Correctness of binding-time analysis

JENS PALSBERG
Computer Science Department, Aarhus University
Ny Munkegade, DK–8000 Aarhus C, Denmark
(*e-mail*:palsberg@daimi.aau.dk)

Abstract

A binding-time analysis is correct if it always produces *consistent* binding-time information. Consistency prevents partial evaluators from 'going wrong'. A sufficient and decidable condition for consistency, called well-annotatedness, was first presented by Gomard and Jones. In this paper we prove that a weaker condition implies consistency. Our condition is decidable, subsumes the one of Gomard and Jones, and was first studied by Schwartzbach and the present author. Our result implies the correctness of the binding-time analysis of Mogensen, and it indicates the correctness of the core of the binding-time analyses of Bondorf and Consel. We also prove that *all* partial evaluators will on termination have eliminated all 'eliminable'-marked parts of an input which satisfies our condition. This generalizes a result of Gomard. Our development is for the pure λ-calculus with explicit binding-time annotations.

Capsule review

Palsberg's paper defines a general notion of top-down partial evaluator which subsumes most published partial evaluators when restricted to the pure lambda calculus. A new notion of well-annotatedness is introduced, weaker than that of Gomard and Jones.

Modularizing a correctness proof of binding-time analysis is a nontrivial task that involves finding an interface between binding-time information and the subsequent specialization process.

The paper gives a well structured account of correctness of binding-time analysis for the pure λ-calculus via a careful decomposition of the problem. In essence the correctness requirement is that a partial evaluator should never 'go wrong' when blindly following binding-time information. The decomposition is as follows:

(1) The binding-time analysis is viewed as realizing a decidable *well-annotatedness* condition for λ-terms. (2) Well-annotatedness implies a *consistency* condition, relating to reductions of static redexes. (3) A certain family of partial evaluators which follow a simple *top-down* reduction strategy cannot 'go wrong' if their input is consistent.

The modularity of this argument not only simplifies the structure of the proof, but allows for some independent variation in the binding-time analysis and the partial evaluation phases, and facilitates comparisons (both formal and informal) with existing approaches.

1 Introduction

A partial evaluator is an implementation of Kleene's S_n^m theorem. When given a program and some of its expected input, a partial evaluator produces a so-called

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

residual program. The residual program will, when given the remaining input, produce the same result as would have the original program when given all of the input. The challenge for a partial evaluator is to produce residual programs that are significantly faster than the input programs. Partial evaluators can for example be used to generate compilers from interpreters (Bondorf, 1991; Gomard and Jones, 1991).

Most partial evaluators use a binding-time analysis in a pre-processing phase. A binding-time analysis annotates a program by marking as 'eliminable' those parts which may be evaluated during partial evaluation and by marking remaining parts as 'residual'. Partial evaluation then proceeds by attempting to evaluate the eliminable parts.

A binding-time analysis is correct if it always produces *consistent* binding-time information. Consistency prevents partial evaluators from 'going wrong'. A partial evaluator 'goes wrong' if it commits a so-called 'projection error' (Gomard and Jones, 1991), that is, if it trusts a part of the program to be of a particular form, of which it is not.

A sufficient and decidable condition for consistency, called well-annotatedness, was first presented by Gomard and Jones (1991). They proved that a particular partial evaluator cannot 'go wrong' when given a well-annotated program. Gomard also presented a binding-time analysis that always produces well-annotated programs (Gomard 1990, 1991).

In this paper we prove that a weaker condition than that of Gomard and Jones implies consistency. Our condition is decidable, subsumes the one of Gomard and Jones, and was first studied by Schwartzbach and the present author (Palsberg and Schwartzbach, 1992). Our result implies the correctness of the binding-time analysis of Mogensen (1992), and it indicates the correctness of the core of the binding-time analyses of Bondorf (1991) and Consel (1990). By 'the core' of an analysis we mean its restriction to the λ-calculus.

Finally, we prove that *all* partial evaluators will on termination have eliminated all 'eliminable'-marked parts of an input which satisfies our condition. This generalizes a result of Gomard who proved that a particular partial evaluator will on termination have eliminated all 'eliminable'-marked parts of a well-annotated program.

Our consistency criterion is true of the output of radically different binding-time analyses. The binding-time analysis of Gomard and Jones is based on so-called partial type inference. The analysis of Mogensen extends it with the use of recursive types. The binding-time analyses of Bondorf and Consel are based on an abstract interpretation called closure analysis.

Our results demonstrate that a partial evaluation strategy can be changed without affecting the correctness of the chosen binding-time analysis. We focus on a novel family of so-called *top-down* partial evaluators and prove that if such one is given a consistent input, then it cannot 'go wrong'.

Our development is for the pure λ-calculus with explicit binding-time annotations. This allows us to concentrate on the higher-order aspects of binding-time analysis.

In the following section we define the pure λ-calculus with explicit binding-time annotations, the so-called 2-level λ-calculus. In Section 3 we introduce the key

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

concept of consistent 2-level λ-terms, and we present the family of top-down partial evaluators for the 2-level λ-calculus. In Section 4 we recall the consistency condition studied by Schwartzbach and the present author. In Section 5 we state our results, and in Section 6 we prove the required lemmas. Finally, in Section 7 we conclude.

2 The 2-level λ-calculus

As a basis for discussing the input to binding-time analyses we start by recalling the pure λ-calculus (Barendregt, 1981).

Definition 1
The language of λ-terms is defined by the grammar:

$$\begin{array}{lll} E ::= & x & (\textit{variable}) \\ & \mid \lambda x.E & (\textit{abstraction}) \\ & \mid E_1 \mathbin{@} E_2 & (\textit{application}) \end{array}$$

An occurrence of $(\lambda x.E) \mathbin{@} E'$ is called a redex. The semantics is as usual given by the rewriting rule scheme:

$$(\lambda x.E) \mathbin{@} E' \rightarrow E[E'/x] \qquad \text{(beta-reduction)}$$

Here, $E[E'/x]$ denotes the term E with E' substituted for free occurrences of x (after renaming bound variables if necessary). We write $E_S \rightarrow^* E_T$ to denote that E_T has been obtained from E_S by 0 or more beta-reductions. A term without redexes is said to be in normal form. We use the convention that application associates to the left. □

To simplify matters, we will assume that the input to a binding-time analysis is just one λ-term that encodes both the term to be analyzed and the known input. For example, suppose we want to analyze the term E which takes its input through the free variables x and y. Suppose also that the value of x is known to be E' and that the value of y is unknown. We will then supply the analysis with the term $(\lambda x.E) \mathbin{@} E'$. Notice that y is free also in this term. Thus, free variables henceforth correspond to unknown input.

The output of a binding-time analysis can be presented as an *annotated* version of the analyzed term. In the annotated term, all residual abstractions and applications are underlined. The language of annotated terms is usually called a 2-level λ-calculus (Nielson and Nielson, 1988) and is defined as follows.

