Binding Time Analysis: Abstract Interpretation versus Type Inference

Jens Palsberg
palsberg@daimi.aau.dk

Michael I. Schwartzbach
mis@daimi.aau.dk

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

April 1992

Abstract

Binding time analysis is important in partial evaluators. Its task is to determine which parts of a program can be specialized 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 the abstract interpretation approach of Bondorf, and the second is the type inference approach of Gomard. Both apply to the untyped lambda calculus. We prove that Bondorf's analysis is better than Gomard's.

1 Introduction

In this paper we compare two techniques for doing *binding time analysis* of terms in the untyped lambda calculus, see figure 1. The binding times we

are concerned with are "known" (compile-time) and "unknown" (run-time).

$$E ::= x \mid \lambda x.E \mid E_1 \mathbin{@} E_2$$

Figure 1: The lambda calculus.

The input to a binding time analysis is a term together with some of its expected input. The output is information about which applications can be reduced without knowledge of the remaining input. This information can be used by a partial evaluator to obtain a specialized term. The specialization proceeds by reducing applications that have been determined to be reducible. Note that an application is reducible if the function part is known.

For example, consider the term $(\lambda x.x \mathbin{@} y) \mathbin{@} z$ and suppose that the values of y and z are unknown. If a binding time analysis can determine that the function part of the outermost application is known at partial evaluation-time, then the term can be specialized to $z \mathbin{@} y$. Notice that the other application cannot be reduced.

The output of a binding time analysis can be presented as an *annotated* version of the analyzed term. In the annotated term, all applications that cannot be reduced are underlined; lambda abstractions that cannot take part in a reduction will also be underlined. The language of annotated terms is usually called a 2-level lambda calculus, see figure 2. Its semantics is given by the semantics of the underlying pure term. Given a 2-level term, a specializer proceeds by reducing applications that are not underlined.

$$E ::= x \mid \lambda x.E \mid E_1 \mathbin{@} E_2 \mid \underline{\lambda} x.E \mid E_1 \mathbin{\underline{@}} E_2$$

Figure 2: The 2-level lambda calculus.

For example, consider again the term $(\lambda x.x \mathbin{@} y) \mathbin{@} z$. A binding time analysis may here produce the annotated term $(\lambda x.x \mathbin{@} y) \mathbin{@} z$. It could also produce$(\underline{\lambda} x.x \mathbin{\underline{@}} y) \mathbin{\underline{@}} z$.

Following Gomard and Jones [5], we partially order the set of 2-level terms as follows. Given 2.level terms E and E', $E \sqsubseteq E'$ iff they have the same underlying pure term and E' has the same and possibly more underlinings than E. For example, $(\lambda x.x \mathbin{\underline{@}} y) \mathbin{@} z) \sqsubseteq ((\underline{\lambda} x.x \mathbin{\underline{@}} y) \mathbin{\underline{@}} z)$.

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 better than another, if it always produces $\sqsubseteq$-smaller 2-level terms than the other.

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

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 [8], Bondorf [1], Consel [2], and Hunt and Sands [7]. For examples of the latter that are also applicable to higher-order languages, see the work of Nielson and Nielson [9], and Gomard [4]. Only little is known, however, about the relative quality of these analyses.

We will compare two recent binding time analyses. Both can be executed in polynomial time and both are successfully used in existing partial evaluators. The first is the abstract interpretation approach of Bondorf [1] which is used in the Similix partial evaluator. The second is the type inference approach of Gomard [4] which is used in the Lambdamix partial evaluator.

The two chosen analyses are defined for slightly different languages. Bondorf's is defined for a subset of SCHEME, whereas Gomard's is defined for a lambda calculus with constants. To be able to compare them we restrict both their definitions to merely the pure lambda calculus.

This paper proves that the binding time analysis of Bondorf is better than that of Gomard, in the above sense.

In following section we give the definitions of the two binding analyses, and in section 3 we prove our result.

2 The Formal Systems

To simplify matters, we will assume that the input to the binding time analysis is just one 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 $(\lambda x.E)(E')$. Note that y is free also in this term. Thus, free variables henceforth correspond to unknown input.

Not all annotated versions of a term are acceptable. Informally, an annotation is *sound* if the specializer indeed can reduce all applications that are not underlined. Note that the soundness of 2-level terms is in general undecidable.

