Deriving algorithms from type inference systems: Application to strictness analysis

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

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

Abstract

The rôle of non-standard type inference in static program analysis has been much studied recently. Early work emphasised the efficiency of type inference algorithms and paid little attention to the correctness of the inference system. Recently more powerful inference systems have been investigated but the connection with efficient inference algorithms has been obscured. The contribution of this paper is twofold: first we show how to transform a program logic into an algorithm and, second, we introduce the notion of lazy types and show how to derive an efficient algorithm for strictness analysis.

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 majority of the effort on improving the efficiency of abstract interpretation has concentrated on *frontiers*-based algorithms [13] or widening techniques [6, 9]. The former still has unacceptable performance for some commonly occurring higher-order programs. The latter is a general approach for accelerating convergence in fixed point computations which, in the finite case, leads to some loss in accuracy.

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 [18]. A natural question arises concerning the relationship between this approach and abstract interpretation. Kuo and Mishra's system is strictly weaker than the standard denotational approaches but Jensen [14] has shown how it can be extended to regain this equivalence. However, no type inference algorithm has been proposed for this system so far. 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.

The goal of this paper is to bridge the gap between these two contrasting approaches. The contribution of the paper is twofold:

- **Methodological**: we start from Jensen's logical system and we derive equivalent systems corresponding to the standard implementation of abstract interpretation and to the frontiers optimisation. The derivation relies on restrictions of the type language. In particular we show that frontiers are special forms of strict types. We believe that describing the various implementation techniques in a common framework and expressing optimisations as particular type restrictions sheds a new light on the algorithmic aspects of static analysis.
- **Technical**: we propose a refinement of the language of types, called *lazy types* and we derive a complete and sound inference system for this language. This algorithm has the same power as usual implementations of abstract interpretation but does not exhibit the same inefficiency problems.

1.1 Overview

We will use strictness analysis as a case study in this paper but the techniques are generally applicable [15, 19]. Abstract interpretation represents the strictness property of a function by an abstract function defined on boolean domains [23]. For instance $g_{abs}\ \mathbf{t}\ \mathbf{f} = \mathbf{f}$ means that the result of a call to 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 $\mathbf{t}$ and $\mathbf{f}$ are now (non-standard) types. Conjunctive types are required to retain the power of abstract interpretation: a strict function like $+$ must have type $(\mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{f}) \wedge (\mathbf{t} \rightarrow \mathbf{f} \rightarrow \mathbf{f})$. Also an entailment relation is defined on types and the corresponding type inference system includes a weakening rule. This is enough to make type inference a non trivial task (let us notice however that such a system does not suffer the undecidability problem of more powerful intersection type systems [1, 2]: this is because we are working with the simply typed $\lambda-$calculus). We first define a 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 weakening rule and to derive an algorithm which corresponds to the naïve 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 for its accuracy but also for its inefficiency. We show that we can avoid some of this inefficiency without losing any of the power of abstract interpretation. The point is that 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_{abs} $\mathbf{t}\ \mathbf{f}\ \mathbf{t}\ \mathbf{f}\ \mathbf{f}\ =\ \mathbf{f}$ 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 [13].

$$\begin{array}{lcl} foldr\ g\ l\ b & = & \mathbf{if}\ l\ =\ nil \\ & & \mathbf{then}\ b \\ & & \mathbf{else}\ g\ (head\ l)\ (foldr\ g\ (tail\ l)\ b) \end{array}$$

$$cat\ l\ =\ foldr\ append\ l\ nil$$

The analysis of this function by abstract interpretation, using a simple frontiers-based approach, is intractable when the abstract domain for lists is the usual four point domain [25]. We consider only simple strictness in this paper but our argument applies to more complex domains as well (and with more algorithmic significance) [15, 20]. Assume that we want to know 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 $(Bool \rightarrow Bool \rightarrow Bool) \rightarrow Bool \rightarrow Bool \rightarrow Bool$ 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 are computed to find that *cat* needs its argument. In our algorithm, the original property to prove is $cat : \mathbf{f} \rightarrow \mathbf{f}$ and this requires proving the following property: $foldr : append \rightarrow \mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{f}$ where the first 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 $l\ =\ nil$ and the body of *foldr* as well. This example shows that 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.

1.2 Paper Organisation

The next section is a brief account of Jensen's logic which is the starting point of the work described here. We define the notion of most general type in Section 3 and we describe the corresponding system. We establish its correctness and completeness with respect to Jensen's logic. Three algorithms are derived from this system: a brute force implementation, an implementation of abstract interpretation and the frontiers optimisation. These are presented in Section 4 as different instantiations of a single generic abstract machine. Section 5 introduces our notion of lazy types and presents a lazy types system equivalent to the previous ones. The algorithm for lazy types construction is described in Section 6 with its correctness and completeness properties. Section 7 is a review of related work and conclusions. Appendix 1 provides more details about the derivation of the abstract machines and Appendix 2 is an example illustrating the lazy types algorithm.

2 Jensen's Strictness Logic

Jensen considers a simply typed $\lambda-$calculus with constants. The terms, Λ_T, are defined by the following syntax:

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

The ordering on types and the program logic are defined in Fig. 1. Γ is an environment mapping variables to formulae (i.e. strictness types). In the rule **Cond-1** σ represents the standard type of e_2 (or e_3). The rule states that the conditional is undefined if the predicate is; the type subscript allows the choice of a representation for undefined which has a type structure which is compatible with the outer context of the conditional expression. **Cond-2** states that any type which can be inferred for both branches is valid for the conditional expression; this is the counterpart of the least upper bound operation used in traditional abstract interpretation. We define $=$ as the equivalence induced by the ordering on types:

$$\sigma = \tau \Leftrightarrow \sigma \leq \tau \text{ and } \tau \leq \sigma$$

3 Most General Types

We introduce a slightly restricted language of strictness formulae in Fig. 2; 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

$$\mathbf{f} \le \phi \qquad \phi \le \phi \qquad \phi \le \mathbf{t} \qquad \mathbf{t}_{\sigma\to\tau} \le \mathbf{t}_\sigma \to \mathbf{t}_\tau$$

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

$$\phi \to \psi_1 \wedge \phi \to \psi_2 \le \phi \to (\psi_1 \wedge \psi_2) \qquad \frac{\phi' \le \phi, \psi \le \psi'}{\phi \to \psi \le \phi' \to \psi'}$$

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

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

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

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

Figure 1: Jensen's Strictness Logic

$$\mathbf{t}, \mathbf{f} \in T_S \qquad \frac{\sigma \in T_I \qquad \psi \in T_S}{\sigma \to \psi \in T_S} \qquad \frac{\phi_1 \in T_S \ldots \phi_n \in T_S}{\phi_1 \wedge \ldots \wedge \phi_n \in T_I}$$

Figure 2: The language T_I

restriction is convenient because it does not weaken the expressive power of the system and it makes type manipulation easier.

