Accurate Binding-Time Analysis For Imperative Languages: Flow, Context, and Return Sensitivity

Luke Hornof
hornof@irisa.fr
Irisa
Campus Universitaire de Beaulieu
35042 Rennes Cedex, France

Jacques Noyé
noye@emn.fr
École des Mines de Nantes
4 rue Alfred Kastler
44070 Nantes Cedex 03, France

Abstract

Since a binding-time analysis determines how an off-line partial evaluator will specialize a program, the accuracy of the binding-time information directly determines the degree of specialization. We have designed and implemented a binding-time analysis for an imperative language, and integrated it into our partial evaluator for C, called Tempo [9]. This binding-time analysis includes a number of new features, not available in any existing partial evaluator for an imperative language, which are critical when specializing existing programs such as operating system components [24, 25].

- *Flow sensitivity.* A different binding-time description is computed for each program point, allowing the same variable to be considered static at one program point and dynamic at another.
- *Context sensitivity.* Each function call is analyzed with the context of the call site, generating multiple binding-time annotated instances of the same function definition.
- *Return sensitivity.* A different binding-time description is computed for the side-effects and the return value of a function

1 Introduction

Automatic program specialization is emerging as a key software engineering concept which allows software to be generic without sacrificing performance. The motivation for our work on Tempo [9], a partial evaluator for C, is to demonstrate that partial evaluation can provide a realistic basis for automatic program specialization. Therefore, we have chosen to deal with a widely used language, namely C, and focus on optimizing existing, realistic applications. One of the main areas of applications we are looking at is operating system code. Indeed, this is an area where the conflict between generality (an operating system must, by definition, deal with a wide variety of situations) and performance is especially acute. It is therefore not surprising that many opportunities for applying partial evaluation to operating systems code have been identified [11, 12, 28].

However, we have discovered that existing partial-evaluation technology is not sufficiently advanced to effectively specialize the corresponding programs. This is due to a lack of accuracy of binding-time analyses in dealing with typical features of imperative programs, such as pointers, aliases, and side-effecting functions. We have found that *flow*, *context*, *return*, and *use sensitivity* are necessary in a binding-time analysis in order to successfully specialize systems programs.

Use sensitivity is addressed in [18]. The basic idea is that, at specialization time, the value of a variable is allowed to be computed in certain contexts even if the variable identifier is residualized in others. An accurate handling of pointers and structures makes it essential that a single residualized use of an object does not force all other uses to be residualized. This led us to develop an analysis in two different phases. The first phase determines which parts of the program can be computed at specialization time, whereas the second phase determines the actual transformations which will be applied at specialization time.

This paper focuses on the first phase of the analysis, which determines which parts of the program are *static*, *i.e.* can be computed at specialization time, and describes how to obtain *flow*, *context*, and *return* sensitivity. Firstly, flow sensitivity allows a different binding time to be associated with a variable at different program points, *i.e.* a variable is allowed to be static at one point and dynamic at another. Secondly, systems code contains calls to the same function which occur in different system states. Context sensitivity permits each call to be analyzed with respect to its specific state, allowing the different static values in each

Permission to make digital/hard copy of part or all this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
PEPM '97 Amsterdam, ND
© 1997 ACM 0-89791-917-3/97/0006...$3.50

state to be exploited by each call. Finally, a system procedure typically returns some sort of constant error status. Return sensitivity allows the binding-time analysis to take advantage of this constant value, even if the system function contains dynamic constructs.

We have implemented an inter-procedural flow, context, and return-sensitive binding-time analysis and integrated it into Tempo. The analysis deals with a wide subset of C, including in particular multiple returns, pointers, and structures. As a result, significant existing applications can be handled without major rewriting. The results of the analysis are used to drive both Tempo's compile-time and run-time specializer [9, 10]. We have found that, with this extra precision obtained by our analysis, we are able to effectively specialize systems code [24, 25, 31]. It has also been applied successfully to many other application domains such as domain-specific language interpreters [30], and, in the context of run-time specialization, scientific programming and image processing [27].

In the next section, Sect. 2, we explain flow, context, and return sensitivity and show how they improve the precision of the binding-time analysis. The details of the analysis are then presented in Sect. 3. Existing applications on which this analysis is being applied are given in Sect. 4. Related work is addressed in Sect. 5 and final remarks are made in Sect. 6.

2 Sensitivities

Let us look at a few examples that exhibit flow, context, and return sensitivity. For each example, we first present the initial source program. Then we give the program annotated by the first phase of the binding-time analysis, where static constructs are overlined and dynamic constructs underlined. The second phase of the binding-time analysis determines which *action* (*i.e.* transformation) to apply to each construct during specialization. The *evaluate* action instructs the specializer to evaluate the construct and *residualize* instructs the specializer to residualize it. We present the action-annotated program, where overlined constructs are to be evaluated and underlined constructs are to be residualized. Finally, we show the resulting specialized program.

