Binding-Time Analysis for Both Static and Dynamic Expressions

Kenichi ASAI

Department of Information Science, Faculty of Science, University of Tokyo
7-3-1 Hongo Bunkyo-ku 113-0033 Japan
"Information and Human Activity," PRESTO, JST

asai@is.s.u-tokyo.ac.jp

Received 18 June 2001

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, data structures can be statically accessed while they are residualized at the same time. Previously, such data structures were treated as completely dynamic, which prevented their components from being accessed statically. The technique presented in this paper effectively allows data structures to be lifted 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.

Keywords Partial Evaluation, Offline Specialization, Binding-Time Analysis, Type System, Functional Languages.

§1 Introduction

Given a function $f(x, y)$ and some of its arguments $x = a$, program specialization (or partial evaluation) transforms f into a more efficient specialized program $f_a(y)$ by reducing expressions in f using the information in its known

argument[16]. 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[9] but also for the construction of compilers and compiler generators from interpreters[11, 16]. It is also applied to runtime code generation[10, 19].

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[17, 13] determines binding-times (*i.e.*, if an expression is static or dynamic) *during* specialization. Since the decision is made according to the actual values used for specialization, it leads to more accurate binding-times and hence better specialization.

An offline specializer[16], 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:

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

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-

[*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.

time analysis annotates P as follows:

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

where static and dynamic subterms are annotated with superscripts S and D, respectively. The variable x (which is bound to $cons\,1\,2$) is classified as dynamic because the program text $cons\,1\,2$ is required to residualize it. Consequently, $car\,x$ becomes dynamic, and the original term itself is returned as the result of specialization without performing any specialization.[*2]

Readers might think that $cons\,1\,2$ can be annotated as S as follows:

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

This is not the case because the residual program would then contain *specialization-time values* rather than program texts: $cons\,1\,(1\ .\ 2)$.

The problem can be avoided if we could *lift* data structures: we transform the specialization-time value $(1\ .\ 2)$ into its program text $cons\,1\,2$ as in:

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

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. 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

[*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 $(\lambda^S x.\, cons^D\,(car^S\,x)\,(cons^D\,(lift\,x)\,(lift\,x)))@^S\,(cons^S\,1\,2)$ will duplicate $cons\,1\,2$ as in: $cons\,1\,(cons\,(cons\,1\,2)\,(cons\,1\,2))$.

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[3]. 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. Soundness of the typing rules with respect to the specializer is proven in Sect. 6. In Sect. 7, these typing rules are transformed into typing rules for unannotated terms, giving constraint generation rules. The equivalence of the two typing rules is proven in Sect. 8. To solve the generated constraints, binding-times are first interpreted as a pair in Sect. 9. A two-phase constraint satisfaction algorithm is presented in Sect. 10, and its correctness is proven in Sect. 11. Its time complexity is shown to be almost linear time in Sect. 12. Related work is discussed in Sect. 13 and the paper concludes in Sect. 14.

§2 The Language

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

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

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:

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

$\lambda^S x.M$ 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.

Using the annotation B and coercion operators, we can annotate the term P as follows:

$$(\lambda^S x.\, cons^D\, (car^S\, x{\downarrow}_S^B)\, x{\downarrow}_D^B)@^S\, (cons^B\, 1\, 2)$$

which will be specialized into $cons\, 1\, (cons\, 1\, 2)$ using the specializer shown in the next section.

§3 Specializer

Figures 1 and 2 show a specializer for an annotated term M, written in the form of denotational semantics.[*4] Here, a variable with a superscript $^\diamond$ indicates that it is a fresh dynamic variable. The rule for *cdr* is omitted here and in the following sections because it is similar to the rule for *car*.

Given a term and an environment ρ, the specializer $\mathcal{S}$ reduces static parts and residualizes dynamic parts as usual. For the static expressions, $\mathcal{S}$ behaves exactly the same as an interpreter. For the dynamic expressions, $\mathcal{S}$

[*4] Strictly speaking, it is not denotational semantics since it contains creation of fresh variables. The problem can be avoided through the standard technique that passes an extra parameter for variable creation.

$$
\begin{array}{rcl}
\mathcal{S} : Exp \to Env & \to & W + \{\mathsf{error}\} \\
\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 & = & \mathit{build}\text{-}\lambda(x^\diamond, \mathcal{S}\,[\![M]\!]\,\rho[x^\diamond/x]) \\
\mathcal{S}\,[\![\lambda^B x.\,M]\!]\,\rho & = & (\mathit{build}\text{-}\lambda(x^\diamond, \mathcal{D}(\mathcal{S}\,[\![M]\!]\,\rho[x^\diamond/x])), \lambda x'.\,\mathcal{S}\,[\![M]\!]\,\rho[x'/x]) \\
\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 & = & \mathit{build}\text{-}@(\mathcal{S}\,[\![M_1]\!]\,\rho, \mathcal{S}\,[\![M_2]\!]\,\rho) \\
\mathcal{S}\,[\![\mathit{cons}^S M_1\, M_2]\!]\,\rho & = & (\mathcal{S}\,[\![M_1]\!]\,\rho, \mathcal{S}\,[\![M_2]\!]\,\rho) \\
\mathcal{S}\,[\![\mathit{cons}^D M_1\, M_2]\!]\,\rho & = & \mathit{build}\text{-}\mathit{cons}(\mathcal{S}\,[\![M_1]\!]\,\rho, \mathcal{S}\,[\![M_2]\!]\,\rho) \\
\mathcal{S}\,[\![\mathit{cons}^B M_1\, M_2]\!]\,\rho & = & \quad \mathbf{let}\ (m_1, m_2) = (\mathcal{S}\,[\![M_1]\!]\,\rho, \mathcal{S}\,[\![M_2]\!]\,\rho) \\
 & & \quad \mathbf{in}\ (\mathit{build}\text{-}\mathit{cons}(\mathcal{D}(m_1), \mathcal{D}(m_2)), (m_1, m_2)) \\
\mathcal{S}\,[\![\mathit{car}^S M]\!]\,\rho & = & \#_1(\mathcal{S}\,[\![M]\!]\,\rho) \\
\mathcal{S}\,[\![\mathit{car}^D M]\!]\,\rho & = & \mathit{build}\text{-}\mathit{car}(\mathcal{S}\,[\![M]\!]\,\rho) \\
\mathcal{S}\,[\![c^S]\!]\,\rho & = & c \\
\mathcal{S}\,[\![c^D]\!]\,\rho & = & \mathit{lift}(c) \\
\mathcal{S}\,[\![c^B]\!]\,\rho & = & (\mathit{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 & = & \mathit{build}\text{-}+(\mathcal{S}\,[\![M_1]\!]\,\rho, \mathcal{S}\,[\![M_2]\!]\,\rho) \\
\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{array}
$$

Fig. 1 Specializer for annotated terms written in direct style

builds residual code using auxiliary *build-* functions. The *build-* functions for Scheme are shown in Fig. 3.

When the expression to be specialized is both static and dynamic, both the residual code and the static value are created and returned as a pair. When the static or dynamic part of the pair is required, the corresponding element is extracted using coercion operations. The $\mathcal{D}(M)$ function appearing in the rule for $\lambda^B x.\,M$ and $\mathit{cons}^B M_1\, M_2$ takes the dynamic part of M when it is both static and dynamic. See Fig. 2. This is to ensure that the residual code does not contain any static values.