Both Bondorf's and Gomard's analyses are algorithms that compute sound annotations. Furthermore, for each of them there is a soundness *predicate* on 2-level terms such that the algorithm computes the $\sqsubseteq$-*least* 2-level term that is sound with respect to this predicate. Both these soundness predicates approximate the optimal (undecidable) soundness predicate, such that they always say "unsound" when the optimal one does.

The soundness predicates can be understood as *specifications* of binding time analyses. For each of the predicates, we will say that if an algorithm always produces a term for which the predicate is true, then it is a binding time analysis; it meets the specification.

Both soundness predicates can be presented via constraint systems. For each

of the predicates, the idea is that it is true of a 2-level term E_0 iff a constraint system generated from E_0 is solvable. Both constraint systems are generated as follows. First, the lambda 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 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.

Bondorf's and Gomard's analvses emnlov constraints over different domains. In Gomard's analysis, type variables range over the following types:

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

The intuition behind this Dyn is that a term with this type has unknown value; Dyn abbreviates Dynamic (Dyn is called untyped by Gomard).

The constraints are generated inductively in the syntax, see figure 3. We let GA denote the global constraint system. Sometimes we write GA(E_0) to emphasize that the constraint system is generated from E_0. Essentially these constraints were presented by Henglein [6]; they are in the style of Wand [12]. For any pure term, there is a $\sqsubseteq$-least annotated version for which GA is true, for a proof see Gomard's Master's thesis [3]. Henglein gave a pseudo-linear time algorithm for computing this $\sqsubseteq$-least term [6].

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

In addition, for each free variable f in E_0
there is an initial contraint $[\![\mathrm{f}]\!] = \mathsf{Dyn}$.

Figure 3: Gomard's soundness predicate, GA(E_0).

Bondorf's analysis is based on an abstract interpretation called *closure analysis*. The *closures* of a term are simply the subterms corresponding to lambda abstraction. A closure analysis approximates for every subterm the set of

possible closures to which it may evaluate [11, 1]. Bondorf's binding time analysis is simply a closure analysis that in addition to the other 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 by Gomard.

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

Phrase:	Safety constraints:
$\lambda x.E$	$[\![\lambda x.E]\!] \subseteq \textsc{lambda}$
$E_1 \mathbin{@} E_2$	$[\![E_1]\!] \subseteq \textsc{lambda}$

Phrase:	Connecting constraints:
$E_1 \mathbin{@} E_2$	For every $\lambda x.E$ in E_0,
	if $\lambda x \in [\![E_1]\!]$ then
	$[\![E_2]\!] \subseteq [\![x]\!] \wedge [\![E_1 \mathbin{@} E_2]\!] \supseteq [\![E]\!]$
$E_1 \mathbin{\underline{@}} E_2$	if $\mathsf{Dyn} \in [\![E_1]\!]$ then
	$\{\mathsf{Dyn}\} \subseteq [\![E_2]\!] \wedge \{\mathsf{Dyn}\} \subseteq [\![E_1 \mathbin{\underline{@}} E_2]\!]$

In addition, for each free variable f in E_0
there is an initial contraint $[\![\mathsf{f}]\!] \supseteq \{\mathsf{Dyn}\}$.

Figure 4: Bondorf's soundness predicate, BA(E_0).

In Bondorf's analysis, type variables range over sets of closures and Dyn. We denote by LAMBDA the finite set of all lambda tokens in E_0, the main term. The constraints are generated from the syntax, see figure 4. As a conceptual aid, the constraints are grouped into *basic*, *safety*, and *connecting* constraints.

The connecting constraints reflect the relationship between formal and actual arguments and results. The condition $\lambda x \in [\![E]\!]$ been states that the two guarded inclusions are relevant only if the closure denoted by λx is a possible result of E_1. Similarly, the condition $\mathsf{Dyn} \in [\![E_1]\!]$ states that the two guarded

inclusions are relevant only if the value of E_1 is unknown.

We let BA denote the global constraint system. Sometimes we write BA(E_0) to emphasize that the constraint system is generated from E_0. The first basic constraint and the first connecting constraint yield a closure analysis of untyped terms.

It is novel to present the specification of Bondorf's analysis via constraint systems. This idea makes possible the proof of comparison of the two analyses, see later. The BA constraint systems has the same fundamental property as the GA constraint systems, as follows.

