A Type-based Framework for Program Analysis

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

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

Abstract. In this paper we present a general framework for type-based analyses of functional programs. Our framework is a generalisation of our earlier work on strictness analysis and was inspired by Burn's logical framework. The framework is parameterised by a set of types to represent properties and interpretations for constants in the language. To construct a new analysis, the user needs only to supply a model for the types (which properties they denote) and sound rules for the constants. We identify the local properties that must be proven to guarantee the correctness of a specific analysis and algorithm. We illustrate the approach by recasting Hunt and Sand's binding time analysis in our framework. Furthermore we report on experimental results suggesting that our generic inference algorithm can provide the basis for an efficient program analyser.

1 Introduction

The first explicit use of types in program analysis was by Kuo and Mishra [14]. They presented a type system for inferring strictness of functional programs. The potential attraction of their approach, compared to abstract interpretation, was the possibility of designing efficient algorithms to do the analysis. The disadvantage was that the analysis was weaker, providing less useful information than abstract interpretation. This weakness of the approach has been addressed by Jensen [12, 13] and Benton [2, 3]. Both recapture the power of abstract interpretation by adding intersection types but in doing so they lose the direct connection to efficient algorithms. Their work has focused on strictness analysis, although [13] also considers binding time analysis. Burn [4] has generalised the framework but still does not address algorithmic aspects.

Recently, in [5, 6], we have shown how efficient algorithms can be derived from these earlier systems. The key idea, elaborated below, has been the use of "lazy" types. Our main focus has been on strictness analysis. In this paper we follow [4] in generalising our earlier work; we present a schematic logic which is parameterised by the set of basic types and inference rules for constants. To construct a new analysis, the user must provide the following:

1. The list of constants of the language.
2. The list of type constants.
3. A type structure which includes a typing for the language constants and a mapping from types to sets of values in the standard domain.
4. The type inference rules for the language constants and the verification of the associated correctness conditions.

5. Rules defining the treatment of the constants by the abstract machine and their correctness proof.

The properties to be proven for a particular instantiation occur as verification requirements of the correctness property of the generic system.

In the next section we introduce the notion of *type structure* and discuss some general requirements on types and their interpretation. In Section 3 we present a logic which is a subset of Burn's [4] (omitting disjunctions and pairs) and sketch the proof of correctness of this logic. In the next section we present our notion of lazy types and the associated algorithm. Section 5 contains a first instantiation of the framework: a binding time analysis modeled using PER's [11]. Section 6 presents a second instantiation, a strictness analysis of lists, and Section 7 reports on experimental results. Section 8 summarises the achievements of this paper and suggests avenues for further research.

Throughout this paper we will consider a simply typed λ-calculus with constants, which are unspecified for the moment, and fixed points:

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

2 Types and their interpretation

We assume some sets of type constants which are pre-ordered by $\leq$. Examples might be the set:

$$B_{ss} = \{\mathbf{f}, \mathbf{t}\} \text{ with } \mathbf{f} \leq \mathbf{t},$$
$$B_{sl} = \{\mathbf{f}, \infty, \mathbf{f}_{\in}, \mathbf{t}\} \text{ with } \mathbf{f} \leq \infty \leq \mathbf{f}_{\in} \leq \mathbf{t}$$
$$B_{bt} = \{\mathit{static}, \mathit{dynamic}\} \text{ with } \mathit{static} \leq \mathit{dynamic}$$

used respectively in simple strictness logics, strictness analysis of lists and binding time analysis. More complex types can be constructed using $\wedge$ (intersection or conjunction) and $\rightarrow$ (functions). The ordering $\leq$ is extended to constructed types as shown in Fig. 1.

$$\phi \leq \phi$$

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

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

Fig. 1. The ordering on types

A property is a "collection" of objects from some standard domain of discourse, D. Concrete examples of precisely what is meant by the word *collection* are given below. Type formulae represent such properties.

