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 anal-
ysis. 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 knowl-
edge 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 gen-
eral 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 be-
yond 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 re-
strictions 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 de-
terminate 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 character-
izes the computations that may be performed at
compile-time, thus forming a basis for partial evalua-
tion 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 follow-
ing the binding time information, thus the simplifying
program transformation phase. Because binding time
analysis safely determines the static computations in-
dependently of the actual values, binding time informa-
tion is valid as long as the known and unknown
input pattern remains the same. Another motivation
of binding time information is that it avoids process-
ing program parts where there are no static computa-
tions 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 expres-
sions that manipulate static data and freezes the oth-
ers. 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 success-
ful, as reported in [8], where we describe the compi-
lation and the generation of a compiler from an inter-
pretive specification of an Algol-like language [17]
using Schism. Because the binding time analysis han-
dles higher order functions, continuation semantics
can be tackled. Because it handles data structures,
it determines whether the injection tag, in the repre-
sentation 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 lan-
guage; section 3 extends it to handle data structures;
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

\[ k \in \text{Constant} \]
\[ x \in \text{Variable} \]
\[ p \in \text{Primitive} \]
\[ f \in \text{Function-Var} \]
\[ e \in \text{Expression} \]
\[ pr \in \text{Program} \]

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

The program is a set of mutually recursive func-
tions, the first of which \( f_1 \) is the main function of
the program. For simplicity, we assume that all func-
tions 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 condi-
tional has the usual semantics.

2.2 Abstract Values

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

\[ \delta \in Av = \{ \perp_b, Stat, Dyn \} \]

The value \( \perp_b \) denotes an expression that has an
undefined binding time value. The value \( Stat \) denotes
an expression which can be evaluated statically. Fi-
nally, 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

\[ \perp_b \sqsubseteq Stat \sqsubseteq Dyn \]

Note that the value \( Stat \) could have been used as
the initial value rather than introducing the value \( \perp_b \)
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
\( \perp_b \).

2.3 Abstract Environment

Binding time analysis of a program aims to approx-
imate 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 es-
ential 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 \textit{binding time}
signature of a function and is defined as

\[ \pi \in Signature = \text{Av}^n \times \text{Av} \]

As an example, consider the function \texttt{pairlis}.

\[
\text{pairlis}(11, 12) = \\
\quad \text{null}(11) \rightarrow \text{nil} \\
\quad \mid \text{cons}((\text{cons}(\text{car}(11), \text{car}(12)), \\
\quad \text{pairlis}(\text{cdr}(11), \text{cdr}(12)))
\]

Assume that \texttt{pairlis} has static (known) first pa-
rameter and dynamic (unknown) second parameter,
it binding time signature would be \texttt{pairlis} : 
\( Stat \times Dyn \rightarrow Dyn \). Indeed, function \texttt{pairlis}
builds a data which is partially static (11 is static and 12
is dynamic); because we do not deal with data struc-
tures 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 signa-
ture; 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 (de-
scribed in section 3) will yield finite descriptions of
the data structures built and manipulated by a pro-
gram. In the above example, such a description will
capture the fact that \texttt{pairlis} called with a static list
and a dynamic list returns a list of pairs, where each
pair is static in its \texttt{car} and dynamic in its \texttt{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 \textit{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

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

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

$$\text{fold}_f^n : \text{Function-Var} \rightarrow (\text{Av})^n \rightarrow \text{Abs-Env} \rightarrow \text{Abs-Env}$$

$$\text{fold}_f^n = \lambda f. \lambda (\delta_1, \ldots, \delta_n). \lambda \sigma.
\text{let } ((\delta'_1, \ldots, \delta'_n), \delta) = (\sigma f)
\text{ in }
\sigma[f \mapsto ((\delta_1 \cup \delta'_1), \ldots, \delta_n \cup \delta'_n), \delta]$$

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

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

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

The binding time analysis of an expression can be summarized as follows: 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 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.

