Binding Time Analysis: A New PERspective*

Sebastian Hunt† and David Sands‡

Department of Computing, Imperial College
180 Queens Gate, London SW7 2BZ
email: {lsh,ds}@uk.ac.ic.doc

Abstract

Given a description of the parameters in a program that will be known at partial evaluation time, a binding time analysis must determine which parts of the program are dependent solely on these known parts (and therefore also known at partial evaluation time). In this paper a binding time analysis for the simply typed lambda calculus is presented. The analysis takes the form of an abstract interpretation and uses a novel formalisation of the problem of binding time analysis, based on the use of partial equivalence relations. A simple proof of correctness is achieved by the use of logical relations.

1 Introduction

Given a description of the parameters in a program that will be known at partial evaluation time, a binding time analysis must determine which parts of the program are dependent solely on these known parts (and therefore also known at partial evaluation time). A binding time analysis performed prior to the partial evaluation process can have several practical benefits (see [Jon88]), and has been shown to be essential in some approaches to the generation of efficient compilers from interpreters [JSS85].

In this paper we present a form of binding time analysis for a simply typed λ-calculus with constants, and including list-types. The analysis can be viewed as a dependency analysis: given a description of which parts (substructures) of a function's input are fixed (static), we compute (an approximation to) the parts of the result that are completely determined by the static parts of the input (i.e. that are also static). The results of the analysis can be used as the basis of various global (sticky) versions in which parts of a program are annotated with binding time information. Such global forms of the analysis are beyond the scope of this paper, but good illustrations can be found in [Ses86, Lau89].

The key to the approach taken in this paper is a formalisation of the notion of binding times in terms of *partial equivalence relations* (pers). This perspective allows us to construct a binding time analysis, in the form of an abstract interpretation, which extends smoothly to higher-order functions using the essence of the logical relations approach to correctness developed in [Abr90].

1.1 Overview

The remainder of this paper is organised as follows. In section 2 we motivate the use of partial equivalence relations to specify the staticness of structured data, and introduce some of the basic notation used in the rest of the paper. Section 3 introduces the language, and the notion of a semantic *interpretation*. Section 4 defines the standard interpretation (which determines the standard semantic evaluation function) and the interpretations of the key constants. Section 5 defines the abstract interpretation, and introduces the monotone *concretisation maps* which associate a per to each abstract domain point. Section 6 states and proves a correctness condition for the abstract interpretation. Section 7 provides a simple example of the analysis. Section 8 considers semantically related analyses. Section 9 concludes.

*To appear: Symposium on Partial Evaluation and Semantics-Based Program Manipulation, June 17–19, 1991, Yale University, USA.

†Partially supported by ESPRIT BRA 3074, *Semagraph*.

‡Partially supported by ESPRIT BRA 3124, *Semantique*.

2 Describing Binding Times with PERs

This section motivates the use of partial equivalence relations to describe binding times, and a correctness condition for a binding time analysis.

Suppose we have an interpreter $eval : code \times input \rightarrow output$, defined in a functional language in terms of a number of auxiliary functions, $f_1, f_2, \ldots, f_k$. The arguments to *eval* are the program to be interpreted and a vector of values for the program's run-time parameters. A much studied partial evaluation problem is the automatic generation of a compiler from the definition of the interpreter. (Each time the generated compiler is applied to some program, say p, it produces as output a version of the interpreter which is specialised to p.) To generate such a compiler, it is necessary to know, for each use of each f_i in the interpreter, how much of each argument to f_i will be fixed when the code to be compiled becomes known. It is this information which is provided by a binding time analysis. Since the f_i will typically be defined in terms of each other, the core of a binding time analysis is a method which, for each f_i defined in a program, can determine how much of $(f_i\ x)$ will be fixed given information about how much of x is fixed. It is this part of the analysis which we shall be considering.

It is important to note that a method is required which does not need to know the actual value which x is to take (since, for example, a compiler must be generated without knowing what programs are to be compiled). To illustrate the idea, let $\mathcal{D}$ and $\mathcal{E}$ be sets (they could be domains), and consider the simple example of the function $K : \mathcal{D} \times \mathcal{E} \rightarrow \mathcal{D}$, where

$$K(x, y) = x.$$

Clearly, without knowing what value it actually takes, we can say that when x becomes fixed, the value of $K(x, y)$ will also be fixed. Furthermore, this is true whether or not y becomes fixed at the same time. We can sum up this information about K in the formula:

$$\forall x \in \mathcal{D}. \forall y, y' \in \mathcal{E}. K(x, y) = K(x, y') \qquad (1)$$

An even simpler example would be the function $C : \mathcal{D} \rightarrow \mathcal{E}$ such that

$$C\ x = e$$

for some constant $e \in \mathcal{E}$. In this case, the value of $C\ x$ is fixed before x becomes known. Thus:

$$\forall x, x' \in \mathcal{D}. C\ x = C\ x'. \qquad (2)$$

As a final example, consider the function $swap : \mathcal{D} \times \mathcal{E} \rightarrow \mathcal{E} \times \mathcal{D}$ defined by

$$swap(x, y) = (y, x).$$

When x is fixed, the second element of $swap(x, y)$ is fixed and when y is fixed, the first element of $swap(x, y)$ is fixed, i.e.,

$$\pi_2(swap(x, y)) = \pi_2(swap(x, y')) \qquad (3)$$

and

$$\pi_1(swap(x, y)) = \pi_1(swap(x', y)), \qquad (4)$$

for all $x, x' \in \mathcal{D}$, and $y, y' \in \mathcal{E}$.

2.1 Equivalence Relations

The properties of K, C, and *swap* described above can be neatly re-expressed if for each set $\mathcal{D}$ we define the equivalence relations $All_{\mathcal{D}}, Id_{\mathcal{D}} \subseteq \mathcal{D}^2$, where for all $x, x' \in \mathcal{D}$:

$$x\ All_{\mathcal{D}}\ x'$$

$$x\ Id_{\mathcal{D}}\ x' \iff x = x'.$$

For a function $f : \mathcal{D} \rightarrow \mathcal{E}$ and binary relations $P \subseteq \mathcal{D}^2$ and $Q \subseteq \mathcal{E}^2$, we write $f : P \Rrightarrow Q$ iff

$$\forall x, x' \in \mathcal{D}. x\ P\ x' \Rightarrow (f\ x)\ Q\ (f\ x').$$

Then the property of C described by (2), becomes:

$$C : All_{\mathcal{D}} \Rrightarrow Id_{\mathcal{E}}.$$

For binary relations P, Q we define the relation $P \times Q$ by:

$$(x, y)\ P \times Q\ (x', y') \iff x\ P\ x' \wedge y\ Q\ y'.$$

Then (1) becomes

$$K : Id_{\mathcal{D}} \times All_{\mathcal{E}} \Rrightarrow Id_{\mathcal{D}},$$

while (3) and (4) become

$$swap : Id_{\mathcal{D}} \times All_{\mathcal{E}} \Rrightarrow All_{\mathcal{E}} \times Id_{\mathcal{D}}$$

and

$$swap : All_{\mathcal{D}} \times Id_{\mathcal{E}} \Rrightarrow Id_{\mathcal{E}} \times All_{\mathcal{D}},$$

respectively.