Proposition: For any pure term, there is a $\sqsubseteq$-least annotated version for which BA is solvable.
Proof: Consider some pure term E_U. Let A be thy set of annotated versions of E_U for which BA is solvable. Thy idea in the proof is to represent A as the solutions of another constraint system C.

Phrase:	Basic constraints:
$\lambda x.E$	$[\![\lambda x.E]\!] \supseteq \{\lambda x\}$

Phrase:	Connecting constraints:

$E_1 \mathbin{@} E_2$	For every $\lambda x.E$ in E_U,
	if $\lambda x \in [\![E_1]\!]$ then
	$[\![E_2]\!] \subseteq [\![x]\!] \wedge [\![E_1 \mathbin{@} E_2]\!] \supseteq [\![E]\!]$
	if $\mathsf{Dyn} \in [\![E_1]\!]$ then
	$\{\mathsf{Dyn}\} \subseteq [\![E_2]\!] \wedge \{\mathsf{Dyn}\} \subseteq [\![E_1 \mathbin{@} E_2]\!]$

In addition, for each free variable f in E_U
there is an initial contraint $[\![\mathrm{f}]\!] \supseteq \{\mathsf{Dyn}\}$.

Figure 5: The constraint system C.

The constraint system C is derived from the pure term E by a three-step process like the one employed above. First, the lambda term is α-converted so that every λ-bound variable is distinct. 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, see figure 5. We let S denote

the set of solutions of C, it is ordered by point-wise inclusion.

It is easy to see that S is closed under point-wise intersection. Further-more, we can always obtain a solution of C by assigning LAMBDA $\cup\{\mathsf{Dyn}\}$ to all variables, Thus, S has a least element.

We now define a function

$$\mathsf{annotate} : \mathrm{S} \to \mathrm{A}$$

Given a solution $\mathrm{L} \in \mathrm{S}$, annotate will underline those abstractions and applications in E_U whose type variable contains Dyn in L. This yields a two-level term E_0.

If we can show that annotate is monotone and surjective onto A, then we can proceed as follows. Let L_M be the least solution of C, and let $E_A \in \mathrm{A}$ be given. Since annotate is surjective, we can find L so that $\mathsf{annotate}(\mathrm{L}) = E_A$. Since L_M is less than L, and since annotate is monotone, we get that $\mathsf{annotate}(\mathrm{L}_M) \sqsubseteq E_A$. Thus, $\mathsf{annotate}(\mathrm{L}_M)$ is the least element of A.

To see that annotate has range A, note that the type variables used in C and $\mathrm{BA}(E_0)$ are the same up to a renaming which removes underlinings. We can thus transform $\mathrm{L} \in \mathrm{S}$ into an assignment L_0 of sets to the type variables used in $\mathrm{BA}(E_0)$. To get that $E_0 \in \mathrm{A}$ we then only need to establish that L_0 is a solution to $\mathrm{BA}(E_0)$. This is straightforward to check, as follows.

Consider first subterms of E_0 of the form $\lambda x.E$. Since $\mathrm{L} \in \mathrm{S}$ we have that $[\![\lambda x.E]\!] \supseteq \{\lambda x\}$. Moreover, since $\lambda x.E$ did not get underlined by annotate, we have that $\mathsf{Dyn} \notin [\![\lambda x.E]\!]$ in L, so $[\![\lambda x.E]\!] \subseteq$ LAMBDA in $\mathrm{BA}(E_0)$.

Consider next subterms of E_0 of the form $\underline{\lambda} x.E$. Since the lambda got underlined, we have that $\mathsf{Dyn} \in [\![\underline{\lambda} x.E]\!]$ in L. Thus, in $\mathrm{BA}(E_0)$ we have that $[\![\underline{\lambda} x.E]\!] \supseteq \{\mathsf{Dyn}\}$.

Subterms of E_0 of the forms $E_1 \mathbin{@} E_2$ and $E_1 \mathbin{\underline{@}} E_2$ are treated in similar fashiuns. Thus, L_0 is a solution of $\mathrm{BA}(E_0)$ so annotate has range A.

It is immediate that annotate is monotone.