Definition 1. A *type structure* $\mathcal{M}$ is a tuple $(X, \sqsubseteq, \sqcap, \Rightarrow, norm)$, where

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

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

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

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

Given a particular structure, $\mathcal{M}$ and an interpretation of the type constants $\mathcal{I} : B \to X$ 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 2. The structure is a model, if for all ϕ and ψ:

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

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, [4], 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.

2.1 First example : complete PERs

Hunt [9] has proposed that strictness properties should be represented by complete (i.e. strict and inductive) partial equivalence relations (CPERs). The motivation for this suggestion is that certain properties which cannot be represented by Scott-closed sets can be represented. Hunt and Sands have used CPERs in binding time analysis [11]. A PER on a set D is a binary relation which is symmetric and transitive. A PER, P, is *strict* if:

$$\perp P \perp$$

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

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

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

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

where

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

The requirements of Definition 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. The binding time analysis example is fully described in Section 5.

2.2 Second example: Scott-closed sets

Scott-closed sets have been used to represent strictness properties. Recall that a Scott-closed set S is:

- left-closed, i.e. if $y \sqsubseteq x$ and $x \in S$ then $y \in S$.
- for all chains $\{x_n\}$ in S, $\bigsqcup\{x_n\} \in S$.

Let $\mathcal{SCS}(D)$ be the set of Scott-closed sets on D. We define the Scott-closed set structure as follows:

$$\mathcal{M}_{scs} = (\mathcal{SCS}(D), \sqsubseteq_{scs}, \sqcap_{scs}, \Rightarrow_{scs}, norm_{scs})$$

where

- $\sqsubseteq_{scs} = \subseteq$
- $\sqcap_{scs} = \cap$
- $S_1 \Rightarrow_{scs} S_2 = \{f \mid \forall s_1 \in S_1 . f\ s_1 \in S_2\}$
- $norm_{scs}(S) = S$

Again the requirements of Definition 1 are trivially satisfied.

3 Type-based analysis

A logic based on Burn's, [4], is shown in Fig. 2. We give a standard denotational semantics in Fig. 3; we assume a (family of typed) function(s) $\mathcal{K}$ which gives the usual interpretations to constants.

Definition 3. We use the following notation:

- $d \models_{el} \phi \equiv d \in norm([\![\phi]\!])$
- $\rho \models_{el} \Gamma \equiv \forall x. \rho\ x \models_{el} \Gamma\ x$
- $\Gamma \models e : \phi \equiv \forall \rho \models_{el} \Gamma . \mathcal{S}[\![e]\!]\rho \models_{el} \phi$

Correctness of the logic in Fig. 2 follows from the following theorem, which is a variant of Theorem 5.2 in [4]. Notice that in common with [2, 4], but in contrast to [12], correctness is established with respect to a standard semantics (as opposed to an abstract interpretation). Burn claims that this makes the addition of disjunctions easier; the cost is some additional complexity in the statement of the correctness (the use of the *norm* function - which is necessary but did not feature in Burn's framework).

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

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

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

Fig. 2. The logic

$$\begin{aligned}
\mathcal{S}[\![x]\!]\rho &= \rho\ x \\
\mathcal{S}[\![c]\!]\rho &= \mathcal{K}[\![c]\!] \\
\mathcal{S}[\![e_1\ e_2]\!]\rho &= (\mathcal{S}[\![e_1]\!]\rho)(\mathcal{S}[\![e_2]\!]\rho) \\
\mathcal{S}[\![\lambda x.e]\!]\rho &= \lambda d.\mathcal{S}[\![e]\!]\rho[x := d] \\
\mathcal{S}[\![\mathbf{fix}\lambda g.e]\!]\rho &= \textstyle\bigsqcup_{i \geq 0}(\mathcal{S}[\![\lambda g.e]\!]\rho)^i \bot
\end{aligned}$$

Fig. 3. The denotational semantics

Definition 4. A rule for an n-ary constant:

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

is *sound* if, under the assumption that:

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

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

Theorem 5. *Suppose that each rule for each n-ary constant is sound. Let e be a term of type σ and let Γ be an environment. We then have:*

$$\Gamma \vdash_T e : \phi \textit{ implies } \Gamma \models e : \phi$$

where ϕ is a formula consistent with σ.

