Binding-Time Analysis for Both Static and Dynamic Expressions

Kenichi Asai

Department of Information Science, Faculty of Science, University of Tokyo*
"Information and Human Activity," PRESTO, JST

Abstract. This paper presents a specializer and a binding-time analyzer for a functional language where expressions are allowed to be used as both static and dynamic. With both static and dynamic expressions, we can statically access data structures while residualizing them at the same time. Previously, such data structures were treated as completely dynamic, which prevented us from accessing their components statically. The technique presented in this paper effectively allows us to lift data structures which was prohibited in the conventional partial evaluators. The binding-time analysis is formalized as a type system and the solution is obtained by solving constraints generated by the type system. We prove the correctness of the constraint solving algorithm and show that the algorithm runs efficiently in almost linear time.

1 Introduction

Given a function $f(x, y)$ and some of its argument $x = a$, program specialization (or partial evaluation) transforms f into a more efficient specialized program $f_a(y)$ by reducing expressions in f using the information on its known argument[14]. Because expressions that depend only on the known argument can be performed at specialization time, the execution of the resulting program $f_a(y)$ is usually (far) more efficient than the execution of the original function applied to the same argument: $f(a, y)$. The specialization technique is widely used not only for program optimization[8] but also for the construction of compilers and compiler generators from interpreters[10, 14]. It is also applied to runtime code generation[9, 17].

Specialization is different from normal execution in that it involves expressions whose value is unknown at specialization time. They are called *dynamic* expressions.[1] The expressions with known values are called *static*. Since static expressions can be safely executed at specialization time whereas dynamic expressions can not, a specializer needs additional facilities to distinguish them. There are mainly two approaches to cope with this: *online* and *offline*.

An online specializer[15] determines binding-times (i.e., if an expression is static or dynamic) *during* specialization. Since the decision is made according to

* 7-3-1 Hongo Bunkyo-ku 113-0033 Japan (`http://www.is.s.u-tokyo.ac.jp/~asai`)

[1] Dynamic expressions are said to be *residualized* in the specialized (or *residual*) program, so that they are executed later when unknown arguments are available.

A. Cortesi, G. Filé (Eds.): SAS'99, LNCS 1694, pp. 117–133, 1999.
© Springer-Verlag Berlin Heidelberg 1999

the actual values used for specialization, it leads to more accurate binding-times and hence better specialization.

An offline specializer[14], on the other hand, determines binding-times of expressions at a binding-time analysis (BTA) phase performed prior to specialization. Because the binding-time analysis does not use the actual values for the analysis, the result usually contains some kind of approximation. However, since all the binding-times are known at specialization time, specialization process itself becomes much simpler and faster. This enables us to practically self-apply the specializer to obtain compilers and compiler generators from interpreters.

In this paper, we demonstrate how to remove one of the existing limitation of offline specializers, i.e., limitation caused from a static data structure in a dynamic context. When a data structure (such as a cons cell and an abstraction) occurs in a dynamic context, i.e., when a data structure needs to be residualized, the conventional offline specializer makes the binding-time of this data structure completely dynamic, even if it is used statically somewhere else. For example, consider the following term P:

$$P = (\lambda x.\, cons\,(car\,x)\,x)@(cons\,1\,2) \quad .$$

When this term is specialized using *online* specializers, the result naturally becomes $cons\,1\,(cons\,1\,2)$. When we use *offline* specializers, however, the binding-time analysis classifies $x (= cons\,1\,2)$ dynamic because it is residualized in the final result. (Otherwise, the residual program would contain *specialization-time values* rather than program texts: $cons\,1\,(1\ .\ 2)$.) Consequently, $car\,x$ becomes dynamic, and the original term itself is returned as the result of specialization without performing any specialization.[2]

The problem can be avoided if we could *lift* data structures: we transform the specialization-time value $(1\ .\ 2)$ into its program text $cons\,1\,2$. In the conventional specializers, however, lifting is allowed only for constants and not for data structures because it leads to code duplication.[3] Furthermore, it is not obvious how to lift closures.

The problem here is that x is used as both static and dynamic. Even though it is used statically at $car\,x$, it becomes dynamic because it is residualized. The solution we take here is to carry *both* the residualized form and the static value for such data (after residualizing it in a let-expression). In the subsequent specialization, we take the dynamic part for residualization and the static part for reduction. This technique enables us to access statically the components of residualized data structures. It effectively allows us to lift data structures.

This flexibility does not come for free, however. Carrying both values spoils one of the good properties of offline specialization: static computation can be executed efficiently using standard evaluators. To minimize the overhead of carrying both values, we use data structures with both values only when necessary.

[2] The assignment of $cons\,1\,2$ into x is performed. However, a let-expression is inserted to avoid code duplication to produce **let** $x = cons\,1\,2$ **in** $cons\,(car\,x)\,x$.

[3] For example, specialization of $(\lambda x.\, cons\,(car\,x)\,(cons\,x\,x))@(cons\,1\,2)$ will duplicate $cons\,1\,2$ as in: $cons\,1\,(cons\,(cons\,1\,2)\,(cons\,1\,2))$.

To this end, we handle three kinds of binding-times: static, dynamic, and both. Static expressions are reduced efficiently as in the conventional specializers. Dynamic expressions are residualized in let-expressions so that they will never be duplicated. Expressions that are reduced statically but are also residualized will carry both static and dynamic values with some extra overhead.

In this paper, we present a monovariant binding-time analysis which identifies expressions that are actually used as both static and dynamic. It is formalized as a type system, and the result of binding-time analysis is obtained by solving constraints generated by the type system. The analysis is efficient: its time complexity is almost linear to the length of a program, achieving the same complexity as the conventional binding-time analysis.

A polyvariant binding-time analysis is a more general approach to give multiple binding-times to the same expression. However, it is known to be quite expensive[2]. Rather than employing expensive analyses, we stay in a monovariant framework to obtain more efficient almost linear time algorithm. Whether or not it is feasible to use a general polyvariant analysis in this situation is outside the scope of this paper.

The paper is organized as follows. After presenting our language in Sect. 2, we show a specializer for it in Sect. 3. Binding-time types are defined in Sect. 4 and typing rules for annotated terms are presented in Sect. 5. In Sect. 6, these typing rules are transformed into typing rules for unannotated terms, giving constraint generation rules. To solve the generated constraints, binding-times are first interpreted as a pair in Sect. 7. Two-phase constraint satisfaction algorithm is presented in Sect. 8, and its correctness is discussed in Sect. 9. Its time complexity is shown to be almost linear time in Sect. 10. Related work is discussed in Sect. 11 and the paper concludes in Sect. 12.

2 The Language

The language we consider is a λ-calculus with data structures (pairs), constants, and primitive operators (of which we include only arithmetic addition here). The syntax is defined as follows:

$$L = x \mid \lambda x.\, L \mid L@L \mid \mathit{cons}\, L\, L \mid \mathit{car}\, L \mid \mathit{cdr}\, L \mid c \mid L + L \ .$$

The language does not contain recursion explicitly, but the framework presented here can be easily extended to cope with recursion.

