Binding Time Analysis for Higher Order Untyped Functional Languages

Charles Consel

Research Report YALEU/DCS/RR-780
April 1990

This work is supported by the Darpa grant N00014-88-K-0573.

Binding Time Analysis for Higher Order Untyped Functional Languages

Charles Consel*
Yale University
Department of Computer Science
e-mail: consel-charles@cs.yale.edu

Abstract

When some inputs of a program are known at compile-time, certain expressions can be processed statically; this is the basis of the notion of *partial evaluation*. Identifying these early computations can be determined independently of the actual values of the input by a static analysis called *binding time analysis*. Then, to process a program, one simply follows the binding time information: evaluate compile-time expressions and defer the others to run-time.

Using abstract interpretation, we present a binding time analysis for an untyped functional language which provides an effective treatment of both higher order functions and data structures. To our knowledge it is the first such analysis. It has been implemented and is used in a partial evaluator for a side-effect free dialect of Scheme. The analysis is general enough, however, to be valid for non-strict typed functional languages such as Haskell. Our approach and the system we have developed solve and go beyond the open problem of partially evaluating higher order functions described in [3] since we also provide a method to handle data structures.

Our analysis improves on previous work [5, 15, 4] in that: (1) it treats both higher order functions and data structures, (2) it does not impose syntactic restrictions on the program being processed, and (3) it does not require a preliminary phase to collect the set of possible functions that may occur at each site of application.

1 Introduction

Analyzing the binding times of a program aims to determine when the value of a variable is available: if the value is known at compile-time it is said to be *static*; if it is not known until run-time it is *dynamic*. This information is important because it characterizes the computations that may be performed at compile-time, thus forming a basis for *partial evaluation* and a generalization of *constant folding* for use in optimizing compilers. Knowing which expressions are static and which are dynamic allows one to process the static semantics of a program by simply following the binding time information, thus the simplifying program transformation phase. Because binding time analysis safely determines the static computations independently of the actual values, binding time information is valid as long as the known and unknown input pattern remains the same. Another motivation of binding time information is that it avoids processing program parts where there are no static computations to perform.

A partial evaluator [3] is the most natural user of binding time information, since in essence it is a static semantics processor [20] that executes those expressions that manipulate static data and freezes the others. Indeed, our analysis has been implemented and is currently used in a self-applicable partial evaluator called Schism [6, 7]. This effort has been very successful, as reported in [8], where we describe the compilation and the generation of a compiler from an interpretive specification of an Algol-like language [17] using Schism. Because the binding time analysis handles higher order functions, continuation semantics can be tackled. Because it handles data structures, it determines whether the injection tag, in the representation of an element of a sum, is static. Therefore, syntax analysis, scope resolution, storage calculation and type checking are actually performed at compile-time.

Our approach is described in three steps: section 2 presents a binding time analysis for a first order language; section 3 extends it to handle data structures;

*This research was supported by Darpa under Grant N00014-88-k-0573

finally section 4 addresses how to handle higher order functions. Section 5 compares this approach with related work and section 6 concludes.

2 First Order Binding Time Analysis

We first present the binding time analysis for a first order functional language.

2.1 A First Order Functional Language

$k \in$ *Constant*
$x \in$ *Variable*
$p \in$ *Primitive*
$f \in$ *Function-Var*
$e \in$ *Expression*
$pr \in$ *Program*

$$pr ::= \{f_1(x_1,\ldots,x_n) = e_1,\ \ldots,\ f_k(x_1,\ldots,x_n) = e_k\}$$
$$e ::= k \mid x \mid e_1 \rightarrow e_2 \,[\!]\, e_3 \mid p(e_1,\ldots,e_n) \mid f(e_1,\ldots,e_n)$$

The program is a set of mutually recursive functions, the first of which (f_1) is the main function of the program. For simplicity, we assume that all functions have the same arity. An expression is either a constant, a variable, a primitive call, or a function call. A call consists of an operator and a list of one or more arguments.

The semantics are straightforward. The calling mechanism is applicative (call-by-value). The conditional has the usual semantics.

2.2 Abstract Values

Binding time analysis is based on an abstract interpretation [1]; for a first order language the following domain is used

$$\delta \in Av = \{\bot_b, Stat, Dyn\}$$

The value $\bot_b$ denotes an expression that has an undefined binding time value. The value *Stat* denotes an expression which can be evaluated statically. Finally, the value *Dyn* denotes a frozen expression (a partial evaluator would generate a residual expression in this case).

This domain forms a chain, with ordering

$$\bot_b \sqsubseteq Stat \sqsubseteq Dyn$$

Note that the value *Stat* could have been used as the initial value rather than introducing the value $\bot_b$ in the abstract domain. However, in practice, this value is useful; it gives extra information about the program. For instance, it allows one to determine the functions which are never invoked during the analysis: their binding time signature only consists of the value $\bot_b$.

2.3 Abstract Environment

Binding time analysis of a program aims to approximate a binding time value for each expression of a program by propagating the specification of its input (*i.e.,* which input are known and which are unknown). In fact, because the present language is first order and referentially transparent, we do not need to annotate each expression with a binding time value; the essential information is the binding time value of each parameter of a function and the binding time value of its result. We call this information the ***binding time signature*** of a function and is defined as

$$\pi \in Signature = Av^n \times Av$$

As an example, consider the function `pairlis`.

```
pairlis(l1, l2) =
    null(l1)  → nil
    [] cons(cons(car(l1), car(l2)),
            pairlis(cdr(l1), cdr(l2)))
```

Assume that `pairlis` has static (known) first parameter and dynamic (unknown) second parameter, its binding time signature would be `pairlis` : *Stat* $\times$ *Dyn* $\rightarrow$ *Dyn*. Indeed, function `pairlis` builds a data which is partially static (`l1` is static and `l2` is dynamic); because we do not deal with data structures yet, the result of such a call is dynamic.

