A Unified Framework for Binding-Time Analysis

Peter Thiemann

Wilhelm-Schickard-Institut, Universität Tübingen, Sand 13, D-72076 Tübingen, Germany, E-mail: thiemann@informatik.uni-tuebingen.de

Abstract. Binding-time analysis is a crucial part of offline partial evaluation. It is often specified as a non-standard type system. Many type-based binding-time analyses are reminiscent of simple type systems with additional features like recursive types. We make this connection explicit by expressing binding-time analysis with annotated type systems that separate the concerns of type inference from those of binding-time annotation. The separation enables us to explore a design space for binding-time analysis by varying the underlying type system and the annotation strategy independently. The result is a classification of different monovariant binding-time analyses which allows us to compare their relative power. Due to the systematic approach we uncover some novel analyses.

A partial evaluator separates the computation of a source program into two or more stages [7, 20]. Using the (*static*) input of the first stage it transforms a source program into a specialized *residual program*. Application of the residual program to the (*dynamic*) input of the second stage yields the same answer as application of the source program to the entire input. The *binding time* of an input is the information whether it is static or dynamic.

Binding-time analysis (BTA) is a prepass of a partial evaluator that annotates each expression in the source program with the earliest (static is earlier than dynamic) time at which it can be evaluated. The actual specializer is a mere interpreter of annotated programs that executes the static expressions and generates code for the remaining ones.

Binding-time analyses come in two flavors: A *monovariant* BTA computes a single mapping of program points to binding-times, whereas a *polyvariant* BTA allows for several such mappings. Both alternatives have their merits. Monovariant BTAs are simple and efficient to implement [4, 14, 15, 18, 20]. However, in some applications static and dynamic values flow through the same program points, which forces a monovariant BTA to annotate the program points as dynamic. A polyvariant BTA [5, 6] yields better results in these cases, but is also considerably more expensive.

In the current work we concentrate on monovariant BTAs for the lambda calculus as they are used in many partial evaluators [4, 14, 15, 18]. Our analyses achieve some degree of polyvariance because we admit liberal binding-time coercions and rely on a more precise inclusion-based flow analysis framework.

BTA is often presented as a monolithic analysis which makes it unnecessarily hard to understand and to reason about [2, 3, 14, 18, 21]. Recent work has shown the possibility to modularize BTA into several stages [4, 12, 13]. All of these works rely more or less implicitly on using type systems and extending them with annotations. However, none of the latter works really exploits the potential of the modularization, namely comparing variants of the analyses with respect to their accuracy.

We consider a modest staging of BTA in two phases, building on the ideas of annotated type systems: flow-type analysis and binding-time annotation. Building a BTA on top of a flow-type system has some advantages over approaches where types and binding times are intermingled [2, 14, 18]. First, it is easier to implement a modular algorithm. Second, the approach applies to typed and untyped languages. Third, it clearly separates different concerns: binding-time propagation and type correctness. By not confusing them, we avoid a problem in Henglein's type inference algorithm [18], which was discovered by Birkedal and Welinder [2]. We investigate two variations of flow-type systems, an equational system and an inclusion-based system which adds subtyping to the equational system. On top of these, we investigate two binding-time annotation strategies, a local one and a global one. The essential difference between these two lies in additional binding-time coercion rules of the local strategy. We also identify an instance of our framework that is equivalent to Gomard's BTA [14].

We present a modular algorithm for this BTA framework. Its run time ranges from almost-linear (for the equational system with the global annotation strategy) to exponential (for the inclusion-based system with the local annotation strategy), relying on well-known algorithms for flow-type analysis [18,30].

We have compared the relative strengths of the different instantiations of our algorithm. We can show that the equational and inclusion-based approaches are equivalent under the global annotation scheme. This is somewhat unexpected and shows that a simple-minded annotation strategy can throw away information which is present in the underlying flow-type system. Otherwise, we show that the local strategy produces strictly better results than the global strategy for both type systems. Furthermore, the local variant of the inclusion-based system is strictly better than the local variant of the equational system. Finally, we show how to improve the results of a local equational BTA by using eta-expansors [11]. Figure 1 gives an overview of our results.

As far as we know, the following issues have not been investigated previously:

- the generic algorithm for BTA based on type automata and annotations,
- the combination of equational flow-typing with a local annotation scheme,
- inclusion-based BTA in a type-based setting, and
- an algorithm to improve the results of BTA using eta-expansors.