To see that annotate is surjective onto A, we proceed as follows. Let $E_A \in \mathrm{A}$ be given. Let L_0 be a solution of $\mathrm{BA}(E_A)$. Since the type variables used in C and $\mathrm{BA}(E_A)$ are the same up to renaming, we can transform L_0 into

an assignment L of sets to type variables used in C. It is immediate that L is a solution of C because it must satisfy fewer constraints than in BA. To see that $\mathsf{annotate}(\mathrm{L}) = E_0$, note that in E_0 an abstraction or an application is underlined iff the corresponding type variable contains Dyn in L_0. This completes the proof. □

Given a pure term, the $\sqsubseteq$-least annotated version for which BA is solvable can be computed in cubic time. The idea is to compute the least solution of the C constraint system and then apply $\mathsf{annotate}$, see [10]. Bondorf's original algorithm [1] was given in a compositional style; its time complexity appears to be worse than cubic time.

Note that Gomard's analysis currently can be computed faster than Bondorf's. Note also that GA may force applications to be underlined because of typing problems, even though they have a known function part. This is not so in Bondorf's analysis. For example, consider the pure term $(\lambda h.(h \mathbin{@} I) \mathbin{@} (h \mathbin{@} f)) \mathbin{@} I$ where $I = \lambda x.x$ and f is a free variable. Bondorf's algorithm will produce *no* underlinings at all, whereas Gomard's algorithm will produce $(\lambda h.(h \mathbin{\underline{@}} I') \mathbin{\underline{@}} (h \mathbin{\underline{@}} f)) \mathbin{@} I'$ where $I' = \underline{\lambda} x.x$. Thus, intuitively Bondorf's analysis does better than Gumard's analysis. This is indeed so, we now prove it formally.

3 Comparison

We now show that the binding time analysis of Bondorf produces at most as many underlinings as the analysis of Gomard. We do this by proving for all 2-level lambda terms that if GA is solvable, then so is BA. This implies the desired result because given any pure term, BA is in particular solvable for the $\sqsubset$-*least* annotated version for which GA is solvable.

The proof involves several lemmas, see figure 6. The main technical problem to be solved is that BA and GA are constraint systems over two different domains, sets versus types. This makes a direct comparison hard. We overcome this problem by applying solvability preserving maps into constraints over a common two-point domain.

We first show that the possibly *conditional* constraints of BA are equivalent to a set of *unconditional* constraints (UBA). UBA is obtained from BA by

repeated transformations. A set of constraints can be described by a pair (C, U) where C contains the conditional constraints and U the unconditional ones. We have two different transformations:

a) If U is solvable and c holds in the minimal solution, then $(C \cup \{c \Rightarrow K\}, U)$ becomes $(C, U \cup \{K\})$.

b) Otherwise, (C, U) becomes $(\emptyset, U)$.

This process clearly terminates, since each transformation removes at least one conditional constraint.

Lemma 1: BA is solvable *iff* UBA is solvable.
Proof: We show that each transformation preserves solvability.

a) We know that U is solvable, and that c holds in the minimal solution. Assume that $(C \cup \{c \Rightarrow K\}, U)$ is solvable. The condition c must hold and, hence, so must K. But then $(C, U \cup \{K\})$ is solvable. Conversely, assume that $(C, U \cup \{K\})$ is solvable. Then so is $(C \cup \{c \Rightarrow K\}, U)$, since K holds whether c does or not.

b) If (C, U) is solvable, then clearly so is $(\emptyset, U)$. Assume now that $(\emptyset, U)$ is solvable, and that no condition in C holds in the minimal solution of U. Then clearly (C, U) can inherit this solution.

It follows that solvability is preserved for any sequence of transformations. □

We now introduce a particularly simple kind of constraints, which we cal *2-constraints*. Here variables range over the set $\{\lambda, \mathsf{Dyn}\}$. We define a function ϕ which maps UBA constraints into 2-constraints. Individual constraints are mapped as follows:

UBA	ϕ(UBA)
$X \subseteq Y$	$X = Y$
$X \subseteq$ LAMBDA	$X = \lambda$
$X \supseteq \{\lambda x\}$	$X = \lambda$
$X \supseteq \{\mathsf{Dyn}\}$	$X = \mathsf{Dyn}$

It turns out that ϕ preserves solvability in the inverse direction.

Lemma 2: If ϕ(UBA) is solvable, then so is UBA.
Proof: Assume that L is a solution of ϕ(UBA). We obtain a (non-minimal) solution of UBA by assigning LAMBDA to X if $L(X) = \lambda$, and assigning $\{\mathsf{Dyn}\}$ to X if $L(X) = \mathsf{Dyn}$. □

Next, we define the closure $\overline{\text{GA}}$ as the smallest set that contains GA and is closed under symmetry, reflexivity, and transitivity of $=$, and also closed under the following property: if $\alpha \to \beta = \alpha \to \beta'$, then $\alpha = \alpha'$ and $\beta = \beta'$. Hardly surprising, this closure preserves solvability.