The condition in the theorem is a property of constants that has to be checked for any particular instantiation of the general scheme.

Proof. This is proved by induction over the shape of the derivation tree. We consider two cases.
Var: follows since $\rho \models_{el} \Gamma$.
Abs: by IH $\Gamma[x \mapsto \phi] \models e : \psi$. That is, for any ρ' ($\equiv \rho[x \mapsto v]$) such that $\rho' \models_{el} \Gamma[x \mapsto \phi]$, we have $\mathcal{S}[\![e]\!]\rho' \models_{el} \psi$ and $\rho'\ x \models_{el} \phi$. Then $\mathcal{S}[\![\lambda x.e]\!]\rho \models_{el} \phi \rightarrow \psi$.

4 Lazy types and algorithms

The logic in Fig. 2 is not immediately suggestive of an algorithm. Two problems are multi-premise rules and the weakening rule. Multi-premise rules pose a strategy problem and the weakening rule introduces some non-determinism. In [5, 6] we show how to derive an algorithm by modifying the logic using techniques similar to those used in [7, 8]. Jensen's work [12] assures us that an algorithm exists because he establishes an exact correspondence with abstract interpretations for which algorithms are known. Unfortunately, the derived algorithms have the same complexity as the abstract interpretation algorithms; they compute every property of the function when only a few are actually needed. We overcome this by introducing *lazy types*: we allow types to have "closures", introduced by the application rule, embedded in them which are lazily evaluated as required. The following presentation is slightly different from [5, 6] but the formal definition of lazy types remains unchanged.

We introduce two new parameterised type symbols: ∇_e^Γ and Δ_e^Γ. The type ∇_e^Γ symbolically represents the set of types that are derivable for e, using the assumptions Γ:

$$\nabla_e^\Gamma = \{\psi \mid \Gamma \vdash_T e : \psi\}$$

and Δ_e^Γ symbolically represents the conjunctive type $\bigwedge \nabla_e^\Gamma$. Notice that e is an expression in the simply typed λ-calculus and ψ is required to be a finite conjunction of formulae which have the same arrow structure as the type of e; as a consequence, if we have a finite set of constant types then ∇_e^Γ is a finite set.

Formulae may now include types of the form Δ_e^Γ and we extend the ordering on types in the following way:

$$\frac{\Gamma \vdash_G e : \phi}{\Delta_e^\Gamma \leq \phi}\ (\phi \neq \Delta_e'^\Gamma) \qquad\qquad \frac{\forall \phi . (\Gamma \vdash_G e : \phi) \Rightarrow \psi \leq \phi}{\psi \leq \Delta_e^\Gamma}$$

We define a function *Expand* which converts from lazy types to standard types; the function acts as the identity on constants and:

$$\begin{aligned} Expand(\sigma_1 \wedge \sigma_2) &= Expand(\sigma_1) \wedge Expand(\sigma_2) \\ Expand(\sigma_1 \rightarrow \sigma_2) &= Expand(\sigma_1) \rightarrow Expand(\sigma_2) \\ Expand(\Delta_e^\Gamma) &= \bigwedge\{\psi \mid Expand(\Gamma) \vdash_T e : \psi\} \end{aligned}$$

The lazy types system is shown in Fig. 4; correctness of this logic follows from the following theorem.

Theorem 6.
(i) $\psi_1 \leq \psi_2 \ \Rightarrow\ Expand(\psi_1) \leq Expand(\psi_2)$
(ii) $\Gamma \vdash_G e : \phi \ \Rightarrow\ Expand(\Gamma) \vdash_T e : Expand(\phi)$

Proof. Both parts are proved by a simultaneous induction over the $\leq$ and $\vdash_G$ relations. We just consider a few cases.

