Science of Computer Programming 25 (1995) 219-249

Science of Computer Programming

Lazy type inference and program analysis

Chris Hankin[a],*, Daniel Le Métayer[b]

[a] *Department of Computing, Imperial College, London SW7 2BZ, UK*
[b] *INRIA/IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France*

Abstract

Approaches to static analysis based on nonstandard 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 introduction of laziness into type inference. We present the basic notions in the context of a higher-order strictness analysis of list-processing functions. We also present 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 advanced techniques in compilers. The most significant contributions for improving the efficiency of abstract interpretation include widening techniques [9, 14], chaotic iteration sequences [8, 37] (and the related minimal function graphs [27]), and *frontiers*-based algorithms [23, 36]. 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 the abstract interpretation, the type inference systems are routinely implemented as part of production quality compilers This has led some researchers to develop program analyses based on nonstandard type inference. One of the earliest

* Corresponding author.

SSDI 0167-6423(95)00012-7

examples is Kuo and Mishra's strictness analysis [28]. 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 [25] has shown how it can be extended to regain this equivalence.

Abstract interpretation represents the strictness property of a function by an abstract function defined on boolean domains. For instance $g_{\mathrm{abs}}\,1\,0 = 0$ means that the result of a call to g is undefined if its second argument is undefined (0 is the abstract value representing an undefined element and 1 is an abstraction of the whole domain, thus represents the absence of information). In terms of types, this property is represented by $g\colon \mathbf{t} \to \mathbf{f} \to \mathbf{f}$. Notice that $\mathbf{t}$ and $\mathbf{f}$ are nonstandard types: $\mathbf{f}$ is the type of the undefined value and $\mathbf{t}$ is the type of all values. As observed by Jensen [25], conjunction types are required for the type system to retain the power of abstract interpretation: a strict function like $+$ must have type $(\mathbf{f} \to \mathbf{t} \to \mathbf{f}) \wedge (\mathbf{t} \to \mathbf{f} \to \mathbf{f})$.

Jensen's 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 [15], we introduce the notion of *most general type* which is equivalent to the conjunction of all the types of an expression. The restriction to most general types allows us to get rid of the the weakening rule and to derive an algorithm which corresponds to the naive implementation of abstract interpretation. The most general type can be seen as a representation of the tabulation of the function. Then we proceed by showing that a further restriction on most general types naturally leads to the frontiers optimisation. The basic idea behind frontiers is to take advantage of monotonicity during the calculation of least fixed points. The restriction on types amounts to representing a conjunction of types by its minimal elements.

The fact that abstract interpretation computes the most general type of an expression accounts not only for its accuracy but also for its inefficiency. We show in this paper that some of this inefficiency can be avoided without losing any of the power of abstract interpretation. The point is that the abstract interpretation often provides much more information than really required. If g is a function of n arguments, the abstract version of g considers all possible combinations of the abstract values of these n arguments: for instance $g_{\mathrm{abs}}\,1\,0\,1\,0\,0 = 0$ means that a call to g is undefined if its second, fourth and fifth arguments are undefined. In some cases this particular piece of information will be useful to show that g is strict in one of its arguments but in many cases it will not be useful at all. The basic idea behind our algorithm is to *compute the strictness types on demand* rather than deriving systematically the most precise information as abstract interpretation does. The corresponding notion of lazy types is defined by allowing source expressions to occur inside types. Formally such a lazy type is equivalent to the most general type of the expression, but it is in unevaluated form, very much like a closure in lazy languages. We give a simple example to provide some intuition about lazy types. This example is traditionally used to illustrate the inefficiency of abstract interpretation [23]:

$$foldr\; b\; g\; \mathbf{nil} = b$$

$$foldr\; b\; g\; \mathbf{cons}(x, y) = g\; x\; (foldr\; b\; g\; y)$$

$$cat\; l = foldr\; \mathbf{nil}\; append\; l$$

Assume that we use a naïve implementation of abstract interpretation to decide if *cat* is strict. The abstract version of *cat* is defined in terms of the abstract version of *foldr*. The abstract version of *foldr* is a function in the domain $Bool \to (Bool \to Bool \to Bool) \to Bool \to Bool$ and its representation is a table of size 64. Two iteration steps are required to find the least fixed point, so two functions of this size are built. In terms of types this means that 128 types were computed to find that *cat* needs its argument. In our algorithm, the original property to prove is *cat*: $\mathbf{f} \to \mathbf{f}$ and this requires proving the following property: *foldr*: $\mathbf{t} \to append \to \mathbf{f} \to \mathbf{f}$ where the second component of the type is an unevaluated closure which corresponds to the conjunction of all the types of *append*. This returns *True* directly because if l has type $\mathbf{f}$ then so does the body of *foldr*. This example shows that a naïve implementation of abstract interpretation is unnecessarily expensive because it considers all possible abstract values for the arguments of a function when only some of them are really useful. This problem becomes crucial in the presence of higher-order functions. In contrast, our algorithm finds information about *append* without computing unnecessary information about its arguments: in this example *append* is left unevaluated in the type of *foldr* because it is not necessary to answer the original question. This case is extreme because we do not need any information about *append* at all. A different original question might require proving that *append* possesses a particular type.

Simple strictness analysis returns information about whether 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:

$$sum\; \mathbf{nil} = 0$$

$$sum\; \mathbf{cons}(x, y) = x + (sum\; y)$$

$$append\; \mathbf{nil}\; l = l$$

$$append\; \mathbf{cons}(x, y)\; l = \mathbf{cons}\; (x, (append\; y\; l))$$

$$H\; l_1\; l_2 = sum(append\; l_1\; l_2)$$

Rather than suspending the evaluation of each recursive call to *append* and returning the weak head normal form $\mathbf{cons}\,(x, (append\; y\; l))$, 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, 30, 38, 39]. Previous approaches, either ideal-based or projection-based, 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, 23]. We illustrate an interesting feature of lazy type inference in this paper: we show that it extends naturally to domains of any depth and it explores the domains only at the particular depth required by the original question.

In the next section we introduce a simply typed λ-calculus and describe a strictness logic based on Jensen's work. Lazy types are introduced in Section 3 and an algorithm for checking types is presented in Section 4. In Section 5, we present some examples. Richer domains for lists are considered in Section 6. Section 7 illustrates that our techniques are applicable to other (safety) analyses; we present a binding time analysis. Experimental results are reported in Section 8. We compare our approach with related work in Section 9 and we conclude in Section 10.