The rules for constants return a static constant, a dynamic constant, or both. To make a dynamic constant, a standard *lift* operator is used.[*5] In this paper, the three kinds of constants are strictly distinguished to treat constants

[*5] *Lift* operator transforms a static constant into its dynamic form (program text).

$$
\begin{array}{rcl}
W & = & W_S + W_D + W_B \\
W_S & = & \mathsf{Int} + (W \to W) + (W \times W) \\
W_D & = & \mathit{Code} \\
W_B & = & W_D \times W_S \\
\mathit{Env} & = & \mathit{Var} \to W
\end{array}
$$

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

Fig. 2 Semantic Domain and $\mathcal{D}(\cdot)$

$$
\begin{array}{lcl}
\mathit{build}\text{-}\lambda(x, N) & = & \texttt{(lambda (}x\texttt{) }N\texttt{)} \\
\mathit{build}\text{-}@(N_1, N_2) & = & \texttt{(}N_1\ N_2\texttt{)} \\
\mathit{build}\text{-}\mathit{cons}(N_1, N_2) & = & \texttt{(cons }N_1\ N_2\texttt{)} \\
\mathit{build}\text{-}\mathit{car}(N) & = & \texttt{(car }N\texttt{)} \\
\mathit{build}\text{-}+(N_1, N_2) & = & \texttt{(+ }N_1\ N_2\texttt{)}
\end{array}
$$

Fig. 3 *build-* functions for Scheme

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:

$$
\begin{array}{lcllcl}
\mathcal{S}[\![c^S]\!]\rho & = & c & \mathcal{S}[\![c{\downarrow}_D^B]\!]\rho & = & \mathit{lift}(c) \\
\mathcal{S}[\![c^D]\!]\rho & = & \mathit{lift}(c) & \mathcal{S}[\![c{\downarrow}_S^B]\!]\rho & = & c \\
\mathcal{S}[\![c^B]\!]\rho & = & c & & &
\end{array}
$$

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

For simplicity, the specializer $\mathcal{S}$ does not perform *let-insertion*[4] to residualize code but returns code directly. This sometimes causes problems such as code duplication and elimination. It can be avoided by simply adopting the standard let-insertion technique whenever *build-* functions are called.

Although $\mathcal{S}$ in Fig. 1 does not explicitly show any error-handling, $\mathcal{S}$ raises an error when non-function values are applied or when *build-* functions are called with non-code arguments.

To specialize a given annotated term M, we execute $\mathcal{S}$ with M and an empty environment ρ_{init}: $\mathcal{S}[\![M]\!]\rho_{init}$. Here, ρ_{init} is an environment which raises an error ("unbound variable") for all variables.

§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:

$$\pi = S \mid D \mid B \qquad \tau = \mathsf{Int}^{\pi} \mid \tau \to^{\pi} \tau \mid \tau \times^{\pi} \tau \ .$$

Binding-time values are either S (static), D (dynamic), or B (both static and dynamic). Binding-time types are either a constant, a function type, or a pair type. They are attached with a binding-time value to indicate their binding-times. We denote τ^{π} to separate a (toplevel) binding-time value π from its type, and say that it has a binding-time value π.

We assume for simplicity that the program to be specialized is simply typed with respect to the types without binding-time values. Then, the main task of the binding-time analysis is to infer the binding-time values of each type.

Not all the binding-time types defined above are well-formed types. Functions and pairs have constraints on the possible binding-time values of toplevel types and component types. For example, in $\tau_1 \times^{D} \tau_2$, both the binding-time values of τ_1 and τ_2 must be D, because if the pair as a whole is residualized as code, so are its components. To state the relationship of binding-time values, we first define two relations:

Definition 4.1
Relations $\pi_1 \triangleright \pi_2$ and $\pi_1 \triangleright' \pi_2$ on binding-time values are defined as follows:

$S \triangleright S$	$S \triangleright D$	$S \triangleright B$	$D \triangleright D$	$B \triangleright D$	$B \triangleright B$
$S \triangleright' S$	$S \triangleright' D$	$S \triangleright' B$	$D \triangleright' D$	$B \triangleright' D$	

If π_1 is static, no restriction on π_2. If π_1 is dynamic, π_2 has to be dynamic, too. 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 ($\triangleright'$).

Using these relations, we define the well-formed types as follows.

Definition 4.2

1. Int^{π} is well-formed (regardless of the value of π).
2. $\tau_1^{\pi_1} \to^{\pi} \tau_2^{\pi_2}$ is well-formed if $\pi \triangleright' \pi_1$ and $\pi \triangleright \pi_2$.
3. $\tau_1^{\pi_1} \times^{\pi} \tau_2^{\pi_2}$ is well-formed if $\pi \triangleright \pi_1$ and $\pi \triangleright \pi_2$.

When π is S or D, the definition coincides with the standard requirements on well-formed types. Namely, if the toplevel binding-time value is static, no restriction on the components, but if the toplevel binding-time value is dynamic,

so must be the components. In addition, we need to consider the case where π is B.

For a pair type $\tau_1^{\pi_1} \times^B \tau_2^{\pi_2}$ to be well-formed, we require that π_1 and π_2 must be either D or B. They are not allowed to be S because the specialization of $\tau_1^{\pi_1} \times^B \tau_2^{\pi_2}$ produces code (via *build-cons*) using code of its components. This is why we have $\pi \triangleright \pi_1$ and $\pi \triangleright \pi_2$ for $\tau_1^{\pi_1} \times^{\pi} \tau_2^{\pi_2}$.

For a function type $\tau_1^{\pi_1} \rightarrow^B \tau_2^{\pi_2}$ to be well-formed, we first require that π_2 has to be either D or S. This is the same reason as for the pair type. For π_1, we impose stronger restriction: π_1 has to be D (and not B). In other words, the argument of a function whose binding-time value is B must be D.

To understand the consequence of this restriction, 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 always has 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.[*6] 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.

[*6] 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.

$$\frac{\Gamma \vdash_1 M_1 : \tau_2^{\pi_2} \to^S \tau_1^{\pi_1} \quad \Gamma \vdash_1 M_2 : \tau_2^{\pi_2}}{\Gamma \vdash_1 M_1 @^S M_2 : \tau_1^{\pi_1}} \qquad \frac{\Gamma \vdash_1 M_1 : \tau_2^D \to^D \tau_1^D \quad \Gamma \vdash_1 M_2 : \tau_2^D}{\Gamma \vdash_1 M_1 @^D M_2 : \tau_1^D}$$

$$\Gamma, x : \tau^\pi \vdash_1 x : \tau^\pi \qquad \frac{\Gamma, x : \tau_1^{\pi_1} \vdash_1 M : \tau_2^{\pi_2}}{\Gamma \vdash_1 \lambda^S x.\, M : \tau_1^{\pi_1} \to^S \tau_2^{\pi_2}} \qquad \frac{\Gamma, x : \tau_1^D \vdash_1 M : \tau_2^D}{\Gamma \vdash_1 \lambda^D x.\, M : \tau_1^D \to^D \tau_2^D}$$

$$\frac{\Gamma \vdash_1 M_1 : \tau_1^{\pi_1} \quad \Gamma \vdash_1 M_2 : \tau_2^{\pi_2} \quad \pi_i \in \{B, D\}}{\Gamma \vdash_1 \mathit{cons}^B M_1\, M_2 : \tau_1^{\pi_1} \times^B \tau_2^{\pi_2}} \qquad \frac{\Gamma, x : \tau_1^D \vdash_1 M : \tau_2^{\pi_2} \quad \pi_2 \in \{B, D\}}{\Gamma \vdash_1 \lambda^B x.\, M : \tau_1^D \to^B \tau_2^{\pi_2}}$$

$$\frac{\Gamma \vdash_1 M_1 : \tau_1^{\pi_1} \quad \Gamma \vdash_1 M_2 : \tau_2^{\pi_2}}{\Gamma \vdash_1 \mathit{cons}^S M_1\, M_2 : \tau_1^{\pi_1} \times^S \tau_2^{\pi_2}} \qquad \frac{\Gamma \vdash_1 M : \tau_1^{\pi_1} \times^S \tau_2^{\pi_2}}{\Gamma \vdash_1 \mathit{car}^S M : \tau_1^{\pi_1}} \qquad \frac{\Gamma \vdash_1 M : \tau^B}{\Gamma \vdash_1 M \downarrow_S^B : \tau^S}$$

$$\frac{\Gamma \vdash_1 M_1 : \tau_1^D \quad \Gamma \vdash_1 M_2 : \tau_2^D}{\Gamma \vdash_1 \mathit{cons}^D M_1\, M_2 : \tau_1^D \times^D \tau_2^D} \qquad \frac{\Gamma \vdash_1 M : \tau_1^D \times^D \tau_2^D}{\Gamma \vdash_1 \mathit{car}^D M : \tau_1^D} \qquad \frac{\Gamma \vdash_1 M : \tau^B}{\Gamma \vdash_1 M \downarrow_D^B : [\tau]^D}$$

$$\frac{\Gamma \vdash_1 M_1 : \mathsf{Int}^S \quad \Gamma \vdash_1 M_2 : \mathsf{Int}^S}{\Gamma \vdash_1 M_1 +^S M_2 : \boxed{\mathsf{Int}^\pi}} \qquad \frac{\Gamma \vdash_1 M_1 : \mathsf{Int}^D \quad \Gamma \vdash_1 M_2 : \mathsf{Int}^D}{\Gamma \vdash_1 M_1 +^D M_2 : \mathsf{Int}^D} \qquad \Gamma \vdash_1 c^\pi : \mathsf{Int}^\pi$$

Fig. 4 Typing rules for annotated terms (type-checking rules)

§5 Typing Rules

Figure 4 shows typing rules for annotated terms. We call them *type-checking rules*. A judgement $\Gamma \vdash_1 M : \tau$ represents that M has a binding-time type τ under a type assumption Γ. $[\tau]^D$ appearing in the rule for $\tau \downarrow_D^B$ is a type obtained by replacing all the binding-time values in τ^D with D. These rules are fairly standard and do not need particular explanation. We only note that 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.