$\Delta_e^\Gamma \leq \phi$ because $\Gamma \vdash_G e : \phi$: by IH $Expand(\Gamma) \vdash_T e : Expand(\phi)$, and thus $\bigwedge\{\psi \mid Expand(\Gamma) \vdash_T e : \psi\} \leq Expand(\phi)$.

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

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

$$\textbf{App}\quad \frac{\Gamma \vdash_G e_1 : \Delta^{\Gamma}_{e_2} \to \psi}{\Gamma \vdash_G e_1 e_2 : \psi}$$

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

Fig. 4. The Lazy Types system

Var: $Expand(\Gamma) \vdash_T e : Expand(\psi_1)$ but then $Expand(\Gamma) \vdash_T e : Expand(\psi_2)$ follows from the IH and **Weak**.

App: by IH $Expand(\Gamma) \vdash_T e_1 : Expand(\Delta^{\Gamma}_{e_2} \to \psi)$, i.e.

$$Expand(\Gamma) \vdash_T e_1 : \bigwedge\{\phi \mid Expand(\Gamma) \vdash_T e_2 : \phi\} \to Expand(\psi)$$

For any $\psi \in \{\phi \mid Expand(\Gamma) \vdash_T e_2 : \phi\}$, we clearly have:

$$Expand(\Gamma) \vdash_T e_2 : \psi$$

and thus, repeated use of **Conj**,

$$Expand(\Gamma) \vdash_T e_2 : \bigwedge\{\phi \mid Expand(\Gamma) \vdash_T e_2 : \phi\}$$

Finally, using **App**, we have

$$Expand(\Gamma) \vdash_T e_1\ e_2 : Expand(\psi)$$

In [5], we also consider completeness of the lazy types system but this correctness result is sufficient for our purposes here. In *op. cit.* we also show how to derive an abstract machine for strictness analysis from the type system. The machine derived from the system of Fig. 4 is shown in Fig. 5. The state of the machine is represented by a triple: a stack, an environment of type assumptions and the code (which contains pairs consisting of an expression and a putative type). The *Inf* instruction computes the $\leq$ predicate, it is defined in Fig. 6; this definition must be extended to take account of the type constants for a particular analysis.

The correctness of the strictness machine is considered in some detail in [5]. A generalised statement of correctness is given below:

$$
\begin{array}{rl}
\langle S, E, (e, \phi_1 \wedge \phi_2) : C\rangle & \rhd_G \langle S, E, (e, \phi_1) : (e, \phi_2) : And : C\rangle \\[1em]
\langle S, E, (\lambda x.e, \sigma \to \tau) : C\rangle & \rhd_G \langle S, (x : \sigma) : E, (e, \tau) : D(x) : C\rangle \\[1em]
\langle S, E, (e_1 e_2, \phi) : C\rangle & \rhd_G \langle S, E, (e_1, (E, e_2) \to \phi) : C\rangle \\[1em]
\langle S, E[x \mapsto \phi], (x, \psi) : C\rangle & \rhd_G \langle S, E[x \mapsto \phi], Inf(\phi, \psi) : C\rangle \\[1em]
\langle S, (x : \sigma) : E, (D(x)) : C\rangle & \rhd_G \langle S, E, C\rangle \\[1em]
\langle S, E, (\mathbf{fix}(\lambda g.e), \phi) : C\rangle & \rhd_G \\
 & \quad \langle S, (g :_r \phi) : E, (e, \phi) : Iter(g, e) : C\rangle \\
\langle S, E[g \mapsto_r \phi], (g, \psi) : C\rangle & \rhd_G \\
 & \quad \langle S, E[g \mapsto_r \phi], Inf(\phi, \psi) : Rec(g, \psi) : C\rangle \\
\langle True : S, E, Rec(g, \phi) : C\rangle & \rhd_G \langle True : S, E, C\rangle \\
\langle S_1 : S, E, Rec(g, \phi) : C\rangle & \rhd_G \langle \phi : S, E, C\rangle \\
 & \quad S_1 \neq True \\[1em]
\langle S_1 : S, (g :_r \phi) : E, Iter(g, e) : C\rangle & \rhd_G \langle S_1 : S, E, C\rangle \\
 & \quad S_1 = True \quad or \quad S_1 = False \\[1em]
\langle (\phi_1 \vee \ldots \vee \phi_n) : S, (g :_r \phi) : E, Iter(g, e) : C\rangle & \rhd_G \\
\multicolumn{2}{r}{\langle S, E, (\mathbf{fix}(\lambda g.e), \phi \;\wedge\; \phi_1) : \ldots : (\mathbf{fix}(\lambda g.e), \phi \;\wedge\; \phi_n) : Or : \ldots : Or : C\rangle} \\[1em]
\langle S_1 : S_2 : S, E, Op : C\rangle & \rhd_G \langle (Op\ S_1\ S_2) : S, E, C\rangle \\
 & \quad Op = And \quad or \quad Op = Or
\end{array}
$$