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

Definition **3.1**

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

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

$$\begin{array}{lcl} ct(\mathbf{t}) & = & \mathbf{t} \\ ct(\mathbf{f}) & = & \mathbf{f} \\ ct(\sigma \wedge \tau) & = & ct(\sigma) \wedge ct(\tau) \\ ct(\sigma \to \tau) & = & CT(\sigma) \to ct(\tau) \end{array}$$

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

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

Definition **3.2 (Most General Types)**

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

The logic for computing most general types is shown in Fig. 3. C_σ is the set of T_S types compatible with σ (with the same arrow structure). In the following, we always consider types modulo the congruence derived from the following equivalence:

$$a \wedge a \equiv a \qquad a \wedge b \equiv b \wedge a \qquad (a \wedge b) \wedge c \equiv a \wedge (b \wedge c)$$

Modulo this congruence, there is only a finite number of types σ' such that $\sigma = \sigma'$.

The following theorems account for the correctness and the completeness of $\vdash_S$ (with respect to $\vdash_T$) :

Theorem **3.3 (Correctness)**

$$\Gamma \vdash_S e : \sigma \Rightarrow \Gamma \vdash_T e : \sigma$$

Theorem **3.4 (Completeness)**

$$CT(\Gamma) \vdash_S e : MGT(\Gamma, e)$$

The proof of correctness is straightforward. The completeness theorem states that the type derived for an expression e by $\vdash_S$ is (equivalent to) the conjunction of all the types derived for e by $\vdash_T$. Its proof relies on the correctness theorem and the following property:

Property **3.5**

$$\Gamma \le \Delta, \Delta \vdash_T e : \sigma, \sigma \le \sigma' \Rightarrow \exists \tau. CT(\Gamma) \vdash_S e : ct(\sigma') \wedge \tau$$

This property is shown by induction on the length of the proof of $\Delta \vdash_T e : \sigma$.

4 Abstract Interpretation and Frontiers

In this section we present three abstract machines derived from the above logic. All of the machines are instances of a single scheme. The first machine implements the above logic in a fairly direct manner; the other two are optimisations of the first. The first computes the type of abstractions and fixed points by exhaustive search. There are thus two areas for optimisation: the first directs the search for fixed points by building an ascending chain of approximations, the resulting machine is equivalent to a naïve abstract interpretation; the second directs the search for abstractions

$$\textbf{Var}\quad \Gamma[x \mapsto \phi] \vdash_S x : \phi \qquad\qquad \textbf{Taut}\quad \Gamma \vdash_S c : \mathbf{t}$$

$$\textbf{Abs}\quad \frac{\Gamma[x \mapsto \phi_1] \vdash_S e : \psi_1^1 \wedge \ldots \wedge \psi_{n_1}^1 \quad \ldots \quad \Gamma[x \mapsto \phi_k] \vdash_S e : \psi_1^k \wedge \ldots \wedge \psi_{n_k}^k}{\Gamma \vdash_S \lambda x.e : ((\phi_1 \to \psi_1^1) \wedge \ldots \wedge (\phi_1 \to \psi_{n_1}^1) \wedge \ldots \wedge (\phi_k \to \psi_{n_k}^k))}$$

$$\textbf{App}\quad \frac{\begin{array}{l}\Gamma \vdash_S e_1 : ((\phi_1^1 \wedge \ldots \wedge \phi_{n_1}^1) \to \psi_1) \wedge \ldots \wedge ((\phi_1^m \wedge \ldots \wedge \phi_{n_m}^m) \to \psi_m) \\ \Gamma \vdash_S e_2 : \theta_1 \wedge \ldots \wedge \theta_k\end{array}}{\Gamma \vdash_S e_1 e_2 : \bigwedge_i \psi_i \qquad i \ such\ that \left\{\phi_1^i, \ldots \phi_{n_i}^i\right\} \equiv \{\theta_1, \ldots, \theta_k\}}$$

$$\textbf{Fix}\quad \frac{\Gamma \vdash_S (\lambda g.e) : \sigma}{\Gamma \vdash_S \mathbf{fix}(\lambda g.e) : \bigwedge_{i=1}^n \theta_i \qquad with\ (\theta_1 \wedge \ldots \wedge \theta_n \to \theta_i) \in \sigma\ for\ all\ i}$$

$$\textbf{Cond-1}\quad \frac{\Gamma \vdash_S e_1 : \mathbf{f}}{\Gamma \vdash_S \mathbf{cond}(e_1, e_2, e_3) : \bigwedge CT(C_\sigma)}$$

$$\textbf{Cond-2}\quad \frac{\Gamma \vdash_S e_1 : \mathbf{t} \quad \Gamma \vdash_S e_2 : \phi_1^2 \wedge \ldots \wedge \phi_{n_2}^2 \quad \Gamma \vdash_S e_3 : \phi_1^3 \wedge \ldots \wedge \phi_{n_3}^3}{\Gamma \vdash_S \mathbf{cond}(e_1, e_2, e_3) : \bigwedge\{\phi_i \mid \exists k, l. \phi_i = \phi_k^2 = \phi_l^3\}}$$

In the rule **Abs**, we generate the premises by taking all $\phi_i = CT(\sigma_i)$ such that $\sigma_i \in C_\sigma$ with σ the standard type of x.

Figure 3: The Most General Types system

by using a more compact representation of the set of types akin to the notion of frontiers [13].

