strategy applied to approach the least fixed point: the chaotic iteration sequence reaches it “from below” starting with the strongest (but possibly wrong) assumption when the lazy types algorithm starts with the weakest assumption (the initial question) strengthening it if necessary. It is not clear however whether this variation in the strategy leads to a significantly different behaviour in practice.

An interesting avenue for further research would be to reexpress abstract graph reduction and chaotic fixed point iteration in terms of type inference as suggested here to be able to relate the techniques on a formal basis. As an aside this might also provide some insight for a simpler correctness proof of abstract graph reduction.

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, [24], 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.

## Acknowledgements

We are grateful to Valérie Gouranton and Ronan Gaugne for implementing the algorithm and conducting the experiments. We thank Sebastian Hunt for providing the benchmark examples. Pascal Fradet provided some valuable feedback on earlier versions of this paper. The first author is partially funded by ESPRIT Working Group 6809 (Semantique).

## 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. L. Burn, *A Logical Framework for Program Analysis*, in *Proceedings of the 1992 Glasgow Functional Programming Workshop*, Springer Verlag Workshops in Computer Science, 1992.
- [6] 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.
- [7] 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.
- [8] P. Cousot and R. Cousot, *Static determination of dynamic properties of recursive procedures*, in E. J. Neuhold (ed.), *Formal Description of Programming Concepts*, North-Holland, 1978.
- [9] 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.
- [10] O. Danvy and J. Hatcliff, *CPS transformation after strictness analysis*, Technical Report, Kansas State University, to appear in ACM LOPLAS.
- [11] M. van Eekelen, E. Goubault, C. Hankin and E. Nöcker, *Abstract reduction: a theory via abstract interpretation*, in R. Sleep et al (eds), *Term graph rewriting: theory and practice*, John Wiley & Sons Ltd, 1992.
- [12] A. Ferguson and R. J. M. Hughes, *Fast abstract interpretation using sequential algorithms*, in *Proceedings WSA '93*, LNCS 724, Springer Verlag, 1993.
- [13] 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.
- [14] 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.
- [15] C. L. Hankin and D. Le Métayer, *Deriving algorithms from type inference systems: Application to strictness analysis*, in *Proceedings of POPL'94*, ACM Press, 1994.
- [16] C. L. Hankin and D. Le Métayer, *A type-based framework for program analysis*, *Proceedings of the Static Analysis Symposium*, LNCS, Springer Verlag, 1994 (to appear).
- [17] J. J. Hannan, *Investigating a proof-theoretic meta-language*, PhD thesis, University of Pennsylvania, DIKU Technical Report Nr 91/1, 1991.
- [18] J. Hannan and D. Miller, *From Operational Semantics to Abstract Machines*, *Mathematical Structures in Computer Science*, 2(4), 1992.
- [19] 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.



- [20] L. S. Hunt, *Abstract Interpretation of Functional Languages: From Theory to Practice*, PhD thesis, Imperial College, 1991.
- [21] L. S. Hunt and C. L. Hankin, *Fixed Points and Frontiers: A New Perspective*, Journal of Functional Programming, 1(1), 1991.
- [22] L. S. Hunt and D. Sands, *Binding Time Analysis: A new PERSpective*, in Proceedings of the ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation, 1991.
- [23] 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.
- [24] T. P. Jensen, *Abstract Interpretation in Logical Form*, PhD thesis, University of London, 1992. Also available as DIKU Technical Report 93/11.
- [25] N. D. Jones and A. Mycroft, *Data-flow analysis of applicative programs using minimal function graphs*, in *Proceedings of the ACM Conference on Principles of Programming Languages*, 1986.
- [26] 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.
- [27] J. Launchbury, *Strictness and binding time: two for the price of one*, in *Proceedings of the ACM Conference on Programming Languages Design and Implementation*, 1991.
- [28] 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.
- [29] J. C. Mitchell, *Type inference with simple subtypes*, Journal of Functional Programming, 1(3), 1991.
- [30] L. Mauborgne, *Abstract interpretation using TDGs*, *Proceedings of the Static Analysis Symposium*, LNCS, Springer Verlag, 1994 (to appear).
- [31] A. Mycroft, *Abstract Interpretation and Optimising Transformations for Applicative Programs*, PhD thesis, University of Edinburgh, December 1981.
- [32] E. Nöcker, *Strictness analysis using abstract reduction*, in *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture*, ACM Press, 1993.
- [33] S. L. Peyton Jones and C. Clack, *Finding Fixed Points in Abstract Interpretation*, in S. Abramsky and C. L. Hankin (eds), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.
- [34] M. Rosendhal, *Higher-order chaotic iteration sequences*, in *Proceedings of the 5th Int. Symp. Programming Language Implementation and Logic Programming*, LNCS 714, Springer-Verlag, 1993.
- [35] P. Wadler, *Strictness Analysis on Non-flat Domains*, in S. Abramsky and C. L. Hankin (eds), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

- [36] 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.