2.1 Flow Sensitivity

A flow-sensitive analysis associates a different state with each program point. This allows variables that are read and written multiple times to be associated with different binding-times at different locations in a program.

In the example in Fig. 1, the function is analyzed with an initial binding-time description specifying that parameters `x`, `y`, and `p` are all static, and that the global variable `d` is dynamic. The variable `x` is read and written multiple times, and is static at some program points and dynamic at others. The left-hand side, or *lvalue*, of an assignment is considered static if it depends only on static data. This is why, in the example, all of the variables which occur on the left-hand side of an assignment are static.

Pointers and aliasing may create an *ambiguous* definition, an assignment for which the analysis cannot statically determine which location will be modified at run time. In the example, we assume that pointer `p` may point to either `x` or `y`, which creates an ambiguous assignment (binding-time annotated aliases appear in comments next to dereferenced pointers). Since the assignment is dynamic, both locations must become dynamic.

The action annotations only slightly differ from the binding-time annotations. Static constructs become evaluate constructs, and dynamic constructs become residualize constructs. The only exceptions to this are the static left-hand sides of dynamic assignments, which are annotated residualize, instructing the specializer to residualize the variable identifiers on the left-hand sides (instead of evaluating the variable and lifting the resulting value).

The subsequent specialization phase is guided by the action annotations. Evaluate constructs are evaluated and residualize constructs are residualized. Evaluate statements disappear completely. Evaluate expressions are evaluated, and the resulting value is lifted into the residual code. Residualize expressions and statements are residualized.

2.2 Context Sensitivity

Context sensitivity enables a function to be analyzed with respect to different states, or *contexts*, producing an annotated instance of the function for each context. Since annotated instances are separate, each one can exploit the static values of its specific context.

The second example shows a function `f()` which contains a sequence of calls to `g()`, as given in Fig. 2. Function `f()` is analyzed with an initial binding-time description specifying that global `d` is dynamic. The context of the first call consists of a static actual parameter, a static non-local variable `x`, and a dynamic non-local variable `y` (binding times of the non-local variables appear in comments). An instance of the function is then annotated with respect to this context. Notice that `x` becomes dynamic while analyzing the body of `g()`, which creates a different context for the second call to `g()`. Therefore, a second instance of the function is created and annotated with respect to this new context. The third call to `g()` has the same context as the second call, so a new instance is not created.

The corresponding actions are then produced and are used to specialize the program. In the residual program, each different instance of function `g()` produces a different residual function definition. Since the third call to `g()` had the same context as the second call, it also shares the same residual function definition.

2.3 Return Sensitivity

Return sensitivity allows a function to return a static value even though the function contains dynamic side-effects and is therefore residualized.

In the third example, shown in Fig. 3, the function is analyzed with an initial binding-time description specifying that global variable `d` is dynamic. Return sensitivity allows the static value returned by `g()` to be used at its call site, which in turn enables the multiplication to be considered static as well. At the function's definition, we indicate that the function contains dynamic side-effects by annotating the identifier `g` as dynamic and that it returns a static value by annotating its return type `int` as static. At the call site, the identifier is annotated as both static and dynamic.

The specializer exploits the static return value returned by `g()` to perform the multiplication, and residualizes the call in order to residualize its side-effects. Notice that the specialized definition of `g()` no longer returns a value.

Source code

```
int d;
void f(int x, int y, int *p)
{
  x = x + y;
  x = d;
  x = x + y;

  *p = d;
  x = x + y;
}
```

Action annotated code

```
int d;
void f(int x, int y, int *p)
{
  x = x + y;

  x = d;

  x = x + y;

  * p = d;     /* alias:  p → {x, y} */
  x = x + y;
}
```

Binding-time annotated code

```
int d;
void f(int x, int y, int *p)
{
  x = x + y;

  x = d;

  x = x + y;

  * p = d;     /* alias:  p → { x, y } */
  x = x + y;
}
```

Specialized code (w.r.t. x = 2, y = 3)

```
int d;
void f(int x, int y, int *p)
{
  x = d;
  x = x + 3;

  *p = d;
  x = x + y;
}
```

Figure 1: Flow sensitivity

3 The Binding-Time Analysis

We shall make the above-mentioned ideas precise by describing our binding-time analysis using a data-flow analysis framework (see, for instance, [1, 20]) on the subset of C described in Fig. 4. For the sake of conciseness, this subset contains only a limited number of expressions and statements; further details on the intra-procedural aspects of the analysis can be found in [18]. Note also that non-void function calls are assumed to assign their return value directly to an identifier, which can then be used in subsequent calculations. This strategy simplifies the analysis without restricting its applicability. We assume that all programs are transformed prior to the analysis, if needed, so that they conform with this constraint. Also, the analysis presented is further simplified by the fact that it does not handle recursive functions.