Definition 2
The language of 2-level λ-terms is defined by the grammar:

$$\begin{array}{lll} E ::= & x & (\textit{variable}) \\ & \mid \lambda x.E & (\textit{static abstraction}) \\ & \mid E_1 \mathbin{@_i} E_2 & (\textit{static application}) \\ & \mid \underline{\lambda} x.E & (\textit{dynamic abstraction}) \\ & \mid E_1 \mathbin{\underline{@}_i} E_2 & (\textit{dynamic application}) \end{array}$$

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

Notice that we require the application symbols to be indexed. The role of the indexes will be explained later. The semantics of 2-level λ-terms is given by the four rewriting rule schemes:

$$(\lambda x.E) \; @_i \; E' \rightarrow E[E'/x]$$
$$(\underline{\lambda} x.E) \; \underline{@}_i \; E' \rightarrow E[E'/x]$$
$$(\lambda x.E) \; \underline{@}_i \; E' \rightarrow E[E'/x]$$
$$(\underline{\lambda} x.E) \; @_i \; E' \rightarrow E[E'/x]$$

Thus, the semantics of 2-level λ-terms is essentially the same as that of λ-terms.

An occurrence of $(\lambda x.E) \; @_i \; E'$ *is called a* static *redex. An occurrence of* $(\underline{\lambda} x.E) \; \underline{@}_i \; E'$ *is called a* dynamic *redex. Occurrences of* $(\lambda x.E) \; \underline{@}_i \; E'$ *and* $(\underline{\lambda} x.E) \; @_i \; E'$ *are called* confused *redexes. A term without static redexes is in* static normal form. *If there is a reduction sequence from E to E′ and E′ is in static normal form, then E is said to have the static normal form E′.*

We write $E \rightarrow_s E'$ *to denote that E′ has been obtained from E by reducing a static redex. Such a reduction is called a static reduction. We write* $E_S \rightarrow_s^* E_T$ *to denote that* E_T *has been obtained from* E_S *by* 0 *or more static reductions. Such a reduction sequence is called a static reduction sequence. If there is a static reduction sequence from E to E′ and E′ is in static normal form, then E is said to have the static reduction normal form E′. Notice that if a term has a static reduction normal form, then it also has a static normal form. The opposite implication is false: if* $\Omega = (\lambda a.a \; @_1 \; a) \; @_2 \; (\lambda b.b \; @_3 \; b)$, *then* $(\underline{\lambda} x.y) \; \underline{@}_4 \; \Omega$ *has a static normal form but no static reduction normal form.*

The language of 2-level λ-terms is partially ordered by $\sqsubseteq$ *as follows. Given 2-level λ-terms E and E′,* $E \sqsubseteq E'$ *if and only if they are equal except for underlinings and E′ has the same and possibly more underlinings than E. For example,* $((\lambda x.x \; \underline{@}_1 \; y) \; @_2 \; z) \sqsubseteq ((\underline{\lambda} x.x \; \underline{@}_1 \; y) \; \underline{@}_2 \; z)$. *Notice that* $\sqsubseteq$ *admits greatest lower bounds for two terms that are equal except for underlinings.* □

Intuitively, 'static' means 'statically known', and 'dynamic' means 'not statically known'. Thus, the static entities are the eliminable parts, and the dynamic entities are the residual parts.

Definition 3
A partial evaluator for the 2-level λ-calculus implements a partial function from 2-level λ-terms to 2-level λ-terms by doing repeated reductions. If it is defined on an argument, then it yields a static normal form of the argument. □

If a partial evaluator is defined on an argument E, then we will say that the partial evaluation of E terminates. Notice that a partial evaluator may perform an arbitrary reduction sequence, as long as it gets rid of the static redexes. Most partial evaluators, however, use binding-time information to perform a sequence of only static reductions. The following observation expresses a Church–Rosser property of static reductions.

Fact 4
A 2-level λ-term has at most one static reduction normal form.

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

Proof
Consider the following translation F of 2-level λ-terms into λ-terms.

$$\begin{array}{rcl} F(x) & = & x \\ F(\lambda x.E) & = & \lambda x.F(E) \\ F(E_1 \mathbin{@_i} E_2) & = & F(E_1) \mathbin{@} F(E_2) \\ F(\underline{\lambda} x.E) & = & y_x \mathbin{@} F(E) \\ F(E_1 \mathbin{\underline{@}_i} E_2) & = & (z_i \mathbin{@} F(E_1)) \mathbin{@} F(E_2) \end{array}$$

Here y_x and z_i are disjoint families of fresh variables. They may be thought of as functions that emit code. Clearly, any 2-level λ-term E has at most one static reduction normal form if and only if $F(E)$ has at most one normal form. The result then follows from the Church–Rosser theorem for λ-terms. □

Thus, if two partial evaluations using only static reductions both yield static normal forms, then these static normal forms are equal. Notice that, in general, two partial evaluations *can* yield different static normal forms.

3 Consistency of 2-level λ-terms

We now define the key concept of consistent 2-level λ-terms. Our definition of consistency is *independent* of particular partial evaluators.

Definition 5
For a 2-level λ-term E, an application is said to be enabled *in E if it is a static application which is not a subterm of any static abstraction in E, and whose function part is not a static application. A 2-level λ-term is* well-formed *if all its enabled applications are static redexes. A 2-level λ-term is* consistent *if every static reduction sequence yields a well-formed 2-level λ-term.* □

For example, $((\underline{\lambda} x.x \mathbin{\underline{@}_1} y) \mathbin{@_2} z)$ is not well-formed. As another example, the term $((\lambda x.x \mathbin{\underline{@}_1} y) \mathbin{@_2} z)$ is well-formed and consistent, since $((\lambda x.x \mathbin{\underline{@}_1} y) \mathbin{@_2} z) \rightarrow_s z \mathbin{\underline{@}_1} y$ and $z \mathbin{\underline{@}_1} y$ is well-formed and in static normal form. Finally, $((\lambda x.x \mathbin{@_1} y) \mathbin{@_2} z)$ is well-formed but not consistent, since $((\lambda x.x \mathbin{@_1} y) \mathbin{@_2} z) \rightarrow_s z \mathbin{@_1} y$ and $z \mathbin{@_1} y$ is not well-formed.

Intuitively, an enabled application in a 2-level λ-term is located near the root of the syntax-tree for E; the application is 'above' every static abstraction. The intuition behind why the function part of an enabled application cannot be a static application is that we want the function part of an enabled application to be 'fully' evaluated. Notice that there may be more than one enabled application in a 2-level λ-term. Consider for example $(x \mathbin{@_1} y) \mathbin{@_2} ((\lambda z.z \mathbin{@_3} z) \mathbin{@_4} y)$, where both $@_1$ and $@_4$ are enabled, but $@_2$ and $@_3$ are not.

Fact 6
Consistency of 2-level λ-terms is preserved by static reduction.

Proof
Suppose E is a consistent 2-level λ-term. Suppose also that a static reduction of E yields E'. Consider then any static reduction sequence starting with E'. This

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

sequence can be prefixed by the static reduction from E to E', yielding a static reduction sequence starting with E. Since E is consistent, the reduction sequence from E yields a well-formed 2-level λ-term. □

We will now define a novel family of so-called *top-down* partial evaluators for the 2-level λ-calculus. In practice it is too inefficient to search for a static redex before every reduction. Instead, most partial evaluators trust that all enabled applications are static redexes. This will not 'go wrong' if the input is consistent.