1 Basic Framework

For concreteness of our exposition, we have chosen a lambda calculus extended with numbers, pairs, and conditionals.

$$e ::= x \mid \lambda x.e \mid e@e \mid 0 \mid \mathrm{succ}\ e \mid (e,e) \mid \pi_1 e \mid \pi_2 e \mid \mathrm{if0}\ e\ e\ e$$

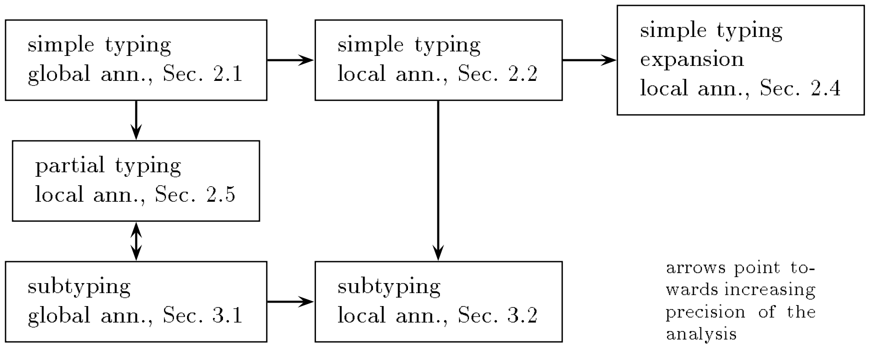

Fig. 1. Overview of Results

In the subsequent text, we use let-expressions "let $x = e_1$ in e_2" as syntactic sugar for "$(\lambda x.e_2)@e_1$." We assume that each subterm e is identified by a unique *program point* $\ell \in L$ which we indicate by superscripting e^ℓ where necessary.

We also define an annotated version of the syntax which serves to express the output of the BTA. β ranges over binding-time (bt) annotations, for example S and D. $\mathrm{lift}^{\beta,\beta'}\ E$ denotes a binding-time coercion (from β to β') for integers.

$$E ::= x \mid \lambda^\beta x.E \mid E@^\beta E \mid 0 \mid \mathrm{succ}^\beta\ E \mid (E, E)^\beta \mid \pi_i^\beta E \mid \mathrm{if0}^\beta\ E\ E\ E \mid \mathrm{lift}^{\beta,\beta'}\ E$$

We define $|E|$ as the term obtained by dropping all annotations and lifts from E. E is a *completion* of e if $e = |E|$.

We employ a standard type language with $\perp$ and $\top$ types denoting the empty type and the type of all values, which are used in the inclusion-based system. Types can be recursive without an explicit fixpoint constructor in the language.

$$\tau ::= \perp \mid \top \mid \mathrm{int} \mid \tau \to \tau \mid \tau \times \tau$$

1.1 Type Inference

From an abstract operational view, type inference takes a term as input and constructs a directed graph, every node of which is annotated by a type constructor, and a mapping M from the set L of program points to the nodes of the graph. We call this directed graph a type automaton (cf. [29]).

Definition 1. A *type automaton* over a set of program points L is a Moore machine [19] $\mathcal{A} = (Q, \Sigma, X, \delta, \mathrm{lab})$ where Q is the set of states, $\Sigma = \{1, 2, \ldots\}$ is the input alphabet, X is the set of labels, which are type constructors, $\delta : Q \times \Sigma \to Q$ is the partial transition function, and $\mathrm{lab} : Q \to X$ the labeling function. For any state ϕ, the transitions $\delta(\phi, 1), \ldots, \delta(\phi, n)$ are defined if and only if $\mathrm{lab}(\phi)$ is an n-ary type constructor.

The additional mapping M from the set L of program points to Q determines for each program point ℓ a subautomaton $\mathcal{A}(\ell)$ with initial state $M(\ell)$ that describes the type of the construct at ℓ. Recursive types arise naturally in this framework: they are type automata that recognize infinite languages.

It is helpful to consider states of the type automaton as type variables and type inference as a process that refines an initial non-deterministic automaton by unification until unification fails or the automaton is deterministic.

1.2 Binding-Time Annotation

We relate binding times to states of an automaton by giving a map $B : Q \to \mathrm{BT}$. BT can be the standard domain $\{0, 1\}$ aka $\{S, D\}$, as well as a multi-level binding-time domain $\{0, \dots, D\}$ where $D \geq 1$ is the maximum binding time [13]. Each map B corresponds to an annotation of the occurrences of type constructors in the recursive type denoted by the automaton. However, it is often more convenient to talk about binding-time-annotated types ρ.

