In Proc. *ICCL'94, Fifth IEEE International Conference on Computer Languages*, pages 289–298.

Binding-time Analysis: Abstract Interpretation versus Type Inference

Jens Palsberg Michael I. Schwartzbach
palsberg@daimi.aau.dk mis@daimi.aau.dk

Computer Science Department, Aarhus University
Ny Munkegade, DK–8000 Aarhus C, Denmark

Abstract

Binding-time analysis is important in partial evaluators. Its task is to determine which parts of a program can be evaluated if some of the expected input is known. Two approaches to do this are abstract interpretation and type inference. We compare two specific such analyses to see which one determines most program parts to be eliminable. The first is a an abstract interpretation approach based on closure analysis and the second is the type inference approach of Gomard and Jones. Both apply to the pure λ-calculus. We prove that the abstract interpretation approach is more powerful than that of Gomard and Jones: the former determines the same and possibly more program parts to be eliminable as the latter.

1 Introduction

In this paper we compare two techniques for doing *binding-time analysis* of terms in the pure λ-calculus [1]. The binding-times we are concerned with are "static" (compile-time) and "dynamic" (run-time).

Binding-time analysis has been formulated in various settings. Most of them are based on either abstract interpretation or type inference. For examples of the former that are applicable to higher-order languages, see the work of Mogensen [11, 12], Bondorf, [2], Consel [3], and Hunt and Sands [8]. For examples of the latter that are also applicable to higher-order languages, see the work of Nielson and Nielson [13], and Gomard and Jones [6]. Only little is known, however, about the relative quality of these analyses. For comparison with strictness analysis, note that Jensen [9] has proved the equivalence of two strictness analyses based on abstract interpretation and type inference, respectively.

We will compare two fundamentally different binding-time analyses. The first is a an abstract interpretation approach based on closure analysis [14] and the second is the type inference approach of Gomard and Jones [6]. The first is intended to capture the binding-time analyses of Bondorf [2] and Consel [3], when restricted to the λ-calculus. We have not given a proof of this connection, however, so this analysis may turn out to be novel. The binding-time analyses of Bondorf, Consel, and Gomard and Jones are successfully used in the SIMILIX, SCHISM, and LAMBDA-MIX partial evaluators, respectively.

Our comparison concentrates on the higher-order aspects of the two binding-time analyses. Thus, we define the abstract interpretation based analysis for the λ-calculus, and we restrict the definition of Gomard and Jones' analysis to the same language.

A natural extension would be the introduction of first-order values. This introduces the possibility of "lifting" static values to dynamic values, and this is, to the best of our knowledge, done differently in Bondorf's and Gomard and Jones' analyses. Our comparison shows that the two chosen analyses differ even when first-order values are omitted.

We assume that a λ-term to be analyzed takes its input through the free variables. We also assume that all input is dynamic. The definition of Gomard and Jones' analysis remains the same if we introduce static input. The definition of the abstract interpretation based analysis, however, depends on having no static higher-order input. Since we treat the λ-calculus, there are no first-order values, hence no static input. Again, our comparison shows that the two chosen analyses differ even in the absence of static input.

The output of a binding-time analysis can be presented as an *annotated* version of the analyzed λ-term. The language of annotated λ-terms is usually called a 2-level λ-calculus [13]. It is defined by the grammar:

$$\begin{array}{rcll} E & ::= & x & \text{(variable)} \\ & | & \lambda x.E & \text{(static abstraction)} \\ & | & E_1 \mathbin{@} E_2 & \text{(static application)} \\ & | & \underline{\lambda} x.E & \text{(dynamic abstraction)} \\ & | & E_1 \mathbin{\underline{@}} E_2 & \text{(dynamic application)} \end{array}$$

Intuitively, "static" means "statically known", and "dynamic" means "not statically known". The static entities are those that can be eliminated during partial evaluation. The purpose of a binding-time analysis for the λ-calculus is to produce *consistent* 2-level λ-terms [14]. Consistency prevents partial evaluators from "going wrong" [14]. For example, the 2-level λ-term $(\underline{\lambda} x.x) \mathbin{@} y$ is inconsistent: even though the function part of the static application is an abstraction, it is marked as dynamic.

For another example, consider $(\lambda x.x \mathbin{@} y) \mathbin{@} z$. A binding-time analysis may here produce the annotated term $(\lambda x.x \mathbin{\underline{@}} y) \mathbin{@} z$. It could also produce $(\underline{\lambda} x.x \mathbin{\underline{@}} y) \mathbin{\underline{@}} z$. Both these 2-level λ-terms are consistent. In the first case, a partial evaluator will do a single reduction and obtain $z \mathbin{\underline{@}} y$.

Following Gomard and Jones [6], we partially order the set of 2-level λ-terms 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, $((\lambda x.x \mathbin{\underline{@}} y) \mathbin{@} z) \sqsubseteq ((\underline{\lambda} x.x \mathbin{\underline{@}} y) \mathbin{\underline{@}} z)$. Notice that $\sqsubseteq$ admits greatest lower bounds, written $\sqcap$, for two terms that are equal except for underlinings.

The quality of a binding-time analysis has at least two aspects:

- How many applications does it determine to be reducible? and
- How fast can it be executed?

The first aspect determines the effect of the specializer, and the second aspect influences the overall execution time of the partial evaluator.

This paper focuses on the first quality aspect. Our quality measure is the standard one:

> One binding-time analysis is more powerful than another, if it always produces $\sqsubseteq$-smaller 2-level λ-terms than the other.

The intuition is "the fewer underlinings, the better".