Definition 5.1
An annotated term p_{ann} is *well-annotated* if $\vdash_1 p_{\text{ann}} : D$ is derivable.

Definition 5.2
Among all the well-annotated term p_{ann} of p, p^*_{ann} has the *best* annotation if the following two conditions hold for any subterm of p:

1. If the binding-time value is S in p_{ann}, it is also S in p^*_{ann}.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash_1 x : D \times^B D}{\Gamma \vdash_1 x\downarrow_S^B : D \times^S D}\quad \dfrac{\Gamma \vdash_1 x : D \times^B D}{\Gamma \vdash_1 x\downarrow_D^B : D}}{\Gamma \vdash_1 car^S x\downarrow_S^B : D}}{\Gamma \vdash_1 cons^D (car^S x\downarrow_S^B)\, x\downarrow_D^B : D}}{\vdash_1 \lambda^S x.\, cons^D (car^S x\downarrow_S^B)\, x\downarrow_D^B : D \times^B D \to^S D}\quad \dfrac{\vdash_1 1 : D \quad \vdash_1 2 : D}{\vdash_1 cons^B 1\,2 : D \times^B D}}{\vdash_1 (\lambda^S x.\, cons^D (car^S x\downarrow_S^B)\, x\downarrow_D^B)@^S (cons^B 1\,2) : D}$$

Fig. 5 Example of type checking (Γ represents $x : D \times^B D$ and Int^π is written π)

2. (Applied only when the first condition is satisfied.)
   If the binding-time value is B in p^*_{ann}, it is also B in p_{ann}.

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, we can obtain the same specialized program 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 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.

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

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

then, its derivation becomes as shown in Fig. 5.

The typing rules shown in Fig. 4 are an extension of the conventional binding-time analyses. By substituting all B with D, we find that the rules are standard 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.

$$
\begin{aligned}
V_{\mathsf{Int}^S} &= \mathsf{Int} \\
V_{\tau_1^{\pi_1} \to^S \tau_2^{\pi_2}} &= V_{\tau_1^{\pi_1}} \to V_{\tau_2^{\pi_2}} \\
V_{\tau_1^{\pi_1} \times^S \tau_2^{\pi_2}} &= V_{\tau_1^{\pi_1}} \times V_{\tau_2^{\pi_2}} \\
V_{\tau^D} &= \mathit{Code} \\
V_{\tau^B} &= V_{\tau^D} \times V_{\tau^S}
\end{aligned}
$$

Fig. 6 Definition of V_{τ^π}

§6 Soundness of Type-Checking Rules

This section shows that the type-checking rules presented in Fig. 4 are sound with respect to the specializer $\mathcal{S}$. In other words, we will show that if a term has a binding-time type τ^π (according to the type-checking rules), $\mathcal{S}$ will return a value of that type (without producing an error). Define V_{τ^π}, a domain of values of type τ^π, as in Fig. 6. In the following, we write $\tau^\pi = \Gamma(x)$ when x has a type τ^π in Γ.

Definition 6.1
A specialization-time environment ρ *respects* a type environment Γ if $\rho(x) \in V_{\Gamma(x)}$ holds for all x such that $\rho(x)$ is defined.

Then, the soundness of the type-checking rules is formally stated as follows.

Theorem 6.1 (Soundness of type-checking rules)
If $\Gamma \vdash_1 M : \tau^\pi$ is derivable using the type-checking rules in Fig. 4, then for all ρ that respects Γ, $\mathcal{S}\,[\![M]\!]\,\rho \in V_{\tau^\pi}$ holds.

Proof By induction on the depth of the derivation tree for $\Gamma \vdash_1 M : \tau^\pi$. Below, we will show the cases of M only for x, $\lambda^S x.\,M$, $\lambda^D x.\,M$, and $\lambda^B x.\,M$. Other cases are similar.

(the case for x) The derivation tree is $\Gamma, x : \tau^\pi \vdash_1 x : \tau^\pi$. For all ρ that respects $\Gamma, x : \tau^\pi$, we have $\rho(x) \in V_{\tau^\pi}$. Thus, $\mathcal{S}\,[\![x]\!]\,\rho = \rho(x) \in V_{\tau^\pi}$

(the case for $\lambda^S x.\,M$) The bottom of the derivation tree becomes:

$$\frac{\Gamma, x : \tau_1^{\pi_1} \vdash_1 M : \tau_2^{\pi_2}}{\Gamma \vdash_1 \lambda^S x.\,M : \tau_1^{\pi_1} \to^S \tau_2^{\pi_2}}$$

Assume that ρ respects Γ. Since $\rho[x'/x]$ respects $\Gamma, x : \tau_1^{\pi_1}$ for all x' such that $x' \in V_{\tau_1^{\pi_1}}$, we have $\mathcal{S}\,[\![M]\!]\,\rho[x'/x] \in V_{\tau_2^{\pi_2}}$ from the induction

hypothesis. Thus,

$$\begin{aligned}\mathcal{S}\,[\![\lambda^S x.\,M]\!]\,\rho &= \lambda x'.\,\mathcal{S}\,[\![M]\!]\,\rho[x'/x] \\ &\in V_{\tau_1^{\pi_1}} \to V_{\tau_2^{\pi_2}} \\ &= V_{\tau_1^{\pi_1} \to^S \tau_2^{\pi_2}}\end{aligned}$$

(the case for $\lambda^D x.\,M$) The bottom of the derivation tree becomes:

$$\frac{\Gamma, x : \tau_1^D \vdash_1 M : \tau_2^D}{\Gamma \vdash_1 \lambda^D x.\,M : \tau_1^D \to^D \tau_2^D}$$

Assume that ρ respects Γ. Since $\rho[x^\diamond/x]$ respects $\Gamma, x : \tau_1^D$ for all $x^\diamond$ such that $x^\diamond \in V_{\tau_1^D}$, we have $\mathcal{S}\,[\![M]\!]\,\rho[x^\diamond/x] \in V_{\tau_2^D}$ from the induction hypothesis. As the arguments of *build*-λ below are code, we have

$$\begin{aligned}\mathcal{S}\,[\![\lambda^D x.\,M]\!]\,\rho &= \mathit{build}\text{-}\lambda(x^\diamond, \mathcal{S}\,[\![M]\!]\,\rho[x^\diamond/x]) \\ &\in \mathit{Code} \\ &= V_{\tau_1^D \to^D \tau_2^D}\end{aligned}$$

(the case for $\lambda^B x.\,M$) The bottom of the derivation tree becomes:

$$\frac{\Gamma, x : \tau_1^D \vdash_1 M : \tau_2^{\pi_2} \quad \pi_2 \in \{B, D\}}{\Gamma \vdash_1 \lambda^B x.\,M : \tau_1^D \to^B \tau_2^{\pi_2}}$$

Assume that ρ respects Γ. Since $\rho[x'/x]$ respects $\Gamma, x : \tau_1^D$ for all x' such that $x' \in V_{\tau_1^D}$, we have $\mathcal{S}\,[\![M]\!]\,\rho[x'/x] \in V_{\tau_2^{\pi_2}}\,(\pi_2 \in \{B, D\})$ from the induction hypothesis. Thus,

$$\begin{aligned}\lambda x'.\,\mathcal{S}\,[\![M]\!]\,\rho[x'/x] &\in V_{\tau_1^D} \to V_{\tau_2^{\pi_2}} \\ &= V_{\tau_1^D \to^S \tau_2^{\pi_2}}\end{aligned}$$