Definition 2 Binding-Time-Annotated Types.

$$\rho ::= \bot^{\beta} \mid \top^{\beta} \mid \mathrm{int}^{\beta} \mid \rho \to^{\beta} \rho \mid \rho \times^{\beta} \rho$$

Annotated types may also be recursive. We write $\upsilon^{\beta} = \rho$ in order to peel off the top-level binding-time annotation.

Not every binding-time annotation is admissible. For example, the annotated type $\mathrm{int}^0 \to^D \mathrm{int}^0$ does not make sense because it specifies a dynamic function where the argument and result are available at time 0, i.e., statically. Clearly, the same kind of restriction must be imposed on all other type constructors: the components of a constructed value are not available before the constructor.

Definition 3 Well-formedness. Let $\mathcal{A} = (Q, \Sigma, X, \delta, \mathrm{lab})$ be a type automaton and $B : Q \to \mathrm{BT}$ a binding-time annotation.

$(\mathcal{A}, B)$ is *well-formed* if for all $\phi \in Q$: either $\mathrm{lab}(\phi) = \top$ and $D = B(\phi)$, or $\forall i.\ \phi' = \delta(\phi, i)$ defined $\Rightarrow B(\phi) \leq B(\phi')$.

Equivalently, we can express the well-formedness criterion as a predicate on annotated types [12]:

Definition 4 Well-formed Types. An annotated type ρ is well-formed if wft ρ is derivable from the axioms wft $\top^D$ and wft int^{β} and the rules

$$\frac{\mathrm{wft}\ \upsilon_1^{\beta_1} \quad \mathrm{wft}\ \upsilon_2^{\beta_2} \quad \beta \leq \beta_1 \quad \beta \leq \beta_2}{\mathrm{wft}\ \upsilon_1^{\beta_1} \to^{\beta} \upsilon_2^{\beta_2}} \qquad \frac{\mathrm{wft}\ \upsilon_1^{\beta_1} \quad \mathrm{wft}\ \upsilon_2^{\beta_2} \quad \beta \leq \beta_1 \quad \beta \leq \beta_2}{\mathrm{wft}\ \upsilon_1^{\beta_1} \times^{\beta} \upsilon_2^{\beta_2}}$$

Def. 3 provides a set of inequalities that every well-formed annotation must fulfill. These inequalities give rise to a set of *binding-time constraints* (BTC) from a type automaton in the obvious way: Associate a binding-time variable β_ϕ with each state ϕ of the automaton. This variable captures the annotation $\beta_\phi = B(\phi)$. The BTC sets only involve BTCs of the forms:

$$1.\ i \leq \beta \text{ (for } 0 \leq i \leq D), \quad 2.\ \beta_1 \leq \beta_2, \text{ and} \quad 3.\ \beta_1 = \beta_2$$

where $\leq$ and $=$ are the usual relations over $\mathrm{BT} \subseteq \mathbf{N}$.

Definition 5. A *solution* for a set of BTCs is an assignment from binding-time variables to BT which satisfies all constraints.

Fact. Every set of BTCs has a unique least solution.

Lemma 6. *There is an algorithm to compute the least solution for BTC set S in $O(|S|)$ time.*

Proof. First rewrite inequations of the form $a \leq \beta$ to equations of the form $\beta = a \sqcup \beta$ (where $\sqcup$ denotes maximum). This takes $O(|S|)$ time and preserves all solutions. A theorem of Seidl [32, Theorem 10] provides an algorithm to compute the least solution to such a set of equations over $\mathbf{N}$ in $O(|S|)$ time.

1.3 Annotation Strategies

Given a term and its type automaton, we have two choices to perform a binding-time annotation. One choice is the *global strategy* that provides a single well-annotation for the entire type automaton. It is similar to what standard algorithms provide [2, 14, 18].

Another choice is the *local strategy.* For each program point ℓ, it constructs the subautomaton $\mathcal{A}(\ell)$ of the type automaton. For each of these automata we have to provide well-annotations, but now we also need to respect *phase constraints* that relate the types of "neighboring" program points. These phase constraints can prescribe binding-time coercions that need to be inserted into the term to make it well-annotated. Below, we will make this notion more precise.