We prove that the abstract interpretation based analysis is more powerful than that of Gomard and Jones, in the sense that it always produces $\sqsubseteq$-smaller 2-level λ-terms.

Our proof technique is a generalization of one used in [16]. That paper also compares two analyses based on abstract interpretation and type inference, respectively. The key difference is that the abstract interpretation in [16] uses a lattice whereas the one in this paper does not. Our new proof technique can handle both cases.

In the following section we discuss the key concept of well-annotatedness, in Section 3 we define the two analyses, and in Section 4 we prove our result.

2 Well-annotatedness

A sufficient and decidable condition for consistency, called well-annotatedness, was first presented by Gomard and Jones [6]. Their binding-time analysis always produces well-annotated terms.

In this paper we present another condition for consistency. It is defined to capture the outputs of the binding-time analyses of Bondorf and Consel, when restricted to the λ-calculus. We have not given a proof of this connection, however, so our condition may turn out to be novel.

This paper proves that the new consistency condition is weaker than that of Gomard and Jones. This justifies calling the new condition "well-annotatedness". For clarity, we will call the new condition "Palsberg/Schwartzbach well-annotatedness". Thus, if a 2-level λ-term is Gomard/Jones well-annotated, then it will also be Palsberg/Schwartzbach well-annotated. It has been proved by the first author [14] that Palsberg/Schwartzbach well-annotatedness, hence also Gomard/Jones well-annotatedness, implies consistency.

The two definitions of well-annotatedness can be understood as *specifications* of binding-time analyses. Such a specification is implemented by any algorithm that always produces well-annotated 2-level λ-terms. We need not be satisfied with any such algorithm, however. Both well-annotatedness criteria have the property that for each λ-term there is a $\sqsubseteq$-least well-annotated version of it. Thus, the best possible implementations are algorithms that produce the $\sqsubseteq$-least well-annotated versions of their inputs.

We compare the best possible implementations of the two well-annotatedness criteria. These algorithms indeed exist, as follows. Both the algorithms of Gomard [5] and Henglein [7] produce the $\sqsubseteq$-least Gomard/Jones well-annotated 2-level version of a given λ-term. We believe (but have not proved) that the algorithms of Bondorf and Consel produces the $\sqsubseteq$-least Palsberg/Schwartzbach well-annotated 2-level version of a given λ-term. A naïve algorithm that computes this $\sqsubseteq$-least 2-level λ-term is sketched below.

Our comparison proceeds by first proving that Gomard/Jones well-annotatedness implies Palsberg/Schwartzbach well-annotatedness. This leads to the desired result because the input/output behavior of the two chosen analyses are fully defined by their specifications.

Gomard and Jones formulated their well-annotatedness criterion via inference rules [6]. For the purpose of this paper, we will rephrase it using constraint systems. The new notion of well-annotatedness will also be phrased using constraint systems. The use of constraint systems makes possible the proof of comparison.

3 The Formal Systems

Both notions of well-annotatedness will be presented via constraint systems. In each case, the idea is that the condition is true of a 2-level λ-term E_0 iff a constraint system generated from E_0 is solvable. Both constraint systems are generated in the style of Wand [20], as follows. First, the 2-level λ-term is α-converted so that every λ-bound (or $\underline{\lambda}$-bound) variable is distinct. This means that every abstraction $\lambda x.E$ (or $\underline{\lambda} x.E$) can be denoted by the unique lambda token λx (or $\underline{\lambda} x$). Second, a type variable $[\![E]\!]$ is assigned to every subterm E. Finally, a finite collection of constraints over these variables is generated from the syntax.

The two definitions of well-annotatedness employ constraints over different domains, we will consider them in turn.

Gomard/Jones Well-annotatedness

In Gomard/Jones well-annotatedness, type variables range over the following types:

$$\tau ::= \mathsf{Dyn} \mid \tau_1 \to \tau_2$$

The intuition behind the constant Dyn is that a term with this type has unknown value; Dyn abbreviates Dynamic (Dyn is called code by Gomard and Jones).

The constraints are generated inductively in the syntax of a 2-level λ-term E_0, as follows:

Phrase:	Initial constraints:
x	If x is free in E_0 then $[\![x]\!] = \mathsf{Dyn}$
E_0	$[\![E_0]\!] = \mathsf{Dyn}$

Phrase:	Type constraint:
$\lambda x.E$	$[\![\lambda x.E]\!] = [\![x]\!] \to [\![E]\!]$
$E_1 \mathbin{@} E_2$	$[\![E_1]\!] = [\![E_2]\!] \to [\![E_1 \mathbin{@} E_2]\!]$
$\underline{\lambda} x.E$	$[\![\underline{\lambda} x.E]\!] = [\![x]\!] = [\![E]\!] = \mathsf{Dyn}$
$E_1 \mathbin{\underline{@}} E_2$	$[\![E_1]\!] = [\![E_2]\!] = [\![E_1 \mathbin{\underline{@}} E_2]\!] = \mathsf{Dyn}$

We let GJ ("Gomard/Jones") denote the global constraint system. Sometimes we write $GJ(E_0)$ to emphasize that the constraint system is generated from E_0. Each type constraint matches an inference rule in Gomard and Jones' formulation of the predicate [6]. The initial constraints of the form $[\![x]\!] = \mathsf{Dyn}$ reflect that the free variables of E_0 correspond to unknown input. The initial constraint $[\![E_0]\!] = \mathsf{Dyn}$ reflects that a partial evaluator is supposed to produce a residual program. Intuitively, underlinings must be introduced if a subterm does not have a simple type. A *solution* of GJ assigns a type to each type variable such that all constraints are satisfied. The constraint system $GJ((\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{\underline{@}} z))$ is shown in figure 1.