3.1 Intra-procedural aspects

Locations and States We refer to the sets of values propagated by the analysis as *states*. States are elements of $Location \rightarrow Bt$, where Bt is the lattice $U \sqsubset S \sqsubset D$ with least upper bound operator $\sqcup$. U stands for undefined, S for static, and D for dynamic. In the intra-procedural case and in the absence of structures, $Location = Identifier$, provided all identifiers have been renamed in order to be unique. That is, each actual memory location associated to a given variable identifier is modeled by a single abstract location denoted by the identifier.

The binary operator $\setminus$ of type $State \times Locations \rightarrow State$ resets a set of locations to the bottom element U.

In the following, we shall use a graph representation of states. The application of a state is modeled by a lookup function which takes a graph (a set of pairs location/binding time) and a location, and returns the corresponding binding time. All the locations do not need to occur in the graph. A location which does not occur in the graph is considered to be undefined (the lookup function returns U).

Pre-processing We assume that, prior to binding-time analysis, an alias analysis and a definition analysis have been executed. The alias analysis gives, for each dereference expression $*^e exp$ at program point e, the set $aliases(e)$ of corresponding aliases, *i.e.* a set of locations. The definition analysis computes, for each statement at program point s, the set of locations $defs(s)$ which *may* be defined (through an assignment) within the statement. The function *unambiguous-defs*() additionally computes, for each assignment, the set of locations unambiguously defined by the assignment. If there is a single location associated to the

Source code

```
int x, y, d;
void f()
{
  x = 1;
  y = d;
  g(5);
  g(5);
  g(5);
}

g(int z)
{
  x = (x + z) + y;
}
```

Action annotated code

```
int x̄, y, d;
void f()
{
  x̄ = 1;
  y = d;
  g(5̄); /* x̄, y */
  g(5̄); /* x, y */
  g(5̄); /* x, y */
}

void g(int z̄) /* x̄, y */
{
    x̄ = (x + z)‾ + y;
}
void g(int z̄) /* x, y */
{
    x = (x + z̄) + y;
}
```

Binding-time annotated code

```
int x, y, d;
void f()
{
  x̄ = 1;
  ȳ = d;
  g(5̄); /* x̄, y */
  g(5̄); /* x, y */
  g(5̄); /* x, y */
}

void g(int z̄) /* x̄,y */
{
    x̄ = (x + z)‾ + y;
}
void g(int z̄) /* x,y */
{
  x̄ = (x + z̄) + y;
}
```

Specialized code

```
int y, d;
void f()
{
  y = d;
  g1();
  g2();
  g2();
}

void g1()
{
  x = 6 + y;
}
void g2()
{
x = (x + 5) + y;
}
```

Figure 2: Context sensitivity

Source code

```
int x, d;
void f()
{
  x = (g(1) * 2) + d;
}

int g(int z)
{
  x = z + d;
  return (z + 3);
}
```

Action annotated code

```
int x, d;
void f()
{
  x = (g(1̄) * 2̄) + d;
}

‾int‾ g(int z̄)
{
  x = z̄ + d;
  ‾return (z̄ + 3̄);‾
}
```

Binding-time annotated code

```
int x, d;
void f()
{
  x̄ = (g(1̄) * 2̄) + d ;
}

‾int‾ g(int z̄)
{
  x̄ = z̄ + d;
  ‾return‾ (z̄ + 3̄);
}
```

Specialized code

```
int x, d;
void f()
{
  g();
  x = 8 + d;
}

void g()
{
  x = 1 + d;
}
```

Figure 3: Return sensitivity

Domains:

$const \in Integer$
$id \in Identifier$
$bop \in BinaryOperator$

Abstract syntax:

exp	::= *const*	constant
	| *id*	variable
	| **&** lexp	reference
	| * exp	dereference
	| exp *bop* exp	binary expression
lexp	::= *id*	variable
	| * exp	dereference
stmt	::= lexp = exp	assignment
	| **if** (exp) stmt **else** stmt	conditional statement
	| **do** stmt **while** (exp)	loop
	| { stmt* }	block
	| id (exp*)	void function call
	| id = id (exp*)	non-void function call
	| **return** exp	function return
	| **return**	void function return
type-spec	::= int | char | ...	base types
	| * type-spec	pointer type
decl	::= type-spec id	declaration
func-def	::= type-spec id (decl*) stmt	function definition
program	::= decl* func-def*	program

Figure 4: Syntax of C subset

left-hand side of the assignment, the assignment is unambiguous; it unambiguously defines the location. Otherwise, there are, because of aliasing, several locations associated to the left-hand side of the assignment, the assignment is ambiguous; the defined location cannot be determined statically. The set of locations unambiguously defined by the assignment is therefore empty. This information is necessary since the binding-time analysis is capable of detecting that a dynamic variable becomes static if it is assigned a static value, but only if the assignment is unambiguous.