$$\text{Bt}: \text{Expression} \rightarrow \text{Env} \rightarrow \text{Abs-Env} \rightarrow \text{Av} \times \text{Abs-Env}$$

$$\text{Bt}[k]_{\rho \sigma} = (\text{Stat}, \sigma)$$

$$\text{Bt}[x]_{\rho \sigma} = (\rho, x), \sigma$$

$$\text{Bt}[e_1 \rightarrow e_2 \mid e_3]_{\rho \sigma} =$$

$$\text{let } (\delta_1, \sigma_1) = \text{Bt}[e_1]_{\rho \sigma}
(\delta_2, \sigma_2) = \text{Bt}[e_2]_{\rho \sigma_1}
(\delta_3, \sigma_3) = \text{Bt}[e_3]_{\rho \sigma_2}
in
((\delta_1 = \bot_k \rightarrow \bot_k \cup \delta_1 = \text{Stat} \rightarrow \delta_2 \cup \delta_3 \cup \text{Dyn}), \sigma_3)$$

$$\text{Bt}[p(e_1, \ldots, e_n)]_{\rho \sigma} =$$

$$\text{let } (\delta_1, \sigma_1) = \text{Bt}[e_1]_{\rho \sigma}
(\delta_2, \sigma_2) = \text{Bt}[e_2]_{\rho \sigma_1}
\ldots
(\delta_n, \sigma_n) = \text{Bt}[e_n]_{\rho \sigma_{n-1}}
in
\text{fold}_f^n ((\delta_1 \cup \delta'_1), \ldots, \delta_n \cup \delta'_n), \sigma)
$$

(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 foldfn, 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

$$\text{graph}(\sigma_0) = \{\text{pairlis}\# \mapsto (((\bot_k, \bot_k), \bot_k))\}$$

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:

$$\text{Bt}: \text{Expression} \rightarrow \text{Env} \rightarrow \text{Abs-Env} \rightarrow \text{Av} \times \text{Abs-Env}$$

$$\text{Bt}[k]_{\rho \sigma} = (\text{Stat}, \sigma)$$

$$\text{Bt}[x]_{\rho \sigma} = (\rho, x), \sigma$$

$$\text{Bt}[e_1 \rightarrow e_2 \mid e_3]_{\rho \sigma} =$$

$$\text{let } (\delta_1, \sigma_1) = \text{Bt}[e_1]_{\rho \sigma}
(\delta_2, \sigma_2) = \text{Bt}[e_2]_{\rho \sigma_1}
(\delta_3, \sigma_3) = \text{Bt}[e_3]_{\rho \sigma_2}
in
((\delta_1 = \bot_k \rightarrow \bot_k \cup \delta_1 = \text{Stat} \rightarrow \delta_2 \cup \delta_3 \cup \text{Dyn}), \sigma_3)$$

$$\text{Bt}[p(e_1, \ldots, e_n)]_{\rho \sigma} =$$

$$\text{let } (\delta_1, \sigma_1) = \text{Bt}[e_1]_{\rho \sigma}
(\delta_2, \sigma_2) = \text{Bt}[e_2]_{\rho \sigma_1}
\ldots
(\delta_n, \sigma_n) = \text{Bt}[e_n]_{\rho \sigma_{n-1}}
in
\text{fold}_f^n ((\delta_1 \cup \delta'_1), \ldots, \delta_n \cup \delta'_n), \sigma)$$

(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 foldfn, 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

$$\text{graph}(\sigma_0) = \{\text{pairlis}\# \mapsto (((\bot_k, \bot_k), \bot_k))\}$$

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(σ₁) = {pairlis# ↦ ((Stat, Dyn), ⊥ₜ)}

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

graph(σ₂) = {pairlis# ↦ ((Stat, Dyn), Dyn)}

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

\[\text{pairlis}(11, 12) = \begin{cases} \text{null}(11) \rightarrow \text{nil} \\ \& \varepsilon_1 : \text{cons}(\varepsilon_2 : \text{cons}(	ext{car}(11), \text{car}(12)), \text{pairlis}(	ext{cdr}(11), \text{cdr}(12))) \end{cases}\]

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

\[\varepsilon_1 = (\varepsilon_2, \varepsilon_1)\]
\[\varepsilon_2 = (\text{Stat, Dyn})\]

The result of the function pairlis would be \(\varepsilon_1\). This abstract pair refers to \(\varepsilon_2\) 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 \(\varepsilon_2\) 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

\[\text{Av} = \{\bot_s, \text{Stat}, C, \text{Dyn}\}\]

Where \(C \in \mathcal{P}(\text{CP})\setminus\emptyset\). 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

\[\bot_s \subseteq \text{Stat}\]
\[\text{Stat} \subseteq C\]
\[C_1 \subseteq C_2\]
\[C \subseteq \text{Dyn}\]

for all \(C \in \mathcal{P}(\text{CP})\setminus\emptyset\)