On the other hand, since $\rho[x^\diamond/x]$ respects $\Gamma, x : \tau_1^D$ for all $x^\diamond$ such that $x^\diamond \in V_{\tau_1^D}$, we have $\mathcal{S}\,[\![M]\!]\,\rho[x^\diamond/x] \in V_{\tau_2^{\pi_2}}\,(\pi_2 \in \{B, D\})$ from the induction hypothesis. As the argument of $\mathcal{D}(\cdot)$ is either B or D, we have $\mathcal{D}(\mathcal{S}\,[\![M]\!]\,\rho[x^\diamond/x]) \in \mathit{Code}$. Thus,

$$\begin{aligned}\mathit{build}\text{-}\lambda(x^\diamond, \mathcal{D}(\mathcal{S}\,[\![M]\!]\,\rho[x^\diamond/x])) &\in \mathit{Code} \\ &= V_{\tau_1^D \to^D \tau_2^{\pi_2}}\end{aligned}$$

Together, we have

$$\begin{aligned} & \mathcal{S}[\![\lambda^B x.\, M]\!]\rho \\ = \;& (\mathit{build}\text{-}\lambda(x^\diamond, \mathcal{D}(\mathcal{S}[\![M]\!]\rho[x^\diamond/x])), \lambda x'.\, \mathcal{S}[\![M]\!]\rho[x'/x]) \\ \in \;& (V_{\tau_1^D \to^D \tau_2^{\pi_2}}) \times (V_{\tau_1^D \to^S \tau_2^{\pi_2}}) \\ = \;& V_{\tau_1^D \to^B \tau_2^{\pi_2}} \end{aligned}$$

■

As a special case of this theorem, we can conclude that the specialization of a well-annotated term always produces code without an error:

Corollary 6.1
If $\vdash_1 M : \tau^D$, then $\mathcal{S}[\![M]\!]\rho_{init} \in V_{\tau^D}$.

Proof From Theorem 6.1 because ρ_{init} respects the empty type environment. ■

§7 Constraint Generation

Since the type-checking rules presented in Fig. 4 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. 7. We call them *constraint generation rules.*

The constraint generation rules are directed over the syntax of unannotated terms. A judgement $\Gamma \vdash_2 L : \tau^\pi \rightsquigarrow M$ represents that an unannotated term L has a binding-time type τ^π and is transformed to annotated term M under a type assumption Γ. The boxed parts indicate the places where coercion rules are incorporated. We will see in the next section that these coercion rules are sufficient. $[\tau]^\pi$ is a type obtained by replacing all the binding-time values in τ^π with D when $\pi = D$, and otherwise τ^π itself. The constraint $\pi \geq \pi'$ shows coercibility from π to π' and is defined as follows:

$$B > S, \quad B > D$$
$$\pi \geq \pi' \Longleftrightarrow (\pi > \pi') \vee (\pi = \pi') \;\; .$$

Coercion is permitted only from B to S or D (not from S to D). If inequality

$$\Gamma \vdash_2 c : \mathsf{Int}^{\pi} \rightsquigarrow c^{\pi} \qquad \Gamma, x : \tau^{\pi} \vdash_2 x : \boxed{[\tau]^{\pi'}} \rightsquigarrow x{\downarrow}^{\pi}_{\pi'}, \quad \{\boxed{\pi \geq \pi'}\}$$

$$\frac{\Gamma, x : \tau_1^{\pi_1} \vdash_2 L : \tau_2^{\pi_2} \rightsquigarrow M}{\Gamma \vdash_2 \lambda x.L : \tau_1^{\pi_1} \rightarrow^{\pi} \tau_2^{\pi_2} \rightsquigarrow \lambda^{\pi} x.M} \ \{\pi \rhd' \pi_1,\ \pi \rhd \pi_2\}$$

$$\frac{\Gamma \vdash_2 L_1 : \tau_2^{\pi_2} \rightarrow^{\pi} \tau_1^{\pi_1} \rightsquigarrow M_1 \quad \Gamma \vdash_2 L_2 : \tau_2^{\pi_2} \rightsquigarrow M_2}{\Gamma \vdash_2 L_1 @ L_2 : \boxed{[\tau_1]^{\pi_1'}} \rightsquigarrow (M_1 @^{\pi} M_2){\downarrow}^{\pi_1}_{\pi_1'}} \left\{ \begin{array}{ll} \boxed{\pi_1 \geq \pi_1'}, & \pi \rhd' \pi_2, \\ \pi \in \{S, D\}, & \pi \rhd \pi_1 \end{array} \right\}$$

$$\frac{\Gamma \vdash_2 L_1 : \tau_1^{\pi_1} \rightsquigarrow M_1 \quad \Gamma \vdash_2 L_2 : \tau_2^{\pi_2} \rightsquigarrow M_2}{\Gamma \vdash_2 \mathit{cons}\, L_1\, L_2 : \tau_1^{\pi_1} \times^{\pi} \tau_2^{\pi_2} \rightsquigarrow \mathit{cons}^{\pi} M_1\, M_2} \ \{\pi \rhd \pi_1,\ \pi \rhd \pi_2\}$$

$$\frac{\Gamma \vdash_2 L : \tau_1^{\pi_1} \times^{\pi} \tau_2^{\pi_2} \rightsquigarrow M}{\Gamma \vdash_2 \mathit{car}\, L : \boxed{[\tau_1]^{\pi_1'}} \rightsquigarrow (\mathit{car}^{\pi} M){\downarrow}^{\pi_1}_{\pi_1'}} \left\{ \begin{array}{ll} \boxed{\pi_1 \geq \pi_1'}, & \pi \rhd \pi_1, \\ \pi \in \{S, D\}, & \pi \rhd \pi_2 \end{array} \right\}$$

$$\frac{\Gamma \vdash_2 L_1 : \mathsf{Int}^{\pi_1} \rightsquigarrow M_1 \quad \Gamma \vdash_2 L_2 : \mathsf{Int}^{\pi_2} \rightsquigarrow M_2}{\Gamma \vdash_2 L_1 + L_2 : \mathsf{Int}^{\pi} \rightsquigarrow M_1 +^{\pi} M_2} \left\{ \begin{array}{l} \pi_1 \rhd \pi,\ \pi_2 \rhd \pi, \\ \pi_1 \rhd \pi_2,\ \pi_2 \rhd \pi_1, \\ \pi_1 \in \{S, D\},\ \pi_2 \in \{S, D\} \end{array} \right\}$$

Fig. 7 Typing rules for unannotated terms (constraint generation rules)

$\pi > \pi'$ holds when a binding-time analysis finishes, an annotation ${\downarrow}^{\pi}_{\pi'}$ is attached to the term corresponding to π'. In the case when $\pi = \pi'$, we interpret $M{\downarrow}^{\pi}_{\pi'} = M$.

Although the rules in Fig. 7 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. 7. Then, these constraints are solved to give types of each subterm.

As an example of constraint generation, consider the term P in Sect. 1. Let Γ be $x : \pi_1 \times^{\pi_3} \pi_2$. ($\mathsf{Int}^{\pi}$ is written π here and in the following derivation.)

After unification of types, we obtain the following derivation:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash_1 x : \pi_1 \times^{\pi_3''} \pi_2}{\Gamma \vdash_1 car\ x : \pi_1'} \quad \Gamma \vdash_1 x : \pi_1 \times^{\pi_3'} \pi_2}{\Gamma \vdash_1 cons\ (car\ x)\ x : \pi_1' \times^{\pi_4} (\pi_1 \times^{\pi_3'} \pi_2)}}{\vdash_1 \lambda x.\ cons\ (car\ x)\ x : \pi_1 \times^{\pi_3} \pi_2 \to^{\pi_5} \pi_1' \times^{\pi_4} (\pi_1 \times^{\pi_3'} \pi_2)} \quad \dfrac{\vdash_1 1 : \pi_1 \quad \vdash_1 2 : \pi_2}{\vdash_1 cons\ 1\ 2 : \pi_1 \times^{\pi_3} \pi_2}}{\vdash_1 (\lambda x.\ cons\ (car\ x)\ x)@(cons\ 1\ 2) : \pi_1' \times^{\pi_4'} (\pi_1 \times^{\pi_3'} \pi_2)}$$

together with the following constraints:

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

The best solution to these constraints (with the initial constraint $\pi_4' = D$) is:[*7]

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

which gives us the following expected result:

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

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.