2. A strictness logic for the analysis of lists

We consider a simply typed language, Λ_L. Standard types are defined by the following syntax:

$$\tau = \iota \mid \tau \to \tau \mid \mathit{list}(\tau)$$

where ι is a base type (e.g. *int*). The terms are defined by the following syntax:

$$e = x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \mathbf{fix}(\lambda g.e) \mid \mathbf{cond}(e_1, e_2, e_3) \mid$$

$$\mathbf{nil} \mid \mathbf{cons}(e_1, e_2) \mid \mathbf{hd}(e) \mid \mathbf{tl}(e) \mid \mathbf{case}(e_1, e_2, e_3)$$

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:

$$\mathit{sum} = \mathbf{fix}(\lambda s.\lambda l.\mathbf{case}(0, \lambda x.\lambda y.x + (sy), l))$$

The loss of accuracy that occurs without the **case** operator is discussed in [38].

We consider strictness analysis of lists as the main case study for the presentation of the lazy type inference technique. As a first stage, we consider a nonstandard type system corresponding to Wadler's 4-point domain [38]. We show that this extension can be generalised later. The four elements of the domain are $\mathbf{f} \leqslant \infty \leqslant \mathbf{f}_{\in} \leqslant \mathbf{t}$ 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

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

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

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

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

Fig. 1. The ordering on types.

∞). Technically these types are defined as downwards closed subsets of the standard domain [25].

The set of types and the associated ordering, which is a form of subtype relation, is described in Fig. 1.

Some occurrences of **t** are subscripted by a standard type because the set of constant types includes a collection of **t** and **f** constants [25] (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: $\sigma = \tau \Leftrightarrow \sigma \leqslant \tau$ and $\tau \leqslant \sigma$. The type inference system is shown in Fig. 2. Γ is an environment mapping variables to formulae (i.e. strictness types). In the weakening rule, $\Gamma \leqslant \Delta$ is a shorthand notation for

$$\forall x . \Gamma[x \mapsto \phi] \quad \text{and} \quad \Delta[x \mapsto \psi] \;\Rightarrow\; \phi \leqslant \psi$$

The tautology rule is justified by the fact that a constant is defined, so the only type it can possess is **t**. In the rule **Cond-1**, σ represents the standard type of e_2 (or e_3). The rules for **hd**, **tl** **cons** and **case** follow from the definition of the types. For example, rule **Cons-2** says that if e_2 is an expression which may contain an undefined value (it has type $\mathbf{f}_{\in}$), then so is $\mathbf{cons}(e_1, e_2)$.

This system is an extension of [15, 25] and the soundness and completeness proofs of the logic (with respect to traditional abstract interpretation) follow straightforwardly from Jensen [26]. As an illustration, we show how the property, *sum*: $\mathbf{f}_{\in} \rightarrow \mathbf{f}$, can be derived in this logic:

$$\begin{array}{ll}
\textbf{Conj} & \dfrac{\dfrac{A \quad B}{[s:\mathbf{f}_{\in} \rightarrow \mathbf{f},\; l:\mathbf{f}_{\in}] \vdash \lambda x . \lambda y . x + (s\, y) : \mathbf{t} \rightarrow \mathbf{f}_{\in} \rightarrow \mathbf{f} \wedge \mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{f}} \qquad C}{[s:\mathbf{f}_{\in} \rightarrow \mathbf{f}, l:\mathbf{f}_{\in}] \vdash \mathbf{case}(\mathbf{0}, \lambda x . \lambda y . x + (s\, y), l) : \mathbf{f}} \\
\textbf{Case-3} &
\end{array}$$

$$\vdots$$

$$\begin{array}{ll}
\textbf{Abs} & \dfrac{\dfrac{}{\vdash (\lambda s . \lambda l . \mathbf{case}(0, \lambda x . \lambda y . x + (s\, y), l)) : (\mathbf{f}_{\in} \rightarrow \mathbf{f}) \rightarrow (\mathbf{f}_{\in} \rightarrow \mathbf{f})}}{\dfrac{\vdash \mathbf{fix}(\lambda s . \lambda l . \mathbf{case}(0, \lambda x . \lambda y . x + (s\, y), l)) : \mathbf{f}_{\in} \rightarrow \mathbf{f}}{\vdash \mathit{sum} : \mathbf{f}_{\in} \rightarrow \mathbf{f}}} \\
\textbf{Fix} &
\end{array}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 2. The strictness logic.

where A is

$$\textbf{Abs}\ \cfrac{\cfrac{\vdots}{\cfrac{[s:\mathbf{f}_\in \to \mathbf{f}, l:\mathbf{f}_\in, x:\mathbf{t}, y:\mathbf{f}_\in] \vdash x + (s\,y):\mathbf{f}}{\vdots}}}{[s:\mathbf{f}_\in \to \mathbf{f}, l:\mathbf{f}_\in] \vdash \lambda x.\lambda y.x + (s\,y):\mathbf{t} \to \mathbf{f}_\in \to \mathbf{f}}$$

B is

$$\textbf{Abs}\ \cfrac{\cfrac{\vdots}{\cfrac{[s:\mathbf{f}_\in \to \mathbf{f}, l:\mathbf{f}_\in, x:\mathbf{f}, y:t] \vdash x + (s\,y):\mathbf{f}}{\vdots}}}{[s:\mathbf{f}_\in \to \mathbf{f}, l:\mathbf{f}_\in] \vdash \lambda x.\lambda y.x + (s\,y):\mathbf{f} \to \mathbf{t} \to \mathbf{f}}$$

and C is

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

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.

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

Fig. 3. The language T_I.

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.

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.

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 of types $\leqslant_G$ 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) represents $MGT(\Gamma, e)$, the conjunction of all of the possible types of the term. This correspondence explains the new rules in the definition of $\leqslant_G$. 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. Another departure from the original proof system of Fig. 2 concerns the absence of a weakening rule. This makes the new system more suitable as the basis for the derivation of an inference algorithm. In order to retain the power of the original

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

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

Fig. 4. The language T_G.

system, a form of weakening is integrated within some other rules (**Var**, **Fix**, **Cond-1**, **Hd**, **Tl-3**, **Case-1**). Notice that the **Fix** rule is a schema. The following definition establishes a correspondence between lazy types and ordinary types, the extension to environments is straightforward:

Definition 3.2.

$$Expand : T_G \to T_I$$

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

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

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

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

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

Basically *Expand* replaces a closure by the most general type of the corresponding expression. 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).

$$\Gamma \vdash_G e : \phi \;\Rightarrow\; Expand(\Gamma) \vdash_T e : Expand(\phi) \quad \phi \in T_G$$

Conjecture 3.4 (Completeness).

$$Expand(\Gamma) \vdash_T e : Expand(\phi) \;\Rightarrow\; \Gamma \vdash_G e : \phi \quad \phi \in T'_S$$

Conjecture 3.5 (Equivalence).

$$\Gamma \vdash_T e : \phi \Leftrightarrow \Gamma \vdash_G e : \phi \quad \Gamma \in Var \to T_I,\; e : \phi \in T_I$$

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 Theorem 3.3.