The analysis Assuming a single function and a single return statement, the analysis propagates forward the initial state, which returns S or D for any input parameter declared static or dynamic respectively, and U for any other location. The join operator $\bigsqcup$ on binding-time states is defined as a pointwise application of the least upper bound operator $\sqcup$ on the *State* function space range.

The data-flow equations relating the state $in(s)$ at the entry point of a statement at program point s and the state $out(s)$ at the output of the same statement are given in Fig. 5 with the transfer functions given in Fig. 6.

The function $t_s()$ describes the evolution of the state caused by data dependencies of an assignment at program point s. Each location in the set of possible definitions is mapped to the assignment binding time, given by *stmt-bt*(s) (see Fig. 7). Note that the assignment binding time depends on the input state. If the assignment is ambiguous, a safe approximation has to be taken: the new binding time of each defined location is the least upper bound of its previous binding time and of the assignment binding time. If the assignment is unambiguous, the new binding time of the defined variable is the assignment binding time.

A second transfer function, $t_{e,s}()$, takes control dependencies into account to compute the state at join points. If the specializer does not duplicate continuations, as is the case for Tempo, join points exist at the end of conditional statements and loops. To compute the proper safe approximation of the state at a join point, the binding time of each location possibly defined within the conditional or loop statement is the least upper bound of its previous binding time and of the binding time of the conditional or loop test, given by *exp-bt*$(e, state)$ (see Fig. 7). If a test is dynamic, all the locations possibly defined in the scope of the test are considered dynamic. In case of a static test, the join operation has no effect; the transfer function is the identity function.

For instance, let us consider the case of a variable which

$\text{lexp}^{e_1} =^{s} \text{exp}^{e_2}$:	$out(s) = t_s(in(s))$
$\texttt{if}^{s}$ (exp^{e}) $\text{stmt}_1^{s_1}$ $\texttt{else}$ $\text{stmt}_2^{s_2}$:	$in(s_1) = in(s)$ $in(s_2) = in(s)$ $out(s) = t_{e,s_1}(out(s_1)) \sqcup t_{e,s_2}(out(s_2))$
$\texttt{do}^{s}$ stmt^{s_0} $\texttt{while}$ (exp^{e}):	$in(s_0) = in(s) \sqcup t_{e,s_0}(out(s_0))$ $out(s) = out(s_0)$
{ $\text{stmt}_1^{s_1} \ldots \text{stmt}_n^{s_n}$ }^{s}:	$in(s_1) = in(s)$ $in(s_{i+1}) = out(s_i), 1 \leq i < n$ $out(s) = out(s_n)$
$\texttt{return}^{s}$ exp^{e}:	$out(s) = in(s)$

Figure 5: Intra-procedural binding-time analysis — data-flow equations for statements

$$t_s(state) = \{(loc, \textit{stmt-bt}(s)) \mid loc \in \textit{defs}(s)\} \sqcup (state \setminus \textit{unambiguous-defs}(s))$$

$$t_{e,s}(state) = \{(loc, \textit{exp-bt}(e, state)) \mid loc \in \textit{defs}(s)\} \sqcup state$$

Figure 6: Intra-procedural binding-time analysis — the transfer functions

is assigned a static value in a branch of a conditional statement whose test is dynamic. At specialization time, the value of the variable after the join point will be unknown. At execution time, if the branch is taken, the variable will be assigned the static value; if not, it will keep the value it had before entering the conditional statement. Therefore, such variables need to be considered dynamic at the join point.

3.2 Inter-procedural aspects

In order to deal with multiple functions, new program points, noted f, are introduced for function definitions. Context sensitivity is obtained by duplicating function definitions and the corresponding data-flow equations according to the different calling contexts encountered in the program for a given function.

These calling contexts are given by the binding times of the call input, *i.e.* the binding times of the actual parameters as well as the binding times of the non-local locations that may be *used* by the function, taking into account other nested calls. In the same way, it is possible to define the return context of a call, which is given by the binding time of the returned value together with the binding times of the non-local locations that may be *defined* by the call, again taking into account any other nested call. Since the number of locations defined by a given program is finite, these contexts are finite.

We assume that the non-local locations used and defined by a function are computed in another pre-processing phase, similar to inter-procedural summary information [5]. Note that this phase must follow (or be combined with) alias analysis. When, as a right-hand side (left-hand side) expression, a pointer potentially points to several locations, all these locations must be considered used (defined). Also, the notion of use and definition actually relates to the analysis rather than to actual executions. In particular, in case of an ambiguous definition of a pointer dereference, all the locations (potentially) pointed by the pointer must also be considered used as their binding times are used to compute the binding times of the (potentially) defined locations.

In order to propagate the binding time of the return value from the function definition to the calling sites, a new type of location is introduced. Each non-void return statement is considered to set the binding time of a *return* location. This return location is considered to be unambiguously defined by the return statement.