In the literature on binding time analysis, the fact that x will be fixed at, say, partial evaluation time, is often described by saying that x is *static*, while the fact that x will *not* be fixed is described by saying that x is *dynamic*. We propose that this terminology should be formalised by equating *dynamic* with *All* and *static* with *Id*. We could then read $K : Id \times All \Rrightarrow Id$ as saying that given a static first argument and a dynamic second argument, K returns a static result. Similarly, we could read $swap : All \times Id \Rrightarrow Id \times All$ as saying that given a dynamic first argument and a static second argument, *swap* returns a pair whose first element is static and whose second element may be dynamic.

While the static versus dynamic terminology is very convenient, it may also be rather misleading. When we say, for example, that f takes dynamic arguments to static results, it may suggest that being static and being dynamic are properties which *values* can satisfy, thus: "if d is a dynamic value, then $(f\ d)$ is a static value". But this would be wrong given our formalisation of the terminology. To see why, we need only consider the above example of C and observe that the fact that C takes dynamic arguments to static results is completely independent of the value e. This is not surprising, since we have equated the terms dynamic and static with relations rather than predicates.

2.2 Partial Equivalence Relations

We have seen how the equivalence relations *All* and *Id* may be used to describe binding times. There are two ways in which we need to extend this idea. Firstly to deal with structured data types, and secondly to deal with higher-order functions.

For structured data it is useful to have more refined notions of binding times than just static and dynamic; we would like to be able to model various *degrees* of staticness ([Mog88]), such as lists with static structure and dynamic elements. It is possible to achieve this with equivalence relations as well. For example, let $\mathcal{L}$ be the domain of finite, partial and infinite lists of integers with the usual ordering, and consider a relation $P \subseteq \mathcal{L} \times \mathcal{L}$ such that

- $\perp P \perp$
- $[\,]\ P\ [\,]$
- $\forall n, m.\, l\ P\ l' \Rightarrow n : l\ P\ m : l'$,

where $[\,]$ is the empty list and $:$ is the cons operation. We can then characterise the functions on lists, $f : \mathcal{L} \rightarrow \mathcal{E}$, which ignore the elements of their arguments as those for which

$$f : P \Rrightarrow Id_{\mathcal{E}}.$$

This is the basis of the treatment of list types in sub-section 5.1.

To extend our ideas to deal with higher-order functions, we use a less restricted class of relations than the equivalence relations. A *partial equivalence relation* (per) on a set $\mathcal{D}$ is a binary relation on $\mathcal{D}$ which is *symmetric* and *transitive*. If P is such a per we define the set $|P|$ by

$$|P| = \{x \in \mathcal{D} \mid x\ P\ x\}.$$

Note that the domain and range of a per P are both equal to $|P|$ (so for any $x, y \in \mathcal{D}$, if $x\ P\ y$ then $x\ P\ x$ and $y\ P\ y$), and that the restriction of P to $|P|$ is an equivalence relation. Clearly, an equivalence relation is just a per which is reflexive (so $|P| = \mathcal{D}$). Partial equivalence relations over various applicative structures have been used to construct models of the polymorphic lambda calculus (see, for example, [Asp90, AP90]). As far as we are aware, the first use of pers in static program analysis is that presented in [Hun90].

For the rest of the paper, we will restrict our attention to Scott domains and continuous maps between them ([SG9]). For each pair of domains $\mathcal{D}$ and $\mathcal{E}$, we will write the domain of continuous functions from $\mathcal{D}$ to $\mathcal{E}$ as $[\mathcal{D} \rightarrow \mathcal{E}]$.

Because we are working with domains, it is natural to consider pers which respect the structure of those domains. In particular, we restrict our attention to those pers which are both *strict* and *inductive*, where a relation on domains $P \subseteq \mathcal{D} \times \mathcal{E}$, is strict iff $\perp\ P\ \perp$, and is inductive iff whenever $\{x_i\}_{i \in \omega} \subseteq \mathcal{D}$ and $\{y_i\}_{i \in \omega} \subseteq \mathcal{E}$ are chains such that $\forall i \in \omega.\, x_i\ P\ y_i$, then

$$\bigsqcup_{i \in \omega} x_i\ P \bigsqcup_{i \in \omega} y_i.$$

Following [AP90], we use the term *complete per* to describe pers which are both strict and inductive. We denote the class of complete pers on $\mathcal{D}$ by $\textsc{cper}(\mathcal{D})$. For any domain $\mathcal{D}$, $\textsc{cper}(\mathcal{D})$ is closed under intersection and thus forms a complete meet semi-lattice (a partially ordered set with all meets) when ordered by subset inclusion. The greatest element of $\textsc{cper}(\mathcal{D})$ is $All_{\mathcal{D}}$.

Given pers $P \in \textsc{cper}(\mathcal{D})$ and $Q \in \textsc{cper}(\mathcal{E})$, we may construct a new per $(\mathcal{D} \Rrightarrow \mathcal{E}) \in \textsc{cper}([\mathcal{D} \rightarrow \mathcal{E}])$ defined by:

$$\begin{gathered} f\ (P \Rrightarrow Q)\ g \\ \Longleftrightarrow \\ \forall x, x' \in \mathcal{D}.\, x\ P\ x' \Rightarrow (f\ x)\ Q\ (g\ x'). \end{gathered}$$

If P is a per, we will write $x : P$ to mean $x \in |P|$. This notation and the above definition of $P \Rrightarrow Q$ are consistent with the notation used previously, since now

$$\begin{aligned} & f : P \Rrightarrow Q \\ & \quad \Longleftrightarrow \quad f\ (P \Rrightarrow Q)\ f \\ & \quad \Longleftrightarrow \quad \forall x, x' \in \mathcal{D}.\, x\ P\ x' \Rightarrow (f\ x)\ Q\ (f\ x'). \end{aligned}$$

Note that even if P and Q are both total (i.e., equivalence relations), $P \Rrightarrow Q$ may be partial. A simple example is $All \Rrightarrow Id$. In sub-section 5.2 we will see how such pers describe staticness at function types in a very natural way. Note also, that $\Rrightarrow$ 'behaves well' with respect to curried functions of more than one argument. That is to say, for function $f : \mathcal{D}_1 \to \ldots \to \mathcal{D}_k \to \mathcal{E}$, and pers $P_1 \subseteq \mathcal{D}_1^2, \ldots P_k \subseteq \mathcal{D}_k^2$ and $Q \subseteq \mathcal{E}^2$, the conditions

$$f : P_1 \Rrightarrow (\ldots (P_k \Rrightarrow Q))$$

and

$$\begin{array}{l}(x_1\ P_1\ x_1') \wedge \ldots \wedge (x_k\ P_k\ x_k') \\ \quad \Rightarrow \quad (f\ x_1\ \ldots\ x_k)\ Q\ (f\ x_1'\ \ldots\ x_k'),\end{array}$$

are equivalent. In future, we will borrow the types convention and write $f : P_1 \Rrightarrow (\ldots (P_k \Rrightarrow Q))$ as $f : P_1 \Rrightarrow \ldots \Rrightarrow P_k \Rrightarrow Q$.