Note that the constraints differ from those presented by Henglein [7]; the latter are defined on pure λ-terms with the purpose of specifying the *computation* (rather than the well-annotatedness) of an annotation.

For any λ-term, there is a $\sqsubseteq$-least annotated version for which GJ is solvable, for a proof see Gomard's Master's thesis [4]. Henglein gave a pseudo-linear time algorithm for computing this $\sqsubseteq$-least term [7].

A New Notion of Well-annotatedness

The new notion of well-annotatedness is based on an abstract interpretation called *closure analysis* [18, 2] (also called *control flow analysis* by Jones [10] and Shivers [19]). The *closures* of a term are simply the subterms corresponding to abstractions. A closure analysis approximates for every subterm the set of possible closures to which it may evaluate [10, 18, 2, 19]. Both Bondorf and Consel's binding-time analyses may be understood as a closure analysis that in addition to closures also incorporates a special value Dyn (Dyn is called D by Bondorf). The intuition behind Dyn is the same as that behind the Dyn used in Gomard/Jones well-annotatedness.

We define the new well-annotatedness criterion as follows. For static entities, we generate the usual constraints of closure analysis [16, 17, 15]; and for dy-

Constraints:

$$\begin{array}{rl}
y & [\![y]\!] = \mathsf{Dyn} \\
(\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{\underline{@}} z) & [\![(\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{\underline{@}} z)]\!] = \mathsf{Dyn} \\
\lambda x.y & [\![\lambda x.y]\!] = [\![x]\!] \rightarrow [\![y]\!] \\
\lambda z.z \mathbin{\underline{@}} z & [\![\lambda z.z \mathbin{\underline{@}} z]\!] = [\![z]\!] \rightarrow [\![z \mathbin{\underline{@}} z]\!] \\
(\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{\underline{@}} z) & [\![\lambda x.y]\!] = [\![\lambda z.z \mathbin{\underline{@}} z]\!] \rightarrow [\![(\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{\underline{@}} z)]\!] \\
z \mathbin{\underline{@}} z & [\![z]\!] = [\![z]\!] = [\![z \mathbin{\underline{@}} z]\!] = \mathsf{Dyn}
\end{array}$$

Solution: The mapping L where

$$\begin{array}{rcl}
L[\![x]\!] = L[\![\lambda z.z \mathbin{\underline{@}} z]\!] & = & \mathsf{Dyn} \rightarrow \mathsf{Dyn} \\
L[\![y]\!] = L[\![(\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{\underline{@}} z)]\!] & = & \mathsf{Dyn} \\
L[\![z]\!] = L[\![z \mathbin{\underline{@}} z]\!] & = & \mathsf{Dyn} \\
L[\![\lambda x.y]\!] & = & (\mathsf{Dyn} \rightarrow \mathsf{Dyn}) \rightarrow \mathsf{Dyn}
\end{array}$$

Figure 1: The constraint system $GJ((\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{\underline{@}} z))$.

namic entities we generate the same constraints as Gomard and Jones. This approach emphasizes both the similarities and differences between Gomard/Jones well-annotatedness and Palsberg/Schwartzbach well-annotatedness.

(The definition of well-annotatedness in [14] is slightly different but equivalent to the one given here.)

In the new notion of well-annotatedness, type variables range over the set D of binding-time values. The set D consists of the value Dyn and all subsets of LAMBDA. LAMBDA is the finite set of static lambda tokens in E_0, the main term. Sometimes we write LAMBDA(E_0) to emphasize that the set is generated from E_0. The set D is partially ordered by $\leq$, as follows:

1. $\mathsf{Dyn} \leq \mathsf{Dyn}$; and

2. if $v, v' \subseteq$ LAMBDA and $v \subseteq v'$, then $v \leq v'$.

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

The constraints are generated from the syntax of a 2-level λ-term E_0, as shown in the table below.

As a conceptual aid, the constraints are grouped into *initial*, *basic*, and *connecting* constraints.

The connecting constraints reflect the relationship between formal and actual arguments and results. The condition $\{\lambda x\} \leq [\![E_1]\!]$ states that the two guarded inclusions are relevant only if the closure denoted by λx is a possible result of E_1. Notice that Dyn is not a possible result of E_1 because of the basic constraint $[\![E_1]\!] \geq \emptyset$.

Phrase:	Initial constraints:
x	If x is free in E_0 then $[\![x]\!] = \mathsf{Dyn}$
E_0	$[\![E_0]\!] = \mathsf{Dyn}$

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

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

We let *WA* ("Well-Annotatedness") denote the global constraint system, i.e., the collection of constraints for every subterm. Sometimes we write $WA(E_0)$ to emphasize that the constraint system is generated from E_0. Intuitively, underlinings must be introduced to ensure that no subexpression can evaluate to both a static and a dynamic value. The constraint system $WA((\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{@} z))$ is shown in figure 2.