It is easy to assign a binding time value for each expression of a function from its binding time signature; this is achieved by propagating the binding time value of each parameter into the body of the function.

The next version of the binding time analysis (described in section 3) will yield finite descriptions of the data structures built and manipulated by a program. In the above example, such a description will capture the fact that `pairlis` called with a static list and a dynamic list returns a list of pairs, where each pair is static in its `car` and dynamic in its `cdr`.

Given a program and a specification of its input, the binding time analysis safely approximates a binding time signature for each function. We call the set of binding time signatures an ***abstract environment***

(*Abs-Env*): it maps each function of a program to its binding time signature. Let Av and *Function-Var* be respectively the set of binding time values and the set of functions of a program, the abstract environment is defined as

$$\sigma \in \mathit{Abs\text{-}Env} = \mathit{Function\text{-}Var} \rightarrow \mathit{Signature}$$

Because we want the binding time signature of a function to be an approximation of all the calls to this function, each call will be "folded" with the corresponding binding time signature in the *Abs-Env*. This folding operation is defined as follows

$$
\begin{array}{l}
\mathit{fold}_{fn} : \mathit{Function\text{-}Var} \rightarrow Av^n \rightarrow \mathit{Abs\text{-}Env} \rightarrow \mathit{Abs\text{-}Env} \\
\mathit{fold}_{fn} = \lambda f .\ \lambda (\delta_1, \ldots, \delta_n).\ \lambda \sigma . \\
\qquad \textbf{let } \langle\langle \delta'_1, \ldots, \delta'_n \rangle, \delta\rangle = (\sigma f) \\
\qquad \textbf{in} \\
\qquad\quad \sigma[f \mapsto \langle\langle \delta_1 \sqcup \delta'_1, \ldots, \delta_n \sqcup \delta'_n \rangle, \delta\rangle]
\end{array}
$$

Note that all changes in *Abs-Env* are monotonic in the domain Av.

2.4 Abstract Interpretation

Given a program and a specification of its input, the binding time analysis computes an abstract environment that approximates a binding time signature for each function of the program. The abstract interpreter manipulates a binding time environment defined as follows

$$\rho \in \mathit{Env} = \mathit{Variable} \rightarrow Av$$

It is a finite mapping from variables into binding time properties.

The binding time analysis of an expression can be summarized as follow: a constant has a static binding time value; the binding time value of a variable is defined by the binding time environment; a conditional is static if every subcomponent is static, otherwise it is dynamic; a primitive call is static if every argument is static, otherwise it is dynamic; finally, a function call is static if its corresponding binding time signature is static in its result part, otherwise it is dynamic. Note that because we want the binding time signature to be a safe approximations of all the calls to a function, the abstract environment is updated each time a function call is analyzed. Also, because we do not consider data structures in this first version, all primitive calls are treated the same: no attempt is made to collect other information than static or dynamic.

$$
\begin{array}{l}
\mathcal{B}t : \mathit{Expression} \rightarrow \mathit{Env} \rightarrow \mathit{Abs\text{-}Env} \rightarrow Av \times \mathit{Abs\text{-}Env} \\
\mathcal{B}t[\![k]\!]\rho\sigma = \langle \mathit{Stat}, \sigma \rangle \\
\mathcal{B}t[\![x]\!]\rho\sigma = \langle \rho[\![x]\!], \sigma \rangle \\
\mathcal{B}t[\![e_1 \rightarrow e_2 \ [\!]\ e_3]\!]\rho\sigma = \\
\quad \textbf{let } \langle \delta_1, \sigma_1 \rangle = \mathcal{B}t[\![e_1]\!]\rho\sigma \\
\qquad\ \ \langle \delta_2, \sigma_2 \rangle = \mathcal{B}t[\![e_2]\!]\rho\sigma_1 \\
\qquad\ \ \langle \delta_3, \sigma_3 \rangle = \mathcal{B}t[\![e_3]\!]\rho\sigma_2 \\
\quad \textbf{in} \\
\qquad \langle (\delta_1 = \bot_b \rightarrow \bot_b \ [\!]\ \delta_1 = \mathit{Stat} \rightarrow \delta_2 \sqcup \delta_3 \ [\!]\ \mathit{Dyn}),\ \sigma_3 \rangle \\
\mathcal{B}t[\![p\,(e_1, \ldots, e_n)]\!]\rho\sigma = \\
\quad \textbf{let } \langle \delta_1, \sigma_1 \rangle = \mathcal{B}t[\![e_1]\!]\rho\sigma \\
\qquad\ \ \langle \delta_2, \sigma_2 \rangle = \mathcal{B}t[\![e_2]\!]\rho\sigma_1 \\
\qquad\ \ \ldots \\
\qquad\ \ \langle \delta_n, \sigma_n \rangle = \mathcal{B}t[\![e_n]\!]\rho\sigma_{n-1} \\
\quad \textbf{in} \\
\qquad \langle (\bigsqcup_{j=1}^{n} \delta_j),\ \sigma_n \rangle \\
\mathcal{B}t[\![f(e_1, \ldots, e_n)]\!]\rho\sigma = \\
\quad \textbf{let } \langle \delta_1, \sigma_1 \rangle = \mathcal{B}t[\![e_1]\!]\rho\sigma \\
\qquad\ \ \langle \delta_2, \sigma_2 \rangle = \mathcal{B}t[\![e_2]\!]\rho\sigma_1 \\
\qquad\ \ \ldots \\
\qquad\ \ \langle \delta_n, \sigma_n \rangle = \mathcal{B}t[\![e_n]\!]\rho\sigma_{n-1} \\
\qquad\ \ \langle \_, \delta \rangle = \sigma f \\
\quad \textbf{in} \\
\qquad \langle \delta,\ \mathit{fold}_{fn}[\![f]\!]\langle \delta_1, \ldots, \delta_n \rangle \sigma_n \rangle
\end{array}
$$