Definition 7
A top-down *partial evaluator for the 2-level λ-calculus attempts only the reduction of enabled applications.* □

The set of consistent 2-level λ-terms yields an appropriate interface between a binding-time analysis and a top-down partial evaluator, as follows.

Fact 8
If a top-down partial evaluator is applied to a consistent 2-level λ-term, then it will not attempt to apply other than a static abstraction to an argument.

Proof
By induction on the length of the reduction sequence. □

We can then summarize the idea of a correct binding-time analysis.

Definition 9
A binding-time analysis of λ-terms is a function mapping λ-terms to 2-level λ-terms such that the output is an annotated version of the input. It is correct *if it always produces consistent 2-level λ-terms.* □

For example, the trivial binding-time analysis maps any λ-term to the same term where all abstractions and applications are underlined. It is clearly correct, since each output contains no static applications, hence no enabled applications.

Most published partial evaluators are top-down. For example, the partial evaluator studied by Gomard and Jones (1991) can be formulated in the following fashion:

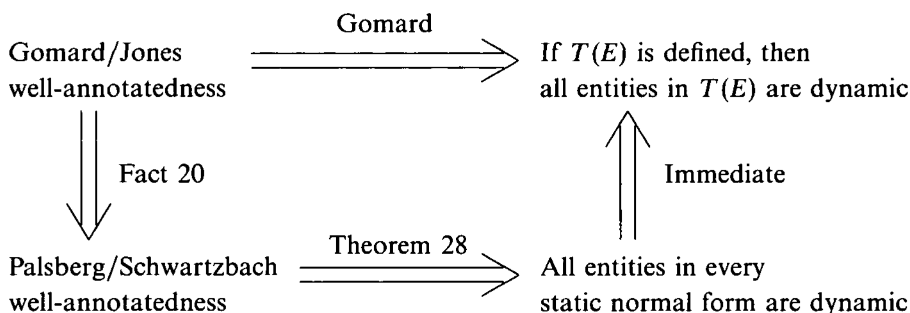
$$\begin{array}{rcl}
T(x) & = & x \\
T((\lambda x.E) @_i E_2) & = & T(E[E_2/x]) \\
T((E_1 @_j E_1') @_i E_2) & = & T(T(E_1 @_j E_1') @_i E_2) \\
T(\underline{\lambda} x.E) & = & \underline{\lambda} x.T(E) \\
T(E_1 \underline{@}_i E_2) & = & T(E_1) \underline{@}_i T(E_2)
\end{array}$$

The partial function T performs the partial evaluation by attempting to reduce enabled applications. If the calculation of $T(E)$ gets into a situation where T occurs but no rule applies then $T(E)$ is undefined. The cases where no rule applies are $T(\lambda x.E)$, $T(x @_i E)$, $T((\underline{\lambda} x.E) @_i E)$, and $T((E_1 \underline{@}_j E_2) @_i E)$. Furthermore, if the calculation of $T(E)$ diverges, then $T(E)$ is undefined. It is easy to prove by induction on the length of the calculation of $T(E)$ that if $T(E)$ is defined, then $T(E)$ is a static normal form of E.

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

Gomard (1991) proved (using a denotational description of T) that if a 2-level λ-term E is 'well-annotated' and if $T(E)$ is defined, then all abstractions and applications in $T(E)$ are dynamic.

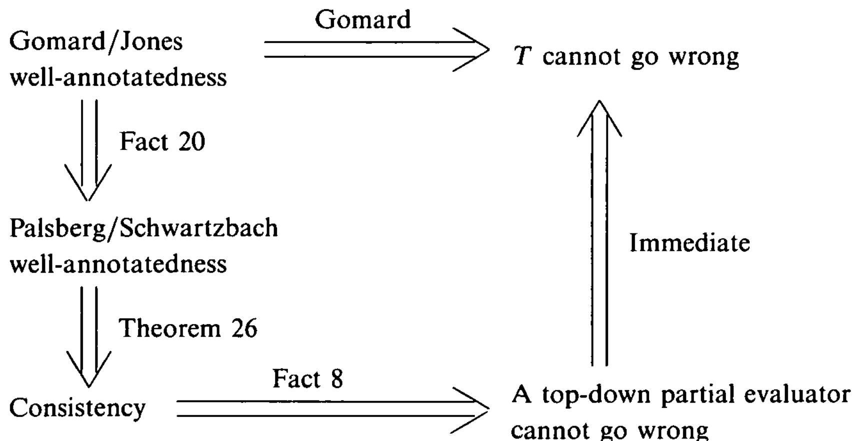
In this paper we generalize Gomard's result. We prove that if a 2-level λ-term satisfies a weaker condition than Gomard and Jones' well-annotatedness, then all abstractions and applications in every static normal form are dynamic. The weaker condition was first studied by Schwartzbach and the present author (Palsberg and Schwartzbach, 1992); we will call it Palsberg/Schwartzbach well-annotatedness. We thus prove that *all* partial evaluators will on termination have eliminated all static parts of a Palsberg/Schwartzbach well-annotated input. This result can be illustrated as follows:



We call our consistency condition 'well-annotatedness', although this word has already been used by Gomard and Jones. We believe that our terminology makes sense because all 2-level λ-terms that are well-annotated according to Gomard and Jones' definition will also be well-annotated according to our definition.

Gomard (1991) also proved that when given a well-annotated input, the above partial evaluator will not attempt to apply other than a static abstraction to an argument.

We also generalize that result by proving that Palsberg/Schwartzbach well-annotatedness implies consistency. This result can be illustrated as follows:



https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

4 Well-annotatedness of 2-level λ-terms

We will now define the Palsberg/Schwartzbach well-annotatedness condition on 2-level λ-terms. The condition is defined using constraint systems. For each 2-level λ-term E we will define a constraint system $WA(E)$. Then E is well-annotated if and only if $WA(E)$ is solvable. The symbol WA abbreviates 'well-annotated'.

Palsberg/Schwartzbach well-annotatedness is defined to capture the outputs of the binding-time analyses of Bondorf (1991) and Consel (1990), when restricted to the λ-calculus. We have not given a proof of this connection, however. These analyses are based on an abstract interpretation called *closure analysis* (Sestoft, 1989; Bondorf, 1991) (also called *control flow analysis* by Jones (1981) and Shivers (1991)). The *closures* of a term are simply the subterms corresponding to lambda abstractions. A closure analysis approximates for every subterm the set of possible closures to which it may evaluate (Jones, 1981; Sestoft, 1989; Bondorf, 1991; Shivers, 1991).

Bondorf used his analysis in the partial evaluator Similix, and Consel used his in the partial evaluator Schism. Bondorf/Consel's binding-time analysis may be understood as a closure analysis that in addition to closures also incorporates a special value Dyn. The intuition behind the constant Dyn is that a term with this binding time has unknown value; Dyn abbreviates Dynamic. Our constraint systems combine a closure analysis with the Dyn binding time.