It is important to observe that every global annotation gives rise to a local annotation. Given a type automaton $\mathcal{A} = (Q, \ldots)$ and a global annotation B we construct the local annotation B_e of $\mathcal{A}(\ell) = (Q_\ell, \ldots)$ for expression e^ℓ. By construction of the $\mathcal{A}(\ell)$ there is an injective mapping $\iota : Q_\ell \to Q$ and we can define B_e by $B_e := B \circ \iota$, i.e., by restricting B to the set of states of $\mathcal{A}(\ell)$.

2 Equational Binding-Time Analysis

$$\text{(var)}\ \Gamma\{x : \tau\} \vdash x : \tau \qquad \text{(const)}\ \Gamma \vdash 0 : \text{int} \qquad \text{(succ)}\ \frac{\Gamma \vdash e : \text{int}}{\Gamma \vdash \text{succ}\ e : \text{int}}$$

$$\text{(abs)}\ \frac{\Gamma\{x : \tau_2\} \vdash e : \tau_1}{\Gamma \vdash \lambda x.e : \tau_2 \to \tau_1} \qquad \text{(app)}\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 @ e_2 : \tau_1}$$

$$\text{(pair)}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \text{(proj)}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i e : \tau_i}$$

$$\text{(if)}\ \frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if0}\ e_1\ e_2\ e_3 : \tau}$$

Fig. 2. Simple Types

The underlying type system of equational BTA is the system of simple types with recursion. Figure 2 gives the standard rules.

$$\Gamma\{x:\rho\}\vdash x \rightsquigarrow x:\rho \qquad \Gamma\vdash 0 \rightsquigarrow \mathrm{lift}^{0,\beta}\ 0 : \mathrm{int}^\beta \qquad \frac{\Gamma\vdash e \rightsquigarrow E : \mathrm{int}^\beta}{\Gamma\vdash \mathrm{succ}\ e \rightsquigarrow \mathrm{succ}^\beta\ E : \mathrm{int}^\beta}$$

$$\frac{\Gamma\{x:\rho_2\}\vdash e \rightsquigarrow E:\rho_1 \quad \mathrm{wft}\ \rho_2 \to^\beta \rho_1}{\Gamma\vdash \lambda x.e \rightsquigarrow \lambda^\beta x.E : \rho_2 \to^\beta \rho_1} \qquad \frac{\Gamma\vdash e_1 \rightsquigarrow E_1 : \rho_2 \to^\beta \rho_1 \quad \Gamma\vdash e_2 \rightsquigarrow E_2 : \rho_2}{\Gamma\vdash e_1 @ e_2 \rightsquigarrow E_1 @^\beta E_2 : \rho_1}$$

$$\frac{\Gamma\vdash e_1 \rightsquigarrow E_1:\rho_1 \quad \Gamma\vdash e_2 \rightsquigarrow E_2:\rho_2 \quad \mathrm{wft}\ \rho_1 \times^\beta \rho_2}{\Gamma\vdash (e_1,e_2) \rightsquigarrow (E_1,E_2)^\beta : \rho_1 \times^\beta \rho_2} \qquad \frac{\Gamma\vdash e \rightsquigarrow E : \rho_1\times^\beta \rho_2}{\Gamma\vdash \pi_i e \rightsquigarrow \pi_1^\beta E : \rho_i}$$

$$\frac{\Gamma\vdash e_1 \rightsquigarrow E_1 : \upsilon_1^\beta \quad \Gamma\vdash e_2 \rightsquigarrow E_2 : \rho \quad \Gamma\vdash e_3 \rightsquigarrow E_3 : \rho}{\Gamma\vdash \mathrm{if0}\ e_1\ e_2\ e_3 \rightsquigarrow \mathrm{if0}^\beta\ E_1\ E_2\ E_3 : \rho}$$

Fig. 3. Translation Rules for Global Equational BTA

2.1 Global Variant

For the global variant we only need the well-annotatedness constraints, phase constraints are not necessary.

Definition 7 Global Equational Binding-Time Analysis.

- Construct $\mathcal{A}$ by type reconstruction for simple types with recursion.
- Build a BTC set from the well-formedness constraints derived from $\mathcal{A}$ and the initial binding times, i.e., $b_i \leq \beta_{x_i}$.
- Solve the BTC set to obtain a minimal well-formed bt annotation of $\mathcal{A}$.