(The notation "_" means that the corresponding value in the right hand side is unused.)

2.4.1 Fixpoint Iteration

Binding time analysis is done by a fixpoint iteration. The abstract interpretation of a program starts by updating the initial *Abs-Env*: the binding signature of the main function is updated with the specification of the input (static/dynamic). Then, each iteration recomputes the definition of each function with respect to its current binding time signature (the binding time value of the parameters).

Since all changes of *Abs-Env*, performed by fold_{fn}, are monotonic in the domain Av, a fixpoint will be reached in a finite number of iterations.

2.5 An Example

Consider a program defining the function `pairlis`. The initial *Abs-Env* is

$$\mathit{graph}(\sigma_0) = \{\mathit{pairlis\#} \mapsto \langle\langle \bot_b, \bot_b \rangle, \bot_b \rangle\}$$

where *pairlis#* is the abstract version of function `pairlis`. Assume that the function `pairlis` has static first parameter and dynamic second parameter, the *Abs-Env* is:

$$graph(\sigma_1) = \{pairlis\# \mapsto \langle\langle Stat,\ Dyn\rangle,\ \bot_b\rangle\}$$

After the first iteration of the binding time analysis the *Abs-Env* is

$$graph(\sigma_2) = \{pairlis\# \mapsto \langle\langle Stat,\ Dyn\rangle,\ Dyn\rangle\}$$

This *Abs-Env* is unchanged at the next iteration: it is the final *Abs-Env*. Because our analysis so far does not handle data structures it is unable to detect that, in this context, function **pairlis** returns a list of pairs, where each pair is static in its **car** and dynamic in its **cdr**. Instead, it has inferred that function **pairlis** returns a dynamic value. Therefore the static data cannot be used and the expressions that manipulate them cannot be executed statically.

3 First Order Binding Time Analysis with Data Structures

In this section, we propose a second binding time analysis that extends the previous approach to handle structured values that contain both static and dynamic parts. These values are called ***partially static structures*** [15]. This extension is achieved by constructing finite descriptions of data structures manipulated by a program.

3.1 Finite Description of Data Structures

The analysis of function **pairlis**, given previously, may generate an infinite number of abstract pairs (*i.e.,* pairs of abstract values) even if the set of abstract values is finite. In order to have a finite description of the data returned by this function, the analysis needs to manipulate a unique reference to each abstract pair instead of the abstract pair itself.

To uniquely reference a data structure we assign a label, similar to a program point, to each occurrence of a data structure constructor. The function **pairlis** now becomes

```
pairlis(l1, l2) =
    null(l1)  → nil
      ▯ ε1:cons(ε2:cons(car(l1), car(l2)),
                pairlis(cdr(l1), cdr(l2)))
```

Assume that the function **pairlis** has its first parameter static and its second parameter dynamic, the data structures it builds would be described by

$$\begin{aligned} \varepsilon_1 &= \langle \varepsilon_2,\ \varepsilon_1\rangle \\ \varepsilon_2 &= \langle Stat,\ Dyn\rangle \end{aligned}$$

The result of the function **pairlis** would be ε_1. This abstract pair refers to ε_2 in its **car** and to itself in its **cdr**. This self-reference expresses the fact that **pairlis** builds a list of arbitrary length. The abstract pair ε_2 contains a static and a dynamic data. In other words, for such abstract values, **pairlis** builds a list of pairs whose **car** is static and whose **cdr** is dynamic.

For the rest of this paper we will only consider the pair data structure. However, it is straightforward to extend this approach to other data structures.

The label attached to each occurrence of the constructor **cons** is called a *cons point* and CP represents the set of cons points in a program.

3.2 Abstract Values and Cons Points

We now define a new set of abstract values that includes the cons points

$$Av = \{\bot_b,\ Stat,\ C,\ Dyn\}$$

Where $C \in \mathcal{P}(CP)\backslash\emptyset$. Indeed, like a collecting interpretation [11], the binding time analysis will collect the set of possible cons points that an expression may use. The domain Av is a lattice with the ordering

$$\begin{array}{ll} \bot_b \sqsubseteq Stat & \\ Stat \sqsubseteq C & \textit{for all } C \in \mathcal{P}(CP)\backslash\emptyset \\ C_1 \sqsubseteq C_2 & \textit{iff } C_1 \subseteq C_2 \textit{ (subset inclusion)} \\ C \sqsubseteq Dyn & \textit{for all } C \in \mathcal{P}(CP)\backslash\emptyset \end{array}$$

This ordering captures the fact that a partially static structure lies between a static and a dynamic value.