The constraints for the pure terms yield a closure analysis. A *solution* of *WA* assigns an element of D to each type variable such that all constraints are satisfied. Below we prove that for any λ-term, there is

Constraints:

$$\begin{array}{rl}
y & [\![y]\!] = \mathsf{Dyn} \\
(\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{@} z) & [\![(\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{@} z)]\!] = \mathsf{Dyn} \\
\lambda x.y & [\![\lambda x.y]\!] \geq \{\lambda x\} \\
\lambda z.z \mathbin{@} z & [\![\lambda z.z \mathbin{@} z]\!] \geq \{\lambda z\} \\
(\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{@} z) & [\![\lambda x.y]\!] \geq \emptyset \\
z \mathbin{@} z & [\![z]\!] \geq \emptyset \\
(\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{@} z) & \left[\begin{array}{l}
\{\lambda x\} \leq [\![\lambda x.y]\!] \;\Rightarrow \left[\begin{array}{l} [\![\lambda z.z \mathbin{@} z]\!] \leq [\![x]\!] \\ [\![(\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{@} z)]\!] \geq [\![y]\!] \end{array}\right. \\
\{\lambda z\} \leq [\![\lambda x.y]\!] \;\Rightarrow \left[\begin{array}{l} [\![\lambda z.z \mathbin{@} z]\!] \leq [\![z]\!] \\ [\![(\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{@} z)]\!] \geq [\![z \mathbin{@} z]\!] \end{array}\right.
\end{array}\right. \\
z \mathbin{@} z & \left[\begin{array}{l}
\{\lambda x\} \leq [\![z]\!] \;\Rightarrow \left[\begin{array}{l} [\![z]\!] \leq [\![x]\!] \\ [\![z \mathbin{@} z]\!] \geq [\![y]\!] \end{array}\right. \\
\{\lambda z\} \leq [\![z]\!] \;\Rightarrow \left[\begin{array}{l} [\![z]\!] \leq [\![z]\!] \\ [\![z \mathbin{@} z]\!] \geq [\![z \mathbin{@} z]\!] \end{array}\right.
\end{array}\right.
\end{array}$$

Solution: The mapping L where

$$\begin{array}{rcl}
L[\![x]\!] = L[\![\lambda z.z \mathbin{@} z]\!] & = & \{\lambda z\} \\
L[\![y]\!] = L[\![(\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{@} z)]\!] & = & \mathsf{Dyn} \\
L[\![z]\!] = L[\![z \mathbin{@} z]\!] & = & \emptyset \\
L[\![\lambda x.y]\!] & = & \{\lambda x\}
\end{array}$$

Figure 2: The constraint system $WA((\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{@} z))$.

a $\sqsubseteq$-least annotated version for which WA is solvable. We also give an algorithm for computing this $\sqsubseteq$-least term.

Informal Comparison

The two notions of well-annotatedness use the same initial constraints and the same constraints on dynamic entities. They differ in their treatment of static entities: GJ uses a type discipline to handle abstractions whereas WA uses abstract interpretation.

Gomard and Jones' analysis can apparently be computed faster than the new analysis [7]. In return for the longer running time, the new analysis produces better results.

For example, consider $(\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{@} z)$. The abstract interpretation approach yields *no* underlinings at all because $WA((\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{@} z))$ is solvable, see figure 2. Gomard and Jones' approach must yield at least one underlining because this λ-term does not have a simple type. Specifically, it yields the 2-level λ-term $(\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{\underline{@}} z)$, that is, just one underlining. For a solution of $GJ((\lambda x.y) \mathbin{@} (\lambda z.z \mathbin{\underline{@}} z))$, see figure 1.

4 Comparison

We now show that the abstract interpretation based analysis produces at most as many underlinings as the analysis of Gomard and Jones. We do this by proving for all 2-level λ-terms that if GJ is solvable, then so is WA. This implies the desired result because given any λ-term, WA is in particular solvable for the $\sqsubseteq$-*least* annotated version for which GJ is solvable.

The main technical challenge is that WA and GJ are constraint systems over two different domains, sets versus types. Our proof introduces the closure $\overline{WA}$ as a convenient "stepping stone" between GJ and WA. The structure of the proof can be illustrated as follows:

$$GJ \implies \overline{WA} \iff WA$$

First we define the closure $\overline{WA}$. The set WA can be described as a disjoint union of C and U where C

contains the connecting constraints and U the initial and basic constraints. The closure $\overline{WA}$ is the smallest set such that

- U is included in $\overline{WA}$.
- If $c \Rightarrow K$ is in C and c is in $\overline{WA}$, then K is in $\overline{WA}$.
- If $r \leq s$ and $s \leq t$ both are in $\overline{WA}$, then $r \leq t$ is in $\overline{WA}$.

The closure $\overline{WA}$ contains only constraints of the forms $\{\lambda x\} \leq X$, $\emptyset \leq X$, $X = \mathsf{Dyn}$, and $X \leq Y$, where X, Y are type variables. We first prove that $\overline{WA}$ has a useful property. The point is that in the proof of Lemma 4.2 below, we need a solution L of $\overline{WA}$ with the property that $\lambda x \in L(X)$ if and only if $\{\lambda x\} \leq X$ is in $\overline{WA}$.

Lemma 4.1 *If $\overline{WA}$ is solvable, then it has a solution L such that for all type variables X and all λx in $L(X)$, the constraint $\{\lambda x\} \leq X$ is in $\overline{WA}$.*

Proof. Suppose $\overline{WA}$ has solution L'. Define L as follows. If $L'(X) = \mathsf{Dyn}$, then $L(X) = \mathsf{Dyn}$. Otherwise, $L(X) = L'(X) \setminus S_X$, where S_X is the set of those λx in $L'(X)$ where the constraint $\{\lambda x\} \leq X$ is *not* in $\overline{WA}$. Clearly, for all type variables X and all λx in $L(X)$, the constraint $\{\lambda x\} \leq X$ is in $\overline{WA}$.