Lemma 3: GA is solvable *iff* $\overline{\text{GA}}$ is solvable.
Proof: The implication from right to left is immediate. Assume that GA is solvable. Equality is by definition symmetric, reflexive, and transitive. The additional property will also be true for any solution. Hence, $\overline{\text{GA}}$ inherits all solutions of GA. □

We define a function ψ which maps $\overline{\text{GA}}$ into 2-constraints. Individual constraints are mapped as follows:

$\overline{\text{GA}}$	$\psi(\overline{\text{GA}})$
$X = Y$	$X = Y$
$X = \alpha \to \beta$	$X = \lambda$
$X = \mathsf{Dyn}$	$X = \mathsf{Dyn}$

We show that ψ preserves solvability.

Lemma 4: If $\overline{\text{GA}}$ is solvable, then so is $\psi(\overline{\text{GA}})$.
Proof: Assume that L is a solution of $\overline{\text{GA}}$. We obtain a solution of $\psi(\overline{\text{GA}})$ by assigning λ to X if $L(X) = \alpha \to \beta$, and assigning Dyn to X if $L(X) = \mathsf{Dyn}$. Thus, the function ψ acts as a quotient map on constraint systems. □

We now show the crucial connection between Bondorf's and Gomard's analyses.

Lemma 5: The UBA constraints are contained in the $\overline{\text{GA}}$ constraints, in the sense that $\phi(\text{UBA}) \subseteq \psi(\overline{\text{GA}})$.
Proof: We perform an induction in the number of transformations performed on BA.

The induction base is the BA configuration (C, U). Here U contains all the basic, safety, and initial cunstraints. For any $\lambda x.E$, BA yields the constraint $[\![\lambda x.E]\!] \supseteq \{\lambda x\}$ which by ϕ is mapped to $[\![\lambda x.E]\!] = \lambda$. GA yields the constraint $[\![\lambda x.E]\!] = [\![x]\!] \to [\![E]\!]$ which by ψ is mapped to $[\![\lambda x.E]\!] = \lambda$ as well. A similar argument applies to the other four kinds of basic and safety constraints, and also to the initial constraints, Thus, we have established the induction base.

For the induction step we assume that $\phi(U) \subseteq \psi(\overline{\mathrm{GA}})$. If we use the b)-transformation and move from (C, U) to $(\emptyset, U)$, then the result is immediate. Assume therefore that we apply the a)-transformation. Then U is solvable, and some condition has been established in the minimal solution. There are two cases, one for $E_1 \mathbin{@} E_2$ and one fur $E_1 \mathbin{\underline{@}} E_2$.

Assume first that some condition $\lambda x \in [\![E_1]\!]$ been has been established for the application $E_1 \mathbin{@} E_2$ in the minimal solution. This opens up for two new connecting constraints: $[\![E_2]\!] \subseteq [\![x]\!]$ and $[\![E_1 E_2]\!] \supseteq [\![E]\!]$. We must show that corresponding equalities hold in $\overline{\mathrm{GA}}$. The only way to enable the condition in the minimal solution of U is to have a chain of U-constraints:

$$\{\lambda x\} \subseteq [\![\lambda x.E]\!] \subseteq X_1 \subseteq X_2 \subseteq \cdots \subseteq X_n \subseteq [\![E_1]\!]$$

From the definitions of ϕ and ψ on constraints between variables, and by applying the induction hypothesis, we get that in $\overline{\mathrm{GA}}$ we have

$$[\![\lambda x.E]\!] = X_1 = X_2 = \cdots = X_n = [\![E_1]\!]$$

From the GA constraints $[\![\lambda x.E]\!] = [\![x]\!] \to [\![E]\!]$ and $[\![E_1]\!] = [\![E_2]\!] \to [\![E_1 E_2]\!]$ and the closure properties of $\overline{\mathrm{GA}}$ it follows that $[\![E_2]\!] = [\![x]\!]$ and $[\![E_1 E_2]\!] = [\![E]\!]$, which was our proof obligation.