The machines have three components: an S stack for partial results, an environment E and a code C containing sub-expressions. The schematic machine is shown as a transition system in Fig. 4. We use the following convention: $E[x \mapsto \phi]$ is the property "x is bound to ϕ in E" and $(x : \phi) : E$ represents an environment which is equal to E except that x is bound to ϕ.

To specify a particular machine, we must define the following operations:

$$init_{abs}, initial_{abs}, update_{abs}, iter_{abs}, last_{abs}, comb_{abs},$$

$$init_{fix}, initial_{fix}, update_{fix}, iter_{fix}, last_{fix}, comb_{fix},$$

app, *cond*, *fix* (Fig. 4). To motivate the definition of the schematic abstract machine, consider the logic in Fig. 3. The only complicated transition rule corresponds to the implementation of **Abs**. **Abs** requires an iteration to generate all the $\sigma_i \in C_\sigma$ as indicated in Fig. 3. Since the "body" of a fixed point expression is always an abstraction (see the abstract syntax), fixed point computations also involve iterations. In our later optimisations we will want to treat these two types of iterations (abstractions and fixed points) differently, thus we have separated them in the schematic semantics. With this motivation in mind, the *init* functions pick a start point for the iteration, the *iter* functions choose the next step in the iteration, the *last* functions check for termination and the *comb* functions combine the results of the iterations. The *initial* functions generate a set from which the next iteration is determined; these sets are *update*d on each iteration.

This generic abstract machine can be derived from the type system of the previous section in several stages very much in the spirit of [11]. More details about this derivation are given in Appendix 1. The result of these refinements is an inference system which is an *Abstract Evaluation System* in the terminology of [11]. Such an inference system can alternatively be presented as an abstract machine as we have done in Fig. 4. The correctness proof of the generic machine depends on conditions on the generic instructions that we do not present here. These conditions ensure that the iteration is "complete" with respect to the **Abs** rule of Fig. 3. It can be shown that the three instantiations of this machine described below satisfy these requirements.

Theorem **4.1**

$$\langle S,\ \ \Gamma,\ \ e : C\rangle \rhd^* \langle \phi : S,\ \ \Gamma,\ \ C\rangle \qquad \Leftrightarrow \qquad \Gamma \vdash_S e : \phi$$

4.1 The naïve machine

The following definitions are motivated by consideration of the **Abs** rule in the logic. The operator *initial* generates the set of complete types which correspond to the types in the premises of the rule (except for the type selected by *first*). On each iteration, *iter* chooses another element from this set and *update* removes it. Termination, decided by the predicate *last*, occurs when the set is empty. The *comb* operator builds the result type.

$$\begin{array}{llllll}
init_{abs}(\sigma) & = & init_{fix}(\sigma) & = & first(\sigma) \\
initial_{abs}(\sigma) & = & initial_{fix}(\sigma) & & \\
 & = & CT(C_\sigma)\ -\ first(\sigma) & & \\
iter_{abs}(\Sigma, \sigma, r) & = & iter_{fix}(\Sigma, \sigma, r) & = & \phi \in \Sigma \\
update_{abs}(\Sigma, \sigma, r) & = & update_{fix}(\Sigma, \sigma, r) & & \\
 & = & \Sigma - iter_{abs}(\Sigma, \sigma, r) & & \\
last_{abs}(\Sigma, \sigma, r) & = & last_{fix}(\Sigma, \sigma, r) & = & (\Sigma = \emptyset) \\
comb_{abs}(r, s) & = & comb_{fix}(r, s) & = & r \wedge s
\end{array}$$

We assume that the choice of ϕ is deterministic and *first* selects an arbitrary complete type compatible with σ. The operators *app*, *cond* and *fix* are defined as implied by the type system.

Given these definitions, the rule for fixed points can be simplified to a single rule:

$$\langle S,\ \ E,\ \ \mathbf{fix}(\lambda g.e) : C\rangle \rhd \langle S,\ \ E,\ \ (\lambda g.e) : Fix : C\rangle$$

$$
\begin{array}{rcl}
\langle S,\ \ E[x \mapsto \phi],\ \ x : C\rangle & \rhd & \langle \phi : S,\ \ E[x \mapsto \phi],\ \ C\rangle \\
\langle S,\ \ E,\ \ c : C\rangle & \rhd & \langle \mathbf{t} : S,\ \ E,\ \ C\rangle \\
 & & \\
\langle S,\ \ E,\ \ (\lambda x_\sigma . e) : C\rangle & \rhd & \langle S,\ \ (x : init_{abs}(\sigma)) : E,\ \ e : Abs : Iter_{abs}(x, e, \Sigma) : C\rangle \\
 & & \text{where } \Sigma = initial_{abs}(\sigma) \\
\langle \bigwedge \tau_i : S,\ \ (x : \sigma) : E,\ \ Abs : C\rangle & \rhd & \langle \bigwedge(\sigma \to \tau_i) : S,\ \ (x : \sigma) : E,\ \ C\rangle \\
\langle r : S,\ \ (x : \sigma) : E,\ \ Iter_{abs}(x, e, \Sigma) : C\rangle & \rhd & \\
\multicolumn{3}{r}{\langle r : S,\ \ (x : iter_{abs}(\Sigma, \sigma, r)) : E,\ \ e : Abs : Comb_{abs} : Iter_{abs}(x, e, \Sigma') : C\rangle} \\
 & & \text{where } \Sigma' = update_{abs}(\Sigma, \sigma, r) \\
 & & \text{if } \neg last_{abs}(\Sigma, \sigma, r) \\
\langle r : S,\ \ (x : \sigma) : E,\ \ Iter_{abs}(x, e, \Sigma) : C\rangle & \rhd & \langle r : S,\ \ E,\ \ C\rangle \\
 & & \text{if } last_{abs}(\Sigma, \sigma, r) \\
\langle r_1 : r_2 : S,\ \ E,\ \ Comb_{abs} : C\rangle & \rhd & \langle comb_{abs}(r_1, r_2) : S,\ \ E,\ \ C\rangle \\
 & & \\
\langle S,\ \ E,\ \ (e_1\, e_2) : C\rangle & \rhd & \langle S,\ \ E,\ \ e_1 : e_2 : App : C\rangle \\
\langle S,\ \ E,\ \ \mathbf{cond}(e_1, e_2, e_3) : C\rangle & \rhd & \langle S,\ \ E,\ \ e_1 : e_2 : e_3 : Cond : C\rangle \\
 & & \\
\langle S,\ \ E,\ \ \mathbf{fix}(\lambda g_\sigma . e) : C\rangle & \rhd & \langle S,\ \ (x : init_{fix}(\sigma)) : E,\ \ e : Abs : Iter_{fix}(g, e, \Sigma) : Fix : C\rangle \\
 & & \text{where } \Sigma = initial_{fix}(\sigma) \\
\langle r : S,\ \ (g : \sigma) : E,\ \ Iter_{fix}(g, e, \Sigma) : C\rangle & \rhd & \\
\multicolumn{3}{r}{\langle r : S,\ \ (g : iter_{fix}(\Sigma, \sigma, r)) : E,\ \ e : Abs : Comb_{fix} : Iter_{fix}(g, e, \Sigma') : C\rangle} \\
 & & \text{where } \Sigma' = update_{fix}(\Sigma, \sigma, r) \\
 & & \text{if } \neg last_{fix}(\Sigma, \sigma, r) \\
\langle r : S,\ \ (g : \sigma) : E,\ \ Iter_{fix}(g, e, \Sigma) : C\rangle & \rhd & \langle r : S,\ \ E,\ \ C\rangle \\
 & & \text{if } last_{fix}(\sigma, r) \\
\langle r_1 : r_2 : S,\ \ E,\ \ Comb_{fix} : C\rangle & \rhd & \langle comb_{fix}(r_1, r_2) : S,\ \ E,\ \ C\rangle \\
 & & \\
\langle r_1 : \ldots : r_n : S,\ \ E,\ \ Op : C\rangle & \rhd & \langle op(r_k, \ldots, r_1) : S,\ \ E,\ \ C\rangle \\
 & & Op \in \{App, Fix, Cond\} \text{ of arity } k
\end{array}
$$