To see that L is a solution of $\overline{WA}$, we consider the constraints in $\overline{WA}$ in turn. Constraints of the forms $X \geq \emptyset$ and $X = \mathsf{Dyn}$ have solution L because they have solution L'. Constraints of the form $\{\lambda x\} \leq X$ have solution L because they have solution L' and because λx is *not* in S_X. Finally, constraints of the form $X \leq Y$ yield two cases. In the first case, $L'(X) = L'(Y) = \mathsf{Dyn}$, so $L(X) = L(Y) = \mathsf{Dyn}$. In the second case, $L'(X)$ and $L'(Y)$ are both sets with $L'(X) \subseteq L'(Y)$. We need to prove that $(L'(X) \setminus S_X) \subseteq (L'(Y) \setminus S_Y)$. To do this, suppose λx is in $L'(X) \setminus S_X$. Thus, λx is in $L'(X)$ and the constraint $\{\lambda x\} \leq X$ is in $\overline{WA}$. Then, by transitivity of $\subseteq$ and by $\overline{WA}$ being closed under the transitivity of $\leq$, we get that λx is in $L'(Y)$ and that the constraint $\{\lambda x\} \leq Y$ is in $\overline{WA}$. Thus, λx is in $(L'(Y) \setminus S_Y)$. □

We can then prove that closing WA preserves solvability.

Lemma 4.2 *WA is solvable* iff *$\overline{WA}$ is solvable.*

Proof. For the *only if* case, suppose WA has solution L. We will show that also $\overline{WA}$ has solution L. We proceed by induction on the construction of $\overline{WA}$. In the base case, consider U. Since U is a subset of WA, U has solution L. In the induction step there are two cases. In the first case, consider $c \Rightarrow K$ in C and suppose c is in $\overline{WA}$. By the induction hypothesis, c has solution L. Combining this with C having solution L we get that K has solution L. In the second case, consider $r \leq s$ and $s \leq t$ in $\overline{WA}$. By the induction hypothesis, both have solution L. Using the transitivity of $\leq$, also $r \leq t$ has solution L.

For the *if* case, suppose $\overline{WA}$ is solvable. By Lemma 4.1, choose a solution L with the property that for all type variables X and all λx in $L(X)$, the constraint $\{\lambda x\} \leq X$ is in $\overline{WA}$. We will show that also WA has solution L. Clearly, U has solution L, since U is included in $\overline{WA}$. To see that also C has solution L, consider $c \Rightarrow K$ in C and suppose c has solution L. Since c is of the form $\{\lambda x\} \leq X$ we get, by the property of L, that c is in $\overline{WA}$. Hence, also K is in $\overline{WA}$, so K has solution L. □

We now show the fundamental connection between the two well-annotatedness criteria.

Lemma 4.3 *If GJ is solvable, then so is $\overline{WA}$.*

Proof. Suppose GJ has solution L. Define L' as follows. If $L(X) = \mathsf{Dyn}$ then $L'(X) = \mathsf{Dyn}$; and if $L(X) = \alpha \to \beta$ then $L'(X) = \textsc{lambda}$. We will show the following three properties of which the first is the desired conclusion:

1. $\overline{WA}$ has solution L';
2. If $\{\lambda x\} \leq X$ is in $\overline{WA}$ then $L[\![\lambda x.E]\!] = L(X)$; and
3. If $X \leq Y$ is in $\overline{WA}$ then $L(X) = L(Y)$.

Here, X, Y are type variables.

We proceed by induction on the construction of $\overline{WA}$.

In the base case, consider U. For any $\lambda x.E$, GJ yields the constraint $[\![\lambda x.E]\!] = [\![x]\!] \to [\![E]\!]$ and WA yields the constraint $[\![\lambda x.E]\!] \geq \{\lambda x\}$. Since the former constraint has solution L, we get that $L'[\![\lambda x.E]\!] = \textsc{lambda}$, so the second constraint has solution L'. Clearly, $L[\![\lambda x.E]\!] = L[\![\lambda x.E]\!]$.

For any $E_1 \,@\, E_2$, GJ yields the constraint $[\![E_1]\!] = [\![E_2]\!] \to [\![E_1 \,@\, E_2]\!]$ and WA yields the constraint $[\![E_1]\!] \geq \emptyset$. Since the former constraint has solution L, we get that $L'[\![E_1]\!] = \textsc{lambda}$, so the second constraint has solution L'.

For any $\underline{\lambda} x.E$, $E_1 \,\underline{@}\, E_2$, free variable in the main term, or the main term itself, WA and GJ yield the same constraints, of the form $X = \mathsf{Dyn}$. Since each of these constraints $X = \mathsf{Dyn}$ has solution L, we get that $L'(X) = \mathsf{Dyn}$, so $X = \mathsf{Dyn}$ also has solution L'.

In the induction step there are two cases. In the first case, consider

$$\{\lambda x\} \leq [\![E_1]\!] \Rightarrow [\![E_2]\!] \leq [\![x]\!] \wedge [\![E_1 \mathbin{@} E_2]\!] \geq [\![E]\!]$$

in C and suppose $\{\lambda x\} \leq [\![E_1]\!]$ is in $\overline{WA}$. By the induction hypothesis, $L[\![\lambda x.E]\!] = L[\![E_1]\!]$. In GJ we have the constraints $[\![\lambda x.E]\!] = [\![x]\!] \rightarrow [\![E]\!]$ and $[\![E_1]\!] = [\![E_2]\!] \rightarrow [\![E_1 \mathbin{@} E_2]\!]$. Thus, $L[\![E_2]\!] = L[\![x]\!]$ and $L[\![E_1 \mathbin{@} E_2]\!] = L[\![E]\!]$, so $L'[\![E_2]\!] \leq L'[\![x]\!]$ and $L'[\![E_1 \mathbin{@} E_2]\!] \geq L'[\![E]\!]$.