To apply these ideas to the binding time analysis of functional programs, we require an effective test for conditions of the form

$$[\![e]\!] : P_1 \Rrightarrow \ldots \Rrightarrow P_k \Rrightarrow Q,$$

where $[\![e]\!] : \mathcal{D}_1 \to \ldots \mathcal{D}_k \to \mathcal{E}$ is the continuous map denoted by the expression e and $P_1, \ldots P_k$ and Q are complete pers. We can provide such a test by the use of abstract interpretation over finite domains ([AH87]). First we present our functional language.

3 The Language

We start with a definition of the syntax of types. Given a set of base types $\{A, B, \ldots\}$ we build type expressions, $\sigma, \tau, \ldots$ as follows:

$$\tau ::= A \mid \tau_1 \times \tau_2 \mid \mathit{list}(\tau) \mid \tau_1 \to \tau_2$$

A *Language* L consists of a set of base types and for each type σ, a set of *typed constants* ranged over by c_σ.

For each type σ, we assume an infinite set of typed variables $Var_\sigma = \{x^\sigma, y^\sigma, \ldots\}$. Then $\Lambda_T(L)$, the typed lambda calculus over L, consists of typed terms $e : \sigma$ built according to the following rules:

$$\begin{array}{ll}(i) & x^\sigma : \sigma \\[2ex] (ii) & c_\sigma : \sigma \\[2ex] (iii) & \dfrac{e : \tau}{\lambda x^\sigma . e : \sigma \to \tau} \\[2ex] (iv) & \dfrac{e_1 : \sigma \to \tau \quad e_2 : \sigma}{e_1\ e_2 : \tau}\end{array}$$

An *interpretation* I of L is specified by a pair of type-indexed families, $I = (\{\mathcal{D}_\sigma^I\}, \{c_\sigma^I\})$. Each $\mathcal{D}_\sigma^I$ is a Scott-domain, such that

$$\mathcal{D}_{\sigma\to\tau}^I = [\mathcal{D}_\sigma^I \to \mathcal{D}_\tau^I],$$

and for each c_σ, $c_\sigma^I \in \mathcal{D}_\sigma^I$. The terms $\perp_\sigma^I$ and $\top_\sigma^I$ may be used to denote the least element and (when it exists) the greatest element, respectively, of $\mathcal{D}_\sigma^I$.

An interpretation I determines the semantic valuation function:

$$[\![\cdot]\!]^I : \Lambda_T(L) \to Env^I \to \cup\mathcal{D}_\sigma^I$$

$$\begin{array}{lcl}[\![x^\sigma]\!]^I\ \rho & = & \rho\ x^\sigma \\ [\![c_\sigma]\!]^I\ \rho & = & c_\sigma^I \\ [\![\lambda x^\sigma.e]\!]^I\ \rho & = & \lambda d \in \mathcal{D}_\sigma^I . [\![e]\!]^I\ \rho[x^\sigma \mapsto d] \\ [\![e_1\ e_2]\!]^I\ \rho & = & ([\![e_1]\!]^I\ \rho)\ ([\![e_2]\!]^I\ \rho),\end{array}$$

where $Env^I = \{Env_\sigma^I\}$, $Env_\sigma^I = Var_\sigma \to \mathcal{D}_\sigma^I$. Note that for all $e : \sigma$, $\rho \in Env^I$, we have $([\![e]\!]\ \rho) \in \mathcal{D}_\sigma^I$.

If e is closed, the value $[\![e]\!]^I\ \rho$ is clearly independent of ρ, and we will often write this value just as $[\![e]\!]^I$.

We will routinely omit type subscripts if the intended type is either unimportant or clear from the context.

For the rest of this paper we will assume a fixed language L, which includes the base type *bool* and various familiar constants, detailed in the next section.

4 The Standard Interpretation

In this section we describe the standard interpretation, $\natural$. Here and in the rest of the paper we will often omit superscripts when referring to the standard interpretation. Thus $\mathcal{D}_{bool}$ will mean $\mathcal{D}_{bool}^\natural$, $[\![e]\!]$ will mean $[\![e]\!]^\natural$, $\perp_\tau$ will mean $\perp_\tau^\natural$, etc.

The domains for $\natural$ are as follows:

- $\mathcal{D}_{bool}$ is the flat three-point domain $\{\mathit{ff}, \mathit{tt}\}_\perp$.
- Interpretations for any remaining base-types are left unspecified, but are assumed to be non-trivial (i.e., not the one-point domain).
- For each σ, τ, $\mathcal{D}_{\sigma\times\tau} = \mathcal{D}_\sigma \times \mathcal{D}_\tau$.
- For each τ, $\mathcal{D}_{list(\tau)}$ is a solution to the recursive domain equation:

  $$\mathcal{D}_{list(\tau)} \cong \mathbf{1} + (\mathcal{D}_\tau \times \mathcal{D}_{list(\tau)}),$$

  where $+$ is separated sum, with left injection *inl* and right injection *inr*. For elements of $\mathcal{D}_{list(\tau)}$ we will write $\mathit{inl}(.)$ as $[]$ and $\mathit{inr}(x, l)$ as $x : l$.

For the constants, we have the following:

- At each type τ a fixed point combinator $Y_{(\tau\to\tau)\to\tau}$ with standard interpretation given by

  $$Y^\natural(f) = \bigsqcup_{n=0}^{\infty} f^n(\perp_\tau)$$

- At each type τ a conditional $cond_{bool\to\tau\to\tau\to\tau}$ with standard interpretation given by:

$$cond^\natural\ x\ y\ z = \begin{cases} \bot_\tau & \text{if } x = \bot_{bool} \\ y & \text{if } x = tt \\ z & \text{if } x = ff \end{cases}$$

- At each type τ the constants $nil_{list(\tau)}$, $cons_{\tau\to list(\tau)\to list(\tau)}$, $null_{list(\tau)\to bool}$, $hd_{list(\tau)\to\tau}$, and $tl_{list(\tau)\to list(\tau)}$, with standard interpretations:

$$\begin{array}{rcl} nil^\natural & = & [\,] \\ cons^\natural\ x\ l & = & x : l \\ null^\natural\ l & = & \begin{cases} tt & \text{if } l = [\,] \\ ff & \text{if } l = x : l' \\ \bot_{bool} & \text{if } l = \bot_{list(\tau)} \end{cases} \\ hd^\natural\ l & = & \begin{cases} x & \text{if } l = x : l' \\ \bot_\tau & \text{otherwise} \end{cases} \\ tl^\natural\ l & = & \begin{cases} l' & \text{if } l = x : l' \\ \bot_{list(\tau)} & \text{otherwise} \end{cases} \end{array}$$

- For each σ, τ the pairing function $pair_{\sigma\to\tau\to\sigma\times\tau}$, and the projection functions $fst_{\sigma\times\tau\to\sigma}$, and $snd_{\sigma\times\tau\to\tau}$, with standard interpretations:

$$\begin{array}{rcl} pair^\natural\ x\ y & = & (x, y) \\ fst^\natural\ (x, y) & = & x \\ snd^\natural\ (x, y) & = & y \end{array}$$

- Any remaining constants are assumed to be base-type first-order constants, i.e., of the form $c_{A_1\to\ldots\to A_n\to B}$. Their standard interpretations are left unspecified.