Theorem 3.6. $\sigma \leqslant_G \tau \Leftrightarrow Expand(\sigma) \leqslant Expand(\tau)$

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

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

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

$$\frac{\Gamma \vdash_G e : \phi}{(\Gamma, e) \leq_G \phi} \ (\phi \text{ not of the form}(\Gamma', e')) \qquad \frac{\phi' \leq_G \phi, \psi \leq_G \psi'}{\phi \to \psi \leq_G \phi' \to \psi'}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 5. The lazy types system.

Theorem 3.7.

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

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

The proofs of Theorems 3.6 and 3.7 are 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, Fix** and constant (e.g. **Hd** and **Cond-1**) rules. A similar property has been proved for other type systems including a form of weakening [1, 33]. 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.

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 [19]. Each refinement step introduces a new inference system defining a predicate M_i. We describe now in more detail the four main refinements and the associated predicates $M_1, \ldots, M_4$.

4.1. Introduction of the result component

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 (True indicating that a property is provable in the logic, False indicating that it is not provable):

$$M_1 \subseteq env \times (\Lambda_L \times T_G) \times Bool$$

In the following, we omit the brackets around the arguments to M_1. The predicate 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:

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

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

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

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

4.2. Sequentialisation of the computation

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

$$M_2 \subseteq list(env) \times list(\Lambda_L \times T_G) \times list(Bool)$$

such that

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

where we use overlining to represent a list of elements.

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

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

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

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

$$\frac{M_2\ \Gamma{:}E\ (e_1,(\Gamma,e_2)\rightarrow\psi){:}C\ \ S_1{:}S}{M_2\ \Gamma{:}E\ (e_1e_2,\psi){:}C\ \ S_1{:}S}$$

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

$$M_2\ nil\ nil\ nil$$

4.3. Optimisation of environment management

M_2 creates a new environment 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:

$$M_3 \subseteq env \times list(\Lambda_L \times T_G) \times list(Bool)$$

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

M_3 is derived from M_2 in a straightforward way;

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

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

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

$$\frac{M_3\ \Gamma\ (e_1,(\Gamma,e_2)\to\psi):C\ \ S_1:S}{M_3\ \Gamma\ (e_1e_2,\psi):C\ \ S_1:S}$$

4.4. Deriving an abstract machine

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. Notice that the rewrite system is deterministic; although there is an apparent ambiguity between the first and fourth rules, they do not overlap because we are dealing with lazy types and thus ψ in the fourth rule is not a conjunction. 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 (accumulator) argument R which is not modified in the rules and is ultimately instantiated with the result of the computation.

$$M_4 \subseteq env \times (list(\Lambda_L \times T_G + (Bool \times Bool \to Bool)) \times list(Bool) \times Bool$$

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

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

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

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

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

$$\frac{M_4\ \Gamma\ (e_1,(\Gamma,e_2)\to\psi):C\ \ S\ \ R}{M_4\ \Gamma\ (e_1e_2,\psi):C\ \ S\ \ R}$$

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

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

$$M_4\ \Gamma\ \mathit{nil}\ R{:}\mathit{nil}\ R$$

The end result is that we now have an inference system which is an *Abstract Evaluation System* in the terminology of [19]. This means that we can alternatively present it as a rewriting system describing a machine. We just have to rewrite any rule

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

as

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

We have reorganised the components to show the similarity to abstract machines for functional languages which have a stack, environment and control (SEC-machine). Notice that the R component is superfluous; its final value is identical to the value on the top of the S component.

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.

Let us consider 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 ϕ. The simplest subproof that would allow us to succeed would be one that proves $\lambda g\,.\,e$ has type $\phi \to \phi$; this in turn follows from a proof that e has type ϕ under the assumption that g also has type ϕ. 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 ϕ. In other words, the assumption on g has to be strengthened. This motivates the rule for (Rec, g, ψ) in Fig. 6. Basically (Rec, g, ψ) records the fact that the preceding instruction was an attempt to prove that the recursive function g has type ψ. If this proof succeeds, then (Rec, g, ψ) is a null operation; otherwise, its effect is to add the assumption $g:\psi$ in the environment. Notice that we use $:_r$ and $\mapsto_r$ to represent bindings to a recursion variable (the bound variable of the outermost λ in a **fix** expression). The D operation is used to clean up the environment.

Section 5 contains an example illustrating the iteration involved in the treatment of recursion. We do not consider embedded occurrences of **fix** here; the extension is straightforward but would introduce unnecessary complications in the presentation.

The *Leq* operation computes the $\leqslant_G$ predicate and is presented in Fig. 7. The rules mirror the definition of $\leqslant_G$ in Fig. 5. The only complexity is introduced by the rule for $\psi \leqslant_G (\Gamma, e)$ which requires an iteration to prove that any type ϕ_i possessed by e satisfies $\psi \leqslant_G \phi_i$. This motivates the introduction of the conditional operation

$$
\begin{array}{rcl}
\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 \\
\\
\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], Leq(\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 \\
\\
\langle S, E, (\mathbf{fix}(\lambda g.e), \phi) : C\rangle & \rhd_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 & \rhd_G & \langle S, E[g \mapsto_r (e, \phi)], Leq(\phi, \psi) : (Rec, g, \psi) : C\rangle \\
\langle True : S, E, (Rec, g, \psi) : C\rangle & \rhd_G & \langle True : S, E, C\rangle \\
\langle False : S, E[g \mapsto_r (e, \phi)], (Rec, g, \psi) : C\rangle & \rhd_G & \langle S, (g :_r (e, \phi \wedge \psi)) : E[g \mapsto_r (e, \phi)], (e, \psi) : D(g) : C\rangle \\
\\
\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 \\
\\
\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 \\
\\
\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 \\
\\
\langle S, E, (\mathbf{case}(e_1, e_2, e_3), \phi) : C\rangle & \rhd_G &
\end{array}
$$

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

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

Fig. 6. The lazy types algorithm.

$Cond(B_1, B_2)$ which is used to test $\psi \leqslant_G \phi_i$ depending on the outcome of the proof of $e : \phi_i$.

The algorithm presented in Fig. 6 is a slight variant of the algorithm appearing in [15]; this version provides a more uniform treatment of fixed points.