Figure 4: The schematic abstract machine.

4.2 The abstract interpretation machine

In abstract interpretation, the search for a fixed point is no longer random but involves the construction of an ascending chain of approximations. We achieve this by modifying the definitions of $init_{fix}$, $initial_{fix}$, $iter_{fix}$, $update_{fix}$, $last_{fix}$, $comb_{fix}$ and fix. The result of $init_{fix}(\sigma)$ is the least complete type compatible with σ. In this algorithm the next iterate is generated from the result of the previous iteration; the set Σ plays no rôle in the computation and so $initial_{fix}$ and $update_{fix}$ are redundant. The iteration terminates when the result from one iteration is equal to the result from the previous iteration: the ascending chain has stabilised. Since the result from the previous iteration is already accounted for in the type generated by the current iteration, $comb_{fix}$ just discards it.

$$
\begin{array}{lcl}
init_{fix}(\sigma) & = & CT(\mathbf{f}_\sigma) \\
initial_{fix}(\sigma) & = & \emptyset \\
iter_{fix}(\Sigma, \sigma, \bigwedge(\sigma \to r_i)) & = & \bigwedge r_i \\
update_{fix}(\Sigma, \sigma, r) & = & \emptyset \\
last_{fix}(\Sigma, \sigma, r) & = & ((\sigma \to \sigma) = r) \\
comb_{fix}(r, s) & = & r \\
fix & = & id
\end{array}
$$

It is straightforward to show the input/output equivalence of this machine and the previous one. We observe that the set of types modulo = is finite. Moreover, the type used for each iteration can be shown to be monotonically increasing. As a consequence, the algorithm is guaranteed to terminate with a fixed point. The correctness is shown with respect to the brute force algorithm by a routine inductive argument.

4.3 The frontiers machine

Considering first-order functions, the machine of the last subsection effectively computes a truth-table representation of the function. Clack and Peyton Jones [5] proposed an alternative representation of first-order functions: rather than representing a function by its truth table, one could just record the maximal argument values at which the result was 0 – this gives a compact representation from which the truth table could be reconstructed. This representation was called the 0-frontier of the function. Hunt and Hankin [13] have shown how this notion can be generalised to higher-order functions over non-flat domains.

We now show how the frontiers optimisation can be incorporated into our work. We compute a restricted form of type which records the maximal argument types which give a result of type $\mathbf{f}$.

The frontiers machine is considerably more complicated than the other two. For simplicity, we consider unary functions whose result type is a basic type. The extension to n-ary functions with basic type results is relatively straightforward. The extension to functions with more complex result types is possible, but rather complex (see [13] for the encodings required).

In this algorithm, the set Σ is used to store the current trial 0-frontier [17]. Initially, it is empty and $init_{abs}$ "selects" the only candidate point. At the end of each iteration, either the last type corresponds to a "frontier" type (the result is $\mathbf{f}$) and the set is unchanged, or it doesn't and the next lowest elements ($preds$) are added to the set. The process terminates when the trial frontier is empty. We redefine the abstraction operators as follows:

$$\begin{array}{lcl}
init_{abs}(\sigma) & = & CT(\mathbf{t}_\sigma) \\
initial_{abs}(\sigma) & = & \emptyset \\
iter_{abs}(\Sigma,\sigma,r) & = & \phi \in next(\Sigma,\sigma,r) \\
update_{abs}(\Sigma,\sigma,r) & = & next(\Sigma,\sigma,r) - iter_{abs}(\Sigma,\sigma,r) \\
last_{abs}(\Sigma,\sigma,r) & = & next(\Sigma,\sigma,r) = \emptyset \\
comb_{abs}(r,s) & = & CT(r \wedge s) \\
preds(\sigma) & = & \text{the maximal elements of} \\
 & & \{\sigma' \in T_S \mid \sigma' < \sigma\} \\
next(\Sigma,\sigma,r) & = & \Sigma, \text{ if } res(r,\sigma) = \mathbf{f} \\
 & = & \Sigma \cup preds(\sigma), \text{ otherwise}
\end{array}$$

where res selects the result type from a functional type, defined as:

$$res(\bigwedge(\sigma \to r_i) \wedge \tau, \sigma) = \bigwedge r_i$$

The fix operators are the same as for the abstract interpretation machine. It is straightforward to show that this machine is correct with respect to the abstract interpretation machine and thus the original logic.