§8 Equivalence of Type-Checking Rules and Constraint Generation Rules

In this section, we show that the type-checking rules (Fig. 4) and the constraint generation rules (Fig. 7) are equivalent. We first show that the annotation obtained via the constraint generation rules can also be obtained via the type-checking rules.

Theorem 8.1
If $\Gamma \vdash_2 L : \tau^\pi \rightsquigarrow M$, then $\Gamma \vdash_1 M : \tau^\pi$.

Proof We show that the derivation tree of the constraint generation rules can be transformed to a derivation tree of type-checking rules. We show the cases for $\lambda x.\, L$, $L_1 @ L_2$, and $L_1 + L_2$. Other cases are similar.

(the case for $\lambda x.\, L$) The constraint generation rule is transformed to one of the rules for $\lambda^S x.\, M$, $\lambda^D x.\, M$, or $\lambda^B x.\, M$ according to $\pi = S, D$, or B, respectively. The two constraints $\pi \triangleright' \pi_1$ and $\pi \triangleright \pi_2$ are used to satisfy

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

the well-formedness of types. When $\pi = S$, these constraints are trivially satisfied, and we obtain the rule for $\lambda^S x.\, M$. When $\pi = D$, the constraints force $\pi_1 = \pi_2 = D$, and we obtain the rule for $\lambda^D x.\, M$. When $\pi = B$, the constraints force $\pi_1 = D$ and $\pi_2 \in \{B, D\}$, and we obtain the rule for $\lambda^B x.\, M$.

(the case for $L_1@L_2$) The constraint generation rule is transformed to the rules for either $M_1@^S M_2$ or $M_1@^D M_2$ according to $\pi = S$ or D, respectively. The two constraints $\pi \triangleright' \pi_2$ and $\pi \triangleright \pi_1$ are used to satisfy the well-formedness of types. When $\pi_1 > \pi_1'$ holds, the coercion rule $\downarrow^B_{\pi_1'}$ is additionally used. For example,

$$\frac{\Gamma \vdash_2 L_1 : \tau_2^{\pi_2} \to^S \tau_1^B \rightsquigarrow M_1 \quad \Gamma \vdash_2 L_2 : \tau_2^{\pi_2} \rightsquigarrow M_2}{\Gamma \vdash_2 L_1@L_2 : [\tau_1]^{\pi_1'} \rightsquigarrow (M_1@^S M_2)\downarrow^B_{\pi_1'}}$$

is transformed to

$$\frac{\dfrac{\Gamma \vdash_1 M_1 : \tau_2^{\pi_2} \to^S \tau_1^B \quad \Gamma \vdash_1 M_2 : \tau_2^{\pi_2}}{\Gamma \vdash_1 M_1@^S M_2 : \tau_1^B}}{\Gamma \vdash_1 (M_1@^S M_2)\downarrow^B_{\pi_1'} : [\tau_1]^{\pi_1'}} \quad .$$

(the case for $L_1 + L_2$) The constraint generation rule is transformed to the rule for $M_1 +^S M_2$ when $\pi_1 = \pi_2 = S$ and to the rule for $M_1 +^D M_2$ when $\pi_1 = \pi_2 = D$. The other cases are excluded by the two constraints $\pi_1 \triangleright \pi_2$ and $\pi_2 \triangleright \pi_1$. The constraints $\pi_1 \triangleright \pi$ and $\pi_2 \triangleright \pi$ ensure that if $\pi_1 = \pi_2 = D$, then $\pi = D$.

■

To show the other direction, namely, that the derivation tree obtained via the type-checking rules can be obtained via the constraint generation rules, we need a lemma.

Lemma 8.1
In a derivation tree for a term with the best annotation, coercion rules are applied only to x, $M_1@^\pi M_2$, or $car^\pi M$ (or $cdr^\pi M$). (In other words, coercion rules are not applied to c^π, $\lambda^\pi x.\, M$, $cons^\pi M_1\, M_2$, or $M_1 +^\pi M_2$.)

Proof Suppose that a coercion rule is applied to $\lambda^\pi x.\, M$. There are two possibilities: $(\lambda^B x.\, M)\downarrow^B_D$ or $(\lambda^B x.\, M)\downarrow^B_S$. In either case, it is not the best annotation because if we replace them with $\lambda^D x.\, M$ or $\lambda^S x.\, M$, respectively, we

obtain a better annotation: the binding-time value of the lambda expression is changed from B to S or D (without changing the binding-time values of other subterms). This concludes the proof for $\lambda^\pi x.M$. Other cases are similar. ■

Define $|M|$ to be an unannotated term obtained by removing all the annotation and the coercion rules from M. The next theorem states that the best annotation obtained via the type-checking rules can be obtained via the constraint generation rules, too.

Theorem 8.2
If $\Gamma \vdash_1 M : \tau^\pi$ and M has the best annotation, then $\Gamma \vdash_2 |M| : \tau^\pi \rightsquigarrow M$.

Proof We show that the derivation tree of the type-checking rules for a term having the best annotation can be transformed to a derivation tree of constraint generation rules.

For the cases other than coercion rules, type checking rules can be transformed directly to the corresponding constraint generation rules. For coercion rules, we know from Lemma 8.1 that they are applied only to either x, $M_1 @^\pi M_2$, or $car^\pi M$ (or $cdr^\pi M$). For all the cases, the corresponding constraint generation rules contain $\pi \geq \pi'$. Thus, we can transform a type-checking rule together with a coercion rule to a constraint generation rule. ■

§9 Interpreting Types as Pairs

In the following sections, we describe constraint satisfaction algorithm. Rather than directly solving constraints obtained by the constraint generation rules in Fig. 7, 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 following table summarizes the correspondence between π and (s, d).

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

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.[*8] In this case, we can treat L in an arbitrary way. Here, we denote it as $\bot$.

Under this interpretation of binding-time values, the best annotation is rephrased as follows:

Definition 9.1
Among all the well-annotated term p_{ann} of p, p^*_{ann} has the *best* annotation if the following two conditions hold for any subterm of p:

1. If the value of d is false in p_{ann}, it is also false in p^*_{ann}.
2. (Applied only when the first condition is satisfied.)
   If the value of s is false in p_{ann}, it is also false in p^*_{ann}.

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.

If we interpret the case where $s = d = \mathsf{false}$ as S rather than $\bot$, Definition 9.1 coincides with Definition 5.2. We used $\bot$ here because it simplifies the constraint solving algorithm. In the following, we use Definition 9.1 as the definition of the best annotation.

The constraints appearing in Fig. 7 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 $\wedge$.

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

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.

Note that for all the constraints $(A) \Rightarrow (B)$, A and B have always the form $s = \mathsf{false}$ or $d = \mathsf{true}$ (and not $s = \mathsf{true}$ or $d = \mathsf{false}$). This observation is important in designing constraint satisfaction algorithm. In the following sections, we express constraints as $(p = x) \Rightarrow (p' = x')$, where p and p' are s or d and x and x' are true or false.

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

§10 Solving Constraints

Given an unannotated term L, the binding-time analysis first generates constraints as described in the previous sections. Assume that the type of L itself is given by $\pi_{top} = (s_{top}, d_{top})$. Then, the constraints are solved together with the initial constraint $d_{top} = \mathsf{true}$. 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_{top}) and true to all s's. Interpret a constraint $(p = x) \Rightarrow (p' = x')$ as "if $p = x$ holds, then change the value of appropriate s or d so that $p' = x'$ holds." Repeat this operation until no changes are possible.

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, $(p = x) \Rightarrow (p' = x')$ is interpreted as "if $p' = x'$ does not hold, then change the value of appropriate s or d so that $p = x$ does not hold." Repeat this operation until no changes are possible.

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 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[*9] 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 B 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

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

important) is fulfilled. The latter is satisfied at step 2 to the extent that the former is not violated.

Step 2 is concerned with s.[*10] It identifies expressions that are actually used as both static and dynamic among the expressions that are classified as B in 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 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.