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

Conjecture 4.1. (1) $\langle S, T, (e, \phi) : C\rangle \rhd_G^* \langle True : S, \Gamma, C\rangle \Leftrightarrow \Gamma \vdash_G e : \phi$

(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

$$\begin{array}{rcl}
\langle S, E, Leq(\mathbf{f}, \phi) : C\rangle & \rhd_G & \langle True : S, E, C\rangle \\
\langle S, E, Leq(\phi, \phi) : C\rangle & \rhd_G & \langle True : S, E, C\rangle \\
\langle S, E, Leq(\infty, \mathbf{f}_\in) : C\rangle & \rhd_G & \langle True : S, E, C\rangle \\
\langle S, E, Leq(\phi, \mathbf{t}) : C\rangle & \rhd_G & \langle True : S, E, C\rangle \\
\langle S, E, Leq(\phi_1 \to \ldots \to \phi_n \to \phi, \psi_1 \to \ldots \to \psi_n \to \mathbf{t}) : C\rangle & \rhd_G & \langle True : S, E, C\rangle \\
\langle S, E, Leq(\phi_1 \wedge \ldots \wedge \phi_n, \psi_1 \wedge \ldots \wedge \psi_m) : C\rangle & \rhd_G & \\
\multicolumn{3}{r}{\langle S, E, Leq(\phi_1, \psi_1) : \ldots : Leq(\phi_n, \psi_1) : Or : \ldots Leq(\phi_1, \psi_m) : \ldots : Leq(\phi_n, \psi_m) : Or : And : C\rangle} \\
\langle S, E, Leq(\psi, (\Gamma, e)) : C\rangle & \rhd_G & \\
\multicolumn{3}{l}{\langle S, \Gamma, (e, \phi_1) : Cond(False, True) : Leq(\psi, \phi_1) : \ldots (e, \phi_k) : Cond(False, True) : Leq(\psi, \phi_k) : And : Setenv(E) : C\rangle} \\
 & & \text{with } \phi_1, \ldots \phi_k \text{ the } T_S \text{ types} \\
 & & \text{compatible with the standard type of } e. \\
 & & \\
\langle S, E, Leq((\Gamma, e), \phi) : C\rangle & \rhd_G & \langle S, \Gamma, (e, \phi) : Setenv(E) : C\rangle \\
 & & \text{with } \phi \neq (\Gamma', e') \\
 & & \\
\langle S, E, Leq(\phi \to \psi, \phi' \to \psi') : C\rangle & \rhd_G & \langle S, E, Leq(\phi', \phi) : Leq(\psi, \psi') : And : C\rangle \\
\langle S, E, Setenv(E') : C\rangle & \rhd_G & \langle S, E', C\rangle \\
\langle B_1 : S, E, Cond(B_1, B_2) : C_0 : C\rangle & \rhd_G & \langle B_2 : S, E, C\rangle \\
\langle B_1' : S, E, Cond(B_1, B_2) : C_0 : C\rangle & \rhd_G & \langle S, E, C_0 : C\rangle \\
 & & \text{with } B_1 \neq B_1'
\end{array}$$

Fig. 7. Implementation of Leq.

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

Conjecture 4.2. (1) $\langle S, \Gamma, Leq(\phi, \psi) : C\rangle \rhd_G^* \langle True : S, \Gamma, C\rangle \Leftrightarrow \phi \leqslant_G \psi$
(2) $\langle S, \Gamma, Leq(\phi, \psi) : C\rangle \rhd_G^* \langle False : S, \Gamma, C\rangle \Leftrightarrow \neg(\phi \leqslant_G \psi)$
if Γ, ϕ and ψ do not contain any $\mapsto_r$ assumption

The restrictions on the 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 when the second rule for *Rec* is applied, the new type bound to the recursion variable is strictly less than the previous binding; i.e. $\phi \wedge \psi <_G \phi$.

The algorithm derived 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 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 avoids the expense of building a suspension for an argument whose value (type) is already known; this is a common optimisation used in the implementation of lazy functional languages.

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 [28] 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 $T_1 = \mathbf{f} \to \mathbf{t} \to \mathbf{t} \to \mathbf{f}$. 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 ($Leq(T_1, T_2)$ fails with $T_2 = \mathbf{t} \to \mathbf{f} \to \mathbf{t} \to \mathbf{f}$). 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:

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

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

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

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

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

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:

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

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

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

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

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

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

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

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

$((g, T_2):And:Or:D(z):D(y):D(x):D(g)\rangle \ \triangleright^*_G$

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

$(Leq(T_1, T_2):(Rec, g, T_2):\ And:Or:D(z):D(y):D(x):D(g)\rangle \ \triangleright^*_G$

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

$(Rec, g, T_2):And:Or:D(z):D(y):D(x):D(g)\rangle \ \triangleright^*_G$

$\langle True:False, (g:_r(E', T_1 \wedge T_2)):(z:\mathbf{t}):(y:\mathbf{t}):(x:\mathbf{f}):(g:_r(E', T_1)),$

$(E', T_2):D(g):And:Or:D(z):D(y):D(x):D(g)\rangle \ \triangleright^*_G$

$\langle True:False, (z:\mathbf{t}):(y:\mathbf{f}):(x:\mathbf{t}):(g:_r(E', T_1 \wedge T_2)):(z:\mathbf{t}):(y:\mathbf{t}):(x:\mathbf{f}):(g:_r(E', T_1)),$

$(E, \mathbf{f}):D(z):D(y):D(x):D(g):And:Or:D(z):D(y):D(x):D(g)\rangle \triangleright^*_G$

$\langle True:False:True:False,$

$(z:\mathbf{t}):(y:\mathbf{f}):(x:\mathbf{t}):(g:_r(E', T_1 \wedge T_2)):(z:\mathbf{t}):(y:\mathbf{t}):(x:\mathbf{f}):(g:_r(E', T_1)),$

$(g, T_1):And:Or:D(z):D(y):D(x):D(g):And:Or:D(z):D(y):D(x):D(g)\rangle \ \triangleright^*_G$

$\langle True:True:False:True:False, (z:t):(y:\mathbf{f}):(x:\mathbf{t}):(g:_r(E', T_1 \wedge T_2)):(z:\mathbf{t}):(y:\mathbf{t}):$

$(x:\mathbf{f}):(g:_r(E', T_1)),$

$And:Or:D(z):D(y):D(x):D(g):And:Or:D(z):D(y):D(x):D(g)\rangle \triangleright^*_G$

$\langle True, nil, nil\rangle$

5.2. Higher-order and lists

The function *foldr* presented in the introduction was used in [23] 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:

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

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

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

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

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

$$\langle nil, nil, (cat, \mathbf{f} \to \mathbf{f})\rangle \ \rhd_G$$

$$\langle nil, (l{:}\mathbf{f}), (foldr\,\mathbf{nil}\,append\,l, \mathbf{f}){:}D(l)\rangle \ \rhd_G$$

$$\langle nil, (l{:}\mathbf{f}), (foldr\,\mathbf{nil}\,append, \mathbf{f} \to \mathbf{f}){:}D(l)\rangle \ \rhd_G$$

$$\langle nil, (l{:}\mathbf{f}), (foldr\,\mathbf{nil}, ((l{:}\mathbf{f}), append) \to \mathbf{f} \to \mathbf{f}){:}D(l)\rangle \ \rhd_G$$

$$\langle nil, (l{:}\mathbf{f}), (foldr, \mathbf{t} \to ((l{:}\mathbf{f}), append) \to \mathbf{f} \to \mathbf{f}){:}D(l)\rangle \ \rhd_G^*$$

$$\langle nil, (l{:}\mathbf{f}){:}(g, ((l{:}\mathbf{f}), append)){:}(b{:}\mathbf{t}){:}(f{:}_r(E, \phi)){:}(l{:}\mathbf{f}),$$

$$(\mathbf{case}\ \dots, \mathbf{f}){:}D(l){:}D(g){:}D(b){:}D(f){:}D(l)\rangle \ \rhd_G^*$$

$$\langle nil, \dots, (l, \mathbf{f}){:}\dots{:}Or{:}D(l){:}D(g){:}D(b){:}\ \dots\rangle \ \rhd_G^*$$

$$\langle True, \dots, D(l){:}D(g){:}D(b){:}\dots\rangle \ \rhd_G^*$$

$$\langle True, (f{:}_r(E, \phi)){:}(l{:}\mathbf{f}), D(f){:}D(l)\rangle \ \rhd_G$$

$$\langle True, (l{:}\mathbf{f}), D(l)\rangle \ \rhd_G$$

$$\langle True, nil, nil\rangle$$

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 [38], 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 $\mathbf{f} \leqslant_0 \mathbf{t}$. Then

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

with:

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

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

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

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

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

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:

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

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

and the ordering on types is extended with the rules:

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

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 *Leq* 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 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 (as mentioned earlier, the fact that the underlying language is typed plays a crucial rôle to this respect). We first restate the definition of *append* as a term of Λ_L:

$$append = \mathbf{fix}(\lambda app\,.\,\lambda u\,.\,\lambda v\,.\,\mathbf{case}(v, \lambda x\,.\,\lambda y\,.\,\mathbf{cons}(x,(app\; y\; v)), u))$$

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

$$\begin{array}{ll}
 & \vdots \\
 & A \qquad B \\ \cline{2-2}
\textbf{Conj} & \Gamma \vdash (\lambda x\,.\,\lambda y\,.\,g\,x(f b g\, y))\colon (\mathbf{t} \to \infty_\epsilon \to \infty) \wedge (\infty \to \mathbf{t} \to \infty) \qquad C \\ \cline{2-2}
\textbf{Case-3} & \Gamma \vdash \mathbf{case}(b, \lambda x \lambda y\,.\,g\,x(f b g\, y), l)\colon \infty \\
 & \\
 & \vdots \\ \cline{2-2}
\textbf{Abs} & \vdash \lambda f\,.\,\lambda b\,.\,\lambda g\,.\,\lambda l\,.\,\mathbf{case}(b, \lambda x \lambda y\,.\,g x(f b g\, y), l)\colon \\
 & (\mathbf{t} \to append \to \infty_\epsilon \to \infty) \to (\mathbf{t} \to append \to \infty_\epsilon \to \infty) \\ \cline{2-2}
\textbf{Fix} & \vdash \mathbf{fix}(\lambda f\,.\,\lambda b\,.\,\lambda g\,.\,\lambda l\,.\,\mathbf{case}(b, \lambda x \lambda y\,.\,g\,x(f\; b\, g\, y), l))\colon \mathbf{t} \to append \to \infty_\epsilon \to \infty \\ \cline{2-2}
 & \vdash foldr\colon \mathbf{t} \to append \to \infty_\epsilon \to \infty
\end{array}$$

where Γ is $[f\colon \mathbf{t} \to append \to \infty_\epsilon \to \infty,\ b\colon \mathbf{t}, g\colon append, l\colon \infty_\epsilon]$. and A is

$$\begin{array}{ll}
 & \dfrac{\begin{array}{c}\vdots \\ \Gamma' \vdash \lambda x\,.\,\lambda y\,.\,\mathbf{cons}(x,(app\; y\; v))\colon \mathbf{t} \to \mathbf{t} \to \infty\end{array} \qquad \dfrac{\dfrac{\begin{array}{c}\vdots \\ \Gamma'' \vdash f b g\, y\colon \infty\end{array}}{(\Gamma'', f b g\, y) \leqslant \infty}}{\Gamma' \vdash v\colon \infty}}{\Gamma' \vdash \mathbf{case}(v, \lambda x\,.\,\lambda y\,.\,\mathbf{cons}(x,(app\; y\; v)), u)\colon \infty} \quad \textbf{Case-4} \\
 & \\
\textbf{App} & \Gamma'' \vdash g\colon t \to (f b g\, y) \to \infty \\ \cline{2-2}
\textbf{App} & \Gamma'' \vdash g\, x\colon (f b g\, y) \to \infty \\ \cline{2-2}
\textbf{App} & \Gamma'' \vdash g\, x(f b g\, y)\colon \infty \\
 & \vdots \\ \cline{2-2}
\textbf{Abs} & \Gamma \vdash (\lambda x\,.\,\lambda y\,.\,g\, x(f b g\, y))\colon (\mathbf{t} \to \infty_\epsilon \to \infty)
\end{array}$$

where

$$\Gamma' \;= [app\colon (t \to (\Gamma'', (f b g\, y)) \to \infty), u\colon \mathbf{t}, v\colon (\Gamma'', (f b g\, y))]\colon \Gamma''$$

$$\Gamma'' = [x\colon \mathbf{t}, y\colon \infty_\epsilon]\colon \Gamma$$

the proof tree for B is similarly constructed and C is $\Gamma \vdash l\colon \infty_\epsilon$. So the domain is explored up to depth 2 (D_2). If we now ask the question $foldr\colon \mathbf{t} \to append \to \mathbf{f}_\epsilon \to \infty$, 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 [17], 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 $\leqslant$ 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 $(X, \sqsubseteq, \sqcap, \Rightarrow, \mathit{norm})$, 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 $\rightarrow$.
- $\mathit{norm} : X \rightarrow \mathfrak{P}(D)$ maps any property to its underlying set of domain elements.