5 Lazy Types

The frontier representation improves the basic algorithm by avoiding the computation of certain types which can be derived from the canonical form. But frontiers do not change the essence of the implementation and still lead to impractical analyses in the presence of larger domains. We take a more radical approach in this section: rather than returning all possible pieces of 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. Lazy types are defined in Fig. 5. The ordering on types and the logic are shown in Fig. 6.

The key idea is that an expression from the term language (with its environment) may appear as part of a type; this plays the rôle of a closure. More formally, a closure (Γ, e) stands for $MGT(\Gamma, e)$, the conjunction of all of the 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 **5.1**

$$Expand : T_G \to T_I$$

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

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

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

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

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

Theorem **5.2 (Correctness)**

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

Theorem **5.3 (Completeness)**

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

Theorem **5.4 (Equivalence)**

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

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 theorems 5.2 and 5.3.

Theorem **5.5**

$$\sigma \leq_G \tau \Leftrightarrow Expand(\sigma) \leq Expand(\tau)$$

Theorem **5.6**

$$\begin{array}{lcl}
\Gamma \vdash_G e : (\phi_1 \wedge \ldots \wedge \phi_n) & \Leftrightarrow & (\Gamma \vdash_G e : \phi_1) \text{ and } \ldots \\
 & & \text{and}(\Gamma \vdash_G e : \phi_n) \\[1ex]
\Gamma \vdash_T e : (\phi_1 \wedge \ldots \wedge \phi_n) & \Leftrightarrow & (\Gamma \vdash_T e : \phi_1) \text{ and } \ldots \\
 & & \text{and}(\Gamma \vdash_T e : \phi_n)
\end{array}$$

Theorem 5.5 can be proved by induction on the proof of the left hand side. Theorem 5.6 is shown by deriving a proof of the right hand side from a proof of the left hand side (it is quite straightforward). Theorem 5.6 allows us to prove theorem 5.2 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 **Cond** rules. A similar property has been proved for other type systems including a form of weakening [1, 21]. Then we use theorems 5.5 and 5.6 and proceed by induction on e to prove completeness.

6 The lazy types algorithm

Applying the same techniques as in Section 4, we can derive an algorithm from the lazy type inference system. Space considerations prevent us from describing the derivation steps and we just present the result in the form of an abstract machine in Fig. 7. The implementation of the Inf instruction is omitted for the same reason; $Inf(\phi, \psi)$ computes $\phi \leq_G \psi$ as defined in Fig. 6 (Theorem 6.2). Notice that a stack element S_i is either a boolean value or a disjunction of types. $True$ (resp. $False$) is installed at the top of the stack if and only if the original property (of the form (e, ϕ)) in the code is (resp. is not) provable in $\vdash_G$. Values which are neither $True$ nor $False$ in the stack are disjunctions of T_G types $(\phi_1 \vee \ldots \vee \phi_n)$. The occurrence of such a value at the top of the stack means that the original property is true if (and only if) the recursive function currently being analysed possesses one of the ϕ_i types (in order to make the presentation simpler we do not consider embedded occurrences of **fix** here; the extension is straightforward). In order to prove that $\mathbf{fix}(\lambda g.e)$ has type ϕ we add the assumption $(g :_r \phi)$ in the environment and try to prove $e : \phi$. If the result is $True$ or $False$ then the case is settled. Otherwise a list of conditions ϕ_i is returned and the algorithm iterates to try to show that one of them is satisfied (rule for $Iter$). Instruction Rec is used to remember that we were trying to prove a property on a recursively defined variable (denoted by $\mapsto_r$ in the environment); so if it fails we just return this property in the stack rather than $False$. Appendix 2 develops an example

$$nil \in env \qquad \frac{\Gamma \in env \quad \sigma \in T_G}{\Gamma[x \mapsto \sigma] \in env}$$

$$\mathbf{t}, \mathbf{f} \in T'_S \qquad \frac{\sigma \in T_G \quad \psi \in T'_S}{\sigma \rightarrow \psi \in T'_S}$$

$$\frac{\Gamma \in env \quad e \in exp}{(\Gamma, e) \in T_G} \qquad \frac{\phi_1 \in T'_S \ldots \phi_n \in T'_S}{\phi_1 \wedge \ldots \wedge \phi_n \in T_G}$$

Figure 5: The language T_G

$$\phi \leq_G \mathbf{t} \qquad \mathbf{f} \leq_G \phi \qquad \phi_1 \rightarrow \ldots \rightarrow \phi_n \rightarrow \phi \leq_G \psi_1 \rightarrow \ldots \rightarrow \psi_n \rightarrow \mathbf{t}$$

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

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

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

$$\mathbf{Abs}\ \frac{\Gamma[x \mapsto \phi] \vdash_G e : \psi}{\Gamma \vdash_G \lambda x.e : (\phi \rightarrow \psi)} \qquad \mathbf{App}\ \frac{\Gamma \vdash_G e_1 : ((\Gamma, e_2) \rightarrow \psi)}{\Gamma \vdash_G e_1 e_2 : \psi}$$

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

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

Figure 6: The Lazy Types system

illustrating the treatment of recursion in the lazy type algorithm. It analyses a function used in [18] to demonstrate the limitations of a type system without conjunction.

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

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

Theorem **6.1**

1. $\langle S, \Gamma, (e, \phi) : C \rangle \rhd^*_G \langle True : S, \Gamma, C \rangle \Leftrightarrow \Gamma \vdash_G e : \phi$

2. $\langle S, \Gamma, (e, \phi) : C \rangle \rhd^*_G \langle False : S, \Gamma, C \rangle \Leftrightarrow \neg(\Gamma \vdash_G e : \phi)$
if Γ *and* ϕ *do not contain any* $\mapsto_r$ *assumptions*

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

Theorem **6.2**

1. $\langle S, \Gamma, Inf(\phi, \psi) : C \rangle \rhd^*_G \langle True : S, \Gamma, C \rangle \Leftrightarrow \phi \leq_G \psi$

2. $\langle S, \Gamma, Inf(\phi, \psi) : C \rangle \rhd^*_G \langle False : S, \Gamma, C \rangle \Leftrightarrow \neg(\phi \leq_G \psi)$
if Γ, ϕ *and* ψ *do not contain any* $\mapsto_r$ *assumption*

Most of the derivation steps to reach the abstract machine are very similar to the ones described in Section 4. The most difficult part of the proof concerns the implementation of **fix**. We have two main facts to prove: (1) the iteration terminates and (2) the result is accurate. Termination is proved by showing that each type $\phi \wedge \phi_i$ satisfies $\phi \wedge \phi_i <_G \phi$. It is easy to show that the result is accurate when the iteration terminates with the *True* answer. In order to show that the initial property cannot be satisfied if the answer is *False*, we prove that at least one of the ϕ_i types returned by the iteration step is a necessary condition to prove the original property (in other words, we do not "bypass" the least fixed point).