Fig. 5. The Lazy Types algorithm

Theorem 7.

Suppose that each transition rule for each constant is sound, then:

1. $\langle S, \Gamma, (e, \phi) : C\rangle \ \rhd_G^* \ \langle True : S, \Gamma, C\rangle \Rightarrow \Gamma \vdash_G e : \phi$
2. $\langle S, \Gamma, (e, \phi) : C\rangle \ \rhd_G^* \ \langle False : S, \Gamma, C\rangle \Leftarrow \neg(\Gamma \vdash_G e : \phi)$
   if Γ and ϕ do not contain any $\mapsto_r$ assumptions

The condition in the theorem is a property of constants that has to be checked for any particular instantiation of the general scheme.

Since there is close correspondence between the transition rules of the machine and the type system, the proof is relatively straightforward; the only subtlety is in the case of fixed points which are implemented by an iterative process in the abstract machine – see [5] for a more detailed discussion.

$$\langle S, E, Inf(\phi,\phi) : C\rangle \ \triangleright_G \langle True : S, E, C\rangle$$
$$\langle S, E, Inf(\phi_1 \wedge \ldots \wedge \phi_n, \psi_1 \wedge \ldots \wedge \psi_m) : C\rangle \ \triangleright_G$$
$$\langle S, E, Inf(\phi_1,\psi_1) : \ldots : Inf(\phi_n,\psi_1) : Or : \ldots Inf(\phi_1,\psi_m) :$$
$$\ldots : Inf(\phi_n,\psi_m) : Or : And : C\rangle$$
$$\langle S, E, Inf(\psi, \Delta_e^\Gamma) : C\rangle \ \triangleright_G$$
$$\langle S, \Gamma, (e,\phi_1) : Cond(False, True) : Inf(\psi,\phi_1) :$$
$$\ldots (e,\phi_k) : Cond(False, True) : Inf(\psi,\phi_k) : And : Setenv(E) : C\rangle$$
with $\phi_1, \ldots \phi_k$ the types compatible with the standard type of e.

$$\langle S, E, Inf(\Delta_e^\Gamma, \phi) : C\rangle \ \triangleright_G \langle S, \Gamma, (e,\phi) : Setenv(E) : C\rangle$$
with $\phi \neq \Delta_{e'}^{\Gamma'}$

$$\langle S, E, Inf(\phi \to \psi, \phi' \to \psi') : C\rangle \ \triangleright_G$$
$$\langle S, E, Inf(\phi',\phi) : Inf(\psi,\psi') : And : C\rangle$$
$$\langle S, E, Setenv(E') : C\rangle \ \triangleright_G \langle S, E', C\rangle$$
$$\langle B_1 : S, E, Cond(B_1, B_2) : C_0 : C\rangle \ \triangleright_G \langle B_2 : S, E, C\rangle$$
$$\langle B_1' : S, E, Cond(B_1, B_2) : C_0 : C\rangle \ \triangleright_G \langle S, E, C_0 : C\rangle$$
with $B_1 \neq B_1'$

Fig. 6. The Inf instruction

5 First example: binding time analysis

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. This analysis has been extensively studied; in this section we show how the analysis of [11] can be recast in our framework.