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

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

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

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

Definition 7.2. The structure is a model, if for all ϕ and ψ:

$$\phi \leqslant \psi \text{ implies } [\![\phi]\!] \sqsubseteq [\![\psi]\!]$$

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

$$\bot\, P\, \bot$$

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. Hurt and Sands have used CPERs in binding time analysis [24].

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

$$\mathscr{M}_{cper} = (\mathscr{CPER}(D), \sqsubseteq_{cper}, \sqcap_{cper}, \Rightarrow_{cper}, norm_{cper})$$

where

- $\sqsubseteq_{cper} = \subseteq$ (set inclusion)
- $\sqcap_{cper} = \cap$ (set intersection)
- $Q \Rightarrow_{cper} R = \{(f, g) \mid \forall q \, q' . q \, Q \, q' \Rightarrow (f\,q) \, R \, (g\,q')\}$
- $norm_{cper}(P) = \{x \in D \mid x \, P \, x\}$

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:

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

Definition 7.4. A rule for an *n*-ary constant:

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

is *sound* if, *under* the assumption that:

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

then $\Gamma \models c\, e_1 \ldots e_n : \phi$.

In our earlier work [17] we extend a result of Burn [5], and show that soundness of the constant rules ensures soundness of the logic. This gives a local correctness condition. In [17], 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 [17] 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. The constants

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

7.3. Type structure

We model constants as PERs in the following way:

$$I_{cper}(static) = \{(x,x) \mid x \in D\} \quad (= Id)$$

$$I_{cper}(dynamic) = \{(x,y) \mid x, y \in D\} \quad (= All)$$

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

$$\phi \leqslant \psi \text{ implies } [\![\phi]\!] \sqsubseteq [\![\psi]\!]$$

and the structure is a model.

7.4. Type inference rules

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

Const $\vdash_G c : static$

$$\begin{array}{rcl}
\langle S, E, \boldsymbol{Leq}(\boldsymbol{static}, \boldsymbol{dynamic}) : C\rangle & \rhd_G & \langle \boldsymbol{True} : S, E, C\rangle \\[1em]
\langle S, E, (c, \boldsymbol{static}) : C\rangle & \rhd_G & \langle \boldsymbol{True} : S, E, C\rangle \\
\langle S, E, (c, \boldsymbol{dynamic}) : C\rangle & \rhd_G & \langle \boldsymbol{True} : S, E, C\rangle \\[1em]
\langle S, E, (+(e_1, e_2), \boldsymbol{static}) : C\rangle & \rhd_G & \\
\multicolumn{3}{r}{\langle S, E, (e_1, \boldsymbol{static}) : (e_2, \boldsymbol{static}) : \boldsymbol{And} : C\rangle} \\
\langle S, E, (+(e_1, e_2), \boldsymbol{dynamic}) : C\rangle & \rhd_G & \langle \boldsymbol{True} : S, E, C\rangle \\[1em]
\langle S, E, (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, \phi) : C\rangle & \rhd_G & \\
\multicolumn{3}{r}{\langle S, E, (e_1, \boldsymbol{static}) : (e_2, \phi) : (e_3, \phi) : \boldsymbol{And} : \boldsymbol{And} : C\rangle} \\[1em]
\langle S, E, (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, \boldsymbol{dynamic}) : C\rangle & \rhd_G & \langle \boldsymbol{True} : S, E, C\rangle
\end{array}$$

Fig. 8. The new transitions for the binding time analysis algorithm.

which is sound since *norm*($[\![static]\!]$) is D. Notice that we have located this axiom in the lazy types system; the same axiom would also be used Jensen's system. The rule scheme for $+$ and the conditional are respectively

$$+\quad \frac{\Gamma \vdash_G e_1 : static \qquad \Gamma \vdash_G e_2 : static}{\Gamma \vdash_G +(e_1, e_2) : static}$$

$$+\quad \frac{\Gamma \vdash_G e_1 : \phi \qquad \Gamma \vdash_G e_2 : \psi}{\Gamma \vdash_G +(e_1, e_2) : dynamic}$$

$$\textbf{Cond-1}\quad \frac{\Gamma \vdash_G b : static \qquad \Gamma \vdash_G e_1 : \phi \quad \Gamma \vdash_G e_2 : \phi}{\Gamma \vdash_G \textbf{if } b \textbf{ then } e_1 \textbf{ else } \mathbf{e}_2 : \phi}$$

$$\textbf{Cond-2}\quad \Gamma \vdash_G \textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2 : dynamic$$

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

$$Expand(\Gamma) \models b : static$$

$$Expand(\Gamma) \models e_1 : Expand(\phi)$$

$$Expand(\Gamma) \models e_2 : Expand(\phi)$$

Now if $\mathscr{S}[\![b]\!]\rho$ is $\perp$ then $\mathscr{S}[\![\textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2]\!]\rho = \perp$ and thus

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

since $Expand(\phi)$ is a CPER. If $\mathscr{S}[\![b]\!]\rho \neq \perp$ then the soundness result is immediate.

7.5. Transition rules

We add the new rules shown in Fig. 8. Since these rules are derived from the typing rules in a fairly direct manner, their correctness is immediate.

$$
\begin{array}{lcl}
\mathit{foldr}\ g\ \mathbf{nil}\ b & = & b \\
\mathit{foldr}\ g\ \mathbf{cons}(x,y)\ b & = & g\ x\ (\mathit{foldr}\ g\ y\ b) \\
\\
\mathit{append}\ \mathbf{nil}\ l & = & l \\
\mathit{append}\ \mathbf{cons}(x,y)\ l & = & \mathbf{cons}(x,(\mathit{append}\ y\ l)) \\
\\
\mathit{cat}\ l & = & \mathit{foldr}\ \mathit{append}\ l\ \mathit{nil} \\
\\
\mathit{Cfoldr}\ g\ \mathbf{nil}\ b\ c & = & c\ b \\
\mathit{Cfoldr}\ g\ \mathbf{cons}(x,y)\ b\ c & = & \mathit{Cfoldr}\ g\ y\ b\ (\lambda y.g\ x\ y\ c) \\
\\
\mathit{Cappend}\ \mathbf{nil}\ l\ c & = & c\ l \\
\mathit{Cappend}\ \mathbf{cons}(x,y)\ l\ c & = & \mathit{Cappend}\ y\ l\ (\lambda y.c\ (\mathbf{cons}(x,y))) \\
\\
\mathit{Ccat}\ l\ c & = & \mathit{Cfoldr}\ \mathit{Cappend}\ l\ \mathit{nil}\ c \\
\\
K\ x\ y & = & x \\
\\
\mathit{isnil}\ \mathbf{nil} & = & \mathbf{True} \\
\mathit{isnil}\ \mathbf{cons}(x,y) & = & \mathbf{False} \\
\\
\mathit{length}\ \mathbf{nil} & = & 0 \\
\mathit{length}\ \mathbf{cons}(x,y) & = & 1+(\mathit{length}\ y) \\
\\
\mathit{sum}\ \mathbf{nil} & = & 0 \\
\mathit{sum}\ \mathbf{cons}(x,y) & = & x+(\mathit{length}\ y) \\
\\
\mathit{test}_1\ l & = & \mathit{Ccat}\ l\ (K\ 0) \\
\mathit{test}_2\ l & = & \mathit{Ccat}\ l\ \mathit{isnil} \\
\mathit{test}_3\ l & = & \mathit{Ccat}\ l\ \mathit{length} \\
\mathit{test}_4\ l & = & \mathit{Ccat}\ l\ \mathit{sum}
\end{array}
$$