Before defining the $WA(E)$ constraint systems, we first introduce a set of type variables, a set of binding-time values, and an auxiliary family of constraint systems. As explained later, type variables range over the set of binding-time values. Thus, it might better to use the phrase 'binding-time variable', rather than 'type variable'. We prefer 'type variable' because it is shorter and because a type might in general be understood as a property of a program part, for example a binding time value.

(The definition of the consistency criterion that was given by Palsberg and Schwartzbach (1992) is slightly different but equivalent to the one given here; the difference helps simplifying our proofs.)

Definition 10
A type variable is of one of the forms $\llbracket x \rrbracket$, $\llbracket \lambda x \rrbracket$, $\llbracket @_i \rrbracket$, $\llbracket \underline{\lambda} x \rrbracket$, *and* $\llbracket \underline{@}_i \rrbracket$. *Each 2-level λ-term is assigned a type variable by the function* var, *defined as follows:*

$$
\begin{aligned}
\mathsf{var}(x) &= \llbracket x \rrbracket \\
\mathsf{var}(\lambda x.E) &= \llbracket \lambda x \rrbracket \\
\mathsf{var}(E_1 \; @_i \; E_2) &= \llbracket @_i \rrbracket \\
\mathsf{var}(\underline{\lambda} x.E) &= \llbracket \underline{\lambda} x \rrbracket \\
\mathsf{var}(E_1 \; \underline{@}_i \; E_2) &= \llbracket \underline{@}_i \rrbracket
\end{aligned}
$$

For each 2-level λ-term E we define $\mathsf{Varset}(E)$ *to be the set of type variables assigned to subterms of E including E itself:*

$$\mathsf{Varset}(E) = \{\mathsf{var}(E') \mid E' \textit{ is a subterm or a bound variable of } E \textit{ (or both) } \}$$

Notice that $\mathsf{Varset}(E)$ *is finite for all E.* □

Notice that a type variable may be associated with more than one 2-level λ-term.

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

Notice also that the type variable assigned to a 2-level λ-term is determined from the root of the syntax-tree for that term. Finally, notice that different variables of the same name are assigned the same type variable. The key property of the definition of var is expressed by the following observation.

Fact 11
If $E \rightarrow^{*} E'$, then $\mathsf{Varset}(E) \supseteq \mathsf{Varset}(E')$.

Before partial evaluation begins, we can obtain by appropriate renamings that all bound variables and application indexes are distinct. However, abstractions and applications may be copied during reduction. The key role of the application indexes is to maintain some distinction between the applications while guaranteeing that the Varset is decreasing during reduction.

Definition 12
Type variables range over the set D of binding-time values. The set D consists of the value Dyn *and all finite subsets of* LAMBDA. LAMBDA *is an infinite set of so-called 'lambda tokens', i.e. symbols of the form λx. The set D is partially ordered by $\leq$, as follows:*

1. $\mathsf{Dyn} \leq \mathsf{Dyn}$; *and*
2. *if d and d' are sets and $d \subseteq d'$, then $d \leq d'$.*

Notice that D is not *a lattice, since* Dyn *is incomparable to all other values.* □

We will now introduce an auxiliary family of constraint systems. They will all be (equivalent to) finite sets of Horn clauses over inequalities of the form $d \leq d'$, where d and d' are either type variables or elements of D. A *solution* of a constraint system assigns an element of D to each type variable such that all constraints are satisfied.

Convention 13
To express as a constraint that a type variable v must assume a set (*not* Dyn) in any solution, we will write the constraint $v \geq \emptyset$. This works since all sets are comparable to the empty set while Dyn is not.

Definition 14
Let E_0 be a 2-level λ-term. We define a constraint system $C_{E_0}(E)$ for every subterm E of E_0 as follows. Let $C_{E_0}(E)$ be the collection of constraints for every subterm of E, generated from the syntax as follows.

Phrase:	*Basic constraints:*
$\lambda x.E$	$[\![\lambda x]\!] \geq \{\lambda x\}$
$E_1 \ @_i\ E_2$	$\mathsf{var}(E_1) \geq \emptyset$
$\underline{\lambda} x.E$	$[\![\underline{\lambda} x]\!] = [\![x]\!] = \mathsf{var}(E) = \mathsf{Dyn}$
$E_1 \ \underline{@}_i\ E_2$	$\mathsf{var}(E_1) = \mathsf{var}(E_2) = [\![\underline{@}_i]\!] = \mathsf{Dyn}$

Phrase:	*Connecting constraints:*
$E_1 \ @_i\ E_2$	For every $\lambda x.E$ in E_0, if $\{\lambda x\} \leq \mathsf{var}(E_1)$ then $\mathsf{var}(E_2) \leq [\![x]\!] \wedge [\![@_i]\!] \geq \mathsf{var}(E)$

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

The set of type variables used in $C_{E_0}(E)$ *is a subset of* $\mathsf{Varset}(E)$. *(Free variables need not be mentioned in any constraint).* □

Fact 15

If $C_{E_0}(E)$ has solution L and E' is a subterm of E, then $C_{E_0}(E')$ has solution L.

Proof

Immediate, since $C_{E_0}(E')$ is a subset of $C_{E_0}(E)$. □

Fact 16

If $C_{E_0}(E)$ has solution L, then no redex in E is confused.

Proof

A redex is confused if it is of either of the forms $(\lambda x.E) \mathbin{\underline{@}_i} E'$ and $(\underline{\lambda} x.E) \mathbin{@_i} E'$. In the first case, we get both $L[\![\lambda x]\!] \geq \{\lambda x\}$ and $L[\![\lambda x]\!] = \mathsf{Dyn}$, a contradiction. In the second case, we get both $L[\![\underline{\lambda} x]\!] = \mathsf{Dyn}$ and $L[\![\underline{\lambda} x]\!] \geq \emptyset$, also a contradiction. □

The constraint system in effect combines a closure analysis with a binding-time analysis. As a conceptual aid, the constraints of $C_{E_0}(E)$ are grouped into *basic* and *connecting* constraints. The connecting constraints reflect the relationship between formal and actual arguments and results. The condition $\{\lambda x\} \leq \mathsf{var}(E_1)$ states that the two guarded inequalities are relevant only if a closure denoted by λx is a possible result of E_1. Note that Dyn is not a possible result of E_1 because of the basic constraint $\mathsf{var}(E_1) \geq \emptyset$.

Definition 17

Let E_0 *be a 2-level* λ*-term. We define the constraint system* $WA(E_0)$ *to be the union of* $C_{E_0}(E_0)$ *and the following constraints:*

- *For all free variables* x *of* E_0*: the constraint* $[\![x]\!] = \mathsf{Dyn}$*; and*
- *The constraint* $\mathsf{var}(E_0) = \mathsf{Dyn}$.

The set of type variables used in $WA(E_0)$ *is* $\mathsf{Varset}(E)$. □

Fact 18

If $WA(E_0)$ has solution L, then $C_{E_0}(E_0)$ has solution L.