5 The Abstract Interpretation

In this section we define an abstract interpretation $\#$, in which each $\mathcal{D}^\#_\sigma$ is a finite lattice. To each value $a \in \mathcal{D}^\#_\sigma$ we will associate a per $\gamma_\sigma(a) \in \text{CPER}(\mathcal{D}_\sigma)$. The interpretations of the constants will be chosen in such a way that we can prove, for any closed[1] $e : \sigma_1 \to \ldots \to \sigma_k \to \tau$, for all $a_1 \in \mathcal{D}^\#_{\sigma_1}, \ldots, a_k \in \mathcal{D}^\#_{\sigma_k}$ and $b \in \mathcal{D}^\#_\tau$:

$$\begin{array}{c} [\![e]\!]^\#\ a_1\ \ldots\ a_k = b \\ \Rightarrow \\ [\![e]\!] : \gamma_{\sigma_1}(a_1) \Rrightarrow \ldots \Rrightarrow \gamma_{\sigma_k}(a_k) \Rrightarrow \gamma_\tau(b). \end{array} \tag{5}$$

So for example, in the case that $k = 1$, if $\gamma_{\sigma_1}(a) = P$ and $\gamma_\tau(b) = Q$, we can test for the condition that $[\![e]\!] : P \Rrightarrow Q$ by evaluating $[\![e]\!]^\#$ and seeing if $[\![e]\!]^\#\ a = b$. (In fact, for our analysis, the weaker condition $[\![e]\!]^\#\ a \sqsubseteq b$ will suffice.) Note that in general, (5) will not hold when the implication is reversed, i.e., our test will be *sound* but not *complete*. On the other hand, since the abstract domains are finite, all functions can be finitely tabulated and so the test will be *effective*. The trading of completeness for effectiveness is unavoidable, since one of the tests we wish to make is for the condition that

$$\forall x, y.\, [\![e]\!]\ x = [\![e]\!]\ y,$$

which is clearly not decidable for all e.

The interpretation consists of two parts: the family of finite domains, and the interpretations of the constants.

5.1 Abstract Domains

Figure 1 gives an inductive definition of the finite abstract domains at each type. At the base types we have just the two point domain, with top element D (*dynamic*), and bottom element S (*static*). Intuitively, the top element of each domain represents "completely dynamic", and decreasing in the domain ordering reflects an increasing degree of "staticness". As specified by the definition of an interpretation, the function-type is interpreted as the continuous function space. The domain construction for lists takes the element domain, and simply adds a new top element, also written D. The remaining points, written SPINE(a), intuitively represent lists with static structure and elements whose staticness is described by a.

With these informal descriptions of the abstract domain points for the ground-types (types not containing $\to$) we can see the correspondence with the finite domains advocated by Launchbury [Lau88, Lau89]. Before giving similar intuitions for the domains at higher types we present a formal description in terms of concretisation maps.

5.2 The Concretisation Maps

At each type τ we define a concretisation map γ_τ which associates to each abstract domain point $a \in \mathcal{D}^\#_\tau$ a complete per $\gamma_\tau(a) \in \text{CPER}(\mathcal{D}_\tau)$.

The definition of the family $\gamma = \{\gamma_\sigma\}$ is given in figure 2 (we write All_σ to mean $All_{\mathcal{D}_\sigma}$ and Id_σ to mean $Id_{\mathcal{D}_\sigma}$). In the remainder of this sub-section we outline the motivations for the definition of γ. We use the following facts:

- $P \subseteq Q \Rightarrow P \cap Q = P$,
- For any per $P \subseteq \mathcal{D}^2$, $(P \Rrightarrow All_\mathcal{E}) = All_{[\mathcal{D}\to\mathcal{E}]}$,

[1] The restriction to closed expressions is dropped in the formal statement of correctness (section 6).

$$
\begin{aligned}
\mathcal{D}^{\#}_{A} &= \{\text{D}, \text{S}\} \text{ where } \text{S} \sqsubset \text{D} \\
\mathcal{D}^{\#}_{\sigma\to\tau} &= [\mathcal{D}^{\#}_{\sigma} \to \mathcal{D}^{\#}_{\tau}] \\
\mathcal{D}^{\#}_{\sigma\times\tau} &= \mathcal{D}^{\#}_{\sigma} \times \mathcal{D}^{\#}_{\tau} \\
\mathcal{D}^{\#}_{list(\tau)} &= \{\text{SPINE}(a) \mid a \in \mathcal{D}^{\#}_{\tau}\} \cup \{\text{D}\} \\
&\quad \text{where for all } a, b \in \mathcal{D}^{\#}_{\tau},\ \text{SPINE}(a) \sqsubseteq \text{SPINE}(b) \iff a \sqsubseteq b, \\
&\quad \phantom{\text{where for all } a, b \in \mathcal{D}^{\#}_{\tau},\ } \text{SPINE}(a) \sqsubset \text{D}
\end{aligned}
$$

Figure 1: Abstract Domains

$$
\gamma_\sigma : \mathcal{D}^{\#}_{\sigma} \to \text{CPER}(\mathcal{D}_\sigma)
$$

$$
\begin{aligned}
\gamma_A(a) &= \begin{cases} All_A & \text{if } a = \text{D} \\ Id_A & \text{if } a = \text{S} \end{cases} \\
\gamma_{\sigma\to\tau}(f) &= \bigcap_{a \in D^{\#}_{\sigma}} (\gamma_\sigma(a) \Rrightarrow \gamma_\tau(fa)) \\
\gamma_{\sigma\times\tau}(a \times b) &= \gamma_\sigma(a) \times \gamma_\tau(b) \\
\gamma_{list(\tau)}(a) &= \begin{cases} All_{list(\tau)} & \text{if } a = \text{D} \\ \mathsf{lfp}\, F_b & \text{if } a = \text{SPINE}(b) \end{cases}
\end{aligned}
$$

where $F_b : \text{CPER}(\mathcal{D}_{list(\tau)}) \to \text{CPER}(\mathcal{D}_{list(\tau)})$ is defined as:

$$
\begin{aligned}
F_b(P) = \ & \{(\bot, \bot)\} \ \cup \\
& \{([\,], [\,])\} \ \cup \\
& \{(x : l, x' : l') \mid (x, x') \in \gamma_\tau(b), (l, l') \in P\}
\end{aligned}
$$

Figure 2: The Concretisation Maps

- $P \supseteq P' \wedge Q \subseteq Q' \ \Rightarrow \ (P \Rrightarrow Q) \subseteq (P' \Rrightarrow Q')$.

The last of these states that $\Rrightarrow$ is anti-monotone in its first argument and monotone in its second.