Dealing with multiple function returns also led us to propagate *return states*. Let us, for instance, consider a conditional statement with return statements in both branches as well as paths that do not return. At the join point, only the non-returning paths should be joined and propagated to the next statement as the output state of the conditional statement. On the other hand, all the returning paths should be joined too, taking into account the possibility of `return` statements under dynamic control.

The corresponding data-flow equations are given in Fig. 8. The functions *used-non-locals*() and *def-non-locals*() return the used non-local and defined non-local locations, respectively. In function calls, the calling context ctx puts together the part of the state relevant to the call by computing the binding time of each actual parameter and associating it to each corresponding formal parameter, as well as computing the binding time of each non-local location (a simple state lookup). The call $\textit{formal-loc}(i, id)$ simply returns the location associated to the i^{th} formal parameter of function id. The output state of a call statement is obtained

statements:

$\text{lexp}^{e_1} =^{s} \text{exp}^{e_2}$:
$stmt\text{-}bt(s) = lexp\text{-}bt(e_1, in(s)) \sqcup exp\text{-}bt(e_2, in(s))$

$\texttt{if}^{s}$ (exp^{e}) $\text{stmt}_1^{s_1}$ $\texttt{else}$ $\text{stmt}_2^{s_2}$:
$stmt\text{-}bt(s) = exp\text{-}bt(e, in(s)) \sqcup stmt\text{-}bt(s_1) \sqcup stmt\text{-}bt(s_2)$

$\texttt{do}^{s}$ stmt^{s_0} $\texttt{while}$ (exp^{e}):
$stmt\text{-}bt(s) = exp\text{-}bt(e, out(s_0)) \sqcup stmt\text{-}bt(s_0)$

$\{ \text{stmt}_1^{s_1} \ldots \text{stmt}_n^{s_n} \}^{s}$:
$stmt\text{-}bt(s) = \sqcup_{1 \le i \le n} stmt\text{-}bt(s_i)$

$\texttt{return}^{s}$ exp^{e}:
$stmt\text{-}bt(s) = exp\text{-}bt(e, in(s))$

expressions:

right-hand side expression:

$const^{e}$:
$exp\text{-}bt(e, \_) = S$

id^{e}:
$exp\text{-}bt(e, state) = lookup(state, id)$

$\&^{e}$ lexp^{e_0}:
$exp\text{-}bt(e, state) = lexp\text{-}bt(e_0, state)$

$*^{e}$ exp^{e_0}:
$exp\text{-}bt(e, state) = exp\text{-}bt(e_0, state) \sqcup (\sqcup_{loc \in aliases(e)} lookup(state, loc))$

$\text{exp}_1^{e_1}$ bop^{e} $\text{exp}_2^{e_2}$:
$exp\text{-}bt(e, state) = exp\text{-}bt(e_1, state) \sqcup exp\text{-}bt(e_2, state)$

left-hand side expression:

id^{e}:
$lexp\text{-}bt(e, state) = S$

$*^{e}$ exp^{e_0}:
$lexp\text{-}bt(e, state) = exp\text{-}bt(e_0, state)$

Figure 7: Intra-procedural binding-time analysis — binding-time annotations

by updating the binding times of the defined non-local locations of the callee. In case of a non-void function call, the binding time of the left-hand side location is additionally set to the binding time of the return location.

The output state of a return statement s is the state $\{\}$, which associates to any location the binding time U. It is the return state *ret-out(s)* which is propagated to the exit point of the function. These return states are propagated along through the *ret-in(s)* and *ret-out(s)* input and output return states. The corresponding equations, very similar to the ones relating *in(s)* and *out(s)*, are omitted. For the assignment statement, *ret-out(s)* is equal to *ret-in(s)*. For conditional statements, loops, and blocks, *ret-in(s)* and *ret-out(s)* are related by exactly the same equations as *in(s)* and *out(s)* (see Fig. 5). In particular, the transfer function $t_{e,s}()$ deals with return locations under conditional control.

A call statement can then be annotated with two binding times: the binding time of the callee return location and the least upper bound of the binding times of the callee defined non-local locations, summarizing the side-effects of the call. Depending on these annotations, the statement will be evaluated away (both binding times are static), rebuilt (the return is dynamic), or reduced (the return is static but there are dynamic side-effects). In the latter case, the non-void function call is residualized into a void function call and the corresponding function definition is residualized into a function returning void.

4 Applications

The Tempo partial evaluator is being used to specialize a wide variety of existing, complex, and real-world applications. In this section we summarize the applications which have already been specialized by Tempo. As well, we give a couple of examples taken from of these applications which show how key features of Tempo's binding-time analysis described in this paper enable static data to be exploited.