Proof

Immediate, since $C_{E_0}(E_0)$ is a subset of $WA(E_0)$. □

The extra constraints in $WA(E_0)$ of the form $[\![x]\!] = \mathsf{Dyn}$ reflect that the free variables of E_0 correspond to unknown input. The extra constraint $\mathsf{var}(E_0) = \mathsf{Dyn}$ reflects that a partial evaluator is supposed to produce a residual program.

We now define the key concept of well-annotated 2-level λ-terms.

Definition 19

A 2-level λ*-term* E *is well-annotated if and only if* $WA(E)$ *is solvable.* □

The following observation justifies our use of the word 'well-annotatedness' for our consistency condition.

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

Fact 20

If a 2-level λ-term is well-annotated according to Gomard and Jones' definition, then so is it according to ours.

Proof

See Palsberg and Schwartzbach (1992). □

Strong static normalization means that every static reduction sequence is finite. Well-annotatedness does *not* imply strong static normalization, as expressed by the following observation.

Fact 21

Well-annotated 2-level λ-terms may yield infinite static reduction sequences.

Proof

Let $\Omega = (\lambda a.a\ @_1\ a)\ @_2\ (\lambda b.b\ @_3\ b)$. Then, $(\lambda x.y)\ @_4\ \Omega$ is well-annotated. □

The set of annotated versions of a λ-term is partially ordered by $\sqsubseteq$. For every λ-term, this set has a least well-annotated element, as expressed by the following observation.

Fact 22

For all λ-terms, there is a $\sqsubseteq$-least well-annotated version.

Proof

See Palsberg and Schwartzbach (1992). □

Mogensen's binding-time analysis produces 2-level λ-terms that are what we here will call Mogensen well-annotated. This notion of well-annotatedness arises by extending the Gomard/Jones approach with recursive types. It follows from Palsberg and Schwartzbach (1992) that if a 2-level λ-term is Mogensen well-annotated, then it is also Palsberg/Schwartzbach well-annotated.

To prove the correctness of the binding-time analysis of Mogensen and the core of the binding-time analysis of Gomard, we need to show that well-annotatedness implies consistency. This will also indicate the correctness of the core of the binding-time analyses of Bondorf and Consel, since we believe (but have not proved) that these analyses produce the $\sqsubseteq$-least well-annotated version of a λ-term.

5 The correctness theorems

We will now state the two promised theorems about our consistency condition. First we need three lemmas which will be proved in the following section. In the statement of the lemmas, we assume that L assigns an element of D to each variable in $\mathsf{Varset}(E_0)$, where E_0 is a 2-level λ-term.

The first lemma expresses that in certain 2-level λ-terms, all abstractions and applications are dynamic.

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

Lemma 23

Suppose E is in static normal form, that $C_{E_0}(E)$ has solution L, and that all free variables x in E has $L[\![x]\!] = \mathsf{Dyn}$. If $L(\mathsf{var}(E)) = \mathsf{Dyn}$, then all abstractions and applications in E are dynamic.

The second lemma expresses a subject-reduction property of reductions.

Lemma 24

If $C_{E_0}(E_S)$ has solution L and $E_S \rightarrow E_T$, then $C_{E_0}(E_T)$ has solution L and $L(\mathsf{var}(E_S)) \geq L(\mathsf{var}(E_T))$.

The third lemma is a substitution lemma which is used in the proof of lemma 24.

Lemma 25

Suppose $C_{E_0}(E), C_{E_0}(U_1), \ldots, C_{E_0}(U_n)$ all have solution L and that the free variables of E are among $x_1, \ldots, x_n$. If $L(\mathsf{var}(U_i)) \leq L[\![x_i]\!]$, then $C_{E_0}(E[U_i/x_i])$ has solution L and $L(\mathsf{var}(E)) \geq L(\mathsf{var}(E[U_i/x_i]))$.

The two correctness theorems can then be stated and proved as follows.

Theorem 26

A well-annotated 2-level λ-term is consistent.

Proof

Suppose E_0 is a well-annotated 2-level λ-term, and that $E_0 \rightarrow_s^* E$. Then $WA(E_0)$ is solvable, so suppose it has solution L. By fact 18, also $C_{E_0}(E_0)$ has solution L. By lemma 24 and an induction on the length of the reduction sequence, $C_{E_0}(E)$ has solution L and $L(\mathsf{var}(E)) = \mathsf{Dyn}$. Since free variables cannot be created during beta-reduction, we also get that all free variables x in E has $L[\![x]\!] = \mathsf{Dyn}$.

We need to show that E is well-formed, so let $E_1 \mathbin{@_i} E_2$ be any enabled application in E. Then $E_1 \mathbin{@_i} E_2$ is not a subterm of any static abstraction in E, and E_1 is not a static application. It is sufficient to show that E_1 is a static abstraction.

Suppose for a contradiction that E_1 is either a variable, a dynamic abstraction, or a dynamic application. In the last two cases we have $L(\mathsf{var}(E_1)) = \mathsf{Dyn}$, contradicting $L(\mathsf{var}(E_1)) \geq \emptyset$. Consider then the case where E_1 is a variable. Since E_1 is not a proper subterm of any static abstraction in E, then E_1 is either free in E or it is bound in a dynamic abstraction. Hence, $L(\mathsf{var}(E_1)) = \mathsf{Dyn}$, contradicting $L(\mathsf{var}(E_1)) \geq \emptyset$. □

Recall that by 'the core' of an analysis we mean its restriction to the λ-calculus.

Corollary 27

The core of the binding-time analysis of Gomard (1990) and the binding-time analysis of Mogensen (1992) are correct.

Proof

Both produce well-annotated 2-level λ-terms (Palsberg and Schwartzbach, 1992). The conclusion then follows from Theorem 26 and Definition 9. □

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

Theorem 28
If a 2-level λ-term is well-annotated, then all abstractions and applications in every static normal form are dynamic.

Proof
Suppose E_0 is a well-annotated 2-level λ-term. Suppose also that E_0 has static normal form E. Then $WA(E_0)$ is solvable, so suppose it has solution L. By fact 18, also $C_{E_0}(E_0)$ has solution L. By Lemma 24 and an induction on the length of the reduction sequence, $C_{E_0}(E)$ has solution L and $L(\mathsf{var}(E)) = \mathsf{Dyn}$. Since free variables cannot be created during beta-reduction, we also get that all free variables x in E has $L[\![x]\!] = \mathsf{Dyn}$. By Lemma 23, all abstractions and applications in E are dynamic.
□

6 The Proofs

Throughout this section, we assume that L assigns an element of D to each variable in $\mathsf{Varset}(E_0)$. We will drop the subscript in $C_{E_0}(E)$ and simply write $C(E)$. We will repeat the statement of the three lemmas (omitting subscripts).

Lemma 23 (*restated*) Suppose E is in static normal form, that $C(E)$ has solution L, and that all free variables x in E has $L[\![x]\!] = \mathsf{Dyn}$. If $L(\mathsf{var}(E)) = \mathsf{Dyn}$, then all abstractions and applications in E are dynamic.

Proof
The proof has two parts: first we prove that all applications are dynamic and then we prove that all abstractions are dynamic.