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

- The implementation of the conditional can avoid processing the second and third term when the first term has type $\mathbf{f}$.

- In the rule for application, when expression e_2 is a constant or a variable then its type ($\mathbf{t}$ 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.

- When an iteration step returns $\phi_1 \vee \ldots \vee \phi_n$ then each of the ϕ_i is a sufficient condition to prove the original property ϕ. So to prove $(g : \phi \wedge \phi_i) : \Gamma \vdash_G e : \phi \wedge \phi_i$ it is enough to prove $(g : \phi \wedge \phi_i) : \Gamma \vdash_G e : \phi_i$.

These optimisations are easy to justify formally and improve the derivation considerably. The reader can easily check that: $\langle nil, nil, (cat, \mathbf{f} \rightarrow \mathbf{f}) : nil \rangle \rhd^*_G \langle True : nil, nil, nil \rangle$ where *cat* is the function defined in the introduction. A more complex example is described in Appendix 2.

$$
\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 \rightarrow \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) \rightarrow \phi) : C \rangle \\
\\
\langle S, E[x \mapsto \phi], (x, \psi) : C \rangle & \rhd_G & \langle S, E[x \mapsto \phi], Inf(\phi, \psi) : C \rangle \\
\\
\langle S, E, (\mathbf{cond}(e_1, e_2, e_3), \phi) : C \rangle & \rhd_G & \langle S, E, (e_1, \mathbf{f}) : (e_2, \phi) : (e_3, \phi) : And : Or : C \rangle \\
\\
\langle S, (x : \sigma) : E, (D(x)) : C \rangle & \rhd_G & \langle S, E, C \rangle \\
\\
\langle S, E, (\mathbf{fix}(\lambda g.e), \phi) : C \rangle & \rhd_G & \langle S, (g :_r \phi) : E, (e, \phi) : Iter(g, e) : C \rangle \\
\langle S, E[g \mapsto_r \phi], (g : \psi) : C \rangle & \rhd_G & \langle S, E[g \mapsto_r \phi], Inf(\phi, \psi) : (Rec, g, \psi) : C \rangle \\
\langle True : S, E, (Rec, g, \phi) : C \rangle & \rhd_G & \langle True : S, E, C \rangle \\
\langle S_1 : S, E, (Rec, g, \phi) : C \rangle & \rhd_G & \langle \phi : S, E, C \rangle \\
 & & S_1 \neq True \\
\\
\langle S_1 : S, (g :_r \phi) : E, Iter(g, e) : C \rangle & \rhd_G & \langle S_1 : S, E, C \rangle \\
 & & S_1 = True \quad or \quad S_1 = False \\
\\
\langle (\phi_1 \vee \ldots \vee \phi_n) : S, (g :_r \phi) : E, Iter(g, e) : C \rangle & \rhd_G & \\
\multicolumn{3}{r}{\langle S, E, (fix \lambda g.e, \phi \wedge \phi_1) : (fix \lambda g.e, \phi \wedge \phi_2) : Or : \ldots : Or : C \rangle} \\
\\
\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}
$$

Figure 7: The Lazy Types algorithm

7 Related work

To summarise: we claim to make two contributions, one methodological and the other technical. We briefly review related work in these two areas before discussing further work.

The techniques we have used in the first stage of our refinements (Sections 3 and 5) are related to previous work on restricting type systems (see for instance [21] for a type system with subtypes and [1] for a type system with conjunction types and subtypes), especially the transformations required to remove weakening. Our approach to the development of abstract machines from logics (Sections 4 and 6) is closely related to Hannan's and Miller's [11]. For example, the first rule for fixed points can be recast as the result of a folding of inference rules [10, 11]. However our presentation of the generic abstract machine is akin to notions found in denotational semantics: the three abstract machines can be viewed as three different "interpretations". As far as methodology is concerned, we believe that our main contributions are to describe the various stages of refinement in a systematic way in the same conceptual framework (even if the abstract machines have been presented using the usual transition rule syntax for better readability) and to show that standard implementations of abstract interpretation can be inserted quite naturally in this context. For example it is nice to see that the frontiers optimisation can be described as a particular restriction on types.

The main technical contribution of the paper is the notion of lazy types and the corresponding type system and algorithm. This addresses an issue that has taxed the abstract interpretation community greatly. The papers [4, 5, 7, 8, 9, 13, 16, 18] all tackle the same issue. The basic problem is that the choice to abstract functions by functions is a disastrous one for the efficiency of the analysis. We can classify the various proposals to circumvent the problem into two categories: (1) some of them [4, 8, 13, 16] strive for a better representation of abstract functions to improve their computation without losing completeness (with respect to the system of Section 2) while (2) others [6, 7, 9, 18] trade a cheaper implementation of the fixed point against a loss of accuracy. Our algorithm falls into the first category because it is complete with respect to the usual abstract interpretation but it has the syntactic flavour of some of the works in the second category [7, 8, 18, 24]. As noticed earlier, lazy types improve on previous work on frontiers [9, 13]. It is closer in spirit to the minimal function graphs approach in which abstract functions are represented by relations representing the portion of the graph of the function that is needed in a particular computation (its so-called *minimal function graph*). However minimal function graphs are only defined for a first-order language and the extension to higher-order does not seem to be easy. The abstract reduction approach [7, 24] is also based on a form of lazy evaluation but there is no notion of types and it is a symbolic form of expressions which is evaluated lazily. As other methods in its category, abstract reduction may entail an arbitrary cut in the fixed point iteration to ensure termination. Of course this is at the price of accuracy.