Let us state the constraint solving algorithm more formally. Let C_{init} be a set of constraints over s and d that are transformed (via the interpretation described in Sect. 9) from the constraints over π obtained via the constraint generation rules in Fig. 7. Let C^*_{init} be a set of contraposition[*11] of the constraints in C_{init}.

Step 1 Let the initial constraint set C_{I} and the initial solution θ_{I} be as follows:

$$\begin{array}{rcl} C_{\mathrm{I}} & = & C_{init} \\ \theta_{\mathrm{I}} & = & \{d_{top} = \mathsf{true}\} \\ & \cup & \{d = \mathsf{false} \mid \text{for all } d(\neq d_{top})\} \\ & \cup & \{s = \mathsf{true} \mid \text{for all } s\} \end{array}$$

Apply the following transformation rule to $\langle C_{\mathrm{I}}, \theta_{\mathrm{I}} \rangle$ as much as possible:

$$\begin{array}{l} \langle \{(p = x) \Rightarrow (p' = x')\} \cup C, \{p = x, p' = x''\} \cup \theta \rangle \\ \rightarrow_{\mathrm{I}} \langle C, \{p = x, p' = x'\} \cup \theta \rangle \end{array}$$

Namely, if $p = x$ exists in the current solution and there exists $(p = x) \Rightarrow (p' = x')$ in the current constraint set, then remove this constraint from the constraint set and update the value of p'. Let the final constraint set and solution be $\langle \tilde{C}_{\mathrm{I}}, \tilde{\theta}_{\mathrm{I}} \rangle$.

Step 2 Let the initial constraint set C_{II} and the initial solution θ_{II} be as follows:

$$\begin{array}{rcl} C_{\mathrm{II}} & = & C^*_{init} \\ \theta_{\mathrm{II}} & = & \{d = x \mid d = x \in \tilde{\theta}_{\mathrm{I}}\} \\ & \cup & \{s = \mathsf{false} \mid \text{for all } s\} \end{array}$$

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

[*11] Contraposition of $(p_1 = x_1) \Rightarrow (p_2 = x_2)$ is given by $(p_2 = \neg x_2) \Rightarrow (p_1 = \neg x_1)$.

Apply the same transformation rule as step 1 to $\langle C_{\mathrm{II}}, \theta_{\mathrm{II}} \rangle$ as much as possible.[*12] (We denote the transformation in step 2 as $\rightarrow_{\mathrm{II}}$.) Let the final constraint set and solution be $\langle \tilde{C_{\mathrm{II}}}, \tilde{\theta_{\mathrm{II}}} \rangle$.

From the construction of the initial solution and the transformation rule, we observe that there exists exactly one $p = x$ in the solution for all p. Thus, given a solution, we can obtain annotation from the $\rightsquigarrow M$ part of the constraint generation rules.

In the following, we write $\theta(p) = x$ if $p = x$ is in θ. $\theta[p \mapsto x]$ is a solution obtained by removing $p = x'$ and adding $p = x$ to θ. When it is important to show explicitly which constraint is removed during a transformation, we will attach a name to a constraint as in $c : (p = x) \Rightarrow (p' = x')$ and write $\langle C, \theta \rangle \xrightarrow{c}_{\mathrm{I}} \langle C', \theta' \rangle$.

§11 Correctness

This section describes the correctness of our algorithm. Assume that boolean values have ordering $\mathsf{true} > \mathsf{false}$. In step 1, we define the ordering of solutions as $\theta \leq_{\mathrm{I}} \theta' \overset{\mathrm{def}}{\Longleftrightarrow} \forall s.\theta(s) \geq \theta'(s) \wedge \forall d.\theta(d) \leq \theta'(d)$. The next lemma shows that a solution does not decrease in this ordering during step 1:

Lemma 11.1
If $\langle C, \theta \rangle \rightarrow_{\mathrm{I}} \langle C', \theta' \rangle$, then $\theta \leq_{\mathrm{I}} \theta'$.

Proof The lemma holds if $\theta = \theta'$. Assume that $\theta \neq \theta'$. Since we have only $s = \mathsf{false}$ or $d = \mathsf{true}$ in the conclusion of $(p = x) \Rightarrow (p' = x')$, there are only two possibilities: (1) $\theta(s) = \mathsf{true}$ and $\theta'(s) = \mathsf{false}$ for some s, or (2) $\theta(d) = \mathsf{false}$ and $\theta'(d) = \mathsf{true}$ for some d. In either case, the lemma holds since $\theta \leq_{\mathrm{I}} \theta'$. ■

Lemma 11.2
The final solution $\tilde{\theta_{\mathrm{I}}}$ of step 1 satisfies all the constraints in C_{init}.

Proof A constraint $(p = x) \Rightarrow (p' = x')$ in C_{init} can be divided into two groups according to whether it is in the final constraint set $\tilde{C_{\mathrm{I}}}$ or not. If it is in $\tilde{C_{\mathrm{I}}}$, it means that the premise $p = x$ did not hold in $\tilde{\theta_{\mathrm{I}}}$, and thus $\tilde{\theta_{\mathrm{I}}}$ satisfies the constraint. If the constraint is not in $\tilde{C_{\mathrm{I}}}$, it means that the following

[*12] Remember that constraints in C_{II} are contraposition of those in C_{I}.

transformation occurred during step 1:

$$\langle\{(p = x) \Rightarrow (p' = x')\} \cup C, \{p = x, p' = x''\} \cup \theta\rangle$$
$$\rightarrow_{\mathrm{I}} \langle C, \{p = x, p' = x'\} \cup \theta\rangle$$

where $(p = x) \Rightarrow (p' = x')$ is removed and the value of p' is updated to x'. Remember that $p' = x'$ has the form $s = \mathsf{false}$ or $d = \mathsf{true}$. Once the value of p is updated to x', we know from Lemma 11.1 that p' will never be updated (decreased) to $s = \mathsf{true}$ or $d = \mathsf{false}$. Thus, at the end of step 1, $p' = x'$ still holds, and the constraint is satisfied. ■

The next lemma shows that the ordering of the application of constraints does not affect the final result.

Lemma 11.3
If $\langle C_0, \theta_0\rangle \xrightarrow{c_1}_{\mathrm{I}} \langle C_1, \theta_1\rangle$ and $\langle C_0, \theta_0\rangle \xrightarrow{c_2}_{\mathrm{I}} \langle C_2, \theta_2\rangle$, then there exists $\langle C_3, \theta_3\rangle$ such that $\langle C_1, \theta_1\rangle \xrightarrow{c_2}_{\mathrm{I}} \langle C_3, \theta_3\rangle$ and $\langle C_2, \theta_2\rangle \xrightarrow{c_1}_{\mathrm{I}} \langle C_3, \theta_3\rangle$.

Proof Let c_1 and c_2 be $(p_1 = x_1) \Rightarrow (p_2 = x_2)$ and $(p_3 = x_3) \Rightarrow (p_4 = x_4)$, respectively. Since both c_1 and c_2 are applicable in $\langle C_0, \theta_0\rangle$, we have $\theta_0(p_1) = x_1$ and $\theta_0(p_3) = x_3$.

Let us apply c_1 first. From $\langle C_0, \theta_0\rangle \xrightarrow{c_1}_{\mathrm{I}} \langle C_1, \theta_1\rangle$, we have $\theta_1 = \theta_0[p_2 \mapsto x_2]$. If $p_2 \neq p_3$, then $\theta_1(p_3) = \theta_0(p_3) = x_3$ and we can apply c_2 to $\langle C_1, \theta_1\rangle$, giving $\theta_3 = \theta_1[p_4 \mapsto x_4] = \theta_0[p_2 \mapsto x_2][p_4 \mapsto x_4]$. If $p_2 = p_3$, on the other hand, we have $x_2 = x_3$ because $p_i = x_i$ always has the form $d = \mathsf{true}$ or $s = \mathsf{false}$. Thus, from $\theta_1(p_3) = \theta_1(p_2) = x_2 = x_3$, we can apply c_2 to $\langle C_1, \theta_1\rangle$ and obtain the same result $\theta_3 = \theta_0[p_2 \mapsto x_2][p_4 \mapsto x_4]$.