5.1 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.

5.2 Type constants

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

5.3 Type structure

We model types using complete PERS as defined in section 2 and the following interpretation of constants:

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

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

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

and the structure is a model.

5.4 Type inference rules

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

$$\textbf{Const} \quad \vdash_G c : static$$

which is sound since $norm([\![static]\!])$ is D. The rule scheme for + and the conditional are respectively:

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

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

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

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

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 $\mathcal{S}[\![b]\!]\rho$ is $\perp$ then $\mathcal{S}[\![\textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2]\!]\rho = \perp$ 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 $\mathcal{S}[\![b]\!]\rho \neq \perp$ then the soundness result is immediate.

5.5 Transition Rules

We add the new rules shown in Fig. 7.

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

$$\langle S, E, Inf(static, dynamic) : C\rangle \ \triangleright_G \langle True : S, E, C\rangle$$

$$\langle S, E, (c, static) : C\rangle \ \triangleright_G \langle True : S, E, C\rangle$$
$$\langle S, E, (c, dynamic) : C\rangle \ \triangleright_G \langle True : S, E, C\rangle$$

$$\langle S, E, (+(e_1, e_2), static) : C\rangle \ \triangleright_G \langle S, E, (e_1, static) : (e_2, static) : And : C\rangle$$
$$\langle S, E, (+(e_1, e_2), dynamic) : C\rangle \ \triangleright_G \langle True : S, E, C\rangle$$

$$\langle S, E, (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, \phi) : C\rangle \ \triangleright_G \langle S, E, (e_1, static) : (e_2, \phi) : (e_3, \phi) : And : And : C\rangle$$

$$\langle S, E, (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, dynamic) : C\rangle \ \triangleright_G \langle True : S, E, C\rangle$$

Fig. 7. The new transitions for the binding time analysis algorithm

6 Second example: strictness analysis of lists

Simple strictness analysis yields properties of the form $f\ \bot = \bot$. The strictness analysis of lists aims at finding more precise information about list manipulating functions. We consider here a set of four properties originally proposed by P. Wadler [16]. We follow the same plan as above to show how our generic framework can be specialised to the strictness analysis of lists.

6.1 Constants of the language

In addition to the conditional function, we consider the usual list manipulation primitives and a case operator used to express pattern matching:

$$\textbf{cond}, \textbf{nil}, \textbf{cons}, \textbf{hd}, \textbf{tl}, \textbf{case}$$

As an example, the function *append*:

$$\begin{aligned} &append\ \textbf{nil}\ l &&= l \\ &append\ \textbf{cons}(x, xs)\ l &&= \textbf{cons}(x, (append\ xs\ l)) \end{aligned}$$

can be rewritten as follows in our kernel language:

$$append\ m\ l\ =\ \textbf{case}(l, f, m) \text{where } f\ x\ xs\ =\ \textbf{cons}(x, (append\ xs\ l))$$

6.2 Type constants

Four type constants correspond to the four strictness properties of [16].

$$B_{sl} = \{\mathbf{f}, \infty, \mathbf{f}_\in, \mathbf{t}\} \text{ with } \mathbf{f} \leq \infty \leq \mathbf{f}_\in \leq \mathbf{t}$$

where ∞ represents infinite lists or lists ending with an undefined element and $\mathbf{f}_\in$ corresponds to finite lists whose elements may be undefined (plus the lists represented by ∞).