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

\[\text{AP} = \text{Av} \times \text{Av}\]

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-Env = (Function-Var → Signature) + (CP → AP)

The initial value of an abstract pair will be (⊥₄, ⊥₄). 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.

\[
\text{fold}_{\text{cp}} : CP \rightarrow AP \rightarrow \text{Abs-Env} \rightarrow \text{Abs-Env}
\]

\[
fold_{\text{cp}} = \lambda e. \lambda (\delta_1, \delta_2). \lambda \sigma. \\
\text{let } (\delta_1', \delta_2') = (\sigma(e)) \\
\text{in } \\
\sigma[\varepsilon \mapsto (\delta_1 \sqcup \delta_1', \delta_2 \sqcup \delta_2')]
\]

3.4 Abstract Primitives

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

\[
\text{cons#} : AP \times CP \rightarrow \text{Abs-Env} \rightarrow (Av \times \text{Abs-Env})
\]

\[
\text{cons#}(\delta_1, \delta_2, e) = \lambda \sigma. \\
\text{let } \sigma' = \text{fold}_{\text{cp}} \varepsilon (\delta_1, \delta_2) \sigma \\
(\delta_1', \delta_2') = (\sigma(e)) \\
\text{in } \\
(\delta_1' \sqcup \delta_1') \land \delta_1' \in \{⊥₄, \text{Stat}, \text{Dyn}\} \rightarrow (\delta_1', \sigma')
\]

\[
\| (\{\varepsilon\}, \sigma')
\]

Notice that the abstract primitive \text{cons#} does not return a cons point when the corresponding abstract pair is completely \text{Stat}, \text{Dyn} or ⊥₄. This is to be consistent with the domain \text{Av} which defines a partially static structure as lying between a completely static value and a completely dynamic value, exclusively.

\[
\text{car#} : Av \rightarrow \text{Abs-Env} \rightarrow (Av \times \text{Abs-Env})
\]

\[
\text{car#} \sqsubseteq = \lambda \sigma. (\sqsubseteq_\text{ Dyn}, \sigma) \\
\text{car#} \text{Stat} = \lambda \sigma. (\text{Stat}, \sigma) \\
\text{car#} \text{Dyn} = \lambda \sigma. (\text{Dyn}, \sigma) \\
\text{car#} = \lambda \delta. \lambda \sigma. (\sqsubseteq, \sigma) \\
\| (\{\delta_1 \sqsubseteq \delta \subseteq \{\varepsilon_1, \ldots, \varepsilon_n\} \land \\
(\delta_1, \delta_1') = (\sigma(e_1)), \sigma))
\]

The abstract primitive \text{cdr#} is defined the same way, except for the last clause where \delta_12 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 \text{null} is defined below; \text{pair#} can be defined similarly.

\[
\text{null#} : Av \rightarrow \text{Abs-Env} \rightarrow (Av \times \text{Abs-Env})
\]

\[
\text{null#} \sqsubseteq = \lambda \sigma. (\sqsubseteq_\text{ Dyn}, \sigma) \\
\text{null#} \text{Stat} = \lambda \sigma. (\text{Stat}, \sigma) \\
\text{null#} = \lambda \delta. \lambda \sigma. (\text{Stat}, \sigma)
\]

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.

\[
\text{bit}[p(e_1, \ldots, e_n)] \rho \sigma = \\
\text{let } (\delta_1, \sigma_1) = \text{bit}[e_1] \rho \sigma \\
(\delta_2, \sigma_2) = \text{bit}[e_2] \rho \sigma_1 \\
\ldots \\
(\delta_n, \sigma_n) = \text{bit}[e_n] \rho \sigma n-1 \\
\text{in } \\
\text{Prim}[p](\delta_1, \ldots, \delta_n) \sigma_n
\]

The primitives are now treated as follows

\[
\text{Prim} : \text{Primitive} \rightarrow Av^\ast \rightarrow \text{Abs-Env} \rightarrow (Av \times \text{Abs-Env})
\]

\[
\text{Prim}[e: \text{cons}] = \lambda (\delta_1, \delta_2). \lambda \sigma. \text{cons#}(\delta_1, \delta_2, e) \sigma \\
\text{Prim}[\text{car}] = \lambda \delta. \lambda \sigma. \text{car#} \delta \sigma \\
\ldots
\]

\[
\text{Prim}[p] = \lambda (\delta_1, \ldots, \delta_n). \lambda \sigma. ((\sqcup_{j=1}^n \delta_j), \sigma)
\]