The concretisation of the points in abstract base domain $\{\text{D}, \text{S}\}$ are just the equivalence relations *All* and *Id* (of the appropriate type).

Perhaps the best motivation for the definition of γ at the function types, is to be found in the proof of correctness (section 6). For an example illustrating that the definition is reasonable, consider one of the simplest function types, $int \to int$. The abstract domain $\mathcal{D}^{\#}_{int\to int}$ is $[\mathcal{D}^{\#}_{int} \to \mathcal{D}^{\#}_{int}]$ where $D^{\#}_{int} = \{\text{S}, \text{D}\}$.

So we have

$$
\mathcal{D}^{\#}_{int\to int} = \begin{array}{c} \lambda x.\text{D} \\ | \\ \lambda x.\, x \\ | \\ \lambda x.\text{S} \end{array}
$$

Now consider the pers corresponding to these points. The concretisation of the top point is the per

$$
\begin{aligned}
& (\gamma_{int}(\text{S}) \Rrightarrow \gamma_{int}(\text{D})) \cap (\gamma_{int}(\text{D}) \Rrightarrow \gamma_{int}(\text{D})) \\
= \ & (Id_{int} \Rrightarrow All_{int}) \cap (All_{int} \Rrightarrow All_{int}) \\
= \ & All_{int\to int} \cap All_{int\to int}
\end{aligned}
$$

which is just $All_{int\to int}$ (informally, dynamic). The concretisation of the middle point is

$$\begin{array}{rl} & (\gamma_{int}(\mathrm{S}) \Rrightarrow \gamma_{int}(\mathrm{S})) \cap (\gamma_{int}(\mathrm{D}) \Rrightarrow \gamma_{int}(\mathrm{D})) \\ = & (Id_{int} \Rrightarrow Id_{int}) \cap (All_{int} \Rrightarrow All_{int}) \\ = & (Id_{int} \Rrightarrow Id_{int}) \cap All_{int \to int} \\ = & Id_{int} \Rrightarrow Id_{int}, \end{array}$$

which relates only equal functions, so is just $Id_{int \to int}$ (informally, static). Interestingly, we now have a domain point below the point corresponding to Id (informally, even more static than static!). To see that this actually makes some sense intuitively, we concretise the bottom point to give

$$\begin{array}{rl} & (\gamma_{int}(\mathrm{S}) \Rrightarrow \gamma_{int}(\mathrm{S})) \cap (\gamma_{int}(\mathrm{D}) \Rrightarrow \gamma_{int}(\mathrm{S})) \\ = & (Id_{int} \Rrightarrow Id_{int}) \cap (All_{int} \Rrightarrow Id_{int}) \\ = & All_{int} \Rrightarrow Id_{int}. \end{array}$$

From the definition of $\Rrightarrow$, we can see that two functions f and g are related by this per as follows:

$$f\ (All \Rrightarrow Id)\ g \iff \forall x, y.\ f\ x = g\ y,$$

i.e., f and g are related iff they are equal *and* they are constant functions (note therefore that the relation is *partial*).

Taking the simplest example of a list type, the abstract domain for *list*(*bool*),

$$\mathcal{D}^{\#}_{list(bool)} = \begin{array}{c} \mathrm{D} \\ | \\ \mathrm{SPINE}(\mathrm{D}) \\ | \\ \mathrm{SPINE}(\mathrm{S}) \end{array}$$

The top point corresponds to All. The concretisations of the other two points are defined as the least fixed points of F_{D} and F_{S} (defined in figure 2). It is straightforward to verify that the functions F_b are monotone. The existence of the least fixed point is thus guaranteed by Tarski's fixed point theorem[2]. The middle point corresponds to the equivalence relation which relates all finite lists of the same length, all partial lists of the same length, and all infinite lists. Intuitively, this describes lists with static structure and dynamic elements. The bottom point just corresponds to Id.

5.3 Abstract Constants

In this section we define the abstract interpretations of the constants. The following lemma aids the intuitive understanding of some of the definitions and is also central to the proofs in section 6 that the abstract interpretations of the constants are correct:

Lemma 5.1 For all σ, for all $a, b \in \mathcal{D}^{\#}_{\sigma}$,

$$a \sqsubseteq b \Rightarrow \gamma_\sigma(a) \subseteq \gamma_\sigma(b).$$

In short, each γ_σ is monotone. □

The interpretations of some of the constants are defined in essentially the same way in $\#$ as in $\natural$ (albeit over the $\mathcal{D}^{\#}_{\sigma}$ rather than the $\mathcal{D}_\sigma$). Thus for each σ, τ, the interpretations of $Y_{(\tau\to\tau)\to\tau}$, $pair_{\sigma\to\tau\to\sigma\times\tau}$, $fst_{\sigma\times\tau\to\sigma}$, and $snd_{\sigma\times\tau\to\tau}$, are as follows:

$$\begin{array}{rcl} Y^{\#}(f) & = & \bigsqcup_{n=0}^{\infty} f^n(\bot^{\#}_{\tau}), \\ pair^{\#}\ x\ y & = & (x, y) \\ fst^{\#}\ (x, y) & = & x \\ snd^{\#}\ (x, y) & = & y. \end{array}$$

The correctness of these interpretations for the pairing and projection functions is fairly clear. The correctness of $Y^{\#}$ is dealt with in the next section.

At each τ, the remaining constants detailed in section 4 are interpreted as follows.

- The conditional:

  $$cond^{\#}\ a\ b\ c = \begin{cases} \top^{\#}_{\tau} & \text{if } a = \mathrm{D} \\ b \sqcup c & \text{if } a = \mathrm{S} \end{cases}$$

  If the condition is dynamic, the result is clearly dynamic. If the condition is static, then the monotonicity of γ_τ can be used to show that it is correct to take the lub of the alternatives.

- The list constants:

  $$\begin{array}{rcl} nil^{\#} & = & \mathrm{SPINE}(\bot^{\#}_{\tau}) \\ cons^{\#}\ a\ l & = & \begin{cases} \mathrm{SPINE}(a \sqcup a') & \text{if } l = \mathrm{SPINE}(a') \\ \mathrm{D} & \text{if } l = \mathrm{D} \end{cases} \\ null^{\#}\ l & = & \begin{cases} \mathrm{S} & \text{if } l = \mathrm{SPINE}(a) \\ \mathrm{D} & \text{if } l = \mathrm{D} \end{cases} \\ hd^{\#}\ l & = & \begin{cases} a & \text{if } l = \mathrm{SPINE}(a) \\ \top^{\#}_{\tau} & \text{if } l = \mathrm{D} \end{cases} \\ tl^{\#}\ l & = & l. \end{array}$$

  The most interesting of these definitions is that for $cons^{\#}$. Adding a new element to a list with dynamic structure results in a list with dynamic structure (of course, we know that the result list has at least one element, but our abstract domains have not been structured to capture such fine-grained information), while adding a new element to a list with static structure results in a list with static structure. As with the

[2] We suspect that the least and greatest fixed points actually coincide. A proof of this would hinge on the fact that the pers we are dealing with are *uniform* (a type of algebraicity condition, see [AP90]).

conditional, the monotonicity of the concretisation maps can be used to prove that taking the lub to describe the staticness of the elements is correct.