6.3 Type structure

We follow [4] and use Scott-closed sets to model the properties of lists. Constants are interpreted as follows:

$$
\begin{aligned}
I_{scs}(\mathbf{f}) &= \{\bot\} \\
I_{scs}(\infty) &= \{l \mid infinite(l) \ or \ \exists n.\mathbf{tl}^n(l) = \bot\} \\
I_{scs}(\mathbf{f}_\in) &= \{l \mid infinite(l) \ or \ \exists n.\mathbf{tl}^n(l) = \bot \ \ or \ \exists n.\mathbf{hd}(\mathbf{tl}^n(l)) = \bot\} \\
I_{scs}(\mathbf{t}) &= D
\end{aligned}
$$

It is straightforward to verify that $I_{scs}(\mathbf{f}) \subseteq I_{scs}(\infty) \subseteq I_{scs}(\mathbf{f}_\in) \subseteq I_{scs}(\mathbf{t})$ and thus $\phi \leq \psi$ implies $[\![\phi]\!] \sqsubseteq [\![\psi]\!]$ and the structure is a model.

6.4 Type inference rules

The type inference rules for this particular instantiation of the framework are described in detail in [6]. So we just give the rule for **cons** here to provide some intuition and we refer the reader to the above reference for a comprehensive presentation.

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

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

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

6.5 Transition rules

We just give the rules for **cons** and we again refer the reader to [6] for the full treatment.

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

7 Experimental Results

The lazy type algorithm has been implemented (in CAML light) as an interpreter realising the abstract machine described in section 4. We report here on experimental results for the stricness analysis of lists described above. This particular instantiation is especially relevant for assessing the efficiency of the analyser because it has proved to be a major challenge in the functional programming community [9]. We use as a testbed two versions of a function concatenating lists

$$
\begin{array}{ll}
K\ x\ y & = x \\
\\
isnil\ \mathbf{nil} & = \mathbf{True} \\
isnil\ \mathbf{cons}(x, xs) & = \mathbf{False} \\
\\
length\ \mathbf{nil} & = 0 \\
length\ \mathbf{cons}(x, xs) & = 1 + (length\ xs) \\
\\
sum\ \mathbf{nil} & = 0 \\
sum\ \mathbf{cons}(x, xs) & = x + (length\ xs) \\
\\
test_1\ l & = Ccat\ l\ K \\
test_2\ l & = Ccat\ l\ isnil \\
test_3\ l & = Ccat\ l\ length \\
test_4\ l & = Ccat\ l\ sum
\end{array}
$$

Fig. 8. Testbed

of lists, *cat* and *Ccat*, the second one being defined in terms of continuations. These examples were provided by S. Hunt to illustrate the limitations of the frontiers optimisation [9, 10].

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

$cat : \mathbf{f}_\in \rightarrow \mathbf{f}$	*False*	$0.02\ s$
$cat : \infty \rightarrow \mathbf{f}$	*False*	$0.03\ s$
$cat : \infty \rightarrow \infty$	*True*	$0.01\ s$
$test_1 : \mathbf{f}_\in \rightarrow \mathbf{f}$	*False*	$0.08\ s$
$test_1 : \infty \rightarrow \mathbf{f}$	*True*	$0.4\ s$
$test_2 : \mathbf{f}_\in \rightarrow \mathbf{f}$	*False*	$0.25\ s$
$test_2 : \infty \rightarrow \mathbf{f}$	*True*	$5.2\ s$
$test_3 : \mathbf{f}_\in \rightarrow \mathbf{f}$	*True*	$0.88\ s$
$test_3 : \infty \rightarrow \mathbf{f}$	*True*	$3.82\ s$
$test_4 : \mathbf{f}_\in \rightarrow \mathbf{f}$	*True*	$0.08\ s$
$test_4 : \infty \rightarrow \mathbf{f}$	*True*	$0.4\ s$

Fig. 9. Experimental results

These results should be compared with other implementations. The contrast with frontiers based "optimisations" of abstract interpretation is striking: the analysis of [10] takes 30 minutes to process *cat* and does not terminate for examples involving *Ccat*. The basic reason is that abstract interpretation based analyses systematically compute all the properties satisfied by a function; when the function is higher-order this can involve a vast amount of information. It turns out that very often only a small part of this information is really necessary. It may be argued that for a fairer comparison we should add the execution times to compute the answers to all possible questions in the lazy type algorithm. Even so our algorithm performs much better (less than 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.