For the first part of the proof, suppose for a contradiction that static applications do occur in E. We can then choose a subterm of E of the form $E_1 \mathbin{@_i} E_2$ so that it is not a subterm of any static abstraction of E and so that E_1 is not a static application. (The existence of $E_1 \mathbin{@_i} E_2$ is easily proved by induction on the structure of E). It is now sufficient to show that there are no possible forms of E_1. There are four cases.

First, E_1 cannot be a static abstraction, since E is in static normal form.

Second, E_1 cannot be a dynamic abstraction or a dynamic application, since they would both imply $L(\mathsf{var}(E_1)) = \mathsf{Dyn}$, contradicting $L(\mathsf{var}(E_1)) \geq \emptyset$.

Third, E_1 cannot be a variable x, since x would be either free in E or bound by a dynamic abstraction and thus have $L(\mathsf{var}(x)) = \mathsf{Dyn}$, contradicting $L(\mathsf{var}(E_1)) \geq \emptyset$.

Fourth, E_1 cannot be a static application, by assumption.

Thus, there are no possible forms of E_1, contradicting the existence of $E_1 \mathbin{@_i} E_2$.

For the second part of the proof, suppose for a contradiction that static abstractions do occur in E. We can then choose a subterm of E of the form $\lambda x.E'$ so that it is not a proper subterm of any other static abstraction in E. It is sufficient to show that this is impossible. There are four cases.

First, $\lambda x.E'$ cannot equal E, since $L[\![\lambda x]\!] \geq \{\lambda x\}$, contradicting $L(\mathsf{var}(E)) = \mathsf{Dyn}$.

Second, $\lambda x.E'$ cannot be the function or argument part of a static application since we know from the first part of the proof that they don't occur in E.

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

Third, $\lambda x.E'$ cannot be the function or argument part of a dynamic application, or the body of a dynamic abstraction, since that would imply $L[\![\lambda x]\!] = \mathsf{Dyn}$, contradicting $L[\![\lambda x]\!] \geq \{\lambda x\}$.

Fourth, $\lambda x.E'$ cannot be the body of a static abstraction, by assumption.

Thus, $\lambda x.E'$ cannot appear anywhere, contradicting its existence. □

Lemma 24 (*restated*) If $C(E_S)$ has solution L and $E_S \rightarrow E_T$, then $C(E_T)$ has solution L and $L(\mathsf{var}(E_S)) \geq L(\mathsf{var}(E_T))$.

Proof

We proceed by induction on the structure of E_S. In the base case, consider x. The conclusion is immediate since x is in normal form.

In the induction step, consider first $\lambda x.E'$. Suppose $\lambda x.E' \rightarrow \lambda x.E''$. Clearly, $L(\mathsf{var}(\lambda x.E')) = L(\mathsf{var}(\lambda x.E'')) = L[\![\lambda x]\!]$. By the induction hypothesis, $C(E'')$ has solution L. Since $C(\lambda x.E')$ has solution L, $L[\![\lambda x]\!] \geq \{\lambda x\}$. Thus, $C(\lambda x.E'')$ has solution L.

Consider then $E_1 \,@_i\, E_2$. There are three cases.

Suppose $E_1 \,@_i\, E_2 \rightarrow E_1' \,@_i\, E_2$. Clearly, $L(\mathsf{var}(E_1 \,@_i\, E_2)) = L(\mathsf{var}(E_1' \,@_i\, E_2)) = L[\![@_i]\!]$. By the induction hypothesis, $C(E_1')$ has solution L and also $L(\mathsf{var}(E_1)) \geq L(\mathsf{var}(E_1'))$. We then need to show that L is a solution of the basic and the connecting constraint for $E_1' \,@_i\, E_2$, that is: $L(\mathsf{var}(E_1')) \geq \emptyset$ and for every $\lambda x.E$ in E_0, if $\{\lambda x\} \leq L(\mathsf{var}(E_1'))$ then $L(\mathsf{var}(E_2)) \leq L[\![x]\!] \wedge L[\![@_i]\!] \geq L(\mathsf{var}(E))$. Both immediately follow from $L(\mathsf{var}(E_1)) \geq \emptyset$ and for every $\lambda x.E$ in E_0, if $\{\lambda x\} \leq L(\mathsf{var}(E_1))$ then $L(\mathsf{var}(E_2)) \leq L[\![x]\!] \wedge L[\![@_i]\!] \geq L(\mathsf{var}(E))$ and $L(\mathsf{var}(E_1)) \geq L(\mathsf{var}(E_1'))$, and the transitivity of $\leq$. Thus, $C(E_1' \,@_i\, E_2)$ has solution L.

Suppose then that $E_1 \,@_i\, E_2 \rightarrow E_1 \,@_i\, E_2'$. This case has a similar proof, we omit the details.

Suppose then that $E_1 = \lambda x.E$ and that $E_1 \,@_i\, E_2 \rightarrow E[E_2/x]$. Since $C(E_1 \,@_i\, E_2)$ has solution L, we have $\{\lambda x\} \leq L[\![\lambda x]\!] = L(\mathsf{var}(\lambda x.E)) = L(\mathsf{var}(E_1))$. Thus we also have $L(\mathsf{var}(E_2)) \leq L[\![x]\!] \wedge L[\![@_i]\!] \geq L(\mathsf{var}(E))$. The conclusion then follows from lemma 25 and the transitivity of $\leq$.

Consider then $\underline{\lambda} x.E'$. Suppose $\underline{\lambda} x.E' \rightarrow \underline{\lambda} x.E''$. Clearly, we have $L(\mathsf{var}(\underline{\lambda} x.E')) = L(\mathsf{var}(\underline{\lambda} x.E'')) = L[\![\underline{\lambda} x]\!]$. By the induction hypothesis, $C(E'')$ has solution L, and $L(\mathsf{var}(E')) \geq L(\mathsf{var}(E''))$. We need to show $L[\![\underline{\lambda} x]\!] = L[\![x]\!] = L(\mathsf{var}(E'')) = \mathsf{Dyn}$. This follows from $L[\![\underline{\lambda} x]\!] = L[\![x]\!] = L(\mathsf{var}(E')) = \mathsf{Dyn}$ and $L(\mathsf{var}(E')) \geq L(\mathsf{var}(E''))$. Thus, $C(\underline{\lambda} x.E'')$ has solution L.

Consider then $E_1 \,\underline{@}_i\, E_2$. There are three cases.

Suppose $E_1 \,\underline{@}_i\, E_2 \rightarrow E_1' \,\underline{@}_i\, E_2$. Clearly, $L(\mathsf{var}(E_1 \,\underline{@}_i\, E_2)) = L(\mathsf{var}(E_1' \,\underline{@}_i\, E_2)) = L[\![\underline{@}_i]\!]$. By the induction hypothesis, $C(E_1')$ has solution L, and also $L(\mathsf{var}(E_1)) \geq L(\mathsf{var}(E_1'))$. We need to show $L(\mathsf{var}(E_1')) = L(\mathsf{var}(E_2)) = L[\![\underline{@}_i]\!] = \mathsf{Dyn}$. This follows from $L(\mathsf{var}(E_1)) = L(\mathsf{var}(E_2)) = L[\![\underline{@}_i]\!] = \mathsf{Dyn}$ and $L(\mathsf{var}(E_1)) \geq L(\mathsf{var}(E_1'))$. Thus, $C(E_1' \,\underline{@}_i\, E_2)$ has solution L.