- The interpretations of any remaining constants are:

$$c^{\#}_{A_1\to\ldots\to A_n\to B}\ a_1\ \ldots\ a_n = \bigsqcup_{i=1}^{n} a_i.$$

6 Correctness

In this section we state a correctness condition for # and prove that # is correct by the use of a logical relation.

Recall that for a closed term $e : \sigma_1 \to \ldots \to \sigma_k \to \tau$, our requirement of the interpretation # is that for all $a_1 \in \mathcal{D}^{\#}_{\sigma_1}, \ldots, a_k \in \mathcal{D}^{\#}_{\sigma_k}$

$$[\![e]\!] : \gamma_{\sigma_1}(a_1) \Rrightarrow \ldots \Rrightarrow \gamma_{\sigma_k}(a_k) \Rrightarrow \gamma_\tau([\![e]\!]^{\#}\ a_1\ \ldots\ a_k).$$

This can be expressed rather more concisely if we make the following observation. For any pair of continuous maps $f : \mathcal{D}_{\sigma_1} \to \ldots \to \mathcal{D}_{\sigma_k} \to \mathcal{D}_\tau$ and $h : \mathcal{D}^{\#}_{\sigma_1} \to \ldots \to \mathcal{D}^{\#}_{\sigma_k} \to \mathcal{D}^{\#}_\tau$, the conditions that

$$f : \gamma_{\sigma_1}(a_1) \Rrightarrow \ldots \Rrightarrow \gamma_{\sigma_k}(a_k) \Rrightarrow \gamma_\tau(h\ a_1\ \ldots\ a_k),$$

for all $a_1 \in \mathcal{D}^{\#}_{\sigma_1}, \ldots, a_k \in \mathcal{D}^{\#}_{\sigma_k}$, and

$$f \in |\gamma_{\sigma_1\to\ldots\to\sigma_k\to\tau}(h)|,$$

are equivalent. Thus the statement that # is correct for a closed expression $e : \sigma$ is simply that $[\![e]\!] \in |\gamma_\sigma([\![e]\!]^{\#})|$.

Before we prove # correct, we need to generalise the correctness condition to allow for non-closed terms. For environments $\rho \in \mathit{Env}$ and $\rho' \in \mathit{Env}^{\#}$, we will say that ρ' *is correct for* ρ iff $\forall x^\tau.\, \rho\ x^\tau \in |\gamma_\tau(\rho'\ x^\tau)|$. Our general statement of correctness is then as follows: # is *correct* iff for all expressions $e : \sigma$, for all environments $\rho \in \mathit{Env}$ and $\rho' \in \mathit{Env}^{\#}$ such that ρ' is correct for ρ,

$$([\![e]\!]\ \rho) \in |\gamma_\sigma([\![e]\!]^{\#}\ \rho')|. \tag{6}$$

An alternative statement of correctness for # uses a family of relations. We define the family $R = \{R_\sigma\}$, with $R_\sigma \subseteq \mathcal{D}_\sigma \times \mathcal{D}_\sigma \times \mathcal{D}^{\#}_\sigma$, by

$$R_\sigma(x, x', a) \iff (x, x') \in \gamma_\sigma(a).$$

We write $R(\rho_1, \rho_2, \rho')$ to mean

$$\forall x^\tau.\, R_\tau(\rho_1\ x^\tau, \rho_2\ x^\tau, \rho'\ x^\tau).$$

Then a simple re-statement of (6) is: # is correct iff for all expressions $e : \sigma$, and environments $\rho \in \mathit{Env}$ and $\rho' \in \mathit{Env}^{\#}$ such that $R(\rho, \rho, \rho')$,

$$R_\sigma([\![e]\!]\ \rho, [\![e]\!]\ \rho, [\![e]\!]^{\#}\ \rho'). \tag{7}$$

It is simple to show that R is a *logical relation*, i.e., that R satisfies the following property: for all σ, τ, for all $f, g \in \mathcal{D}_{\sigma\to\tau}, h \in \mathcal{D}^{\#}_{\sigma\to\tau}$,

$$\begin{gathered} R_{\sigma\to\tau}(f, g, h) \\ \iff \\ \forall x, x' \in \mathcal{D}_\sigma, a \in \mathcal{D}^{\#}_\sigma.\, R_\sigma(x, x', a) \Rightarrow R_\tau(f\ x, g\ x', h\ a). \end{gathered}$$

A straightforward adaptation of the proof of proposition 3.2 in [Abr90] then yields the following:

Theorem 6.1 If, for all constants c_τ, $R_\tau(c^\natural_\tau, c^\natural_\tau, c^{\#}_\tau)$, then for all expressions $e : \sigma$, and environments ρ, ρ' such that $R(\rho, \rho, \rho')$, $R_\sigma([\![e]\!]\ \rho, [\![e]\!]\ \rho, [\![e]\!]^{\#}\ \rho')$. □

For details see [Hun90]. (The fact that the type structure used in this paper is richer than that used in [Abr90] and [Hun90] makes no difference to the proof.)

Thus to prove # correct, it now suffices to prove that for each c_τ, $c^\natural_\tau \in |\gamma_\tau(c^{\#}_\tau)|$. Here we only sketch the proof of the more interesting cases.

For the recursion combinators we need the following:

Lemma 6.2 Each R_σ is strict and inductive. □

The proof of this lemma is trivial given that each $\gamma_\sigma(a)$ is itself strict and inductive. The correctness of each of the $Y^{\#}$ then follows from (a simple adaptation of) proposition 3.5 of [Abr90].

For $\mathit{cond}_{\mathit{bool}\to\tau\to\tau\to\tau}$, the interesting case is $\mathit{cond}^{\#}\ a\ b\ c$ when $a = \mathrm{S}$, in which case $\gamma_{\mathit{bool}}(a) = \mathit{Id}$. By monotonicity of γ_τ we have $\gamma_\tau(b \sqcup c) \supseteq \gamma_\tau(b)$ and $\gamma_\tau(b \sqcup c) \supseteq \gamma_\tau(c)$. Then it suffices to show that for any pers $P, P', Q \in \mathrm{CPER}(\mathcal{D}_\tau)$, if $P, P' \subseteq Q$ then

$$\begin{gathered} (y\ P\ y') \wedge (z\ P'\ z') \\ \Rightarrow \\ \forall x \in \mathcal{D}_{\mathit{bool}}.\, (\mathit{cond}^\natural\ x\ y\ z)\ Q\ (\mathit{cond}^\natural\ x\ y'\ z'). \end{gathered}$$

For $\mathit{cons}_{\tau\to \mathit{list}(\tau)\to \mathit{list}(\tau)}$, the interesting case is

$$\mathit{cons}^{\#}\ a\ \mathrm{SPINE}(a').$$

We have to show that

$$\begin{gathered} (x, x') \in \gamma_\tau(a) \wedge (l, l') \in \gamma_{\mathit{list}(\tau)}(\mathrm{SPINE}(a')) \\ \Rightarrow \\ (x : l, x' : l') \in \gamma_{\mathit{list}(\tau)}(\mathrm{SPINE}(a \sqcup a')). \end{gathered}$$

Let $P = \gamma_\tau(a \sqcup a')$ and $Q = \gamma_{\mathit{list}(\tau)}(\mathrm{SPINE}(a \sqcup a'))$. Now γ_τ and $\gamma_{\mathit{list}(\tau)}$ are monotone, so by assumption we have $(x, x') \in P$ and $(l, l') \in Q$. But $Q = \mathsf{lfp}\ F_{(a\sqcup a')}$, hence $Q = F_{(a\sqcup a')}(Q) \supseteq \{(x : l, x' : l') \mid x\ P\ x', l\ Q\ l'\}$.