Given a term L in the above syntax, a binding-time analyzer produces an annotated term whose syntax is defined as follows:

$$\begin{aligned} M = {} & x \mid \lambda^S x.\, M \mid \lambda^D x.\, M \mid \lambda^B x.\, M \mid M@^S M \mid M@^D M \mid \mathit{cons}^S M\, M \\ & \mid \mathit{cons}^D M\, M \mid \mathit{cons}^B M\, M \mid \mathit{car}^S M \mid \mathit{car}^D M \mid \mathit{cdr}^S M \mid \mathit{cdr}^D M \\ & \mid c^S \mid c^D \mid c^B \mid M +^S M \mid M +^D M \mid M{\downarrow}_S^B \mid M{\downarrow}_D^B \ . \end{aligned}$$

$\lambda^S x.\, M$ and $\lambda^D x.\, M$ are standard static and dynamic abstractions, respectively. In addition, there is a third kind of abstraction $\lambda^B x.\, M$. $\lambda^B x.\, M$ is an abstraction

which is used as both static and dynamic. (B for 'both'.) It becomes static when coerced into a static abstraction using $\downarrow_S^B$. Likewise, it becomes dynamic when coerced into a dynamic abstraction using $\downarrow_D^B$. We call these two operators ($\downarrow_S^B$ and $\downarrow_D^B$) *coercion operators*. In contrast to abstraction, application is either static or dynamic. To apply $\lambda^B x.\, M$, it has to be coerced into a static or dynamic abstraction beforehand using $\downarrow_S^B$ or $\downarrow_D^B$.

Pairs and constants are also split into three kinds: static, dynamic, and both. The same coercion operators are used to take its static or dynamic part.

Note that there exists no coercion operator from S to D. This is the same as the conventional partial evaluator where coercion from a static data structure to a dynamic one is not permitted. As for constants, we will see in the next section that the standard *lift* operator can be used.

3 Specializer

Figure 1 shows a specializer for an annotated term M. In the figure, a variable with a superscript $^\diamond$ indicates that it is a fresh dynamic variable. Expressions with a superscript D (in the righthand side of the figure) are residualized expressions. Among them, $\mathbf{let}^D v^\diamond = A \ \mathbf{in}\ B$ is a residualized let-expression which is used to avoid code duplication. It is an abbreviation of $(\lambda^D v^\diamond.\, B)@^D A.\ \xi k.\, A$ and $\langle B \rangle$ are *shift* and *reset* operations[6]. Shift behaves like call/cc, but takes only the continuation delimited by reset. Shift and reset are used to grab continuation and to insert let-expressions. $\#_1$ and $\#_2$ are projection functions for a pair (A, B):

$$\#_1(A, B) = A, \quad \#_2(A, B) = B \ .$$

Given a term and an environment ρ, the specializer $\mathcal{S}$ reduces static parts and residualizes dynamic parts as usual. When the term is a variable, its value in the environment is returned. When the term is a static abstraction, it simply returns a static lambda closure.

When the term is a dynamic abstraction $\lambda^D x.\, M$, a let-expression is inserted to avoid code duplication/elimination. The rule is essentially the same as CPS-based specialization/let-insertion used in Similix[3], but is written in direct style using shift/reset operations. The rule can be read as follows: it first specializes the body M of the abstraction under the environment where the bound variable x is bound to a fresh dynamic variable $x^\diamond$. It is then given a name $v^\diamond$ and residualized in a let-expression. The result of this dynamic abstraction is this new name, which is then passed to the rest of the specialization (continuation) κ grabbed by the shift operation. Since the dynamic abstraction is residualized in a let-expression and not passed for further specialization, it will never be duplicated/eliminated regardless of how the dynamic name is used in the subsequent specialization. The specialization of the body of the abstraction and the application to the continuation are enclosed by reset operations so that let-expressions inserted within those specializations do not appear before the let-expression introduced here. This is necessary to maintain the scope of inserted let-expressions properly.

$$
\begin{aligned}
\mathcal{S}\,[\![x]\!]\,\rho &= \rho(x)\\
\mathcal{S}\,[\![\lambda^S x.\,M]\!]\,\rho &= \lambda x'.\mathcal{S}\,[\![M]\!]\,\rho[x'/x]\\
\mathcal{S}\,[\![\lambda^D x.\,M]\!]\,\rho &= \xi\kappa.\,\mathbf{let}^D v^\diamond = \lambda^D x^\diamond.\langle \mathcal{S}\,[\![M]\!]\,\rho[x^\diamond/x]\rangle\ \mathbf{in}\ \langle \kappa v^\diamond\rangle\\
\mathcal{S}\,[\![\lambda^B x.\,M]\!]\,\rho &= \xi\kappa.\,\mathbf{let}^D v^\diamond = \lambda^D x^\diamond.\langle \mathcal{D}(\mathcal{S}\,[\![M]\!]\,\rho[x^\diamond/x])\rangle\ \mathbf{in}\ \langle \kappa(v^\diamond, \lambda x'.\mathcal{S}\,[\![M]\!]\,\rho[x'/x])\rangle\\
\mathcal{S}\,[\![M_1 @^S M_2]\!]\,\rho &= (\mathcal{S}\,[\![M_1]\!]\,\rho)(\mathcal{S}\,[\![M_2]\!]\,\rho)\\
\mathcal{S}\,[\![M_1 @^D M_2]\!]\,\rho &= \xi\kappa.\,\mathbf{let}^D v^\diamond = (\mathcal{S}\,[\![M_1]\!]\,\rho)@^D(\mathcal{S}\,[\![M_2]\!]\,\rho)\ \mathbf{in}\ \langle \kappa v^\diamond\rangle\\
\mathcal{S}\,[\![cons^S M_1\ M_2]\!]\,\rho &= (\mathcal{S}\,[\![M_1]\!]\,\rho, \mathcal{S}\,[\![M_2]\!]\,\rho)\\
\mathcal{S}\,[\![cons^D M_1\ M_2]\!]\,\rho &= \xi\kappa.\,\mathbf{let}^D v^\diamond = cons^D(\mathcal{S}\,[\![M_1]\!]\,\rho)\,(\mathcal{S}\,[\![M_2]\!]\,\rho)\ \mathbf{in}\ \langle \kappa v^\diamond\rangle\\
\mathcal{S}\,[\![cons^B M_1\ M_2]\!]\,\rho &= \xi\kappa.\,\mathbf{let}\ (m_1, m_2) = (\mathcal{S}\,[\![M_1]\!]\,\rho, \mathcal{S}\,[\![M_2]\!]\,\rho)\\
&\qquad \mathbf{in}\ \mathbf{let}^D v^\diamond = cons^D \mathcal{D}(m_1)\,\mathcal{D}(m_1)\ \mathbf{in}\ \langle \kappa(v^\diamond, (m_1, m_2))\rangle\\
\mathcal{S}\,[\![car^S M]\!]\,\rho &= \#_1(\mathcal{S}\,[\![M]\!]\,\rho)\\
\mathcal{S}\,[\![car^D M]\!]\,\rho &= \xi\kappa.\,\mathbf{let}^D v^\diamond = car^D(\mathcal{S}\,[\![M]\!]\,\rho)\ \mathbf{in}\ \langle \kappa v^\diamond\rangle\\
\mathcal{S}\,[\![c^S]\!]\,\rho &= c\\
\mathcal{S}\,[\![c^D]\!]\,\rho &= lift(c)\\
\mathcal{S}\,[\![c^B]\!]\,\rho &= (lift(c), c)\\
\mathcal{S}\,[\![M_1 +^S M_2]\!]\,\rho &= (\mathcal{S}\,[\![M_1]\!]\,\rho) + (\mathcal{S}\,[\![M_2]\!]\,\rho)\\
\mathcal{S}\,[\![M_1 +^D M_2]\!]\,\rho &= \xi\kappa.\,\mathbf{let}^D v^\diamond = (\mathcal{S}\,[\![M_1]\!]\,\rho) +^D (\mathcal{S}\,[\![M_2]\!]\,\rho)\ \mathbf{in}\ \langle \kappa v^\diamond\rangle\\
\mathcal{S}\,[\![M\downarrow^B_D]\!]\,\rho &= \#_1(\mathcal{S}\,[\![M]\!]\,\rho)\\
\mathcal{S}\,[\![M\downarrow^B_S]\!]\,\rho &= \#_2(\mathcal{S}\,[\![M]\!]\,\rho)
\end{aligned}
$$