In the second case, consider $r \leq s$ and $s \leq t$ in $\overline{WA}$. By the induction hypothesis, both constraints have solution L', so by the transitivity of $\leq$, also $r \leq t$ has solution L'. If r is on the form $\{\lambda x\}$, then s and t are type variables. By the induction hypothesis, $L[\![\lambda x.E]\!] = L(s)$ and $L(s) = L(t)$, so $L[\![\lambda x.E]\!] = L(t)$. If r is a type variable, then s and t are type variables. By the induction hypothesis, $L(r) = L(s)$ and $L(s) = L(t)$, so $L(r) = L(t)$. □

Lemma 4.4 *If GJ is solvable, then so is WA.*

Proof. Combine Lemmas 4.2 and 4.3. □

To be able to prove that for any λ-term there is a $\sqsubseteq$-least annotated version for which WA is solvable, we need the following lemma.

Lemma 4.5 *Solvability of WA is preserved by binary $\sqcap$ of annotated versions of a λ-term.*

Proof. Let E_U be a λ-term. Suppose E_A and E_B are annotated versions of E_U such that $WA(E_A)$ and $WA(E_B)$ have solutions L_A and L_B, respectively. By Lemma 4.2, it is then sufficient to prove that $\overline{WA}(E_A \sqcap E_B)$ has the solution L, constructed as follows. Let E be a subterm of $E_A \sqcap E_B$. We can then write $E = E'_A \sqcap E'_B$, where E'_A is a subterm of E_A and E'_B is the corresponding subterm of E_B. Finally, we define $L[\![E]\!] = L_A[\![E'_A]\!] \sqcap_{\triangleleft} L_B[\![E'_B]\!]$. The symbol $\sqcap_{\triangleleft}$ denotes the greatest lower bound in the ordering $\triangleleft$ of D. The ordering $\triangleleft$ is an extension of $\leq$ such that also $v \triangleleft \mathsf{Dyn}$ for all v. Thus, (D,$\triangleleft$) is a lattice with Dyn as the maximal element. The property of $\sqcap_{\triangleleft}$ that we need is that Dyn is neutral with respect to it.

We will show the following three properties of which the first is the desired conclusion:

1. $\overline{WA}(E_A \sqcap E_B)$ has solution L;
2. If $\{\lambda x\} \leq X$ is in $\overline{WA}(E_A \sqcap E_B)$, then in both E_A and E_B, the type variables for the two subterms corresponding to λx and X are either both assigned Dyn by L_A and L_B, respectively, or else neither of them are; and
3. If $X \leq Y$ is in $\overline{WA}(E_A \sqcap E_B)$, then in both E_A and E_B, the type variables for the two subterms corresponding to X and Y are either both assigned Dyn by L_A and L_B, respectively, or else neither of them are.

Here, X, Y are type variables.

We proceed by induction on the construction of $\overline{WA}(E_A \sqcap E_B)$.

In the base case, consider U. For any subterm $\lambda x.E$ of $E_A \sqcap E_B$, we must show that $L[\![\lambda x.E]\!] \geq \{\lambda x\}$. Observe that $L[\![\lambda x.E]\!] = L_A[\![E'_A]\!] \sqcap_{\triangleleft} L_B[\![E'_B]\!]$, where E'_A and E'_B both are abstractions of which at most one is dynamic. If neither are dynamic, then both $L_A[\![E'_A]\!] \geq \{\lambda x\}$ and $L_B[\![E'_B]\!] \geq \{\lambda x\}$. Thus, $L_A[\![E'_A]\!] \cap L_B[\![E'_B]\!] \geq \{\lambda x\}$. The desired result is then immediate. If only one is dynamic, then either $L_A[\![E'_A]\!] = \mathsf{Dyn}$ or $L_B[\![E'_B]\!] = \mathsf{Dyn}$. The constraint still holds for the static one. The desired result is then immediate since Dyn is neutral with respect to $\sqcap_{\triangleleft}$. Moreover, since λx and $[\![\lambda x.E]\!]$ correspond to the same subterm of $E_A \sqcap E_B$, the second property above trivially holds.

The case of $E_1 \mathbin{@} E_2$ is treated like the first case. The remaining cases of $\underline{\lambda} x.E$, $E_1 \mathbin{\underline{@}} E_2$, free variables of $E_A \sqcap E_B$, and $E_A \sqcap E_B$ itself, trivially follows from $\mathsf{Dyn} = \mathsf{Dyn} \sqcap_{\triangleleft} \mathsf{Dyn}$.

In the induction step there are two cases. In the first case, consider

$$\{\lambda x\} \leq [\![E_1]\!] \Rightarrow [\![E_2]\!] \leq [\![x]\!] \wedge [\![E_1 \mathbin{@} E_2]\!] \geq [\![E]\!]$$

in C, where $E_1 \mathbin{@} E_2$ and $\lambda x.E$ are subterms of $E_A \sqcap E_B$, and suppose $\{\lambda x\} \leq [\![E_1]\!]$ is in $\overline{WA}(E_A \sqcap E_B)$. The following table gives notation for the subterms of E_A and E_B which correspond to the subterms $E_1 \mathbin{@} E_2, E_1, E_2, \lambda x.E, E$ of $E_A \sqcap E_B$.