Fig. 9. Testbed.

8. Experimental results

The lazy types algorithm has been implemented (in CAML light) as an interpreter realising the abstract machine described in Figs. 6 and 7. We report here on experimental results. We use as a test-bed two versions of a function concatenating lists of lists, the second one being defined in terms of continuations. The functions are presented in Fig. 9. These examples were provided by S. Hunt to illustrate the limitations of the frontiers optimisation [22, 23].

Fig. 10 shows, 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). The figures shown differ slightly from those reported in [17]; these differences arise from the modifications to the algorithm mentioned earlier.

$cat : \mathbf{f}_{\in} \rightarrow \mathbf{f}$	*False*	0.08 *s*
$cat : \infty \rightarrow \mathbf{f}$	*False*	0.17 *s*
$cat : \infty \rightarrow \infty$	*True*	0.02 *s*
$test_1 : \mathbf{f}_{\in\in} \rightarrow \mathbf{f}$	*False*	0.2 *s*
$test_1 : \infty_{\in} \rightarrow \mathbf{f}$	*True*	0.33 *s*
$test_2 : \mathbf{f}_{\in\in} \rightarrow \mathbf{f}$	*False*	0.95 *s*
$test_2 : \infty_{\in} \rightarrow \mathbf{f}$	*True*	3.97 *s*
$test_3 : \mathbf{f}_{\in\in} \rightarrow \mathbf{f}$	*False*	2.9 *s*
$test_3 : \infty_{\in} \rightarrow \mathbf{f}$	*True*	1.7 *s*
$test_4 : \mathbf{f}_{\in\in} \rightarrow \mathbf{f}$	*True*	1.42 *s*
$test_4 : \infty_{\in} \rightarrow \mathbf{f}$	*True*	0.37 *s*

Fig. 10. Experimental results.

These results should be compared with other implementations. The contrast with frontiers based “optimisations” of abstract interpretation is striking: the analysis of [23] takes 30 min to process *cat* and runs out of time 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 their analysis of *cat* requires 5 s and the analysis of *Ccat* around 10 s. In comparison, our algorithm takes 0.5 s for *cat* and about 2 s 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 is very space-consuming.

The analyser described in [31] 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 s and the analysis of *Ccat* 1 h.

It should be noted that our approach has the same worst-case complexity as these other approaches (the problem is inherently exponential as shown in [21]) but we believe that lazy types can provide the basis for more realistic analysers for functional languages because of their ability to tackle higher-order functions in an efficient way. As a consequence, it seems that typical programs do not exhibit worst-case behaviour. Of course, more experience is needed to sustain this claim.

9. Related work

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 [7, 12, 25, 28, 30, 35]. 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 [28, 30]. In contrast, [12, 35] 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 [35], the analysis is expressed as a form of reduction of abstract graphs. As in our work, the 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 $f: t_1 \to t_2 \to t_3$, it turns out to be necessary to prove $f: e_1 \to e_2 \to e_3$. 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 $f: e_1 \to e_2 \to e_3$ (except if $f: e_1 \to e_2 \to e_3$ and $f: t_1 \to t_2 \to t_3$ are syntactically equal) without any attempt to relate the types t_i and e_i. In contrast, the lazy type algorithm tries to prove $t_1 \to t_2 \to t_3 \leqslant e_1 \to e_2 \to e_3$, 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 nontermination if some special measures are not taken. These extra measures can take the form of arbitrarily terminating the derivation (using empirical resource consumption criteria) incurring a loss of accuracy. A neededness analysis called reduction path analysis is also proposed in [35] to allow termination of the computation without throwing away too much information. Because of this parameterisable termination condition, it is difficult to formally quantify the power of abstract graph reduction. An 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 to improving the computation of fixed points is called *chaotic iteration.* It was introduced in [8] and extended to higher-order functional programs in [37]. 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 [37]). As an example, the two algorithms exhibit different behaviours when applied to the following functions:

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

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 [37] 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 [37]. 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 whilst 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.

10. Conclusions

One interesting outcome of the line of work followed here is to reconcile the two main approaches for the static analysis of functional programs. Type inference and abstract interpretation should be seen as two ways of presenting analyses rather than two different options for implementing analysers. We believe that a significant contribution of the type-based approach is to make it easier to decouple the specification of an analysis and its implementation through an algorithm. As shown in [15], this may shed new light on the various choices for optimising the analysers and help in the design of new techniques for program analysis.

We describe now some interesting avenues for further research. Abstract graph reduction and chaotic fixed-point iteration could be reexpressed in terms of type inference as suggested here: this would allow us 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. 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 [26] 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 and Alan Mycroft provided some valuable feedback on earlier versions of this paper. The first author is partially funded by ESPRIT Working Groups 6809 (Semantique) and 6345 (SemaGraph).

References

[1] S. van Bakel, Complete restrictions of the intersection type discipline, *Theoret. Comput. Sci.* **102**(1) (1992).

[2] P.N. Benton, Strictness logic and polymorphic invariance, in: *Proc. 2nd Internat. Symp. on Logical Foundations of Computer Science*, Lecture Notes in Computer Science, Vol. 620 (Springer, Berlin, 1992).

[3] P.N. Benton, Strictness properties of lazy algebraic datatypes, in: *Proc. WSA'93*, Lecture Notes in Computer Science, Vol. 724 (Springer, Berlin, 1993).

[4] G.L. Burn, Evaluation transformers – a model for the parallel evaluation of functional languages (extended abstract), in: *Proc. 1987 Conf. on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, Vol. 274 (Springer, Berlin, 1987).