There are three major directions in which the work presented here can be extended. On the methodological side, we would like to follow [10] in extracting some general transformations on inference rules of the type studied here to derive abstract machines. On the technical side, we would like to study the integration within our framework of approximation techniques such as widening [6, 22] and its implication on the efficiency of the analysis. Also we would like to extend our work to logics involving data types and richer properties [15]. Finally, we are currently implementing the lazy types algorithm; we hope to be able to report results soon.

Acknowledgements

The first author was partially funded by ESPRIT Working Group 6809 (Semantique II). Pascal Fradet and Thomas Jensen gave helpful comments on an earlier draft.

References

[1] S. van Bakel, *Complete restrictions of the intersection type discipline*, Theoretical Computer Science, 102(1):135-163, 1992.

[2] H. Barendregt, M. Coppo, M. Dezani-Ciancaglini, *A filter lambda model and the completeness of type assignment*, Journal of Symbolic Logic, 48(4), 1983.

[3] 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.

[4] 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.

[5] C. Clack and S. L. Peyton Jones, *Strictness Analysis - A Practical Approach*, in J. P. Jouannaud (ed), *Functional Programming Languages and Computer Architecture*, LNCS 201, Springer Verlag, 1985.

[6] 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.

[7] M. van Eekelen, E. Goubault, C. Hankin and E. Nøker, *Abstract reduction: a theory via abstract interpretation*, in R. Sleep et al (eds), *Term graph rewriting: theory and practice*, John Wiley & Sons Ltd, 1992.

[8] A. Ferguson and R. J. M. Hughes, *Fast abstract interpretation using sequential algorithms*, to appear in the Proceedings WSA'93, Springer Verlag, 1993.

[9] 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.

[10] J. J. Hannan, *Investigating a proof-theoretic metalanguage*, PhD thesis, University of Pennsylvania, DIKU Technical Report Nr 91/1, 1991.

[11] J. Hannan and D. Miller, *From Operational Semantics to Abstract Machines*, Mathematical Structures in Computer Science, 2(4), 1992.

[12] F. Henglein, Efficient type inference for higher-order binding time analysis, in *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS 523, Springer Verlag, 1991.

[13] L. S. Hunt and C. L. Hankin, *Fixed Points and Frontiers: A New Perspective*, Journal of Functional Programming, **1**(1), 1991.

[14] 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.

[15] T. P. Jensen, *Abstract Interpretation in Logical Form*, PhD thesis, University of London, 1992. Also available as DIKU Technical Report 93/11.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] J. C. Mitchell, *Type inference with simple subtypes*, Journal of Functional Programming, **1**(3), 1991.

[22] B. Monsuez, *Polymorphic Typing by Abstract Interpretation*, in *Proceedings of 12th Conference FST & TCS*, Springer Verlag, 1992.

[23] A. Mycroft, *Abstract Interpretation and Optimising Transformations for Applicative Programs*, PhD thesis, University of Edinburgh, December 1981.

[24] E. Nöcker, *Strictness analysis using abstract reduction*, in *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture*, ACM Press, 1993.

[25] P. Wadler, *Strictness Analysis on Non-flat Domains*, in S. Abramsky and C. L. Hankin (eds), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

[26] P. Wadler, *Is there a use for linear logic?*, in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM Press, 1991.

Appendix 1: Deriving The Abstract Machines

The abstract machines presented in Section 4 can be derived in a rather systematic way from the Most General Types system presented in Section 3. In this appendix we provide some intuition about this derivation by showing how the rules for variables and application are successively transformed. The original rules are as follows (Fig. 3):

$$\Gamma[x \mapsto \phi] \vdash_S x : \phi$$

$$\frac{\Gamma \vdash_S e_1 : T_1 \qquad \Gamma \vdash_S e_2 : T_2}{\Gamma \vdash_S e_1 e_2 : App(T_1, T_2)}$$

with

$$T_1 = ((\phi^1_1 \wedge \ldots \wedge \phi^1_{n_1}) \rightarrow \psi_1) \wedge \ldots \wedge ((\phi^m_1 \wedge \ldots \wedge \phi^m_{n_m}) \rightarrow \psi_m)$$

$$T_2 = \theta_1 \wedge \ldots \wedge \theta_k$$

and

$$App(T_1, T_2) = \bigwedge_i \psi_i \qquad i \text{ } such \text{ } that \left\{\phi^i_1, \ldots \phi^i_{n_1}\right\} \equiv \{\theta_1, \ldots, \theta_k\}$$

The first reason why the inference system described in Fig. 3 is not an abstract machine is that some inference rules have several premises. The first refinement ensures that all rules have a single premise. This is achieved by defining a predicate

$M_1 \; : \; list(env) \rightarrow list(e) \rightarrow list(T_I) \rightarrow Bool$
such that
$M_1 \; \overline{\Gamma} \; \overline{e} \; \overline{\sigma} \;\; \Leftrightarrow \;\; \forall i \; \Gamma_i \vdash_S e_i \; : \; \sigma_i.$

M_1 is defined as follows for variables and application:

$$\frac{M_1 \quad E \quad C \quad S}{M_1 \quad \Gamma[x \mapsto \phi] : E \quad x : C \quad \phi : S}$$

$$\frac{M_1 \quad \Gamma : \Gamma : E \quad e_1 : e_2 : C \quad T_1 : T_2 : S}{M_1 \quad \Gamma : E \quad (e_1 \; e_2) : C \quad App(T_1, T_2) : S}$$

$$M_1 \quad nil \quad nil \quad nil$$

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

$$\frac{M_2 \quad \Gamma[x \mapsto \phi] \quad C \quad S}{M_2 \quad \Gamma[x \mapsto \phi] \quad x : C \quad \phi : S}$$

$$\frac{M_2 \quad \Gamma \quad e_1 : e_2 : C \quad T_1 : T_2 : S}{M_2 \quad \Gamma \quad (e_1 \; e_2) : C \quad App(T_1, T_2) : S}$$

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

The only reason why M_2 still does not behave like an abstract machine is that some variables in the premises do not occur in the goal (we can see that this is the case for application by considering the definition of App above). In operational terms, this amounts to saying that the system does not exhibit a tail recursive behaviour. This problem is solved by introducing an extra argument R which is not modified in the rules and will ultimately be instantiated with the result of the computation.