$$
\mathcal{D}(M) = \begin{cases} \#_1(M) & \text{if } M \text{ is both} \\ M & \text{if } M \text{ is dynamic} \end{cases}
$$

Fig. 1. Specializer for annotated terms written in direct style

The rule for $\lambda^B x.\,M$ is similar to the rule for $\lambda^D x.\,M$, except that it returns *both* the dynamic variable and the static value instead of returning only a dynamic variable. This enables us to use it in both static and dynamic ways. To use it as a static (dynamic) abstraction, we coerce it using $\downarrow^B_S$ ($\downarrow^B_D$), which is defined to take the appropriate component of the both values. Since $\lambda^B x.\,M$ is also residualized in a let-expression, it will never be duplicated. Note that $\lambda^B x.\,M$ requires overhead of carrying both static and dynamic values as a pair compared to a static or dynamic abstraction.

$\mathcal{D}(M)$ appearing in the rule for $\lambda^B x.\,M$ is used to residualize the body M of $\lambda^B x.\,M$ which can be either B or D.[4] Since residualized expressions must be completely dynamic, $\mathcal{D}(M)$ is used to take the dynamic part of M when M is both static and dynamic. The use of $\mathcal{D}(\cdot)$ does not imply that the binding-time of M needs to be checked at specialization time. Since it is available after the binding-time analysis, the rule for $\lambda^B x.\,M$ can be split into two rules according to the binding-time of M (B or D). This is the same for $cons^B M_1\,M_2$ where four cases (M_1 and M_2 are either B or D) are merged into one rule using $\mathcal{D}(\cdot)$.

Reconfirm here that $\lambda^B x.\,M$ is always residualized in a let-expression. This incurs a restriction to our binding-time analysis: "The argument of an ab-

[4] It cannot be S since it is residualized.

straction of type B has to be dynamic." To understand its consequence, consider the following example: $(\lambda f.\, \mathit{cons}\, f\, (f@2))@(\lambda x.\, x + 3)$. Since $\lambda x.\, x + 3$ is both statically applied and residualized, it is classified as both static and dynamic. We expect to perform the addition $2 + 3$ at specialization time to obtain $\mathit{cons}\, (\lambda x.\, x + 3)\, 5$. Because of the restriction, however, x becomes dynamic (since $\lambda x.\, x + 3$ is both) and so does the addition. Thus, the result we obtain is $\mathit{cons}\, (\lambda x.\, x + 3)\, (2 + 3)$. Because the value of x is unknown when $\lambda x.\, x + 3$ is residualized, x can only be dynamic. Readers might think that we could make x as both static and dynamic. This is not the case because x is *either* static or dynamic, not *both* static and dynamic. When x is residualized as an argument of $\lambda x.\, x + 3$, it is completely dynamic and has no static information. An expression can be both static and dynamic only when it has always both static and dynamic information at the same time.

This restriction comes from our design choice that the binding-time analysis we consider is monovariant.[5] If we used a polyvariant binding-time analysis, we could have assigned to x dynamic only when $\lambda x.\, x + 3$ is residualized and static when $\lambda x.\, x + 3$ is applied statically. However, this requires us to analyze the body of lambda expressions more than once. Rather than going into expensive polyvariant analyses, we stay in a monovariant framework to obtain efficiency. Further investigation is needed to see if the performance penalty of using polyvariant analyses for this particular case is actually serious. Note that even with this restriction, our binding-time analysis produces better results than the conventional monovariant binding-time analyses because lambda expressions of type B were previously treated as completely dynamic.

Let us come back to Fig. 1. The rule for a static application is standard. It specializes the subterms in a compositional way and applies the result in the underlying λ-calculus. A dynamic application is processed in a similar way but the application is residualized in a let-expression. The rules for *cons*, *car*, and a primitive application are similar.

Finally, the rules for constants return a static constant, a dynamic constant, or both. To make a dynamic constant, a standard *lift* operator is used. Here, the type of constants is strictly distinguished to treat constants and data structures in a uniform way. This does not mean that we need to carry both static and dynamic constants for constants of type B. We can handle constants as the same way as the conventional partial evaluators by interpreting:

$$\begin{array}{ll} \mathcal{S}\,[\![c^S]\!]\,\rho = c & \mathcal{S}\,[\![c{\downarrow}^B_D]\!]\,\rho = \mathit{lift}(c) \\ \mathcal{S}\,[\![c^D]\!]\,\rho = \mathit{lift}(c) & \mathcal{S}\,[\![c{\downarrow}^B_S]\!]\,\rho = c \\ \mathcal{S}\,[\![c^B]\!]\,\rho = c & \end{array}$$

and changing the definition of $\mathcal{D}(\cdot)$ accordingly.

[5] A binding-time analysis is called monovariant when each subexpression to be analyzed is allowed to have only one binding-time. When multiple binding-times are allowed, it is called polyvariant.

To specialize a given annotated term M, we execute $\mathcal{S}$ with M and an empty environment ρ_ϕ in the global context: $\langle \mathcal{S}\ [\![M]\!]\ \rho_\phi \rangle$. Here, ρ_ϕ is an environment which raises an error for all variables.