We have seen that each cons point is bound to an abstract pair; the set of abstract pairs is defined as

$$AP = Av \times Av$$

Notice that the domain Av, and thus AP, are finite because the set of cons points is finite.

3.3 Extension of the Abstract Environment to Handle Data Structures

The *Abs-Env*, which represents the state of the analysis, has to be extended to capture the use of cons points.

$$Abs\text{-}Env = (Function\text{-}Var \rightarrow Signature) + (CP \rightarrow AP)$$

The initial value of an abstract pair will be $\langle \bot_b, \bot_b \rangle$. As for the binding time signatures, we define an operation to fold an invocation of **cons** with the corresponding abstract pair contained in the *Abs-Env*.

$$
\begin{aligned}
&fold_{cp} : CP \rightarrow AP \rightarrow Abs\text{-}Env \rightarrow Abs\text{-}Env \\
&fold_{cp} = \lambda\, \varepsilon.\ \lambda\, (\delta_1, \delta_2).\ \lambda\, \sigma. \\
&\qquad \textbf{let } \langle \delta_1\ , \delta_2 \rangle\ = (\sigma\varepsilon) \\
&\qquad \textbf{in} \\
&\qquad\quad \sigma[\varepsilon \mapsto \langle \delta_1 \sqcup \delta'_1, \delta_2 \sqcup \delta'_2 \rangle]
\end{aligned}
$$

3.4 Abstract Primitives

We now define the abstract primitives that operate on the list data type.

$$
\begin{aligned}
&cons\# : AP \times CP \rightarrow Abs\text{-}Env \rightarrow (Av \times Abs\text{-}Env) \\
&cons\#(\delta_1, \delta_2, \varepsilon) = \lambda\sigma. \\
&\quad \textbf{let } \sigma' = fold_{cp}\ \varepsilon\ \langle \delta_1, \delta_2 \rangle\ \sigma \\
&\qquad\ \langle \delta'_1, \delta'_2 \rangle\ = (\sigma'\varepsilon) \\
&\quad \textbf{in} \\
&\qquad (\delta'_1\ = \delta'_2) \wedge \delta'_1 \in \{\bot_b, Stat, Dyn\}\ \rightarrow \langle \delta'_1, \sigma' \rangle \\
&\qquad [\!]\ \langle \{\varepsilon\}, \sigma' \rangle
\end{aligned}
$$

Notice that the abstract primitive *cons#* does not return a cons point when the corresponding abstract pair is completely *Stat*, *Dyn* or $\bot_b$. This is to be consistent with the domain *Av* which defines a partially static structure as lying between a completely static value and a completely dynamic value, exclusively.

$$
\begin{aligned}
&car\# : Av\ \rightarrow Abs\text{-}Env\ \rightarrow (Av \times Abs\text{-}Env) \\
&car\#\ \bot_b = \lambda\sigma.\ \langle \bot_b\ , \sigma \rangle \\
&car\#\ Stat = \lambda\sigma.\ \langle Stat, \sigma \rangle \\
&car\#\ Dyn = \lambda\sigma.\ \langle Dyn, \sigma \rangle \\
&car\# = \lambda\delta.\lambda\sigma.\ \langle (\bigsqcup \{\delta_{i1}\ /\ \delta = \{\varepsilon_1, \ldots, \varepsilon_n\} \wedge \\
&\qquad\qquad\qquad\qquad \langle \delta_{i1}, \delta_{i2} \rangle\ = (\sigma\varepsilon_i)\}, \sigma) \rangle
\end{aligned}
$$

The abstract primitive *cdr#* is defined the same way, except for the last clause where δ_{i2} is considered.

The sub-type of a list (pair or null) can be determined statically when it is either a static value or a partially static structure. The abstract version of the predicate **null** is defined below; *pair#* can be defined similarly.

$$
\begin{aligned}
&null\# : Av\ \rightarrow Abs\text{-}Env\ \rightarrow (Av \times Abs\text{-}Env) \\
&null\#\ \bot_b = \lambda\sigma.\ \langle \bot_b\ , \sigma \rangle \\
&null\#\ Dyn = \lambda\sigma.\ \langle Dyn, \sigma \rangle \\
&null\# = \lambda\delta.\lambda\sigma.\ \langle Stat, \sigma \rangle
\end{aligned}
$$

3.5 Abstract Interpretation

The definition of the abstract interpreter is the same as in section 2.4 except for the treatment of the primitives.

$$
\begin{aligned}
&\mathcal{B}t[\![p\,(e_1, \ldots, e_n)]\!]\rho\sigma = \\
&\quad \textbf{let } \langle \delta_1, \sigma_1 \rangle\ = \mathcal{B}t[\![e_1]\!]\rho\sigma \\
&\qquad\ \ \langle \delta_2, \sigma_2 \rangle\ = \mathcal{B}t[\![e_2]\!]\rho\sigma_1 \\
&\qquad\ \ \ldots \\
&\qquad\ \ \langle \delta_n, \sigma_n \rangle\ = \mathcal{B}t[\![e_n]\!]\rho\sigma_{n-1} \\
&\quad \textbf{in} \\
&\qquad \widehat{Prim}\,[\![p]\!]\,\langle \delta_1, \ldots, \delta_n \rangle\ \sigma_n
\end{aligned}
$$