Specializing systems code has been the main target for which Tempo and its analyses have been designed. Previous work has shown that specializing operating system components with respect to system states that are likely to occur can produce significant speedups [28]. To validate this assertion, Tempo has been used to specialize the Sun Remote Procedure Call (RPC) [24, 25]. As is common for system components, Sun RPC is generic and structured in layers. Therefore, once a given remote procedure call is fixed, the interpretive overhead can be eliminated. Both the client and server functions were specialized, each of which consist of roughly 1000 lines of C code, and speedups of up to 3.75 were achieved.

Specialization is also used in various approaches to design *application generators*, programs which automatically

$id^s\,(exp_1^{e_1} \ldots exp_n^{e_n})$:
$ctx = \{(\textit{formal-loc}(i, id), \textit{exp-bt}(e_i, in(s))) \mid 1 \le i \le n\} \cup \{(loc, \textit{lookup}(in(s), loc)) \mid loc \in \textit{used-non-locals}(id)\}$
$in(f_{id,ctx}) = ctx$
$out(s) = (in(s) \setminus \textit{def-non-locals}(id)) \sqcup out(f_{id,ctx})$
$\textit{ret-out}(s) = \textit{ret-in}(s)$

$id_1 =^s id_2\,(exp_1^{e_1} \ldots exp_n^{e_n})$:
$ctx = \{(\textit{formal-loc}(i, id_2), \textit{exp-bt}(e_i, in(s))) \mid 1 \le i \le n\} \cup \{(loc, \textit{lookup}(in(s), loc)) \mid loc \in \textit{used-non-locals}(id_2)\}$
$in(f_{id_2,ctx}) = ctx$
$out(s) = (in(s) \setminus (\textit{def-non-locals}(id_2) \cup \{id_1\}))$
$\qquad \sqcup\, out(f_{id_2,ctx}) \sqcup \{(id_1, \textit{lookup}(out(f_{id_2,ctx}), RETURN(f_{id_2,ctx})))\}$
$\textit{ret-out}(s) = \textit{ret-in}(s)$

return^s:
$out(s) = \{\}$
$\textit{ret-out}(s) = in(s) \sqcup \textit{ret-in}(s)$

return^s exp^e:
$out(s) = \{\}$
$\textit{ret-out}(s) = \textit{ret-in}(s) \sqcup in(s) \sqcup \{(RETURN(f), \textit{exp-bt}(e, in(s)))\}$

$id^f\,(\textit{formals})\ \textit{body}^s$:
$\textit{ret-in}(s) = \{\}$
$in(s) = in(f)$
$out(f) = \{(loc, \textit{lookup}(\textit{ret-out}(s), loc)) \mid loc \in \textit{def-non-locals}(f)\}$

Figure 8: Inter-procedural binding-time analysis - data-flow equations

translate specifications into applications [6, 7, 30]. Tempo plays a key role in the approach presented in [30], in which a specific application is generated by combining and instantiating generic components. This approach involves defining an abstract machine and a micro-language interpreter, both of which contain interpretation overhead which is eliminated by partial evaluation. Currently, this framework is being applied to automatically generate device drivers for video cards, such as SVGA drivers for the XFree86 X11 server. In this study, the abstract machine implementation consists of about 1000 lines of code, while the interpretor is roughly 4000 lines.

As well, scientific algorithms and image processing functions have been specialized by Tempo [27]. Functions such as Fast Fourier Transform, cubic spline interpolation, and image dithering have been specialized, producing significant speedups. In addition to compile-time specialization, these functions were also specialized at run time, using Tempo's automatic, template-based run-time specializer [10]. Compared with operating systems or application generation programs, these functions are rather small—all consisting of less under 100 lines of code.

Let us now give a couple of examples of how two of the features presented in this paper, flow sensitivity and return sensitivity, were critical in effectively specializing these applications.

The first example illustrates a binding-time improvement which relies on a flow-sensitive analysis. Fig. 9 contains a program fragment where variable `x` is assigned a dynamic value, followed by a number of statements which use `x`. Since the assignment renders `x` dynamic, all of its subsequent uses are dynamic as well. If, however, it is known that there are certain values for `x` which are more common than others,

Original Source Code

```
x = a dynamic expression;
statements in which x is considered dynamic;
```

Transformed Code

```
x = a dynamic expression;
if (x == common_case_value) {
  x = common_case_value;
  statements in which x is considered static;
} else {
  statements in which x is considered dynamic;
}
```

Figure 9: Example of generalized partial computation which relies on flow-sensitive analysis

the program can be transformed in such a way to exploit this information. Specifically, a conditional is introduced to determine if x is in fact equal to some common value. If it is, then by explicitly adding an assignment in the truth branch of the conditional and copying the statements which use x into both branches, the statements in the truth branch can be specialized with respect to this common value for x. This example of *generalized partial computation* [15, 16] has proven useful both with the Sun RPC as well as with application generation. This binding-time improvement is possible because the binding-time analysis is flow sensitive.