8 Conclusions

The contribution of this paper is twofold: we establish a general framework for the construction of algorithms from type-based program analyses and we provide experimental evidence that the inference algorithm based on the lazy evaluation of properties can serve as a basis of an efficient analyser.

The problem of proving the correctness of a particular analysis and algorithm has been reduced to a local correctness proof concerning interpretations of basic types and constants. We have illustrated the approach by a binding time analysis based on [11] which uses complete PERs to represent properties. Our previous work on strictness analysis [5, 6] can be recast into this framework but we use Scott-closed sets for the interpretation of types. The use of CPERs for strictness analysis of lists would allow us to capture richer properties such as head strictness [9].

The techniques that we have developed should be portable to other programming paradigms; this remains a topic for further investigation.

Acknowledgements

The authors were inspired to consider binding time analysis after a conversation with Hanne Riis Nielson. The presentation of lazy types was suggested by Henk Barendregt. We thank Sebastian Hunt for providing the benchmark examples and Valérie Gouranton and Ronan Gaugne for their work on the implementation. The first author is partially funded by ESPRIT Working Group 6809 (Semantique).

References

1. S. van Bakel, *Complete restrictions of the intersection type discipline*, Theoretical Computer Science, 102(1):135-163, 1992.
2. P. N. Benton, *Strictness logic and polymorphic invariance*, in *Proceedings of the 2nd Int. Symposium on Logical Foundations of Computer Science*, LNCS 620, Springer Verlag, 1992.
3. P. N. Benton, *Strictness Properties of Lazy Algebraic Datatypes*, in *Proceedings WSA'93*, LNCS 724, Springer Verlag, 1993.

4. G. L. Burn, *A Logical Framework for Program Analysis*, in *Proceedings of the 1992 Glasgow Functional Programming Workshop*, Springer Verlag Workshops in Computer Science, 1992.
5. C. L. Hankin and D. Le Métayer, *Deriving algorithms from type inference systems: Application to strictness analysis*, in *Proceedings of POPL'94*, ACM Press, 1994.
6. C. L. Hankin and D. Le Métayer, *Lazy type inference for the strictness analysis of lists*, in *Proceedings of ESOP'94*, LNCS 788, Springer Verlag, 1994.
7. J. J. Hannan, *Investigating a proof-theoretic meta-language*, PhD thesis, University of Pennsylvania, DIKU Technical Report Nr 91/1, 1991.
8. J. Hannan and D. Miller, *From Operational Semantics to Abstract Machines*, Mathematical Structures in Computer Science, 2(4), 1992.
9. L. S. Hunt, *Abstract Interpretation of Functional Languages: From Theory to Practice*, PhD thesis, Imperial College, 1991.
10. L. S. Hunt and C. L. Hankin, *Fixed Points and Frontiers: A New Perspective*, Journal of Functional Programming, **1**(1), 1991.
11. L. S. Hunt and D. Sands, *Binding Time Analysis: A new PERspective*, in Proceedings of the ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation, 1991.
12. 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.
13. T. P. Jensen, *Abstract Interpretation in Logical Form*, PhD thesis, University of London, 1992. Also available as DIKU Technical Report 93/11.
14. 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.
15. 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.
16. P. Wadler, *Strictness Analysis on Non-flat Domains*, in S. Abramsky and C. L. Hankin (eds), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.