If c_2 is applied first, we will see using the same argument that c_1 can be applied, resulting in the solution $\theta'_3 = \theta_2[p_2 \mapsto x_2] = \theta_0[p_4 \mapsto x_4][p_2 \mapsto x_2]$. To show $\theta'_3 = \theta_3$, we consider two cases: If $p_2 \neq p_4$, $\theta'_3 = \theta_0[p_4 \mapsto x_4][p_2 \mapsto x_2] = \theta_0[p_2 \mapsto x_2][p_4 \mapsto x_4] = \theta_3$. If $p_2 = p_4$, on the other hand, we have $x_2 = x_4$ (because $p_i = x_i$ always has the form $d = \mathsf{true}$ or $s = \mathsf{false}$), and thus $\theta'_3 = \theta_0[p_4 \mapsto x_4][p_2 \mapsto x_2] = \theta_0[p_2 \mapsto x_2] = \theta_0[p_2 \mapsto x_2][p_4 \mapsto x_4] = \theta_3$

As for C_3, we have $C_3 = C_0 \setminus \{c_1, c_2\}$ for both cases and we are done. ■

The next lemma shows that the final solution of step 1 is unique. Here, $\rightarrow^*$ is a sequence of (zero or more) $\rightarrow$.

Lemma 11.4
If $\langle C, \theta \rangle \rightarrow_{\mathrm{I}}^{*} \langle \tilde{C}_{\mathrm{I}}, \tilde{\theta}_{\mathrm{I}} \rangle$ and $\langle C, \theta \rangle \rightarrow_{\mathrm{I}}^{*} \langle \tilde{C}'_{\mathrm{I}}, \tilde{\theta}'_{\mathrm{I}} \rangle$, then $\langle \tilde{C}_{\mathrm{I}}, \tilde{\theta}_{\mathrm{I}} \rangle = \langle \tilde{C}'_{\mathrm{I}}, \tilde{\theta}'_{\mathrm{I}} \rangle$.

Proof By induction on the length of $\rightarrow_{\mathrm{I}}^{*}$. If $\rightarrow_{\mathrm{I}}$ is not applied at all, the lemma holds because $\langle C, \theta \rangle = \langle \tilde{C}_{\mathrm{I}}, \tilde{\theta}_{\mathrm{I}} \rangle$ and $\langle C, \theta \rangle = \langle \tilde{C}'_{\mathrm{I}}, \tilde{\theta}'_{\mathrm{I}} \rangle$.

Assume $\rightarrow_{\mathrm{I}}$ is applied more than once. Let the first constraint applied in $\langle C, \theta \rangle \rightarrow_{\mathrm{I}}^{*} \langle \tilde{C}_{\mathrm{I}}, \tilde{\theta}_{\mathrm{I}} \rangle$ be $c : (p = x) \Rightarrow (p' = x')$. Since c is applicable to $\langle C, \theta \rangle$, we have $\theta(p) = x$. Remember that $p = x$, as well as the conclusion of all the constraints, has the form $d = \mathsf{true}$ or $s = \mathsf{false}$. This means that $\theta(p) = x$ remains valid throughout all the transformation beginning from $\langle C, \theta \rangle$. In particular, $\theta(p) = x$ holds throughout the second sequence $\langle C, \theta \rangle \rightarrow_{\mathrm{I}}^{*} \langle \tilde{C}'_{\mathrm{I}}, \tilde{\theta}'_{\mathrm{I}} \rangle$. Thus, we know that c is applied somewhere in this sequence. Wherever it is, we can apply c at the beginning of the sequence by changing the order of application using Lemma 11.3 as many times as necessary without changing the final result as in: $\langle C, \theta \rangle \xrightarrow{c}_{\mathrm{I}} \langle C_1, \theta_1 \rangle \rightarrow_{\mathrm{I}}^{*} \langle \tilde{C}'_{\mathrm{I}}, \tilde{\theta}'_{\mathrm{I}} \rangle$. Since we have $\langle C, \theta \rangle \xrightarrow{c}_{\mathrm{I}} \langle C_1, \theta_1 \rangle \rightarrow_{\mathrm{I}}^{*} \langle \tilde{C}_{\mathrm{I}}, \tilde{\theta}_{\mathrm{I}} \rangle$, we can use the induction hypothesis to show $\langle \tilde{C}_{\mathrm{I}}, \tilde{\theta}_{\mathrm{I}} \rangle = \langle \tilde{C}'_{\mathrm{I}}, \tilde{\theta}'_{\mathrm{I}} \rangle$. ■

There exists a trivial solution to step 1: $s = \mathsf{false}$ for all s and $d = \mathsf{true}$ for all d. Since step 1 starts from the smallest solution and increases it monotonically, and the number of possible solutions is finite, it will eventually reach the minimal solution that satisfies constraints. Thus, we have the following theorem:

Theorem 11.1
The annotation obtained from the solution $\tilde{\theta}_{\mathrm{I}}$ of step 1 satisfies the first condition of the best annotation found in Definition 9.1.

Let us move on to step 2. At the beginning of step 2, we reset the value of s to $s = \mathsf{false}$ but use the same d at the end of step 1. The next lemma shows that the value of d no longer changes during step 2.

Lemma 11.5
$\theta_{\mathrm{II}}(d) = \tilde{\theta}_{\mathrm{II}}(d)$ for all d.

Proof We show by induction on the length of a transformation that $\theta(d) = \theta_{\mathrm{II}}(d)$ for all θ during step 2. If a transformation does not occur, we have $\theta_{\mathrm{II}}(d) = \tilde{\theta}_{\mathrm{II}}(d)$ and the lemma trivially holds. Assume that a transformation proceeds from $\langle C^{*}_{init}, \theta_{\mathrm{II}} \rangle$ to $\langle C, \theta \rangle$, and the lemma holds up to here: $\theta(d) = \theta_{\mathrm{II}}(d)$ for all d. We will show that after one more transformation $\langle C, \theta \rangle \xrightarrow{c} \langle C', \theta' \rangle$, the lemma

still holds: $\theta'(d) = \theta(d) = \theta_{\mathrm{II}}(d)$.

Because the constraints used in step 2 are contraposition of the constraints in step 1, they have the form $(p_1 = x_1) \Rightarrow (p_2 = x_2)$ where $p_i = x_i$ is either $s = \mathsf{true}$ or $d = \mathsf{false}$. Among them, only two forms of constraints can change the value of d: $(d_1 = \mathsf{false}) \Rightarrow (d_2 = \mathsf{false})$ or $(s = \mathsf{true}) \Rightarrow (d = \mathsf{false})$. We consider both of the cases in order.

Since $\tilde{\theta}_{\mathrm{I}}$ satisfies C_{init} (Lemma 11.2), it also satisfies C^*_{init}. Because $\tilde{\theta}_{\mathrm{I}}(d) = \theta_{\mathrm{II}}(d)$ and $\theta(d) = \theta_{\mathrm{II}}(d)$ (from the induction hypothesis) for all d, θ satisfies all the constraints of the form $(d_1 = \mathsf{false}) \Rightarrow (d_2 = \mathsf{false})$ in C^*_{init}. Namely, it is (1) $\theta(d_1) = \mathsf{true}$, or (2) $\theta(d_1) = \mathsf{false}$ and $\theta(d_2) = \mathsf{false}$. In the former case, the constraint is not applicable. In the latter case, we have $\theta' = \theta[d_2 \mapsto \mathsf{false}] = \theta$. This concludes the case for $(d_1 = \mathsf{false}) \Rightarrow (d_2 = \mathsf{false})$.

Let us consider the other case: $(s = \mathsf{true}) \Rightarrow (d = \mathsf{false})$. To apply this constraint, we need to have $\theta(s) = \mathsf{true}$, but $s = \mathsf{false}$ for all s at the beginning of step 2. The value of s becomes true when (1) $\theta(d) = \mathsf{false}$ for some d and $(d = \mathsf{false}) \Rightarrow (s = \mathsf{true})$ is applied, or (2) the value of s is propagated through the application of $(s = \mathsf{true}) \Rightarrow (s' = \mathsf{true})$. Assume that $\theta_{\mathrm{II}}(d_0) = \mathsf{false}$ (at the beginning of step 2) and that the application of constraints proceeds as $(d_0 = \mathsf{false}) \Rightarrow (s_0 = \mathsf{true})$, $(s_{i-1} = \mathsf{true}) \Rightarrow (s_i = \mathsf{true})$ $(1 \leq i \leq n)$, $\theta(s_n) = \mathsf{true}$, and finally $(s_n = \mathsf{true}) \Rightarrow (d_n = \mathsf{false})$ to change the value of d_n. Then, we can show (see below) that the value of d_n is already false and the application of the last constraint does not change the value of d. Thus, we have $\theta' = \theta[d_n \mapsto \mathsf{false}] = \theta$ and the case for $(s = \mathsf{true}) \Rightarrow (d = \mathsf{false})$ has been proven.