The specializer shown in Fig. 1 avoids code duplication/elimination by inserting let-expressions for residualization of dynamic expressions. This is important for preserving the identity (pointer equality) of a data structure. Although let-insertion has been used for this purpose in the conventional partial evaluators, data structures became completely dynamic once they were residualized in let-expressions. The use of both values enables us to use data structures freely without worrying about if they are residualized or not.

Readers might think that the introduction of both values would break the identity of data structures since they are duplicated for a static version and a dynamic version. Fortunately, this is not the case because static computation and dynamic computation do not interfere. For static computation, a static value itself with its own identity is used. For dynamic computation, a unique name attached to a data structure serves as its identity. Because the static identity does not remain until the dynamic computation phase,[6] the identity of a data structure is preserved by regarding these static and dynamic identities as the identity of the data structure.

4 Binding-Time Types

The binding-time analysis presented in this paper is formalized as a type system. Binding-time values π and binding-time types τ are defined as follows:

$$\pi = S \mid D \mid B \qquad \tau = \pi \mid \tau \to^\pi \tau \mid \tau \times^\pi \tau \mid \alpha \mid \mu\alpha.\tau \ .$$

Binding-time values are either S (static), D (dynamic), or B (both static and dynamic). They are also used as binding-time types for constants. $\tau \to^\pi \tau$ and $\tau \times^\pi \tau$ are a function type and a pair type, respectively, whose binding-time value is π. α is a type variable and $\mu\alpha.\tau$ is a recursive type. They are used to type terms in untyped languages such as Scheme.[7]

Among binding-time types defined above, only the types derivable from the following rules are well-formed. Here, $\Delta \vdash \tau : \pi$ represents that a binding-time type τ has a binding-time value π under an assumption Δ.

$$\Delta \vdash \pi : \pi \qquad \Delta, \alpha : \pi \vdash \alpha : \pi \qquad \frac{\Delta, \alpha : \pi \vdash \tau : \pi}{\Delta \vdash \mu\alpha.\tau : \pi}$$

$$\frac{\Delta \vdash \tau_1 : \pi_1 \quad \Delta \vdash \tau_2 : \pi_2 \quad \left\{ \begin{array}{l} \pi \triangleright' \pi_1, \\ \pi \triangleright \pi_2 \end{array} \right\}}{\Delta \vdash \tau_1 \to^\pi \tau_2 : \pi} \qquad \frac{\Delta \vdash \tau_1 : \pi_1 \quad \Delta \vdash \tau_2 : \pi_2 \quad \left\{ \begin{array}{l} \pi \triangleright \pi_1, \\ \pi \triangleright \pi_2 \end{array} \right\}}{\Delta \vdash \tau_1 \times^\pi \tau_2 : \pi}$$

Rules for functions and pairs have additional requirements on their component types: $\pi_1 \triangleright \pi_2$ and $\pi_1 \triangleright' \pi_2$. They are similar to the requirements on function and

[6] If it does, it is not static but dynamic.

[7] For example, f in $\lambda f.\, f@f$ has a type $\mu\alpha.\, \alpha \to^\pi \beta$.

pair types found in the conventional type-based binding-time analyses, e.g., "if a pair is dynamic, its components have to be dynamic." Here, they are extended to cope with both static and dynamic case:

$$\begin{array}{llllll} S \triangleright S & S \triangleright D & S \triangleright B & D \triangleright D & B \triangleright D & B \triangleright B \\ S \triangleright' S & S \triangleright' D & S \triangleright' B & D \triangleright' D & B \triangleright' D & \end{array}$$

If π_1 is static, no restriction on π_2. If π_1 is dynamic, π_2 has to be dynamic, too. These are standard requirements on well-formed types. If π_1 is both static and dynamic, there are two cases: the one which allows π_2 to be both static and dynamic ($\triangleright$) and the one which does not ($\triangleright'$). The latter is used for the argument of abstractions of type B, since in that case, B is not permitted as described in the previous section.

When $\vdash \tau : \pi$ is derivable, we say that a well-formed type τ has a binding-time value π. We denote τ^π to separate a (toplevel) binding-time value π from its type. We assume the following equivalence on types:

$$D = D \to^D D = D \times^D D, \quad \mu\alpha.\,\tau = \tau[\mu\alpha.\,\tau/\alpha] \ .$$

5 Typing Rules

Figure 2 shows typing rules for annotated terms. A judgement $\Gamma \vdash M : \tau$ represents that M has a binding-time type τ under a type assumption Γ. These rules are fairly standard and do not need particular explanation. We note only two points. First, the rules produce only well-formed types. (Notice that they do not produce a recursive type.) Second, constants and the result of static primitive applications are allowed to have arbitrary binding-time values. This reflects the fact that static constants can be coerced into dynamic constants using the *lift* operator.

As an example of type checking, consider the term P presented in Sect. 1. If we correctly annotate P as follows:

$$(\lambda^S x.\, cons^D (car^S x \downarrow_S^B) \, x \downarrow_D^B) @ (cons^B 1\, 2)$$

then, its derivation becomes as follows where Γ represents $x : D \times^B D$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash x : D \times^B D}{\Gamma \vdash x \downarrow_S^B : D \times^S D}}{\Gamma \vdash car^S x \downarrow_S^B : D} \quad \dfrac{\Gamma \vdash x : D \times^B D}{\Gamma \vdash x \downarrow_D^B : D}}{\Gamma \vdash cons^D (car^S x \downarrow_S^B) \, x \downarrow_D^B : D}}{\vdash \lambda^S x.\, cons^D (car^S x \downarrow_S^B) \, x \downarrow_D^B : D \times^B D \to^S D} \quad \dfrac{\vdash 1 : D \quad \vdash 2 : D}{\vdash cons^B 1\, 2 : D \times^B D}$$
$$\vdash (\lambda^S x.\, cons^D (car^S x \downarrow_S^B) \, x \downarrow_D^B) @ (cons^B 1\, 2) : D$$

The typing rules shown in Fig. 2 are an extension of the conventional binding-time analyses. By substituting all B with D, we find that the rules are standard