$$\frac{M_3 \quad \Gamma[x \mapsto \phi] \quad C \quad \phi : S \quad R}{M_3 \quad \Gamma[x \mapsto \phi] \quad x : C \quad S \quad R}$$

$$\frac{M_3 \quad \Gamma \quad e_1 : e_2 : App : C \quad S \quad R}{M_3 \quad \Gamma \quad (e_1 \; e_2) : C \quad S \quad R}$$

$$\frac{M_3 \quad \Gamma \quad C \quad App(T_1, T_2) : S \quad R}{M_3 \quad \Gamma \quad App : C \quad T_2 : T_1 : S \quad R}$$

$$M_3 \quad nil \quad nil \quad R : nil \quad R$$

We now have an inference system which is an *Abstract Evaluation System* in the terminology of [10, 11]. This means that we can alternatively present it as a rewriting system describing a machine with three components. We just have to rewrite any rule:

$$\frac{M_3 \quad \Gamma' \quad C' \quad S' \quad R}{M_3 \quad \Gamma \quad C \quad S \quad R}$$

as:

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

Applying this technique and rearranging the order of the arguments, we get the following rules (which are the rules of the schematic abstract machine in Fig. 4).

$$\begin{array}{rcl} \langle S, \;\; E[x \mapsto \phi], \;\; x : C \rangle & \triangleright & \langle \phi : S, \;\; E[x \mapsto \phi], \;\; C \rangle \\ \langle S, \;\; E, \;\; (e_1 \, e_2) : C \rangle & \triangleright & \langle S, \;\; E, \;\; e_1 : e_2 : App : C \rangle \\ \langle T_2 : T_1 : S, \;\; E, \;\; App : C \rangle & \triangleright & \langle App(T_1, T_2) : S, \;\; E, \;\; C \rangle \end{array}$$

Appendix 2: The lazy types algorithm at work

The following function was used in [18] to demonstrate the limitations of a type system without conjunction.

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

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 but the first iteration step returns a necessary condition $(g : T_2)$ which is added to the environment (with $T_2 \;\; = \;\; \mathbf{t} \rightarrow \mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{f}$). 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 $(E, T_1 \wedge T_2)$ from the assumption $(g : (T_1 \wedge T_2))$ and the final result is $True$ as expected.
We use the following notation:

$$\begin{array}{rcl} G & = & \mathbf{fix}(\lambda g.(\lambda x.\lambda y.\lambda z.\mathbf{cond}(eq \; z \; 0)(+ \; x \; y)(g \; y \; x \; (- \; z \; 1)))) \\ E & = & \mathbf{cond}(eq \; z \; 0)(+ \; x \; y)(g \; y \; x \; (- \; z \; 1)) \\ E' & = & (\lambda x.\lambda y.\lambda z.E) \\ T_1 & = & (\mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{t} \rightarrow \mathbf{f}) \\ T_2 & = & (\mathbf{t} \rightarrow \mathbf{f} \rightarrow \mathbf{t} \rightarrow \mathbf{f}) \end{array}$$

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

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

$$\langle nil, (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r T_1) : nil,$$
$$(E, \mathbf{f}) : D(z) : D(y); D(x); Iter(g, E); nil \rangle \quad \triangleright^*_G$$

$$\langle nil, (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r T_1) : nil,$$
$$(((eq \; z \; 0), \mathbf{f}) : ((+ \; x \; y), \mathbf{f}); ((g \; y \; x \; (- \; z \; 1)), \mathbf{f}); And; Or; D(z); D(y); D(x);$$
$$Iter(g, E); nil \rangle \quad \triangleright^*_G$$

$$\langle True : False : nil, (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r T_1) : nil,$$
$$(((g \; y \; x \; (- \; z \; 1)), \mathbf{f}) : And; Or; D(z); D(y); D(x); Iter(g, E); nil \rangle \quad \triangleright^*_G$$

$$\langle True : False : nil, (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r T_1) : nil,$$

$$((g, T_2) : And; Or; D(z); D(y); D(x); Iter(g, E); nil\rangle \quad \rhd^*_G$$

$$\langle False : True : False : nil, (z : \mathbf{t}) : (y : \mathbf{t}) : (x : \mathbf{f}) : (g :_r T_1) : nil,$$
$$(Rec, g, T_2) : And; Or; D(z); D(y); D(x); Iter(g, E); nil\rangle \quad \rhd^*_G$$

$$\langle (T_2) : nil, (g :_r T_1) : nil, Iter(g, E) : nil\rangle \quad \rhd^*_G$$

$$\langle nil, nil, (G, T_1 \wedge T_2) : nil\rangle \quad \rhd^*_G$$

$$\langle nil, (g :_r (T_1 \wedge T_2)) : nil, (E', T_1 \wedge T_2) : Iter(g, E) : nil\rangle \quad \rhd^*_G$$

$$\langle nil, (g :_r (T_1 \wedge T_2)) : nil, (E', T_1) : (E', T_2) : And : Iter(g, E) : nil\rangle \quad \rhd^*_G$$

$$\langle True : nil, (g :_r (T_1 \wedge T_2)) : nil, (E', T_2) : And : Iter(g, E) : nil\rangle \quad \rhd^*_G$$

$$\langle True : False : True : nil, (z : \mathbf{t}) : (y : \mathbf{f}) : (x : \mathbf{t}) : (g :_r (T_1 \wedge T_2)) : nil,$$
$$((g, T_1) : And; Or; D(z); D(y); D(x); And; Iter(g, E); nil\rangle \quad \rhd^*_G$$
$$\langle True : True : False : True : nil, (z : \mathbf{t}) : (y : \mathbf{f}) : (x : \mathbf{t}) : (g :_r (T_1 \wedge T_2)) :$$
$$nil, (Rec, g, T_1) : And; Or; D(z); D(y); D(x); And; Iter(g, E); nil\rangle \quad \rhd^*_G$$

$$\langle True : True : nil, (g :_r (T_1 \wedge T_2)) : nil, And : Iter(g, E); nil\rangle \quad \rhd^*_G$$

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