It is also possible to show that the abstract interpretations of the constants are not only correct but are *best* (i.e., they are the smallest possible correct interpretations), and so give the most precise results possible for our choice of abstract domains and concretisation maps. The issue of best interpretations in per-based analyses is discussed in [Hun90].

7 Example

In this section give a simple example application of the # interpretation to the binding time analysis of a functional program. We will use a recursion equation definition format and omit type superscripts and subscripts from our definitions to make them easier to read (translation into the typed λ-calculus is simple). Suppose our program is

$$map\ snd\ l,$$

where l is of type $list(int \times bool)$ and $map : (int \times bool \to bool) \to list(int \times bool) \to list(bool)$ is defined by

$$\begin{array}{lll} map\ f\ l = cond & (null\ l) \\ & nil \\ & cons\ (f(hd\ l))\ (map\ f\ (tl\ l)) \end{array}$$

The fixed point iteration for *map* under # is given in figure 3. So if l is completely dynamic, the result of our program may also be completely dynamic, since

$$map^{\#}\ snd^{\#}\ \mathrm{D} = \mathrm{D}.$$

But if l has static structure and elements whose second components are all static, the result of our program is completely static, since

$$\begin{array}{rcl} map^{\#}\ snd^{\#}\ \mathrm{SPINE}((\mathrm{D},\mathrm{S})) & = & \mathrm{SPINE}(snd^{\#}(\mathrm{D},\mathrm{S})) \\ & = & \mathrm{SPINE}(\mathrm{S}). \end{array}$$

8 Related Work

Early work on binding time analysis in the MIX project [JSS85, Ses86], and in the TML system [NN88] was only able to classify a variable as either completely static or completely dynamic. Mogensen [Mog88] recognised the importance of being able to make finer distinctions for structured data, and extended the binding time analysis of a first-order dialect of pure lisp to consider partially static structures. Launchbury [Lau88] introduced the use of domain projections to describe binding times in a first-order typed functional language, and in his thesis [Lau89] presented the details of a binding time analysis based on this approach. Our work is intimately connected with Launchbury's, and we will expose this relation in some detail.

8.1 Comparing Projection Analysis and Per Analysis

A projection is a continuous map on a cpo $\alpha : \mathcal{D} \to \mathcal{D}$, such that $\alpha \sqsubseteq \mathrm{id}_{\mathcal{D}}$ and $\alpha \circ \alpha = \alpha$. An intuition behind the use of projections to describe binding times is that the parts of its argument that a projection discards (replaces by bottom) represent the parts about which no information is known, where "no information" is equated with "dynamic".

For example, for domains $\mathcal{D} \times \mathcal{E}$, the projection $left : \mathcal{D} \times \mathcal{E} \to \mathcal{D} \times \mathcal{E}$ which satisfies $left(x, y) = (x, \perp)$, describes the situation where the first argument is static and the second dynamic. The projection $right : \mathcal{D} \times \mathcal{E} \to \mathcal{D} \times \mathcal{E}$ which satisfies $right(x, y) = (\perp, y)$, describes the reverse situation. Using these projections, we have yet another way of stating properties (3) and (4) of the function *swap* (see section 2) as

$$right \circ swap = right \circ swap \circ left$$

and

$$left \circ swap = left \circ swap \circ right,$$

respectively.

The core of Launchbury's binding time analysis is an analysis which, given a projection α describing the static parts of f's argument, finds a projection β such that $\beta \circ f = \beta \circ f \circ \alpha$.

For any function $\alpha : \mathcal{D} \to \mathcal{E}$, there is an equivalence relation $E_\alpha \subseteq \mathcal{D}^2$ defined by

$$x\ E_\alpha\ x' \iff \alpha\ x = \alpha\ x'. \quad (8)$$

For example, we have $E_{\lambda x.x} = Id$ and $E_{\lambda x.\perp} = All$. The connection between a projection based analysis and the approach we have taken is summed up in the following proposition ([Hun90]):

Proposition 8.1 For all projections $\alpha : \mathcal{D} \to \mathcal{D}$, $\beta : \mathcal{E} \to \mathcal{E}$, and functions $f : \mathcal{D} \to \mathcal{E}$:

$$\beta \circ f = \beta \circ f \circ \alpha \iff f : E_\alpha \Rrightarrow E_\beta.$$

□

The above re-statements of (3) and (4) are an instance of this equivalence, where $E_{left} = Id \times All$ and $E_{right} = All \times Id$. Thus in the first-order case, the conditions tested for by our analysis may be seen as a simple re-phrasing of those tested for by Launchbury's. But in making the switch from projections to pers we achieve a new perspective on binding time analysis which leads naturally to a higher order analysis with a clear formal basis.

A correctness condition for the binding time analysis of imperative programs is provided by Jones [Jon88] via the notion of *congruence*. Launchbury has shown

$$map_0^{\#}\ h\ c \;=\; \textsc{spine}(\textsc{s})$$

$$map_1^{\#}\ h\ c \;=\; \begin{cases} \textsc{d} & \text{if } c = \textsc{d} \\ cons^{\#}\ (h(a,b))\ (map_0^{\#}\ h\ \textsc{spine}((a,b))) & \text{if } c = \textsc{spine}((a,b)) \\ = \textsc{spine}(h(a,b) \sqcup \textsc{s}) & \\ = \textsc{spine}(h(a,b)) & \end{cases}$$

$$map_2^{\#}\ h\ c \;=\; \begin{cases} \textsc{d} & \text{if } c = \textsc{d} \\ cons^{\#}\ (h(a,b))\ (map_1^{\#}\ h\ \textsc{spine}((a,b))) & \text{if } c = \textsc{spine}((a,b)) \\ = \textsc{spine}(h(a,b) \sqcup h(a,b)) & \\ = \textsc{spine}(h(a,b)) & \end{cases}$$

$$\vdots$$

$$map^{\#}\ h\ c \;=\; \begin{cases} \textsc{d} & \text{if } c = \textsc{d} \\ \textsc{spine}(h(a,b)) & \text{if } c = \textsc{spine}((a,b)) \end{cases}$$

Figure 3: Fixed Point iteration

a correspondence between the projection-based correctness condition and a restricted notion of congruence. Interestingly, the per-based safety condition can be seen to be even closer to Jones' original notion of congruence (using (8) to derive an equivalence relation from the static part of a program division).