The proof for $\theta(d_n) = \mathsf{false}$ is shown from the fact that the final solution $\tilde{\theta}_{\mathrm{I}}$ of step 1 satisfies all the constraints (Lemma 11.2). Assume (for contradiction) that $\theta(d_n) = \mathsf{true}$. From the induction hypothesis, we have $\tilde{\theta}_{\mathrm{I}}(d_n) = \theta_{\mathrm{II}}(d_n) = \theta(d_n) = \mathsf{true}$. Consider the contraposition of the above sequence of constraints: $(d_n = \mathsf{true}) \Rightarrow (s_n = \mathsf{false})$, $(s_i = \mathsf{false}) \Rightarrow (s_{i-1} = \mathsf{false})$ $(n \geq i \geq 1)$, $(s_0 = \mathsf{false}) \Rightarrow (d_0 = \mathsf{true})$. Since these constraints were satisfied at the end of step 1, we have $\tilde{\theta}_{\mathrm{I}}(d_0) = \mathsf{true}$, which contradicts with the assumption $\tilde{\theta}_{\mathrm{I}}(d_0) = \theta_{\mathrm{II}}(d_0) = \mathsf{false}$ in the previous paragraph. ■

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.[*13]

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

In step 2, we define the following ordering of solutions: $\theta \leq_{II} \theta' \stackrel{\text{def}}{\iff} \forall s.\theta(s) \leq \theta'(s) \wedge \forall d.\theta(d) = \theta'(d)$. Notice that inequality on s is opposite from step 1. Following almost the same discussion as in step 1, we can show the following four lemmas.

Lemma 11.6
If $\langle C, \theta \rangle \rightarrow_{II} \langle C', \theta' \rangle$, then $\theta \leq_{II} \theta'$. (The solution does not decrease during step 2.)

Lemma 11.7
The final solution $\tilde{\theta_{II}}$ of step 2 satisfies all the constraints in C^*_{init} (and hence C_{init}).

Lemma 11.8
If $\langle C_0, \theta_0 \rangle \stackrel{c_1}{\rightarrow}_{II} \langle C_1, \theta_1 \rangle$ and $\langle C_0, \theta_0 \rangle \stackrel{c_2}{\rightarrow}_{II} \langle C_2, \theta_2 \rangle$, then there exists $\langle C_3, \theta_3 \rangle$ such that $\langle C_1, \theta_1 \rangle \stackrel{c_2}{\rightarrow}_{II} \langle C_3, \theta_3 \rangle$ and $\langle C_2, \theta_2 \rangle \stackrel{c_1}{\rightarrow}_{II} \langle C_3, \theta_3 \rangle$. (The ordering of the application of constraints does not affect the final result.)

Lemma 11.9
If $\langle C, \theta \rangle \rightarrow^*_{II} \langle \tilde{C_{II}}, \tilde{\theta_{II}} \rangle$ and $\langle C, \theta \rangle \rightarrow^*_{II} \langle \tilde{C'_{II}}, \tilde{\theta'_{II}} \rangle$, then $\langle \tilde{C_{II}}, \tilde{\theta_{II}} \rangle = \langle \tilde{C'_{II}}, \tilde{\theta'_{II}} \rangle$. (The final solution of step 2 is unique.)

There exists a trivial solution to step 2: the solution of step 1 satisfies all the constraints. Since step 2 starts from the smallest solution and increases it monotonically, and the number of possible solutions is finite, it will eventually reach the minimal solution that satisfies constraints. Because the value of d does not change during step 2, the first condition of Definition 9.1 remains satisfied. Thus, we have the following theorem:

Theorem 11.2
The annotation obtained from the solution $\tilde{\theta_{II}}$ of step 2 is the best annotation.

§12 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[20] 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[20], $O(N\alpha(N,N))$ 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 time 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.

§13 Related Work

Hornof, Consel, and Noyé[15] presented a binding-time analysis for C which handles both static and dynamic data structures. They first perform a forward analysis using the conventional binding-time lattice (S and D) to analyze each use of data structures (*use* binding-times). They then perform a backward analysis using the same binding-time lattice as we used in this paper ($\bot$, S, D, and B) to collect all the uses of each data structure and determine the binding-time of its definition (*definition* binding-times).

Although developed independently, our work can be seen as an extension of their work to functional languages. Unlike their analyses, however, we cannot perform forward and backward analyses separately, since they interfere in our monovariant framework. For example, consider again the following term: $(\lambda f.\, \mathit{cons}\ f\ (f@2))@(\lambda x.\, x+3)$. In Sect. 4, we saw that f is classified as B. Then the restriction on a function with the binding-time value B forces us to classify x as D and hence 2 as D. Notice that we can classify 2 as D only after we know that $\lambda x.\, x+3$ is classified as B. In other words, the definition binding-time of a data structure may change the use binding-time of other data.

For this reason, we perform a *combined* analysis where both forward and backward analyses are embedded as a form of constraints. Because of the contradictory requirements on s (described in Sect. 10), however, we need to separate the combined analysis into two: step 1 for d and step 2 for s.

Almost linear time algorithm for solving constraints was first presented by Henglein[14], and later refined by Bondorf and Jørgensen[5]. It was also applied to multi-level specialization by Glück and Jørgensen[12]. We basically followed the

same technique of using the union/find algorithm[20] to obtain almost linear time algorithm. Because of the introduction of both static and dynamic expressions, our algorithm is split into two steps.

Sperber[18] 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. Based on an offline partial evaluator, 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.

The propagation of both static and dynamic values is a common technique in an online specializers such as FUSE[17]. It is important not only for better specialization but also for preserving pointer equality[2]. 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)[6] is an efficient partial evaluation technique, where specialization is directed by the type of expressions. By systematically applying η-expansion[8] 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.

This paper is an extended version of the previous work[1] and presents soundness of type-checking rules, equivalence of type-checking rules and constraint generation rules, and the correctness of the constraint satisfaction algorithm.

§14 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, data structures can be statically accessed while they are residualized at the same time. Previously, such data structures were treated as completely dynamic, which prevented their components from being accessed statically. The technique presented in this paper effectively allows data structures to be lifted 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.

Acknowledgment I would like to thank anonymous referees for many comments. I am grateful to Robert Glück and Prof. Yoshihiko Futamura for organizing the enjoyable PEDAY colloquium.

References

1) Asai, K. "Binding-Time Analysis for Both Static and Dynamic Expressions," In A. Cortesi and G. Filé, editors, *Static Analysis (LNCS 1694)*, pp. 117–133 (September 1999).

2) 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).

3) 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).

4) 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).

5) 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).

6) Danvy, O. "Type-Directed Partial Evaluation," *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pp. 242–257 (January 1996).

7) Danvy, O., and A. Filinski "Abstracting Control," *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160 (June 1990).

8) 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).

9) 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).

10) 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).

11) Futamura, Y. "Partial evaluation of computation process – an approach to a compiler-compiler," *Systems, Computers, Controls*, Vol. 2, No. 5, pp. 45–50, (1971), reprinted as *Higher-Order and Symbolic Computation*, Vol. 12, No. 4, pp. 381–391, Kluwer Academic Publishers (December 1999).

12) 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).

13) Hatcliff, J. "An Introduction to Online and Offline Partial Evaluation Using a Simple Flowchart Language," In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann editors, *Partial Evaluation, Practice and Theory (LNCS 1706)*, pp. 20–82 (1999).

14) 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).

15) 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).

16) Jones, N. D., C. K. Gomard, and P. Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).

17) 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.

18) 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).

19) 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).

20) Tarjan, R. *Data Structures and Network Flow Algorithms*, Volume CMBS 44 of *Regional Conference Series in Applied Mathematics*, SIAM (1983).