The primitives are now treated as follows

$$
\begin{aligned}
&\widehat{Prim} : Primitive \rightarrow Av^n \rightarrow Abs\text{-}Env\ \rightarrow (Av \times Abs\text{-}Env) \\
&\widehat{Prim}\,[\![\varepsilon\text{:cons}]\!] = \lambda(\delta_1,\delta_2).\ \lambda\sigma.\ cons\#(\delta_1, \delta_2, \varepsilon)\ \sigma \\
&\widehat{Prim}\,[\![\text{car}]\!] = \lambda\delta.\ \lambda\sigma.\ car\#\ \delta\ \sigma \\
&\ldots \\
&\widehat{Prim}\,[\![p]\!] = \lambda(\delta_1, \ldots, \delta_n).\ \lambda\sigma.\ \langle (\bigsqcup_{j=1}^{n} \delta_j), \sigma \rangle
\end{aligned}
$$

3.6 An Example

As an example consider the function **pairlis** which would be called with a static list and a dynamic list.

$$
\begin{aligned}
&\texttt{pairlis(l1, l2) =} \\
&\quad \texttt{null(l1)}\ \rightarrow \texttt{nil} \\
&\qquad [\!]\ \varepsilon_1\texttt{:cons(}\varepsilon_2\texttt{:cons(car(l1), car(l2)),} \\
&\qquad\qquad\qquad \texttt{pairlis(cdr(l1), cdr(l2)))}
\end{aligned}
$$

For this function, the set of cons points is $CP = \{\varepsilon_1, \varepsilon_2\}$. The analysis has to solve the following equation, where *Abs-Env* has been omitted for clarity.

$$
\begin{aligned}
&pairlis\#(Stat, Dyn) = \\
&\quad null\#(Stat)\ \rightarrow Stat \\
&\quad [\!]\ cons\#(cons\#(Stat, Dyn, \varepsilon_2), \\
&\qquad\qquad pairlis\#(Stat, Dyn), \varepsilon_1)
\end{aligned}
$$

The *Abs-Env* containing the initial call to **pairlis** is

$$
\begin{aligned}
graph(\sigma_0)\ = \{\ &[pairlis\# \mapsto \langle \langle Stat, Dyn \rangle, \bot_b \rangle] \\
&[\varepsilon_1 \mapsto \langle \bot_b, \bot_b \rangle] \\
&[\varepsilon_2 \mapsto \langle \bot_b, \bot_b \rangle]\ \}
\end{aligned}
$$

$$
\begin{aligned}
&pairlis\#(Stat, Dyn) = \\
&\quad null\#(Stat)\ \rightarrow Stat \\
&\quad [\!]\ cons\#(cons\#(Stat, Dyn, \varepsilon_2), \bot_b, \varepsilon_1) \\
&\quad = Stat \rightarrow Stat\ [\!]\ cons\#(cons\#(Stat, Dyn, \varepsilon_2), \bot_b, \varepsilon_1) \\
&\quad = Stat \sqcup cons\#(cons\#(Stat, Dyn, \varepsilon_2), \bot_b, \varepsilon_1) \\
&\quad = cons\#(\{\varepsilon_2\}\ , \bot_b, \varepsilon_1)
\end{aligned}
$$

$$\begin{aligned} graph(\sigma_1) &= \{\ [pairlis\# \mapsto \langle\langle Stat,\ Dyn\rangle,\ \perp_b\rangle] \\ &\quad\ [\varepsilon_1 \mapsto \langle\perp_b,\ \perp_b\rangle] \\ &\quad\ [\varepsilon_2 \mapsto \langle Stat,\ Dyn\rangle]\ \} \end{aligned}$$

$$pairlis\#(Stat,\ Dyn) = \{\varepsilon_1\}$$

$$\begin{aligned} graph(\sigma_2) &= \{\ [pairlis\# \mapsto \langle\langle Stat,\ Dyn\rangle,\ \{\varepsilon_1\}\rangle] \\ &\quad\ [\varepsilon_1 \mapsto \langle\{\varepsilon_2\}\ ,\ \perp_b\rangle] \\ &\quad\ [\varepsilon_2 \mapsto \langle Stat,\ Dyn\rangle]\ \} \end{aligned}$$

At the next iteration the result of *pairlis#* is $\{\varepsilon_1\}$. The *Abs-Env* yielded after the second iteration is

$$\begin{aligned} graph(\sigma_n) &= \{\ [pairlis\# \mapsto \langle\langle Stat,\ Dyn\rangle,\ \{\varepsilon_1\}\rangle] \\ &\quad\ [\varepsilon_1 \mapsto \langle\{\varepsilon_2\}\ ,\ \{\varepsilon_1\}\ \rangle] \\ &\quad\ [\varepsilon_2 \mapsto \langle Stat,\ Dyn\rangle]\ \} \end{aligned}$$

This final *Abs-Env* is the solution of this analysis. The *Abs-Env* indicates that the function **pairlis** returns a list of pairs. This list is of any length. Each element of this list is an abstract pair (ε_2) whose **car** is static and **cdr** dynamic.