Assume next that some condition $\mathsf{Dyn} \in [\![E_1]\!]$ has been established for the application $E_1 \mathbin{\underline{@}} E_2$ in the minimal solution. This opens up for two new connecting constraints: $\{\mathsf{Dyn}\} \subseteq [\![E_2]\!]$ and $\{\mathsf{Dyn}\} \subseteq [\![E_1 \mathbin{\underline{@}} E_2]\!]$. They are by ϕ mapped to $[\![E_2]\!] = \mathsf{Dyn}$ and $[\![E_1 \mathbin{\underline{@}} E_2]\!] = \mathsf{Dyn}$. For $E_1 \mathbin{\underline{@}} E_2$, GA generates three constraints, two of which are mapped by ψ to the two constraints from before.

Thus, we have established the induction step. As UBA is obtained by a finite number of transformations, the result follows. □

This allows us to complete the final link in the chain.

Lemma 6: *If* $\overline{\text{GA}}$ is solvable, then so is UBA.
Proof: Assume that $\overline{\text{GA}}$ is solvable. From lemma 4 it follows that so is $\psi(\overline{\text{GA}})$. Since from lemma 5 $\phi(\text{UBA})$ is a subset, it must also be solvable. From lemma 2 it follows that UBA is solvable. □

We conclude that BA is at least as powerful as GA.

Theorem: If GA is solvable, then so is BA.
Proof: We need only to bring the lemmas together, as indicated in figure 6. □

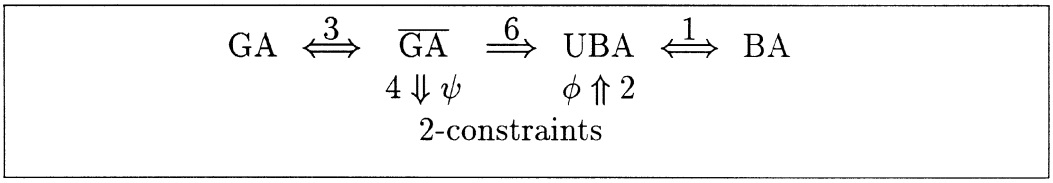

Figure 6: Solvability of constraints.

Corollary: For all pure terms, Bondorf's analysis produces $\sqsubseteq$-smaller annotated versions than does Gomard's analysis.

Proof: Let E be as pure term. Gomard's analysis produces the $\sqsubseteq$-least annotated version E_G of E such that $\text{GA}(E_G)$ is solvable. By the theorem, $\text{BA}(E_G)$ is also solvable. Bondorf's analysis produces the $\sqsubseteq$-least annotated version E_B of E such that $\text{BA}(E_G)$ is solvable Thus, $E_B \sqsubseteq E_G$. □

The example in the previous section shows that for some terms Bondorf's analysis produces *strictly* $\sqsubseteq$-smaller annotated versions than does Gomard's analysis. The proof of our theorem sheds some light on why and how BA accepts more safe terms than GA. Consider a solution of GA that is transformed into a solution of BA according to the strategy implied in figure 6. All closure sets will be the maximal set LAMBDA. Thus, the more fine-grained distinction between individual closures is lost.

Our result is still valid if we allow GA to use recursive types. Here the GA constraints are exactly the same, but the types are changed from finite to regular trees. This allows solutions to constraints such as $X = X \to X$. Only lemma 4 is influenced, but the proof carries through with virtually no

modifications. Even with recursive types, the term $(\lambda h.(h \mathbin{@} I) \mathbin{@} (h \mathbin{@} f)) \mathbin{@} I$ is annotated as before. Hence, BA is also at least as powerful as GA with recursive types.

4 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. The latter may still be preferred in practice, however, because it (currently) can be executed faster.

Strictness analysis is another example of a static analysis for which there are both abstract interpretation approaches and type inference ones. In future work, we hope to compare such analyses.

References

[1] Anders Bondorf. Automatic autoprojection of higher order recursive equations. In *Proc. ESOP'90, European Symposium on Programming.* Springer-Verlag (*LNCS* 432), 1990.

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

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

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

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

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

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

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

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

[10] Jens Palsberg and Michael I. Schwartzbach. Polyvariant analysis of the untyped lambda calculus. Technical Report DAIMI PB-386, Computer Science Department, Aarhus University, 1992. Submitted for publication.

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

[12] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamentae Informaticae*, X:115–1227 1987.