$E_A \sqcap E_B$	E_A	E_B
$E_1 \mathbin{@} E_2$	F_A	F_B
E_1	E_{1A}	E_{1B}
E_2	E_{2A}	E_{2B}
$\lambda x.E$	G_A	G_B
E	E_{BA}	E_{BB}

We must show that

(a) $L[\![E_2]\!] \leq L[\![x]\!]$ and $L[\![E_1 \mathbin{@} E_2]\!] \geq L[\![E]\!]$;

(b) $[\![E_{2A}]\!]$ and $[\![x]\!]$ are either both assigned Dyn by L_A or else neither of them; and also $[\![F_A]\!]$ and $[\![E_{BA}]\!]$ are either both assigned Dyn by L_A or else neither of them; and

(c) $[\![E_{2B}]\!]$ and $[\![x]\!]$ are either both assigned Dyn by L_B or else neither of them; and also $[\![F_B]\!]$ and $[\![E_{BB}]\!]$ are either both assigned Dyn by L_B or else neither of them.

Property (a) is equivalent to $L_A[\![E_{2A}]\!] \sqcap_{\lhd} L_B[\![E_{2B}]\!] \leq L_A[\![x]\!] \sqcap_{\lhd} L_B[\![x]\!]$ and $L_A[\![F_A]\!] \sqcap_{\lhd} L_B[\![F_B]\!] \geq L_A[\![E_{BA}]\!] \sqcap_{\lhd} L_B[\![E_{BB}]\!]$.

By the induction hypothesis (property 2), we get that $[\![G_A]\!]$ and $[\![E_{1A}]\!]$ are either both assigned Dyn by L_A or else neither of them; and we also get that $[\![G_B]\!]$ and $[\![E_{1B}]\!]$ are either both assigned Dyn by L_A or else neither of them.

Observe that G_A and G_B are both abstractions of which at most one is dynamic. Suppose first that both are static. Thus, neither of them are assigned Dyn by L_A and L_B, respectively. Hence, neither of $[\![E_{1A}]\!]$ and $[\![E_{1B}]\!]$ are assigned Dyn by L_A and L_B, respectively. Since L_A and L_B are solutions of $WA(E_A)$ and $WA(E_B)$, respectively, both of F_A and F_B are static applications. We then get that $\{\lambda x\} \leq L_A[\![E_{1A}]\!] \Rightarrow L_A[\![E_{2A}]\!] \leq L_A[\![x]\!] \wedge L_A[\![F_A]\!] \geq L_A[\![E_{BA}]\!]$ and $\{\lambda x\} \leq L_B[\![E_{1B}]\!] \Rightarrow L_B[\![E_{2B}]\!] \leq L_B[\![x]\!] \wedge Ł_B[\![F_B]\!] \geq L_B[\![E_{BB}]\!]$. Moreover, both of the conditions hold because $\{\lambda x\} \leq L[\![E_1]\!] = L_A[\![E_{1A}]\!] \sqcap_{\lhd} L_B[\![E_{1B}]\!]$. The three properties then follow immediately.

Suppose then that one of G_A and G_B is dynamic, say G_B. Thus, $L_B[\![G_B]\!] =$ Dyn. Hence, $L_B[\![E_{1B}]\!] =$ Dyn. Since L_B is a solution of $WA(E_B)$, F_B is a dynamic application, and $L_B[\![E_{2B}]\!] = L_B[\![x]\!] = L_B[\![F_B]\!] = L_B[\![E_{BB}]\!] =$ Dyn. As in the first case, we have $\{\lambda x\} \leq L_A[\![E_{1A}]\!] \Rightarrow L_A[\![E_{2A}]\!] \leq L_A[\![x]\!] \wedge L_A[\![F_A]\!] \geq L_A[\![E_{BA}]\!]$ and the condition hold. The three properties then follow immediately.

In the second case of the induction step, consider $r \leq s$ and $s \leq t$ in $\overline{WA}(E_A \sqcap E_B)$. By the induction hypothesis, both constraints have solution L, so by the transitivity of $\leq$, also $r \leq t$ has solution L. If r is of the form $\{\lambda x\}$, then s and t are type variables. In this case, the desired property (2) for $r \leq t$ follows by using the induction hypothesis on $r \leq s$ (property 2) and on $s \leq t$ (property 3). Similarly, if r is a type variable, then s and t are type variables. In this case, the desired property (3) for $r \leq t$ follows by using the induction hypothesis on $r \leq s$ (property 3) and on $s \leq t$ (property 3). □

We can now prove that the *WA* constraint systems have the same fundamental property as the *GJ* constraint systems, as follows.

Lemma 4.6 *For any λ-term, there is a ⊑-least annotated version for which WA is solvable.*

Proof. Consider some λ-term E_U. We want to establish the following two facts:

1. There are finitely many annotated versions of E_U; and

2. There exists an annotated version of E_U for which *WA* is solvable.

Given these, we can derive the desired result as follows. Let A be the set of all annotated versions of E_U for which *WA* is solvable. Since A is finite (fact 1) and non-empty (fact 2), we can compute its ⊑-greatest lower bound M by a finite sequence of ⊓'s. Since ⊓ preserves solvability of *WA* (Lemma 4.5), we conclude that M ∈ A. Hence, M is the ⊑-least element of A.

To establish the first fact, note that the number of annotated versions of E_U is exponential in the size of E_U, hence finite.