$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda^S x.M : \tau_1 \to^S \tau_2} \qquad \frac{\Gamma, x : D \vdash M : D}{\Gamma \vdash \lambda^D x.M : D} \qquad \frac{\Gamma, x : D \vdash M : \tau \quad \tau \in \{B, D\}}{\Gamma \vdash \lambda^B x.M : D \to^B \tau}$$

$$\Gamma, x : \tau \vdash x : \tau \qquad \frac{\Gamma \vdash M_1 : \tau_2 \to^S \tau_1 \quad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash M_1 @^S M_2 : \tau_1} \qquad \frac{\Gamma \vdash M_1 : D \quad \Gamma \vdash M_2 : D}{\Gamma \vdash M_1 @^D M_2 : D}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash cons^S M_1\, M_2 : \tau_1 \times^S \tau_2} \qquad \frac{\Gamma \vdash M_1 : D \quad \Gamma \vdash M_2 : D}{\Gamma \vdash cons^D M_1\, M_2 : D} \qquad \frac{\Gamma \vdash M : \tau_1 \times^S \tau_2}{\Gamma \vdash car^S M : \tau_1}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma \vdash M_2 : \tau_2 \quad \tau_i \in \{B, D\}}{\Gamma \vdash cons^B M_1\, M_2 : \tau_1 \times^B \tau_2} \qquad \frac{\Gamma \vdash M : D}{\Gamma \vdash car^D M : D}$$

$$\Gamma \vdash c : \pi \qquad \frac{\Gamma \vdash M_1 : S \quad \Gamma \vdash M_2 : S}{\Gamma \vdash M_1 +^S M_2 : \pi} \qquad \frac{\Gamma \vdash M_1 : D \quad \Gamma \vdash M_2 : D}{\Gamma \vdash M_1 +^D M_2 : D}$$

$$\frac{\Gamma \vdash M : B}{\Gamma \vdash M \downarrow_S^B : S} \qquad \frac{\Gamma \vdash M : D \to^B \tau \quad \tau \in \{B, D\}}{\Gamma \vdash M \downarrow_S^B : D \to^S \tau} \qquad \frac{\Gamma \vdash M : \tau_1 \times^B \tau_2 \quad \tau_i \in \{B, D\}}{\Gamma \vdash M \downarrow_S^B : \tau_1 \times^S \tau_2}$$

$$\frac{\Gamma \vdash M : B}{\Gamma \vdash M \downarrow_D^B : D} \qquad \frac{\Gamma \vdash M : D \to^B \tau \quad \tau \in \{B, D\}}{\Gamma \vdash M \downarrow_D^B : D} \qquad \frac{\Gamma \vdash M : \tau_1 \times^B \tau_2 \quad \tau_i \in \{B, D\}}{\Gamma \vdash M \downarrow_D^B : D}$$

Fig. 2. Typing rules for annotated terms

typing rules for two-level terms. The only exception is the coercion rule from B to S, which was previously impossible. This is where we can perform more specialization than before. By classifying previously dynamic expressions as both, more static computation becomes possible through extracting static parts of both values.

An annotated term p_{ann} is well-annotated if $\vdash p_{\text{ann}} : D$ is derivable. It is the best annotation if the following two properties hold where the first one has higher priority:[8]

1. p_{ann} classifies more subterms as S than any other well-annotated terms of p.
2. p_{ann} classifies less subterms as B than any other well-annotated terms of p.

The first property is a standard one stating that p_{ann} can perform static computation more than any other well-annotated terms of p. It also excludes the case where a static value is unnecessarily raised to both and is used only statically by extracting its static part. The second property says that if the same static computation is possible (by the first property), then B should be avoided as much as possible to minimize the overhead of carrying both values. Even though the second property is not fulfilled, the obtained specialized program is the same if the first one is fulfilled. The second property assures that the overhead at specialization time required for carrying both values is minimal. It also excludes the

[8] Here, the words 'more' and 'less' refer to set inclusion of subterms.

$$\Gamma, x : \tau^{\pi} \vdash x : \boxed{\tau^{\pi'}} \ \{\boxed{\pi \geq \pi'}\} \qquad \frac{\Gamma, x : \tau_1^{\pi_1} \vdash L : \tau_2^{\pi_2}}{\Gamma \vdash \lambda x.\, L : \tau_1^{\pi_1} \to^{\pi} \tau_2^{\pi_2}} \left\{ \begin{array}{l} \pi \triangleright' \pi_1, \\ \pi \triangleright \pi_2 \end{array} \right\}$$

$$\Gamma \vdash c : \pi \qquad \frac{\Gamma \vdash L_1 : \tau_2^{\pi_2} \to^{\pi} \tau_1^{\pi_1} \quad \Gamma \vdash L_2 : \tau_2^{\pi_2}}{\Gamma \vdash L_1 @ L_2 : \boxed{\tau_1^{\pi_1'}}} \left\{ \begin{array}{l} \boxed{\pi_1 \geq \pi_1'}, \ \pi \triangleright' \pi_2, \\ \pi \in \{S, D\}, \ \pi \triangleright \pi_1 \end{array} \right\}$$

$$\frac{\Gamma \vdash L_1 : \tau_1^{\pi_1} \quad \Gamma \vdash L_2 : \tau_2^{\pi_2}}{\Gamma \vdash \mathit{cons}\, L_1\, L_2 : \tau_1^{\pi_1} \times^{\pi} \tau_2^{\pi_2}} \left\{ \begin{array}{l} \pi \triangleright \pi_1, \\ \pi \triangleright \pi_2 \end{array} \right\} \qquad \frac{\Gamma \vdash L : \tau_1^{\pi_1} \times^{\pi} \tau_2^{\pi_2}}{\Gamma \vdash \mathit{car}\, L : \boxed{\tau_1^{\pi_1'}}} \left\{ \begin{array}{l} \boxed{\pi_1 \geq \pi_1'}, \ \pi \triangleright \pi_1, \\ \pi \in \{S, D\}, \ \pi \triangleright \pi_2 \end{array} \right\}$$

$$\frac{\Gamma \vdash L_1 : \pi_1 \quad \Gamma \vdash L_2 : \pi_2}{\Gamma \vdash L_1 + L_2 : \pi} \left\{ \begin{array}{l} \pi_1 \triangleright \pi, \ \pi_2 \triangleright \pi, \ \pi_1 \triangleright \pi_2, \ \pi_2 \triangleright \pi_1, \\ \pi_1 \in \{S, D\}, \ \pi_2 \in \{S, D\} \end{array} \right\} \qquad \frac{\Gamma \vdash L : \tau \quad \tau = \tau'}{\Gamma \vdash L : \tau'}$$

Fig. 3. Typing rules for unannotated terms (constraint generation rules)

case where a dynamic value is unnecessarily raised to both and is residualized by extracting its dynamic part.

Given an unannotated term p, a binding-time analysis returns a well-annotated term p_{ann} that has the best annotation in the above sense.

6 Constraint Generation

Since the typing rules presented in the previous section are for annotated terms, they are not directly applicable to type reconstruction of unannotated terms. To reconstruct types for unannotated terms, those rules have to be rewritten so that they are directed over the syntax of unannotated terms. This rewriting is done by (1) combining rules for the same construct but different binding-time values and (2) incorporating coercion rules into the combined rules. The resulting rules are shown in Fig. 3. They are directed over the syntax of unannotated terms. The boxed parts indicate the places where coercion rules are incorporated.