The second example shows how return sensitivity is crucial to specialize the excerpt of the Sun RPC client code [21] shown in Fig. 10. The initial function `Xdr_bytes()` contains code which encodes data in the client buffer by making a call to `Xdr_u_int()` and checking the return value for a success or failure. By following this call interprocedurally, we finally arrive at the function `Xdrmem_putlong` which does the actual encoding. In addition to doing the encoding (performed by the assignment to `*(xdrs->x_private)`), this function also decrements the client buffer size `xdrs->x_handy`, increments the client buffer pointer `xdrs->x_private`, and returns an error value (0) if the buffer was empty (`xdrs->x_handy < 0`) and a success value (1) if not.

If the client buffer is considered to be known at compile time, the binding-time annotations in Fig. 10 are produced. As can be seen, the assignment which performs the encoding is considered dynamic, since the data to be encoded will only become known at run time. However, all of the other operations, namely those which depend on the client buffer, are considered static. For example, a buffer overflow can be statically computed, and the resulting return value can be propagated interprocedurally. In this example, all of the intermediate function calls can be eliminated during specialization and even the initial `if` statement can be reduced. This is due to the fact that return sensitivity allows static return values to be propagated interprocedurally, despite the fact that functions contain dynamic side-effects.

```
int Xdr_bytes(...)
{
  :
  if ((Xdr_u_int(xdrs, sizep) != 0) == 0)
    return 0;
  :
}

int Xdr_u_int(struct str1 *xdrs, unsigned int *up)
{
  :
  return Xdr_u_long(xdrs, up);
  :
}

int Xdr_u_long(struct str1 *xdrs, unsigned int *ulp)
{
  :
  if ((int)(xdrs->x_op) == 0)
    return Xdrmem_putlong(xdrs, (int *)ulp);

  if ((int)(xdrs->x_op) == 2)
    return 1;

  return 0;
  :
}

int Xdrmem_putlong(struct str1 *xdrs, int *lp)
{
  xdrs->x_handy = xdrs->x_handy - 4u;
  if (xdrs->x_handy < 0)
    return 0;

  *(xdrs->x_private) = htonl(*lp);

  xdrs->x_private = 4u + xdrs->x_private;
  return 1;
}
```

Figure 10: Return sensitivity for operating systems code

5 Related Work

There are a number of existing off-line partial evaluators for imperative languages [2, 3, 4, 19, 26] as well as for functional languages [8, 17, 19, 29].

All existing imperative binding-time analyses are *flow-insensitive*; that is, one single description of the binding-time state is maintained for an entire program. In this case, if a variable is dynamic anywhere in the program, its single description would be dynamic, and therefore the variable would be considered dynamic everywhere in the program. In this paper we have obtained flow sensitivity by writing an analysis which is flow sensitive; an alternative approach would be to use a flow-insensitive analysis on an intermediate program flow representation which explicitly encodes flow dependencies, such as Single Static Assignment (SSA) [13]. For example, a binding-time analysis has been described for a simple imperative language, which obtains flow-sensitivity by using a Program Representation Graph, a representation which contains some of the features of SSA [14]. The focus of this work is on providing formal semantics and proving safety conditions of binding-time analyses in order to establish a semantic foundation, and therefore implementation or application issues were not considered. It would be interesting to determine if this framework could be

adapted to handle real programs, for example, by treating a more realistic language containing pointers, data structures, or functions. The idea of flow sensitivity does not apply to functional languages since there is no notion of a state or updates.

Similarly, all existing imperative binding-time analyses are *context insensitive*. Contexts of all the calls to a function are approximated by a single, least precise, context. If a parameter or non-local variable is dynamic at any call site, it will be considered dynamic at every call site. On the other hand, there are a number of existing binding-time analyses for functional languages which are context sensitive, more commonly referred to as *polyvariant* [8, 17, 29]. However, a context-sensitive binding-time analysis for an imperative language is more complicated since contexts must include the binding-times of the non-local variables that are read by a function and the state must be updated with respect to non-local variables that are written. This is further complicated by the possibility of definitions being ambiguous due to aliasing.

Return sensitivity, which prevents the side-effect binding time of a function from interfering with its return binding-time, is a new concept which has not previously been explored. We discovered the need for return sensitivity when applying partial evaluation to a specific application domain, namely operating systems code. Return sensitivity is not applicable for functional languages since pure functions have a return value but do not contain side-effects.

A different approach for obtaining effective specialization of imperative programs has been proposed [22, 23]. Instead of directly treating an imperative program, the original source program is transformed into a functional representation. An existing partial evaluator for a functional language is then used to specialize the program, after which the residual program is transformed back into the original imperative language. The main advantage of this approach is that reusing an existing, mature partial evaluator avoids the need to design and implement a new partial evaluator. Initial results show that this approach may achieve a high degree of specialization; flow, context, and even return sensitivity have been demonstrated for small examples. More experimentation would be needed to determine if this approach could be scaled up to handle the size and complexity of existing, realistic programs.