To establish the second fact, let E_M be the annotated version where all abstractions and applications are dynamic. $WA(E_M)$ is clearly solvable: the assignment of Dyn to all variables yields a solution. □

Theorem 4.7 *For all λ-terms, the abstract interpretation based analysis produces ⊑-smaller annotated versions than does Gomard and Jones' analysis.*

Proof. Let E be a λ-term. Gomard and Jones' analysis produces the ⊑-least annotated version E_G of E such that $GJ(E_G)$ is solvable. By Lemma 4.4, $WA(E_G)$ is also solvable. The abstract interpretation based analysis produces the ⊑-least annotated version E_B of E such that $WA(E_B)$ is solvable (Lemma 4.6). Thus, $E_B \sqsubseteq E_G$. □

The λ-term $(\lambda x.y)$ `@` $(\lambda z.z$ `@` $z)$ from Section 3 shows that in some cases the abstract interpretation based analysis produces strictly ⊑-smaller annotated versions than does Gomard and Jones' analysis.

Mogensen [12] extended the binding-time analysis of Gomard and Jones with the use of recursive types. The type constraints are the same but types can now be regular trees, not only finite ones. This allows solutions to constraints such as $X = X \rightarrow X$. Mogensen's well-annotatedness criterion does have the property that for each λ-term there is a ⊑-least well-annotated version of it [12].

Corollary 4.8 *For all λ-terms, the abstract interpretation based analysis produces ⊑-smaller annotated versions than does Mogensen's analysis.*

Proof. Only Lemma 4.3 is influenced by the introduction of recursive types, and its proof can be reused with a few modifications. □

In some cases the abstract interpretation based analysis produces strictly $\sqsubseteq$-smaller annotated versions than does Mogensen's analysis. For example, consider the λ-term

$$(\lambda x.(x \mathbin{@} (\lambda v.v)) \mathbin{@} y) \mathbin{@} (\lambda z.z \mathbin{@} z)$$

The abstract interpretation approach yields *no* underlinings at all whereas Mogensen's approach yields

$$(\lambda x.(x \mathbin{@} (\underline{\lambda} v.v)) \mathbin{\underline{@}} y) \mathbin{@} (\lambda z.z \mathbin{\underline{@}} z)$$

We leave it to the reader to check this.

We conclude the comparison with demonstrating how the closure $\overline{WA}$ can be useful in an algorithm. The following naïve algorithm computes the $\sqsubseteq$-least Palsberg/Schwartzbach well-annotated version of a given λ-term E.

1. For each possible annotated version E_0 of E do
   (a) Generate $WA(E_0)$
   (b) Derive $\overline{WA}(E_0)$
   (c) Check if $\overline{WA}(E_0)$ is solvable.
2. Compute the greatest lower bound of the well-annotated versions of E, by a sequence of binary $\sqcap$.

Notice that when checking if $\overline{WA}(E_0)$ is solvable, it is sufficient to look for a solution where each type variable is assigned either Dyn or LAMBDA. Since there are exponentially many annotated versions of a λ-term, the algorithm runs in at least exponential time.

5 Conclusion

We have compared two different approaches to binding-time analysis and proved that the abstract interpretation approach produces better results than the type inference approach of Gomard and Jones. The latter may still be preferred in practice, however, because it (currently) can be executed faster.

We have also proved that abstract interpretation approach produces better results than the extended type inference approach of Mogensen.

To summarize, we get the following classification:

The abstract interpretation based approach
is more powerful than
Mogensen's approach
is more powerful than
Gomard and Jones' approach

Acknowledgement. The authors thank Olivier Danvy, Mitchell Wand, and the anonymous referees for helpful comments on drafts of the paper.

References

[1] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1981.

[2] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1–3):3–34, December 1991.

[3] Charles Consel. Binding time analysis for higher order untyped functional languages. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 264–272, 1990.

[4] Carsten K. Gomard. Higher order partial evaluation – HOPE for the lambda calculus. Master's thesis, DIKU, University of Copenhagen, September 1989.

[5] Carsten K. Gomard. Partial type inference for untyped functional programs. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 282–287, 1990.

[6] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[7] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 448–472. Springer-Verlag (*LNCS* 523), 1991.

[8] Sebastian Hunt and David Sands. Binding time analysis: a new PERspective. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 154–165. Sigplan Notices, 1991.

[9] Thomas P. Jensen. Strictness analysis in logical form. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 352–366. Springer-Verlag (*LNCS* 523), 1991.

[10] Neil D. Jones. Flow analysis of lambda expressions. In *Proc. Eighth Colloquium on Automata, Languages, and Programming*, pages 114–128. Springer-Verlag (*LNCS* 115), 1981.

[11] Torben Æ. Mogensen. Binding time analysis for polymorphically typed higher order languages. In *Proc. TAPSOFT'89*, pages 298–312. Springer-Verlag (*LNCS* 352), March 1989.

[12] Torben Æ. Mogensen. Self-applicable partial evaluation for pure lambda calculus. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–121, 1992.

[13] Hanne R. Nielson and Flemming Nielson. Automatic binding time analysis for a typed λ-calculus. *Science of Computer Programming*, 10:139–176, 1988.

[14] Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–363, 1993.

[15] Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 1995. To appear. Also in Proc. CAAP'94, Colloquium on Trees in Algebra and Programming, Springer-Verlag (*LNCS* 787), pages 276–290, Edinburgh, Scotland, April 1994.

[16] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference for partial types. *Information Processing Letters*, 43:175–180, 1992.

[17] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

[18] Peter Sestoft. Replacing function parameters by global variables. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 39–53, 1989.

[19] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, CMU, May 1991. CMU–CS–91–145.

[20] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamentae Informaticae*, X:115–122, 1987.