For interpretive specifications of programming languages **pairlis** might be a function that constructs an environment, binding each variable (static) to its value (dynamic). From the above description we may deduce that the location of the value of a variable in the environment can be determined statically. Indeed, consider the following function

```
assoc(k, l) =
    null(l)  → nil
    ▯ car(car(l)) = k  → car(l)
     ▯ assoc(k, cdr(l))
```

Assume that function **assoc** looks up a variable (**k**) in an environment (**l**) constructed by function **pairlis**, and that the first parameter of function **assoc** is static and its second parameter is cons point ε_1 described above. Then, the analysis will determine the following: by definition of the abstract primitive *null#*, the expression **null(l)** is static because variable **l** is bound to a cons point. Since the expression **car(car(l))** refers to the static part of the environment and variable **k** is static, the expression **car(car(l)) = k** is also static. After fixpoint iteration, the analysis will yield the following information.

$$assoc\#(Stat,\ \varepsilon_1) = \varepsilon_2$$

As a result we know that, in this context, all the tests performed by function **assoc** can be reduced statically. The resulting expression will then be a sequence of **cdr** to go to a given pair of binding variable/value, and a **car** to access it.

4 Higher Order Binding Time Analysis with Data Structures

In this section we extend the last analysis further to handle higher order functions. We will see that functions and primitives are relatively easy to handle. It is the abstractions that necessitate most of the extensions.

4.1 Syntax of a Higher Order Functional Language

$k \in$ *Constant*
$x \in$ *Variable*
$p \in$ *Primitive*
$f \in$ *Function-Var*
$e \in$ *Expression*
$pr \in$ *Program*

$$\begin{aligned} pr &::= \{f_1 = (\lambda(x_1, \ldots, x_n)\ e_1),\ \ldots, \\ &\qquad f_k = (\lambda(x_1, \ldots, x_n)\ e_k)\} \\ e &::= k \mid x \mid e_1 \ \rightarrow e_2 \ [\!]\ e_3 \mid p \mid f \\ &\quad \mid (\lambda(x_1, \ldots, x_n)\ e) \mid e_1(e_2,\ \ldots,\ e_n) \end{aligned}$$

4.2 Finite Description of Closures

We extend the treatment of the functions to include abstractions. First, we define a unique reference for each abstraction. As we did for the data structures, we will attach a label to each abstraction. We call such a label a *closure point* and the set of closure points in a program is denoted by CLP. This unique reference allows us to bind each abstraction to a binding time signature, just as a function.

However, since an abstraction may contain free variables, in addition to the binding time signature, we must also consider the binding time environment. These two elements will represent an *abstract closure*; it is defined as follows

$$AC = Env \times Signature$$

This environment will initially be $\lambda\, x.\emptyset$. It will be used during the iteration to restore the context of the abstraction in order to analyze its body.

4.3 Separating the Operators from the Other Abstract Values

Since we are dealing with an untyped functional language, a value may be of any type, including a function. In the context of a binding time analysis, this means that the same variable may be bound to an abstract value (static or dynamic) and a function. We would like to be able to use the function wherever this variable is used as an operator and its binding time value if it is an operand of some primitives.

To do this, we propose to separate functions from other values. An abstract value will now be a pair defined as

$$Av = Bav \times SPO$$

The first domain (Bav) is the set of abstract values $\{\perp_b, Stat, C, Dyn\}$ defined in the previous section. The second is the set of possible operators: $SPO = \{O, \top_p\}$ where $O \in \mathcal{P}(Primitive \cup Function\text{-}Var \cup CLP)$. Indeed, the binding time analysis will have to approximate the set of possible operators an expression may either return or use. The value $\top_p$ denotes the unknown operator. This value is used when the operator cannot be determined statically. The set SPO is ordered as follows

$O_1 \sqsubseteq O_2$ *iff* $O_1 \subseteq O_2$ *(subset inclusion)*
$O \sqsubseteq \top_p$ *for all* $O \in \mathcal{P}(Primitive \cup Function\text{-}Var \cup CLP)$

Notations: for $\delta \in Av$
δ^v will refer to the first element of δ: Bav
δ^f will refer to the second element: SPO

The least upper bound on Av is defined as

$$\sqcup_{av} = \lambda\delta_1.\lambda\delta_2.\ \langle \delta^v{}_1 \sqcup \delta^v{}_2\ ,\ \delta^f{}_1 \sqcup \delta^f{}_2 \rangle$$

Finally, we extend the $Abs\text{-}Env$ to handle the closures:

$$Abs\text{-}Env = (Function\text{-}Var \rightarrow Signature) + (CLP \rightarrow AC) + (CP \rightarrow AP)$$

4.4 Abstract Interpretation

Elements from SPO and CLP are respectively denoted by o and η.

$\mathcal{B}t$: $Expression \rightarrow Env \rightarrow Abs\text{-}Env \rightarrow Av \times Abs\text{-}Env$
$\mathcal{B}t[\![k]\!]\rho\sigma = \langle\langle Stat, \emptyset\rangle, \sigma\rangle$
$\mathcal{B}t[\![x]\!]\rho\sigma = \langle\rho[\![x]\!], \sigma\rangle$
$\mathcal{B}t[\![e_1 \rightarrow e_2 \,[\!]\, e_3]\!]\rho\sigma =$
  let $\langle\delta_1, \sigma_1\rangle = \mathcal{B}t[\![e_1]\!]\rho\sigma$
    $\langle\delta_2, \sigma_2\rangle = \mathcal{B}t[\![e_2]\!]\rho\sigma_1$
    $\langle\delta_3, \sigma_3\rangle = \mathcal{B}t[\![e_3]\!]\rho\sigma_2$
  in
    $\langle(\delta^v_1 = \perp_b \rightarrow \langle\perp_b, \delta^f_2 \sqcup \delta^f_3\rangle$
     $[\!]\ \delta^v_1 = Stat \rightarrow (\delta_2 \sqcup \delta_3)$
     $[\!]\ \langle Dyn, \top_p\rangle), \sigma_3\rangle$
$\mathcal{B}t[\![p]\!]\rho\sigma = \langle\langle\perp_b, \{p\}\rangle, \sigma\rangle$