Suppose then that $E_1 \,\underline{@}_i\, E_2 \rightarrow E_1 \,\underline{@}_i\, E_2'$. This case has a similar proof, we omit the details.

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

Suppose then that $E_1 = \underline{\lambda}x.E$ and that $E_1 \ \underline{@}_i \ E_2 \rightarrow E[E_2/x]$. Since $C(E_1 \ \underline{@}_i \ E_2)$ has solution L, we have $\mathsf{Dyn} = L\llbracket\underline{\lambda}x\rrbracket = L\llbracket x\rrbracket = L(\mathsf{var}(E)) = L(\mathsf{var}(E_1)) = L(\mathsf{var}(E_2)) = L\llbracket\underline{@}_i\rrbracket$. The conclusion then follows from lemma 25.

Finally note that by fact 16, no redex in E_S is confused. □

Lemma 25 (*restated*) Suppose $C(E), C(U_1), \ldots, C(U_n)$ all have solution L and that the free variables of E are among $x_1, \ldots, x_n$. If $L(\mathsf{var}(U_i)) \leq L\llbracket x_i\rrbracket$, then $C(E[U_i/x_i])$ has solution L and $L(\mathsf{var}(E)) \geq L(\mathsf{var}(E[U_i/x_i]))$.

Proof

We proceed by induction on the structure of E. In the base case, consider x. Since x is being substituted by a term U where $C(U)$ has solution L and $L(\mathsf{var}(U)) \leq L\llbracket x\rrbracket$, the conclusion is immediate.

In the induction step, consider first $\lambda y.E'$. Clearly, we have that $L(\mathsf{var}(\lambda y.E')) = L(\mathsf{var}((\lambda y.E')[U_i/x_i])) = L\llbracket\lambda y\rrbracket$. Moreover, $C((\lambda y.E')[U_i/x_i]) = C(\lambda y.(E'[U_i/x_i]))$. Assume now that y does *not* occur free in any U_i (this can be obtained by the renaming of variables). By the induction hypothesis, if we have a V such that $C(V)$ has solution L and $L(\mathsf{var}(V)) \leq L\llbracket y\rrbracket$, then $C(E'[U_i/x_i, V/y])$ has solution L and $L(\mathsf{var}(E')) \geq L(\mathsf{var}(E'[U_i/x_i, V/y]))$. By taking $V = y$ we get that $C(E'[U_i/x_i])$ has solution L and that $L(\mathsf{var}(E')) \geq L(\mathsf{var}(E'[U_i/x_i]))$. Finally, $L\llbracket\lambda y\rrbracket \geq \{\lambda y\}$, since $C(\lambda y.E')$ has solution L. Thus, $C((\lambda y.E')[U_i/x_i])$ has solution L.

Consider then $E_1 \ @_i \ E_2$. Clearly, $L(\mathsf{var}(E_1 \ @_i \ E_2)) = L(\mathsf{var}((E_1 \ @_i \ E_2)[U_i/x_i])) = L\llbracket @_i\rrbracket$. Moreover, $C((E_1 \ @_i \ E_2)[U_i/x_i]) = C((E_1[U_i/x_i]) \ @_i \ (E_2[U_i/x_i]))$. By the induction hypothesis, $C(E_1[U_i/x_i])$ and $C(E_2[U_i/x_i])$ have solution L, and furthermore $L(\mathsf{var}(E_1)) \geq L(\mathsf{var}(E_1[U_i/x_i]))$ and $L(\mathsf{var}(E_2)) \geq L(\mathsf{var}(E_2[U_i/x_i]))$. We then need to show that L is a solution of the basic and the connecting constraint for $(E_1 \ @_i \ E_2)[U_i/x_i]$, that is: $L(\mathsf{var}(E_1[U_i/x_i])) \geq \emptyset$ and for every $\lambda x.E$ in E_0, if $\{\lambda x\} \leq L(\mathsf{var}(E_1[U_i/x_i]))$ then $L(\mathsf{var}(E_2[U_i/x_i])) \leq L\llbracket x\rrbracket \wedge L\llbracket @_i\rrbracket \geq L(\mathsf{var}(E))$. Both immediately follow from: $L(\mathsf{var}(E_1)) \geq \emptyset$ and for every $\lambda x.E$ in E_0, if $\{\lambda x\} \leq L(\mathsf{var}(E_1))$ then $L(\mathsf{var}(E_2)) \leq L\llbracket x\rrbracket \wedge L\llbracket @_i\rrbracket \geq L(\mathsf{var}(E))$ and $L(\mathsf{var}(E_1)) \geq L(\mathsf{var}(E_1[U_i/x_i]))$ and $L(\mathsf{var}(E_2)) \geq L(\mathsf{var}(E_2[U_i/x_i]))$, and the transitivity of $\leq$. Thus, $C((E_1 \ @_i \ E_2)[U_i/x_i])$ has solution L.

Consider then $\underline{\lambda}y.E'$. Clearly, $L(\mathsf{var}(\underline{\lambda}y.E')) = L(\mathsf{var}((\underline{\lambda}y.E')[U_i/x_i])) = L\llbracket\underline{\lambda}y\rrbracket$. Moreover, $C((\underline{\lambda}y.E')[U_i/x_i]) = C(\underline{\lambda}y.(E'[U_i/x_i]))$. Assume now that y does *not* occur free in any U_i (this can be obtained by the renaming of variables). By the induction hypothesis, if we have a V such that $C(V)$ has solution L and $L(\mathsf{var}(V)) \leq L\llbracket y\rrbracket$, then $C(E'[U_i/x_i, V/y])$ has solution L and $L(\mathsf{var}(E')) \geq L(\mathsf{var}(E'[U_i/x_i, V/y]))$. By taking $V = y$ we get that $C(E'[U_i/x_i])$ has solution L and that $L(\mathsf{var}(E')) \geq L(\mathsf{var}(E'[U_i/x_i]))$. We then need to prove that $L\llbracket\underline{\lambda}y\rrbracket = L\llbracket y\rrbracket = L(\mathsf{var}(E'[U_i/x_i])) = \mathsf{Dyn}$. This follows from $L\llbracket\underline{\lambda}y\rrbracket = L\llbracket y\rrbracket = L(\mathsf{var}(E')) = \mathsf{Dyn}$ and from $L(\mathsf{var}(E')) \geq L(\mathsf{var}(E'[U_i/x_i]))$. Thus, $C((\underline{\lambda}y.E')[U_i/x_i])$ has solution L.