3.6 An Example

As an example consider the function \text{pairlis} which would be called with a static list and a dynamic list.

\[
\text{pairlis}(11, 12) = \\
\text{null}(11) \rightarrow \text{nil} \\
[\varepsilon_1: \text{cons}(\varepsilon_2: \text{cons}(\text{car}(11), \text{car}(12)), \\
\text{pairlis}(\text{cdr}(11), \text{cdr}(12)))]
\]

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

\[
\text{pairlis#}(\text{Stat}, \text{Dyn}) = \\
\text{null#}(\text{Stat}) \rightarrow \text{Stat} \\
\| \text{cons#}(\text{pairlis#}(\text{Stat}, \text{Dyn})) \\
\text{pairlis#}(\text{Stat}, \text{Dyn})
\]

The Abs-Env containing the initial call to \text{pairlis} is

\[
\text{graph}(\sigma_0) = \{ [\text{pairlis#} \mapsto ((\text{Stat}, \text{Dyn}), \sqsubseteq_\text{ Dyn})] \\
[\varepsilon_1 \mapsto (\sqsubseteq_\text{ Dyn}, \sqsubseteq_\text{ Dyn})] \\
[\varepsilon_2 \mapsto (\sqsubseteq_\text{ Dyn}, \sqsubseteq_\text{ Dyn})] \\
\}
\]

\[
\text{pairlis#}(\text{Stat}, \text{Dyn}) = \\
\text{null#}(\text{Stat}) \rightarrow \text{Stat} \\
\| \text{cons#}((\text{pairlis#}(\text{Stat}, \text{Dyn})) \sqsubseteq_\text{ Dyn}, \varepsilon_1) \\
= \text{Stat} \sqsubseteq \text{Stat} \\
= \text{Stat} \sqsubseteq \text{cons#}(\text{pairlis#}(\text{Stat}, \text{Dyn})) \\
= \text{Stat} \sqsubseteq \text{cons#}(\text{pairlis#}(\text{Stat}, \text{Dyn})) \\
= \text{cons#}(\{\varepsilon_2\}, \sqsubseteq_\text{ Dyn}, \varepsilon_1)
\]
\[ \text{graph}(\sigma_1) = \{ \text{pairlis#} \mapsto (\text{Stat, Dyn}, \perp_b) \}
\]
\[\{ e_1 \mapsto (\perp_b, \perp_b) \}
\]
\[\{ e_2 \mapsto (\text{Stat, Dyn}) \} \} \]

\[ \text{pairlis#}(\text{Stat, Dyn}) = \{ e_1 \} \]

\[ \text{graph}(\sigma_2) = \{ \text{pairlis#} \mapsto (\text{Stat, Dyn}, \{ e_1 \}) \}
\]
\[\{ e_1 \mapsto (\{ e_2 \}, \perp_b) \}
\]
\[\{ e_2 \mapsto (\text{Stat, Dyn}) \} \} \]

At the next iteration the result of \text{pairlis#} is \{ e_1 \}. The \text{Abs-Env} yielded after the second iteration is

\[ \text{graph}(\sigma_n) = \{ \text{pairlis#} \mapsto (\text{Stat, Dyn}, \{ e_1 \}) \}
\]
\[\{ e_1 \mapsto (\{ e_2 \}, \{ e_1 \}) \}
\]
\[\{ e_2 \mapsto (\text{Stat, Dyn}) \} \} \]

This final \text{Abs-Env} is the solution of this analysis. The \text{Abs-Env} indicates that the function \text{pairlis} returns a list of pairs. This list is of any length. Each element of this list is an abstract pair \( (e_2) \) whose \text{car} is static and \text{cdr} dynamic.