$\mathcal{B}t[\![f]\!]\rho\sigma = \langle\langle\perp_b, \{f\}\rangle, \sigma\rangle$
$\mathcal{B}t[\![(\eta{:}\lambda(x_1, \ldots, x_n)e)]\!]\rho\sigma =$
  let $\langle\_, \pi\rangle = \sigma\eta$
  in
    $\langle\langle\perp_b, \{\eta\}\rangle, \sigma[\eta \mapsto \langle\rho, \pi\rangle]\rangle$
$\mathcal{B}t[\![e_1(e_2, \ldots, e_n)]\!]\rho\sigma =$
  let $\langle\delta_1, \sigma_1\rangle = \mathcal{B}t[\![e_1]\!]\rho\sigma$
    $\langle\delta_2, \sigma_2\rangle = \mathcal{B}t[\![e_2]\!]\rho\sigma_1$
    ...
    $\langle\delta_n, \sigma_n\rangle = \mathcal{B}t[\![e_n]\!]\rho\sigma_{n-1}$
  in
    $(apply\ o_1\ \langle\delta_2, \ldots, \delta_n\rangle) \circ \ldots \circ$
     $(apply\ o_k\ \langle\delta_2, \ldots, \delta_n\rangle)\ \langle\langle\perp_b, \emptyset\rangle, \sigma_n\rangle$
    where $o_i \in \{o \in \delta^f_1\ /\ \text{arity}(o) = n\}$

Notice that when the test of a conditional expression is dynamic, the binding time value of the whole expression is $\langle Dyn, \top_p\rangle$. The value $\top_p$ expresses the fact that the conditional will not be determined statically, and thus, if the truth or the false branch return some operators, they cannot be considered. This implies that these operators (in the case of a function or a closure) should be considered as having dynamic parameters.

The analysis of an abstraction amounts to updating the $Abs\text{-}Env$ with the current environment for the corresponding closure point. When it is applied, its binding time signature is updated. The function *apply* is defined below.

$apply$: $SPO \rightarrow Av^n \rightarrow Av \times Abs\text{-}Env \rightarrow Av \times Abs\text{-}Env$
$apply[\![f]\!] = \lambda(\delta_1, \ldots, \delta_n).\ \lambda(\delta, \sigma).$
  let $\langle\langle\delta'_1, \ldots, \delta'_n\rangle, \delta'\rangle = (\sigma f)$
  in
    $\langle\delta \sqcup \delta', \sigma[f \mapsto \langle\langle\delta_1 \sqcup \delta'_1, \ldots, \delta_n \sqcup \delta'_n\rangle, \delta'\rangle]\rangle$

$apply[\![\eta]\!] = \lambda(\delta_1, \ldots, \delta_n).\ \lambda(\delta, \sigma).$
  let $\langle\rho', \langle\langle\delta'_1, \ldots, \delta'_n\rangle, \delta'\rangle\rangle = (\sigma\eta)$
  in
    $\langle\delta \sqcup \delta', \sigma[\eta \mapsto \langle\rho', \langle\langle\delta_1 \sqcup \delta'_1, \ldots, \delta_n \sqcup \delta'_n\rangle, \delta'\rangle\rangle]\rangle$

We omit the definition of *apply* for the primitives that operate on lists or other data structures: their definitions are essentially the same as in the previous abstract interpreter.

The iteration process will almost remain the same. Note that the treatment of a closure point is performed by first looking up the binding time environment together with the corresponding binding time signature. The binding time environment is then extended with the binding time signature (the abstract value of the parameters). Finally, the body of the abstraction is analyzed in this environment and the $Abs\text{-}Env$ is updated with the result.

4.5 An Example

The following program combines higher order functions and data structures.

```
{ f(n, l) = map ((η:λ(e) n + e), l)
  map(fun, l) =
   null(l)  → nil
   ▯ ε:cons(fun(car(l)), map(fun, cdr(l))) }
```

The final *Abs-Env* for the function call $f\#(\langle Dyn, \emptyset\rangle, \langle Stat, \emptyset\rangle)$ is

$$
\begin{aligned}
&graph(\sigma_n) \;= \\
&\quad \{\ [f\# \mapsto \langle\langle(Dyn,\ \emptyset),\ (Stat,\ \emptyset)\rangle,\ (\{\varepsilon\},\ \emptyset)\rangle] \\
&\quad\ \ [map\# \mapsto \langle\langle(\bot_b,\ \{\eta\}),\ (Stat,\ \emptyset)\rangle,\ (\{\varepsilon\},\ \emptyset)\rangle] \\
&\quad\ \ [\eta \mapsto \langle\ \{[\mathtt{n} \mapsto (Dyn,\ \emptyset)]\}\ ,\ \langle(Stat,\ \emptyset),\ (Dyn,\ \emptyset)\rangle\rangle] \\
&\quad\ \ [\varepsilon \mapsto \langle(Dyn,\ \emptyset),\ (\{\varepsilon\},\ \emptyset)\rangle]\ \}
\end{aligned}
$$

Using this information and given the list `[1, 2, 3]`, a partial evaluator yields the following residual program.

```
{f'(n) = cons(1+n,cons(2+n,cons(3+n,nil)))}
```

4.6 Some Remarks

Note that our approach to treat higher order functions does not require any prior phase to approximate the set of closures that a given expression may evaluate to. This phase, called closure analysis [21], has been avoided by introducing the set of possible operators in the domain of abstract values.