6 Conclusion

We have designed and implemented a binding-time analysis for imperative programs which accurately handles the complexities found in existing, realistic software systems. We have described how this precision is obtained by presenting a binding-time analysis which is flow, context, and return sensitive. We have validated our approach by applying our partial evaluator to existing, realistic applications. Specifically, we have studied and identified opportunities for specialization in operating systems, application generation, scientific computations, and image processing, and have successfully specialized programs in these domains using a partial evaluator based on the binding-time analysis presented in this paper.

Acknowledgments

The authors would like to thank Julia Lawall, Barbara Moura, and Scott Thibault for their comments on drafts of this paper. This research is supported in part by France Telecom/SEPT, ARPA grant N00014-94-1-0845, and NSF grant CCR-9224375.

References

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

[2] L.O. Andersen. Self-applicable C program specialization. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 54–61, San Francisco, CA, USA, June 1992. Yale University, Hew Haven, CT, USA. Technical Report YALEU/DCS/RR-909.

[3] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[4] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132. Technical Report 94/9, University of Melbourne, Australia, 1994.

[5] J. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[6] D. Batory, S. Vivek, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca model of software-system generators. *IEEE Software*, 11(5):89–94, September 1994.

[7] J. Bell, F. Bellegarde, J. Hook, R. B. Kieburtz, A. Kotov, J. Lewis, L. McKinney, D. P. Oliva, T. Sheard, L. Tong, L. Walton, and T. Zhou. Software design for reliability and reuse: A proof-of-concept demonstration. In *Proceeding of TRI-Ada*, pages 396–404, 1994.

[8] C. Consel. Polyvariant binding-time analysis for applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 66–77. New York: ACM, 1993.

[9] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.

[10] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, January 1996. ACM Press.

[11] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. In *Partial Evaluation*

and Semantics-Based Program Manipulation, pages 44–46, Copenhagen, Denmark, June 1993. ACM Press. Invited paper.

[12] C. Consel, C. Pu, and J. Walpole. Making production OS kernel adaptive: Incremental specialization in practice. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, 1994.

[13] R. Cytron, Ferrante J., B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *ACM Transactions on Programming Languages and Systems*, pages 451–490, 1991.

[14] M. Das, T. Reps, and P. Van Hentenryck. Semantic foundations of binding-time analysis for imperative programs. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 100–110, La Jolla, CA, USA, 1995. ACM Press.

[15] Y. Futamura. Program evaluation and generalized partial computation. In *International Conference on Fifth Generation Computer Systems, Tokyo, Japan*, pages 1–8, 1988.

[16] Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. Amsterdam: North-Holland, 1988.

[17] F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788)*, pages 287–301. Berlin: Springer-Verlag, 1994.

[18] L. Hornof, J. Noyé, and C. Consel. Accurate partial evaluation of realistic programs via use sensitivity. Publication interne 1064, Irisa, Rennes, France, June 1996.

[19] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice-Hall, June 1993.

[20] T.J. Marlowe and B.G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28(2):121–163, December 1990.

[21] Sun Microsystem. NFS: Network file system protocol specification. RFC 1094, Sun Microsystem, March 1989.

[22] B. Moura. *Bridging the Gap between Functional and Imperative Languages.* PhD thesis, University of Rennes I, April 1997.

[23] B. Moura, C. Consel, and J. Lawall. Bridging the gap between functional and imperative languages. Rapport de recherche, Inria, Rennes, France, June 1996.

[24] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. Publication interne 1094, Irisa, Rennes, France, March 1997.

[25] G. Muller, E.N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing a commercial RPC protocol. In *ACM SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, June 1997. ACM Press. To appear.

[26] V. Nirkhe and W. Pugh. Partial evaluation and high-level imperative programming languages with applications in hard real-time systems. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 269–280, Albuquerque, New Mexico, USA, January 1992. ACM Press.

[27] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. Rapport de recherche 1065, Irisa, Rennes, France, November 1996.

[28] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, pages 314–324, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5),ACM Press.

[29] B. Rytz and M. Gengler. A polyvariant binding time analysis. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)*, pages 21–28. New Haven, CT: Yale University, 1992.

[30] S. Thibault and C. Consel. A framework of application generator design. Rapport de recherche RR-3005, Inria, Rennes, France, December 1996. To appear in ACM SIGSOFT Symposium on Software Reusability (SSR'97).

[31] E.N. Volanschi, G. Muller, C. Consel, L. Hornof, J. Noyé, and C. Pu. A uniform automatic approach to copy elimination in system extensions via program specialization. Rapport de recherche 2903, Inria, Rennes, France, June 1996.