The rule for a constant is the same as before. The rule for a variable is merged with the coercion rule. Here, the constraint $\pi \geq \pi'$ shows coercibility from π to π' and is defined as follows:

$$\begin{array}{c} B > S, \quad B > D \\ \pi \geq \pi' \Longleftrightarrow (\pi > \pi') \vee (\pi = \pi') \ . \end{array}$$

Coercion is permitted only from B to S or D (not from S to D). If inequality $\pi > \pi'$ holds when a binding-time analysis finishes, an annotation $\downarrow^{B}_{\pi'}$ is attached to the term corresponding to π'.

The rule for abstraction is obtained by combining three abstraction rules in Fig. 2. Considering the equality $D = D \to^{D} D$, it is easy to see that those three rules correspond to the cases $\pi = S$, D, or B. The side condition on a function

type ensures its well-formedness. Observe that the rule for equality (the last rule of Fig. 3) is required to simulate the dynamic abstraction rule in Fig. 2. Coercion is not required here, because the rule for abstraction can produce the type *after* coercion. For example, if we made an abstraction of type B and immediately coerced it into S (as in $(\lambda^B x.M)\downarrow_S^B$), we could have created a static abstraction from the beginning instead (as in $\lambda^S x.M$).

The rule for application is obtained similarly by combining two application rules in Fig. 2. The constraint $\pi \in \{S, D\}$ shows that there exist only static and dynamic rules for application. The result of application may require coercion if L_1 returns a value of type B.

The rules for *cons* and *car* are similar to the rules for abstraction and application. The equality $D = D \times^D D$ is used to combine three rules for *cons*. Finally, the rule for primitive application says: "If any of the argument becomes dynamic, then the whole primitive application becomes dynamic. In that case, other arguments must also be dynamic." This summarizes the two rules for primitive application in Fig. 2.

Although the rules in Fig. 3 are directed over unannotated terms, types of each subterm are not obvious from these rules. To determine types of each subterm, a binding-time analysis first collects the type constraints generated by the rules in Fig. 3. Then, these constraints are solved to give types of each subterm. Once the constraints are solved and binding-time types are determined, it is easy to annotate a program because the rules in Fig. 3 correspond to the rules in Fig. 2.

As an example of constraint generation, consider the term P in Sect. 1. Let Γ be $x : \pi_1 \times^{\pi_3} \pi_2$. After unification of types, we obtain the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash x : \pi_1 \times^{\pi_3''} \pi_2}{\Gamma \vdash car\ x : \pi_1'} \quad \Gamma \vdash x : \pi_1 \times^{\pi_3'} \pi_2}{\Gamma \vdash cons\ (car\ x)\ x : \pi_1' \times^{\pi_4} (\pi_1 \times^{\pi_3'} \pi_2)}}{\vdash \lambda x.\ cons\ (car\ x)\ x : \pi_1 \times^{\pi_3} \pi_2 \to^{\pi_5} \pi_1' \times^{\pi_4} (\pi_1 \times^{\pi_3'} \pi_2)} \quad \dfrac{\vdash 1 : \pi_1 \quad \vdash 2 : \pi_2}{\vdash cons\ 1\ 2 : \pi_1 \times^{\pi_3} \pi_2}}{\vdash (\lambda x.\ cons\ (car\ x)\ x)@(cons\ 1\ 2) : \pi_1' \times^{\pi_4'} (\pi_1 \times^{\pi_3'} \pi_2)}$$

together with the following constraints:

$$\begin{array}{llllllll} \pi_3 \triangleright \pi_1, & \pi_3 \geq \pi_3'', & \pi_1 \geq \pi_1', & & \pi_3'' \triangleright \pi_1, & \pi_4 \triangleright \pi_1', & \pi_5 \triangleright' \pi_3, & \pi_4 \geq \pi_4', \\ \pi_3 \triangleright \pi_2, & \pi_3 \geq \pi_3', & \pi_3'' \in \{S, D\}, & & \pi_3'' \triangleright \pi_2, & \pi_4 \triangleright \pi_3', & \pi_5 \triangleright \pi_4, & \pi_5 \in \{S, D\}\ . \end{array}$$

The best solution to these constraints (with the initial constraint $\pi_4' = D$) is:[9]

$$\pi_3'' = \pi_5 = S, \quad \pi_1 = \pi_1' = \pi_2 = \pi_3' = \pi_4 = \pi_4' = D, \quad \pi_3 = B$$

which gives us the following expected result:

$$(\lambda^S x.\ cons^D (car^S x \downarrow_S^B)\ x \downarrow_D^B)@(cons^B 1\ 2)\ .$$

Notice that two coercion operators are inserted corresponding to the two inequalities $\pi_3 > \pi_3'$ and $\pi_3 > \pi_3''$ whose equality did not hold.

[9] See the following sections for constraint solving algorithm and the initial constraint.

7 Interpreting Types as Pairs

In the following sections, we describe constraint satisfaction algorithm. Rather than directly solving constraints obtained by the typing rules in Fig. 3, we first interpret a binding-time value π as a pair (s, d). Constraints over π are transformed into constraints over (s, d) accordingly, which are then solved by our constraint satisfaction algorithm. This is fairly a natural approach since B can be regarded as a set $\{S, D\}$.

The interpretation of the pair (s, d) is as follows. Assume that an expression L has a type $\pi = (s, d)$. s is a boolean value indicating if L is used statically. To be more concrete, s becomes true when (1) L is an abstraction and is applied statically, (2) L is a pair and is decomposed statically by *car* or *cdr*, or (3) L is a constant and is applied to a primitive statically. $s = \mathsf{false}$ indicates that L is not used statically. d is a boolean value indicating whether L is residualized or not. If it is, d becomes true. If not, false.

The following table summarizes the correspondence between π and (s, d). When s is true and d is false, it means that L is used but not residualized, i.e., static. When s is false and d is true, L is residualized but not used, i.e., dynamic. $s = d = \mathsf{true}$ means that L is both used and residualized, i.e., L is both. $s = d = \mathsf{false}$ means that L is neither used nor residualized.[10] In this case, we can treat L in an arbitrary way. Here, we denote it as $\bot$.

	$d = \mathsf{false}$	$d = \mathsf{true}$
$s = \mathsf{false}$	$\bot$	D
$s = \mathsf{true}$	S	B

Under this interpretation of binding-time values, the best annotation is rephrased as follows (where the first one has higher priority):

1. Less d is true.
2. Less s is true.

The first criteria says that an expression should be S as much as possible, while the second one says that unnecessary B should be avoided.