Finally, consider $E_1 \ \underline{@}_i \ E_2$. Here, $L(\mathsf{var}(E_1 \ \underline{@}_i \ E_2)) = L(\mathsf{var}((E_1 \ \underline{@}_i \ E_2)[U_i/x_i])) = L\llbracket\underline{@}_i\rrbracket$. Moreover, $C((E_1 \ \underline{@}_i \ E_2)[U_i/x_i]) = C((E_1[U_i/x_i]) \ \underline{@}_i \ (E_2[U_i/x_i]))$. By the induction hypothesis, $C(E_1[U_i/x_i])$ and $C(E_2[U_i/x_i])$ have solution L, and furthermore $L(\mathsf{var}(E_1)) \geq L(\mathsf{var}(E_1[U_i/x_i]))$ and $L(\mathsf{var}(E_2)) \geq L(\mathsf{var}(E_2[U_i/x_i]))$. We then need to prove that $L(\mathsf{var}(E_1[U_i/x_i])) = L(\mathsf{var}(E_2[U_i/x_i])) = L\llbracket\underline{@}_i\rrbracket = \mathsf{Dyn}$. This follows

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

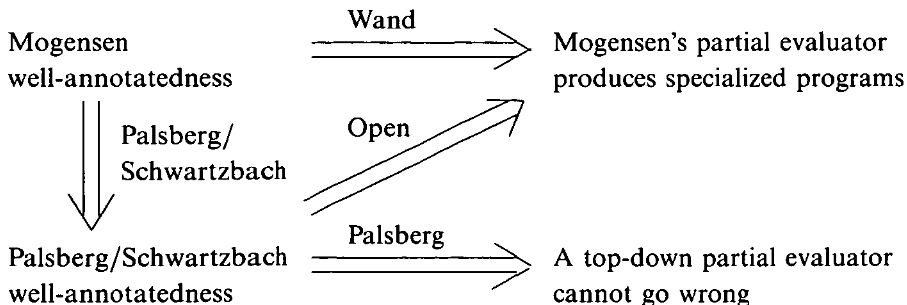
from $L(\mathsf{var}(E_1)) = L(\mathsf{var}(E_2)) = L[\![@_i]\!] = \mathsf{Dyn}$ and $L(\mathsf{var}(E_1)) \geq L(\mathsf{var}(E_1[U_i/x_i]))$ and $L(\mathsf{var}(E_2)) \geq L(\mathsf{var}(E_2[U_i/x_i]))$. Thus, $C((E_1 \; \underline{@}_i \; E_2)[U_i/x_i])$ has solution L. □

7 Conclusion

We have studied a notion of well-annotatedness which subsumes that of Gomard and Jones. We have obtained generalizations of two theorems of Gomard and we have obtained a proof of correctness of the core of the binding-time analysis of Gomard and the binding-time analysis of Mogensen. Our results also indicate the correctness of the binding-time analyses of Bondorf and Consel. We hope that our results can be used as lemmas when proving the correctness of other binding-time analyses. In particular we would like to relate our results to the binding-time analysis for typed λ-calculus of Nielson and Nielson (1988), and to the related results of Schmidt (1987). Notice here that Nielson and Nielson to our knowledge haven't considered subject reduction properties for their 2-level λ-calculus. We would also like to prove that well-annotatedness is preserved by static reduction. This seems difficult because if $E \rightarrow E'$, then the constraint systems $WA(E)$ and $WA(E')$ are rather different. Finally, we would like to prove the correctness of the binding-time analyses of Bondorf (1991), Consel (1990), and Bondorf and Jørgensen (1993).

Mitchell Wand (1993) studied Mogensen's binding-time analysis and partial evaluator for the λ-calculus (Mogensen, 1992). Wand proved that the partial evaluator is correct, in the sense that if it is given an output from the binding-time analysis, then indeed it produces a specialized program. This result relates to our theorem 26 as follows. Wand considers *arbitrary* static reduction sequences under a certain consistency condition. We consider only those static reduction sequences that involve enabled applications, but under a weaker consistency condition. In contrast to Wand, we study the idea of trusting enabled applications to be static redexes.

It remains open if Wand's result can be proved under the weaker consistency condition studied in this paper. In other words, it should be investigated if Mogensen's partial evaluator can be shown to produce a specialized program when given a Palsberg/Schwartzbach well-annotated 2-level λ-term. The open problem and its relation to the solved problems is illustrated below.



https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press

Acknowledgements

The author thanks Torben Amtoft, Olivier Danvy, Fritz Henglein, Neil Jones, Jesper Jørgensen, Karoline Malmkjær, Hanne Riis Nielson, and Kristoffer Rose for discussions on the correctness of binding-time analysis. The author also thanks Torben Amtoft, Peter Mosses, Michael Schwartzbach, Mitchell Wand, and the anonymous referees for helpful comments on a draft of the paper. Finally, the author thanks Olivier Danvy for pointing out an error in a previous version of the definition of enabledness.

References

Barendregt, Henk P. (1981) *The lambda calculus: Its syntax and semantics.* North-Holland.

Bondorf, Anders. (1991) Automatic autoprojection of higher order recursive equations. *Science of computer programming*, **17**(1–3), 3–34.

Bondorf, Anders, & Jørgensen, Jesper. (1993) Efficient analyses for realistic off-line partial evaluation. *Journal of functional programming, special issue on partial evaluation.*

Consel, Charles. (1990) Binding-time analysis for higher order untyped functional languages. *Pages 264–272 of: Proc. ACM conference on lisp and functional programming.*

Gomard, Carsten K. (1990) Partial type inference for untyped functional programs. *Pages 282–287 of: Proc. ACM conference on lisp and functional programming.*

Gomard, Carsten K. (1991) (November). *Program analysis matters.* Ph.D. thesis, DIKU, University of Copenhagen. DIKU Report 91–17.

Gomard, Carsten K., & Jones, Neil D. (1991) A partial evaluator for the untyped lambda-calculus. *Journal of functional programming*, **1(1)**, 21–69.

Jones, Neil D. (1981) Flow analysis of lambda expressions. *Pages 114–128 of: Proc. eighth colloquium on automata, languages, and programming.* Springer-Verlag (*LNCS* 115).

Mogensen, Torben Æ. (1992) Self-applicable partial evaluation for pure lambda calculus. *Pages 116–121 of: Proc. ACM SIGPLAN workshop on partial evaluation and semantics-based program manipulation.*

Nielson, Hanne R., & Nielson, Flemming. (1988) Automatic binding-time analysis for a typed λ-calculus. *Science of computer programming*, **10**, 139–176.

Palsberg, Jens, & Schwartzbach, Michael I. (1992) *Binding-time analysis: Abstract interpretation versus type inference.* Submitted for publication.

Schmidt, David A. (1987) Static properties of partial reduction. *Pages 295–305 of: Proc. partial evaluation and mixed computation, Gl. Avernæs, Denmark.*

Sestoft, Peter. (1989) Replacing function parameters by global variables. *Pages 39–53 of: Proc. conference on functional programming languages and computer architecture.*

Shivers, Olin. (1991) (May). *Control-flow analysis of higher-order languages.* Ph.D. thesis, CMU. CMU–CS–91–145.

Wand, Mitchell. (1993) Specifying the correctness of binding-time analysis. *Journal of functional programming, special issue on partial evaluation.*

https://doi.org/10.1017/S0956796800000770 Published online by Cambridge University Press