There are important aspects of Launchbury's work which we have failed to address. The most significant of these is *domain factorisation*, whereby the projections found in binding time analysis may be used to modify the *type* of a function when the function is specialised to the static part of its argument. A key question with regard to carrying this part of Launchbury's work over to the per setting, is whether or not the individual equivalence classes of the pers involved are Scott domains. We can borrow the relevant result from [Lau89] to answer this question in the affirmative for the ground types, since the pers used by our analysis at those types correspond to Launchbury's projections. For higher types we do not know the answer, nor is it clear how domain factorisation could be exploited at the higher types.

Another feature of [Lau89] is its treatment of polymorphism. This exploits the correspondence between first-order polymorphic functions on the one hand, and natural transformations between functors over a category of Scott domains on the other. Unfortunately, this correspondence does not hold in the higher order case. It is possible to cope with polymorphism simply by resolving each polymorphic function definition in a program into the set of monomorphic instances at which the function is used in the program (for Hindley-Milner types this set is always finite and can be statically determined). This is a rather unsatisfactory thing to do, however, since it makes no attempt to exploit the strong uniformity which undoubtedly exists across the instances of a polymorphic function. A promising approach to extending the theory of abstract interpretation of the simply typed lambda calculus to cope with polymorphism is presented in [AJ91].

In [Mog89], Launchbury's projection based method has been extended to a higher-order language in a way which is rather different from the method we have presented. The approach of [Mog89] is based on a global program analysis in which higher-order functions are treated as "abstract closures" dependent upon the function definitions that appear in a given program. It is certainly easier to reason about correctness using our approach, but we have not yet been able to compare the precision achieved by the resulting analyses.

8.2 Live Variable Analysis

In a live variable analysis [ASU86] we want to know whether or not expressions are *live*, i.e., possibly used in future computations, or *dead*, i.e., definitely not used. Traditionally, a computable test for this property is implemented as a *backwards analysis*, where information about liveness of a program is propagated backwards to variables. Nielson [Nie90] gives a simple functional version of a liveness analysis, presented as a backwards abstract interpretation. For the function K given in section 2, the abstract backwards version $bK^{\#}$ of Nielson's analysis satisfies $bK^{\#}\ live = (live, dead)$.

The connection between liveness and binding time analysis is illustrated by this example since for our analysis,

$$K^{\#}\ (\textsc{s}, \textsc{d}) = \textsc{s}.$$

More interestingly, the formulation of Nielson's correctness condition involves the (indirect) use of a ternary relation between the abstract domain points $\{live, dead\}$ and a pair of standard domain points, which exactly corresponds to the relation R_A we form at the base types.

The correctness condition for liveness analysis is equivalent to the condition for binding time analysis. The fact that our analysis is a forwards analysis does not prevent us from using the results to infer "backwards" properties. Although it may be argued ([Hug87]) that it is more efficient to answer this form of question using a backwards analysis, in the case of a higher-order language (as given here) forwards analysis can capture the crucial interdependency between arguments which would be lost using a simple backwards analysis.

Viewing our analysis as a liveness test, there are several potential applications where first-order liveness analyses have previously been used. Examples include identifying conditions for certain program transformation (e.g., [NN89], [Chi90]), in compile-time garbage collection (e.g., *necessity analysis* in [JM89]), and in the compositional time analysis of lazy higher-order functions [San90].

9 Conclusions

We have presented a local binding time analysis which allows us to describe which parts of data structures are static, and also allows us to analyse higher-order functions, thus overcoming a major limitation of [Lau89]. The analysis is presented as an abstract interpretation, based on the use of partial equivalence relations to describe degrees of staticness, and using the essence of the logical relations approach to correctness developed in [Abr90]. We have shown how abstract domains tailored to each type can be constructed; the generalisation of the list-type to a more general recursive type is straightforward.

Possible further work includes a re-examination of other issues considered in [Lau89]—in particular, domain factorisation, polymorphism, and using the analysis to define some form of program division (i.e. a sticky (global) analysis). Although it is possible to define a global analysis *without* first defining a local analysis (for a good example see [Con90]), potential advantages of formulating an independent local analysis are:

- *efficiency*, since results from a local analysis are reusable in different programs containing common function definitions;
- *proof of correctness*, because of the modularity of the analysis.

Further work is needed to determine what kind of global analysis is appropriate for the specialisation of the higher-order functions in a program.

Acknowledgements

Thanks to Chris Hankin, John Launchbury, various members of the Semantique project and the anonymous referees for numerous useful comments and advice.

References

[Abr90] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Logic and Computation*, 1(1), 1990.

[AH87] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[AJ91] S. Abramsky and T. P. Jensen. A relational approach to strictness analysis of higher order polymorphic functions. In *Proc. ACM Symposium on Principles of Programming Languages*, 1991.

[AP90] M. Abadi and G. Plotkin. A per model of polymorphism and recursive types. In *Logic in Computer Science*. IEEE, 1990.

[Asp90] A. G. Asperti. *Categorical Topics in Computer Science*. PhD thesis, Università di Pisa, 1990.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Chi90] W. N. Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, University of London, 1990.

[Con90] C. Consel. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 264–272. ACM, 1990.

[Hug87] R. J. M. Hughes. Backwards analysis of functional programs. Research Report CSC/87/R3, University of Glasgow, March 1987.

[Hun90] S. Hunt. PERs generalise projections for strictness analysis. Technical Report DOC 14/90, Imperial College, 1990.

[JM89] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Functional Programming Languages and computer architecture, conference proceedings*, pages 54–74. ACM press, 1989.

[Jon88] N.D. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, 1988.

[JSS85] N.D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (LNCS 202)*, pages 124–140. Springer-Verlag, 1985.

[Lau88] J. Launchbury. Projections for specialisation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 299–315. North-Holland, 1988.

[Lau89] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, Department of Computing, University of Glasgow, November 1989.

[Mog88] T. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[Mog89] T. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Diaz and F. Orejas, editors, *TAPSOFT '89 (LNCS 352)*, pages 298–312. Springer-Verlag, 1989.

[Nie90] F. Nielson. Two-level semantics and abstract interpretation — fundamental studies. *Theoretical Computer Science*, (69):117–242, 90.

[NN88] H. R. Nielson and F. Nielson. Automatic binding time analysis for a typed λ-calculus. *Science of Computer Programming*, 10:139–176, 1988.

[NN89] H. R. Nielson and F. Nielson. Transformations on higher-order functions. In *Fourth International Conference on Functional Programming Languages and Computer Architecture, London, England, September 1989*, pages 129–143. ACM Press and Addison-Wesley, 1989.

[San90] D. Sands. *Calculi for Time Analysis of Functional Programs.* PhD thesis, Imperial College, September 1990.

[Ses86] P. Sestoft. The structure of a self-applicable partial evaluator. In H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985. (LNCS 217)*, pages 236–256. Springer-Verlag, 1986.

[SG9] D. Scott and C. Gunter. Semantic domains. In *Handbook of Theoretical Computer Science.* North-Holland, 199-. *(to appear).*