The constraints appearing in Fig. 3 can be transformed into constraints over (s, d) as follows. Assume that $\pi = (s, d)$, $\pi_i = (s_i, d_i)$, and $\Rightarrow$ is an implication having higher precedence than $\wedge$.

$$
\begin{aligned}
\pi_1 \triangleright \pi_2 &: (d_1 = \mathsf{true}) \Rightarrow (d_2 = \mathsf{true}) \wedge (s_1 = \mathsf{false}) \Rightarrow (s_2 = \mathsf{false}) \\
\pi_1 \triangleright' \pi_2 &: (d_1 = \mathsf{true}) \Rightarrow (d_2 = \mathsf{true}) \wedge (d_1 = \mathsf{true}) \Rightarrow (s_2 = \mathsf{false}) \\
\pi_1 \geq \pi_2 &: (d_2 = \mathsf{true}) \Rightarrow (d_1 = \mathsf{true}) \wedge (s_1 = \mathsf{false}) \Rightarrow (s_2 = \mathsf{false}) \\
\pi \in \{S, D\} &: (s = \mathsf{false}) \Rightarrow (d = \mathsf{true}) \wedge (d = \mathsf{true}) \Rightarrow (s = \mathsf{false})
\end{aligned}
$$

By inspecting all possible variations for s and d, it can be verified that each constraint is equivalent (except for $\bot$) to the original one. Intuitively, $\pi_1 \triangleright \pi_2$ is interpreted as "if π_1 is residualized, so is π_2"; $\pi_1 \triangleright' \pi_2$ as "if π_1 is residualized,

[10] For example, 2 in the expression *car* (*cons* 1 2).

π_2 must be dynamic"; $\pi_1 \geq \pi_2$ as "if π_2 is used statically, so is π_1, and if π_2 is residualized, so is π_1" (taking contraposition of the second constraint).

Note that for all the constraints $(A) \Rightarrow (B)$, A and B have always the form $s = \mathsf{false}$ or $d = \mathsf{true}$. This observation is important in designing constraint satisfaction algorithm.

8 Solving Constraints

Given an unannotated term L, the binding-time analysis first generates type constraints as described in the previous sections. Assume that the type of L itself is given by $\pi = (s, d)$. Then, the constraints are solved together with the initial constraint $d = \mathsf{true}$. It ensures that the result of partial evaluation becomes a completely dynamic expression.

Constraint solving is performed in two steps corresponding to the two requirements for the best annotation:

Step 1 Assign false to all d's (except for d used in the initial constraint) and true to all s's. Interpret a constraint $(A) \Rightarrow (B)$ as "if A holds, then change the value of appropriate s or d so that B holds." Repeat this operation until no changes are required.

Step 2 Assign false to all s's. For d, use the value of d at the end of step 1. Interpret all the constraints as their contraposition. That is, $(A) \Rightarrow (B)$ is interpreted as "if B does not hold, then change the value of appropriate s or d so that A does not hold." Repeat this operation until no changes are required.

Step 1 is mainly concerned with d. It starts from the assumption that every expression is static ($s = \mathsf{true}$ and $d = \mathsf{false}$). Then, expressions that have to be residualized ($d = \mathsf{true}$) are identified. Because the fixed-point iteration starts from $d = \mathsf{false}$ and d becomes true only when necessary, the result is the least fixed-point in the sense that the smallest number of d is true.

Step 1 considers s, too, because the value of s sometimes affects the value of d. This happens when the constraint of the form $\pi \in \{S, D\}$ appears (such as in the rule for application). When this constraint appears, π should be D only if it is impossible to assign S (since we want to make as many expressions static as possible). To achieve this, we first set $s = \mathsf{true}$. Only when s needs to be false[11] do we assign $d = \mathsf{true}$.

Notice that we have two contradictory requirements here. We want to assign S as much as possible. At the same time, if an expression is used only as dynamic, we do not want to make it both unnecessarily (because it incurs overhead). The former requirement says that s should be true as much as possible, whereas the latter says s should be false as much as possible. This is the reason why our algorithm has two steps. At step 1, the former requirement (which is more important) is fulfilled. The latter is satisfied at step 2 to the extent that the former is not violated.

[11] For example, an argument of an abstraction of type B cannot be static.

Step 2 is concerned with s.[12] It identifies expressions that are actually used as both static and dynamic among the expressions that are classified as B at step 1. To do this, all expressions that are classified as B (or D) are first assumed to be dynamic (i.e., $s = \mathsf{false}$). Then, expressions that are actually used statically are changed back to B ($s = \mathsf{true}$). Because the fixed-point iteration starts from $s = \mathsf{false}$ and s becomes true only when necessary, the result is the least fixed-point in the sense that the smallest number of s is true.

9 Correctness

This section describes the correctness of our algorithm.[13] We show that the fixed-point iteration changes the solution monotonically (in a certain lattice of finite height), that a solution exists, and hence that the algorithm terminates giving the least solution satisfying constraints. In the following, we assume that $\mathsf{true} > \mathsf{false}$.

Step 1 begins with $s = \mathsf{true}$ and $d = \mathsf{false}$.

Lemma 1. *Application of each constraint in step 1 does not decrease the solution in the following lattice:* $(s_1, d_1) \geq (s_2, d_2) \stackrel{\text{def}}{\Longleftrightarrow} (s_1 \leq s_2) \wedge (d_1 \geq d_2)$. *Furthermore, once a constraint is satisfied, it will never be violated in step 1.*

Since the solution where $s = \mathsf{false}$ and $d = \mathsf{true}$ (for all s's and d's) trivially satisfies all the constraints, there exists a solution. Hence, step 1 returns a least solution in which the smallest number of d's are true.

Step 2 begins with $s = \mathsf{true}$ and d with the value at the end of step 1. We first show that the value of d is the same throughout step 2.

Lemma 2. *The value of d is not modified during step 2.*

The above lemma says that we can forget about d and concentrate on s in step 2. This coincides with our intuition that step 2 changes unnecessary B to D.[14] In the next lemma, we see that step 2 has a similar property to step 1. Note that inequality on s in the definition of lattice is different from Lemma 1.

Lemma 3. *Application of each constraint in step 2 does not decrease the solution in the following lattice:* $(s_1, d_1) \geq (s_2, d_2) \stackrel{\text{def}}{\Longleftrightarrow} (s_1 \geq s_2) \wedge (d_1 = d_2)$. *Furthermore, once a constraint is satisfied, it will never be violated in step 2.*

Because the solution obtained at the end of step 1 satisfies all the constraints, there exists a solution. Hence, step 2 returns a least solution in which the smallest number of s's are true among the solutions in which the smallest number of d's are true.

[12] In the next section, we will see that the value of d does not change during step 2.

[13] Due to the lack of space, proofs for lemmas are not shown here, but will appear in the forthcoming technical report.

[14] It also changes unnecessary S to $\bot$.

10 Complexity

This section shows that the time complexity of solving constraints is almost linear to the length of a program. The number of constraints is linear to the length of a program because the number of constraints generated for each construct is constant. During type inference, we need to unify types as in the conventional type inference. The unification can be efficiently implemented using the union/find algorithm[18] which takes $O(N\alpha(N,N))$ time, where N is the length of a program and α is an inverse of Ackermann's function. Since $\alpha(N,N)$ is known to be less than four for all practical programs[18], $O(N\alpha(N,N))$ can be regarded as almost linear time.