For interpretive specifications of programming languages \text{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

\[ \text{assoc}(k, 1) = \]
\[ \text{null}(1) \rightarrow \text{nil} \]
\[ \| \text{car}(\text{car}(1)) = k \rightarrow \text{car}(1) \]
\[ \| \text{assoc}(k, \text{cdr}(1)) \]

Assume that function \text{assoc} looks up a variable \( k \) in an environment \( 1 \) constructed by function \text{pairlis}, and that the first parameter of function \text{assoc} is static and its second parameter is cons point \( e_1 \) described above. Then, the analysis will determine the following: by definition of the abstract primitive \text{null#}, the expression \text{null}(1) is static because variable 1 is bound to a cons point. Since the expression \text{car}(\text{car}(1)) refers to the static part of the environment and variable \( k \) is static, the expression \text{car}(\text{car}(1)) = k \) is also static. After fixpoint iteration, the analysis will yield the following information.

\[ \text{assoc#}(\text{Stat, } e_1) = e_2 \]

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

\[ \text{4 Higher Order Binding Time Analysis with Data Structures} \]

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.

\[ \text{4.1 Syntax of a Higher Order Functional Language} \]

\[ k \in \text{Constant} \]
\[ x \in \text{Variable} \]
\[ p \in \text{Primitive} \]
\[ f \in \text{Function-Var} \]
\[ e \in \text{Expression} \]
\[ pr \in \text{Program} \]

\[ pr ::= f_1 = (\lambda(x_1, \ldots, x_n) \: e_1), \ldots, f_k = (\lambda(x_1, \ldots, x_n) \: e_k) \]
\[ e ::= k \mid x \mid e_1 \rightarrow e_2 \parallel e_3 \mid p \mid f \]
\[ | (\lambda(x_1, \ldots, x_n) \: e) \mid e_1(e_2, \ldots, e_n) \]

\[ \text{4.2 Finite Description of Closures} \]

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 \textit{closure point} and the set of closure points in a program is denoted by \textit{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 \textit{abstract closure}; it is defined as follows

\[ AC = Env \times Signature \]

This environment will initially be \( \lambda x.0 \). It will be used during the iteration to restore the context of the abstraction in order to analyze its body.

\[ \text{4.3 Separating the Operators from the Other Abstract Values} \]

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 \( \{0, 1, Stat, C, Dyn\} \) defined in the previous section. The second is the set of possible operators: \( SPO = \{O, T_p\} \) where \( O \in P(Primitive \cup Function-Var \cup CLP) \). Indeed, the binding time analysis will have to approximate the set of possible operators an expression may either return or use. The value \( T_p \) denotes the unknown operator. This value is used when the operator cannot be determined statically. The set \( SPO \) is ordered as follows

\[ O_1 \subseteq O_2 \text{ iff } O_1 \subseteq O_2 \text{ (subset inclusion)} \]
\[ O \subseteq T_p \text{ for all } O \in P(Primitive \cup Function-Var \cup CLP) \]

Notations: for \( \delta \in Av \)
\( \delta^\circ \) will refer to the first element of \( \delta : Bav \)
\( \delta^\downarrow \) will refer to the second element: \( SPO \)

The least upper bound on \( Av \) is defined as

\[ \bigcup_{av} = \lambda \delta_1, \lambda \delta_2. (\delta_1 \cup \delta_2, \delta^\downarrow \cup \delta^\downarrow) \]

Finally, we extend the \( Abs-Env \) to handle the closures:

\[ Abs-Env = (Function-Var \rightarrow Signature) + (CLP \rightarrow AC) + (CP \rightarrow AP) \]

4.4 Abstract Interpretation

Elements from \( SPO \) and \( CLP \) are respectively denoted by \( o \) and \( \eta \).

\[ Bt : Expression \rightarrow Env \rightarrow Abs-Env \rightarrow Av \times Abs-Env \]

\[ Bt[k] \rho = (\langle Stat, \emptyset \rangle, \sigma) \]

\[ Bt[x] \rho = \rho[x], \sigma \]

\[ Bt[e_1 \rightarrow e_2] \rho \sigma = \]

let \( \langle \delta_1, \sigma_1 \rangle = Bt[e_1] \rho \sigma \)

\[ \langle \delta_2, \sigma_2 \rangle = Bt[e_2] \rho \sigma_1 \]

\[ \langle \delta_3, \sigma_3 \rangle = Bt[e_3] \rho \sigma_2 \]

in

\[ \langle \delta_1^\downarrow = \bigcup_\delta (\delta_1 \cup \delta_2 \cup \delta_3) \]

\[ (\delta_2^\downarrow = Stat \rightarrow (\delta_2 \cup \delta_3)) \]

\[ (\delta_3^\downarrow = Dyn, T_p)), \sigma_3 \]

\[ Bt [p] \rho \sigma = (\langle \emptyset, \{p\} \rangle, \sigma) \]

\[ Bt[f] \rho \sigma = (\langle \emptyset, \{f\} \rangle, \sigma) \]