[5] G.L. Burn, A logical framework for program analysis, in: *Proc. 1992 Glasgow Functional Programming Workshop*, Workshops in Computer Science (Springer, Berlin, 1992).

[6] G. Burn and D. Le Métayer, proving the correctness of compiler optimisations based on strictness analysis, in: *Proc. 5th Internat. Symp. on Programming Language Implementation and Logic Programming,* Lecture Notes in Computer Science, Vol. 714 (Springer, Berlin, 1993).

[7] T.-R. Chuang and B. Goldberg, A syntactic approach to fixed point computation on finite domains, in: *Proc. 1992 ACM Conf. on Lisp and Functional Programming* (ACM, New York, 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, Amsterdam, 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*, Lecture Notes in Computer Science, Vol. 631, (Springer, Berlin, 1992).

[10] O. Danvy and J. Hatcliff, *CPS trasformation after strictness analysis, ACM Lett. Program. Languages Systems* **1** (3) (1993).

[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* (Wiley, New York, 1992).

[12] A. Ferguson and R.J.M. Hughes, Fast abstract interpretation using sequential algorithms, in: *Proc. WSA'93*, Lecture Notes in Computer Science, Vol. 724 (Springer, Berlin, 1993).
[13] S. Finne and G. Burn, Assessing the evaluation transformer model of reduction on the spineless G-Machine, in *Proc. 6th ACM Conf. on Functional Programming Languages and Computer Architecture* (ACM, New York, 1993).
[14] C.L. Hankin and L.S. Hunt, Approximate fixed points in abstract interpretation, in: B. Krieg-Brückner, ed., *Proc. 4th European Symp. on Programming*, Lecture Notes in Computer Science, Vol. 582 (Springer, Berlin, 1992).
[15] C.L. Hankin and D. Le Métayer, Deriving algorithms from type inference systems: application to strictness analysis, in: *Proc. POPL'94* (ACM, New York, 1994).
[16] C.L. Hankin and D. Le Métayer, Lazy type inference for the strictness analysis of lists, in: *Proc. ESOP'94*, Lecture Notes in Computer Science, Vol. 788 (Springer, Berlin, 1994).
[17] C.L. Hankin and D. Le Métayer, A type-based framework for program analysis, in: *Proc. Static Analysis Symp.*, Lecture Notes in Computer Science, Vol. 864 (Springer, Berlin, 1994).
[18] J.J. Hannan, Investigating a proof-theoretic meta-language, Ph.D. Thesis, University of Pennyslvania; DIKU Technical Report Nr 91/1, 1991.
[19] J. Hannan and D. Miller, From Operational Semantics to Abstract Machines, *Math. Struct. Comput. Sci.* 2(4) (1992).
[20] P.H. Hartel and K.G. Langendoen, Benchmarking implementations of lazy functional languages, in: *Proc. 6th ACM Conf. on Functional Programming Languages and Computer Architecture* (ACM, New York, 1993).
[21] P. Hudak and J. Young, Higher order strictness analysis in untyped lambda calculus, in: *Proc. 13th ACM Symp. on Principles of Programming Languages* (ACM, New York, 1986).
[22] L.S. Hunt, Abstract interpretation of functional languages: from theory to practice, Ph.D. Thesis, Imperial College, 1991.
[23] L.S. Hung and C.L. Hankin, Fixed points and frontiers: A new perspective, *J. Funct. Programming* **1** (1) (1991).
[24] L.S. Hunt and D. Sands, Binding time analysis: a new perspective, in: *Proc. ACM Symp. on Partial Evaluation and Semantics-based Program Manipulation* (ACM, New York, 1991).
[25] T.P. Jensen, Strictness analysis in Logical form, in: J. Hughes, eds, *Proc. 5th ACM Conf. on Functional Programming Languages and Computer Architecture,* Lecture Notes in Computer Science, Vol. 523, (Springer, Berlin, 1991).
[26] T.P. Jensen, Abstract interpretation in logical form, Ph.D. Thesis, University of London, 1992; also available as DIKU Technical Report 93/11.
[27] N.D. Jones and A. Mycroft, Data-flow analysis of applicative programs using minimal function graphs, in: *Proc. ACM Conf. on Principles of Programming Languages* (ACM, New York, 1986).
[28] T.-M. Kuo and P. Mishra, Strictness analysis: a new perspective based on type inference, in: *Proc. 4th ACM Conf. on Functional Programming Languages and Computer Architecture* (ACM, New York, 1989).
[29] J. Launchbury, Strictness and binding time: two for the price of one, in: *Proc. ACM Conf. on Programming Languages Design and Implementation* (ACM, New York, 1991).
[30] A. Leung and P. Mishra, Reasoning about simple and exhaustive demand in higher-order lazy languages, in: *Proc. 5th ACM Conf. on Functional Programming Languages and Computer Architecture,* Lecture Notes in Computer Science, Vol. 523 (Springer, Berlin, 1991).
[31] L. Mauborgne, Abstract interpretation using TDGs, in: *Proc. Static Analysis Symp.*, Lecture Notes in Computer Science, Vol. 864 (Springer, Berlin, 1994).
[32] R. Milner, A theory of type polymorphism in programming, *J. Comput. System Sci.* **17**(3) (1978).
[33] J.C. Mitchell, Type inference with simple subtypes, *J. Funct. Programming* **1**(3) (1991).
[34] A. Mycroft, Abstract interpretation and optimising transformations for applicative programs, Ph.D. Thesis, University of Edinburgh, 1981.
[35] E. Nöcker, Strictness analysis using abstract reduction, in: *Proc. 6th ACM Conf. on Functional Programming Languages and Computer Architecture* (ACM, New York, 1993).
[36] S.L. Peyton Jones and C. Clack, Finding fixed points in abstract interpretation, in: S. Abransky and C.L. Hankin, eds., *Abstract Interpretation of Declarative Languages* (Ellis Horwood, Chichester, UK, 1987).

[37] M. Rosendahl, Higher-order chaotic iteration sequences, in: *Proc. 5th Internat. Symp. Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science, Vol. 714 (Springer, Berlin, 1993).

[38] P. Wadler, Strictness analysis on non-flat domains, in: S. Abramsky and C.L. Hankin, eds., *Abstract Interpretation of Declarative Languages* (Ellis Horwood, Chichester, UK, 1987).

[39] P. Wadler and J. Hughes, Projections for strictness analysis, in: *Proc. 1987 Conf. on Functional Programming Languages and Computer Architecture,* Lecture Notes in Computer Science, Vol. 274 (Springer, Berlin, 1987).