Step 1 starts from the initial constraint $d = \mathsf{true}$ and then proceeds by propagating $d = \mathsf{true}$ or $s = \mathsf{false}$. Since constraints remain satisfied once they are satisfied, each constraint is processed only once. Thus, step 1 finishes in linear to the number of constraints, and hence to the length of a program. Step 2 similarly finishes in linear time by propagating $s = \mathsf{true}$ at most once for each constraint.

11 Related Work

Hornof, Consel, and Noyé[13] presented a binding-time analysis for C which handles both static and dynamic data structures. They performed forward and backward analyses (based on data-flow equations) to identify use and definition of data structures. Roughly speaking, use and definition correspond to s and d, and forward and backward analyses correspond to the two steps of our algorithm. Although developed independently, our work can be seen as an extension of their work to functional languages. We reformulated it in a type-based analysis and included higher-order functions.

Almost linear time algorithm for solving constraints was first presented by Henglein[12], and later refined by Bondorf and Jørgensen[4]. It was also applied to multi-level specialization by Glück and Jørgensen[11]. We basically followed the same technique of using the union/find algorithm[18] to obtain almost linear time algorithm. Because of the introduction of both static and dynamic expressions, our algorithm is split into two steps.

Sperber[16] presented an online partial evaluator that handles multi-valued binding-times: static, dynamic, and unknown. Although his work is similar to our work in that both treat binding-times other than static and dynamic, they are rather independent of each other. Sperber introduces the unknown binding-time when a static expression and a dynamic expression are merged (through, for example, conditional expressions). A value with the unknown binding-time is checked its binding-time at specialization time online. Such expression becomes dynamic in our framework. On the other hand, both static and dynamic expression presented here is an expression that is known to be used both as static and dynamic. Such expressions become dynamic in Sperber's framework.

This work is continuation of our previous work where we showed that propagation of both static and dynamic values is useful for specializing programs

containing tests for pointer equality[1]. Because it was based on an online approach, however, the resulting specializer did not run efficiently compared to existing offline specializers. This paper showed that the same technique can be applied to offline settings without spoiling the advantage of the offline framework.

Type-directed partial evaluation (TDPE)[5] is an efficient partial evaluation technique, where specialization is directed by the type of expressions. By systematically applying η-expansion[7] using types, it *reifies* (lifts) values to their program texts. From the TDPE perspective, our work is close to performing one-level of η-expansion. However, naive application of one-level η-expansion leads to code duplication. Since simple let-insertion will make residualized data completely dynamic, similar mechanism to ours will be necessary to use TDPE as a lifting tool of data structures. Also, it seems difficult to bound the time complexity of TDPE-based lifting. On the other hand, it will be an interesting research topic to see if the type presented here can be used to direct partial evaluation in a TDPE manner.

12 Conclusion

This paper presented a binding-time analysis for a functional language where expressions are allowed to be used as both static and dynamic, together with a specializer for it. With both static and dynamic expressions, we can statically access data structures while residualizing them at the same time. Previously, such data structures were treated as completely dynamic, which prevented us from accessing their components statically. The technique presented in this paper effectively allows us to lift data structures which was prohibited in the conventional partial evaluators. The binding-time analysis is formalized as a type system and the solution is obtained by solving constraints generated by the type system. We have proven the correctness of the constraint solving algorithm and shown that the algorithm runs efficiently in almost linear time. Based on this framework, we are currently making a partial evaluator for Scheme. As a future work, we are planning to extend this framework to multi-level specialization[11] where multiple binding-times are allowed.

Acknowledgements. I would like to thank Hidehiko Masuhara and anonymous referees for many valuable comments.

References

1. Asai, K., H. Masuhara, and A. Yonezawa "Partial Evaluation of Call-by-value λ-calculus with Side-effects," *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*, pp. 12–21 (June 1997).
2. Ashley, J. M., and C. Consel "Fixpoint Computation for Polyvariant Static Analyses of Higher-Order Applicative Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 5, pp. 1431–1448 (September 1994).

3. Bondorf, A., and O. Danvy "Automatic autoprojection of recursive equations with global variables and abstract data types," *Science of Computer Programming*, Vol. 16, pp. 151–195, Elsevier (1991).
4. Bondorf, A., and J. Jørgensen "Efficient analyses for realistic off-line partial evaluation," *Journal of Functional Programming*, Vol. 3, No 3, pp. 315–346, Cambridge University Press (July 1993).
5. Danvy, O. "Type-Directed Partial Evaluation," *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pp. 242–257 (January 1996).
6. Danvy, O., and A. Filinski "Abstracting Control," *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160 (June 1990).
7. Danvy, O., K. Malmkjær, and J. Palsberg "The essence of eta-expansion in partial evaluation," *Lisp and Symbolic Computation*, Vol. 8, No. 3, pp. 209–227, Kluwer Academic Publishers (1995).
8. Dean, J., C. Chambers, and D. Grove "Identifying Profitable Specialization in Object-Oriented Languages," *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '94)*, pp. 85–96 (June 1994).
9. Fujinami, N. "Determination of Dynamic Method Dispatches Using Run-time Code Generation," *Proceedings of the Second International Workshop on Types in Compilation (TIC 1998)*, pp. 135–151 (March 1998).
10. Futamura, Y. "Partial evaluation of computation process – an approach to a compiler-compiler," *Systems, Computers, Controls*, Vol. 2, No. 5, pp. 45–50, (1971).
11. Glück, R., and J. Jørgensen "An Automatic Program Generator for Multi-Level Specialization," *Lisp and Symbolic Computation*, Vol. 10, No. 2, pp. 113–158, Kluwer Academic Publishers (July 1997).
12. Henglein, F. "Efficient Type Inference for Higher-Order Binding-Time Analysis," In J. Hughes, editor, *Functional Programming Languages and Computer Architecture (LNCS 523)*, pp. 448–472 (August 1991).
13. Hornof, L., C. Consel, and J. Noyé "Effective Specialization of Realistic Programs via Use Sensitivity," In Van Hentenryck P., editor, *Static Analysis (LNCS 1302)*, pp. 63-73 (1997).
14. Jones, N. D., C. K. Gomard, and P. Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).
15. Ruf, E. *Topics in Online Partial Evaluation*, Ph.D. thesis, Stanford University (March 1993). Also published as Stanford Computer Systems Laboratory technical report CSL-TR-93-563.
16. Sperber, M. "Self-Applicable Online Partial Evaluation," In O. Danvy, R. Glück, and P. Thiemann editors, *Partial Evaluation (LNCS 1110)*, pp. 465–480 (February 1996).
17. Sperber, M., and P. Thiemann "Two for the price of one: composing partial evaluation and compilation," *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, SIGPLAN Notices, Vol. 32, No 5, pp. 215–225 (June 1997).
18. Tarjan, R. *Data Structures and Network Flow Algorithms*, Volume CMBS 44 of *Regional Conference Series in Applied Mathematics*, SIAM (1983).