Note also that when applied, an abstraction is never analyzed recursively; instead its binding time signature is updated and the iteration process will treat it just as a function. This strategy is crucial to guarantee termination of the analysis. Indeed, in an untyped functional language a set of abstractions may represent a cycle in the call graph.

A typical example is a fixpoint operator (written for eager evaluation)

```
fix f  = let v  = λx . f (λe . ((x x) e))
         in
            (v v)
```

For such functions a recursive analysis of abstractions would cause non-termination of the analysis, as mentioned in [10].

5 Related Work

Partial evaluation has been the primary motivation for this work. The MIX project at the University of Copenhagen has first pointed out the utility of binding time analysis to improve the partial evaluation process and to achieve self-application [14, 5]. A complete system was implemented including a binding time analysis for first order recursive equations.

In [15], a binding time analysis which handles partially static structures is described. As in [12, 13], regular tree grammars are used to obtain finite descriptions of data structures manipulated by a program. This approach is limited to a first order untyped functional language and requires prior transformations of the program being analyzed (alpha-conversion and restricted terms for the arguments of **cons**).

A binding time analysis for a higher order untyped functional language is presented in [4]. It is limited to flat domains and requires a closure analysis.

Binding time analyses for a higher order typed functional language are described in [18] and [16]. The type information of a program are used to deduce binding time descriptions.

In [9], it is shown how binding time information can be further exploited to improve the partial evaluation process. Indeed, binding time information can be compiled into directives, driving the partial evaluator as to *what* to do for each expression, instead of *how* to use the result of partially evaluating an expression.

6 Conclusion

We have presented a method for performing binding analysis of untyped functional languages. Given a program and a specification of its input, our analysis yields the binding time signature of each function as well as the binding time descriptions of the data manipulated by the program. This analysis can be useful for applications such as compile-time optimizations [2, 19], denotational definitions [18] and partial evaluation [14, 5].

We have implemented our binding time analysis and it is used in our partial evaluator for a side-effect free dialect of Scheme.

Acknowledgements

Thanks go to Olivier Danvy, Paul Hudak, Siau Cheng Khoo, and Raman Srinivas Sundaresh for commenting earlier versions of this paper.

References

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

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

[3] D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation.* North-Holland, 1988.

[4] A. Bondorf. Automatic autoprojection of higher order recursive equations. In N. D. Jones, editor, *ESOP'90, 3^{rd} European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[5] A. Bondorf, N. D. Jones, T. Mogensen, and P. Sestoft. Binding time analysis and the taming of self-application. Diku report, University of Copenhagen, Copenhagen, Denmark, 1988.

[6] C. Consel. New insights into partial evaluation: the Schism experiment. In H. Ganzinger, editor, *ESOP'88, 2^{nd} European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 236–246. Springer-Verlag, 1988.

[7] C. Consel. *Analyse de Programmes, Evaluation Partielle et Generation de Compilateurs.* PhD thesis, Université de Paris VI, Paris, France, 1989.

[8] C. Consel and O. Danvy. Static and dynamic semantics processing. Research Report 761, Yale University, New Haven, Connecticut, USA, 1989.

[9] C. Consel and O. Danvy. From interpreting to compiling binding times. In N. D. Jones, editor, *ESOP'90, 3^{rd} European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[10] P. Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. In *ACM Symposium on Principles of Programming Languages*, pages 97–109, 1986.

[11] P. Hudak and J. Young. A collecting interpretation of expressions without powerdomains). In *ACM Symposium on Principles of Programming Languages*, pages 107–118, 1988.

[12] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of lisp-like structures. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[13] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *ACM Conference on Principles of Programming Languages*, pages 66–74, 1982.

[14] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2:9–50, 1989.

[15] 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 325–348. North-Holland, 1988.

[16] T. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Diaz and F. Orejas, editors, *International Joint Conference on Theory and Practice of Software Development*, volume 352 of *Lecture Notes in Computer Science*, pages 298–312. Springer-Verlag, 1989.

[17] M. Montenyohl and M. Wand. Correct flow analysis in continuation semantics. In *ACM Symposium on Principles of Programming Languages*, pages 204–218, 1988.

[18] H. R. Nielson and F. Nielson. Automatic binding time analysis for a typed λ-calculus. In *ACM Symposium on Principles of Programming Languages*, pages 98–106, 1988.

[19] H. R. Nielson and F. Nielson. Tranformations on higher-order functions. In *FPCA'89, 4^{th} International Conference on Functional Programming Languages and Computer Architecture*, pages 129–143, 1989.

[20] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[21] P. Sestoft. Replacing function parameters by global variables. In *FPCA'89, 4^{th} International Conference on Functional Programming Languages and Computer Architecture*, pages 39–53, 1989.