\[ Bt[\lambda x_1, \ldots, x_n e] \rho \sigma = \]

let \( \langle, \pi \rangle = \sigma \eta \)

in

\[ \langle \emptyset, \{\eta\} \rangle, \{x_1 \mapsto \langle \rho \mapsto (\rho, \pi) \rangle \}

\[ Bt[e_1 \ldots e_n] \rho \sigma = \]

let \( \langle \delta_1, \sigma_1 \rangle = Bt[e_1] \rho \sigma \)

\[ \langle \delta_2, \sigma_2 \rangle = Bt[e_2] \rho \sigma_1 \]

\[ \ldots \]

\[ \langle \delta_n, \sigma_n \rangle = Bt[e_n] \rho \sigma_{n-1} \]

in

\[ (apply \ a_1 \langle \delta_3, \ldots, \delta_n \rangle) \circ \ldots \circ \]

\[ (apply \ a_k \langle \delta_3, \ldots, \delta_n \rangle) \langle \emptyset, \emptyset \rangle, \sigma_n \]

where \( a_i \in \{ o \in \delta^\downarrow \mid \text{arity}(o) = n \} \)

Notice that when the test of a conditional expression is dynamic, the binding time value of the whole expression is \( (Dyn, T_p) \). The value \( T_p \) 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 \( Abs-Env \) 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.

\[ apply \ : \ SPO \rightarrow Av^n \rightarrow Av \times Abs-Env \rightarrow Av \times Abs-Env \]

\[ apply[f] = \lambda (\delta_1, \ldots, \delta_n) \lambda (\delta, \sigma). \]

let \( \langle \delta_1^\downarrow, \ldots, \delta_n^\downarrow \rangle, (\delta^\downarrow) = (\sigma f) \)

\[ (\delta \cup \delta^\downarrow, \sigma[f \mapsto \langle \delta_1 \cup \delta_1^\downarrow, \ldots, \delta_n \cup \delta_n^\downarrow, \delta^\downarrow \rangle]) \]

\[ apply[\eta] = \lambda (\delta_1, \ldots, \delta_n) \lambda (\delta, \sigma). \]

let \( \langle \delta^\downarrow, \langle \delta_1, \ldots, \delta_n, \delta^\downarrow \rangle \rangle = (\sigma \eta) \)

\[ (\delta \cup \delta^\downarrow, \sigma[\eta \mapsto \langle \rho, \langle \delta_1 \cup \delta_1^\downarrow, \ldots, \delta_n \cup \delta_n^\downarrow, \delta^\downarrow \rangle \rangle]) \]

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 \( Abs-Env \) is updated with the result.
4.5 An Example

The following program combines higher order functions and data structures.

\[
\begin{align*}
    f(n, 1) &= \text{map } ((\eta; \lambda(e) \ n + e), 1) \\
    \text{map}(\text{fun}, 1) &= \\
    \text{null}(1) &\rightarrow \text{nil} \\
    \{\varepsilon; \text{cons}(\text{fun(car}(l)), \text{map(\text{fun, cdr}(l))) \} \\
\end{align*}
\]

The final Abs-Env for the function call \( f^\#((\text{Dyn}, \emptyset), (\text{Stat}, \emptyset)) \) is:

\[
\begin{align*}
    \text{graph}(\sigma_n) &= \{ [f^\# \mapsto (((\text{Dyn}, \emptyset), (\text{Stat}, \emptyset)), (\{\varepsilon\}, \emptyset))] \\
    &\text{map}^\# \mapsto (((\emptyset, [\eta]), (\text{Stat}, \emptyset)), (\{\varepsilon\}, \emptyset))] \\
    &\text{eta} \mapsto \{ [n \mapsto ((\text{Dyn}, \emptyset)), ((\text{Stat}, \emptyset), (\text{Dyn}, \emptyset)))] \\
    &\varepsilon \mapsto ((\text{Dyn}, \emptyset), (\{\varepsilon\}, \emptyset)) \} \\
\end{align*}
\]

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

\[
\{ f'(n) = \text{cons}(1+n, \text{cons}(2+n, \text{cons}(3+n, \text{nil)))) \}
\]

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)

\[
\text{fix } f = \text{let } v = \lambda x. f (\lambda e. ((x x) e)) \text{ in } (v v)
\]

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, Siu Chong Khoo, and Raman Srinivas Sundaresh for commenting earlier versions of this paper.
References