From the construction of the automaton $\mathcal{A}$ we know that each expression e^ℓ of the original term is associated to a state $\phi_e = M(\ell)$ of $\mathcal{A}$ which in turn is associated with a binding-time annotation $\beta_e = B(\phi_e)$. Using this association we can transform a program into a completion as shown in Fig. 3. The judgement $\Gamma \vdash e \rightsquigarrow E : \rho$ reads "under type assumption Γ term e translates to annotated term E of well-formed annotated type ρ."

2.2 Local Variant

A local BTA associates a local type automaton with each program point and decorates it with binding-time information. The local automaton $\mathcal{A}_e$ for expression e^ℓ is the subautomaton of $\mathcal{A}$ with initial state $M(\ell)$. Each of these local automata $\mathcal{A}_e$ has its own binding-time annotation $B_e : \Phi \to \mathrm{BT}$. Obviously, each of them must be well-formed, according to Def. 3.

Furthermore, we now have to specify phase constraints. This amounts to having binding-time (bt) coercions $\rho \overset{b}{\rightsquigarrow} \rho'$ (read: there is a bt coercion from ρ to ρ', see Fig. 5) in the definition of well-annotatedness. The corresponding translation rule is:

$$\frac{\Gamma\vdash e \rightsquigarrow E : \rho \qquad \rho \overset{b}{\rightsquigarrow} \rho'}{\Gamma\vdash e \rightsquigarrow \langle \rho \overset{b}{\rightsquigarrow} \rho' \rangle E : \rho'}$$

where ρ ranges over annotated types. A bt coercion enables us to use a static function in a dynamic context without compromising the staticness of the function. Bt coercions do not change the shape of the underlying type. Figure 4 defines the coercion relation between bt annotated types, which gives rise to the phase constraints.

$$\rho \overset{b}{\rightsquigarrow} \rho \qquad \frac{\beta \leq \beta'}{\mathrm{int}^{\beta} \overset{b}{\rightsquigarrow} \mathrm{int}^{\beta'}}$$

$$\frac{\rho_1' \overset{b}{\rightsquigarrow} \rho_1 \quad \rho_2 \overset{b}{\rightsquigarrow} \rho_2' \quad \beta \leq \beta' \quad \mathrm{wft}\ \rho_1 \to^{\beta} \rho_2 \quad \mathrm{wft}\ \rho_1' \to^{\beta'} \rho_2'}{\rho_1 \to^{\beta} \rho_2 \overset{b}{\rightsquigarrow} \rho_1' \to^{\beta'} \rho_2'}$$

$$\frac{\rho_1 \overset{b}{\rightsquigarrow} \rho_1' \quad \rho_2 \overset{b}{\rightsquigarrow} \rho_2' \quad \beta \leq \beta' \quad \mathrm{wft}\ \rho_1 \times^{\beta} \rho_2 \quad \mathrm{wft}\ \rho_1' \times^{\beta'} \rho_2'}{\rho_1 \times^{\beta} \rho_2 \overset{b}{\rightsquigarrow} \rho_1' \times^{\beta'} \rho_2'}$$

Fig. 4. Binding-Time Coercion Relation

$$\begin{array}{ll}
\langle \rho \rightsquigarrow \rho \rangle & = \lambda z.z \\
\langle \mathrm{int}^{\beta} \rightsquigarrow \mathrm{int}^{\beta'} \rangle & = \lambda y.\mathrm{lift}^{\beta,\beta'}\ z \\
\langle \rho_1 \to^{\beta} \rho_2 \rightsquigarrow \rho_1' \to^{\beta'} \rho_2' \rangle & = \lambda z.\lambda^{\beta'} x^{\diamond}.\langle \rho_2 \rightsquigarrow \rho_2' \rangle (z @^{\beta} \langle \rho_1' \rightsquigarrow \rho_1 \rangle x^{\diamond}) \\
\langle \rho_1 \times^{\beta} \rho_2 \rightsquigarrow \rho_1' \times^{\beta'} \rho_2' \rangle & = \lambda z.(\langle \rho_1 \rightsquigarrow \rho_1' \rangle \pi_1^{\beta} z, \langle \rho_2 \rightsquigarrow \rho_2' \rangle \pi_2^{\beta} z)^{\beta'}
\end{array}$$

Fig. 5. Higher-Order Binding-Time Coercions

Figure 5 shows an implementation of bt coercions (cf. [8–10]). Coercions can also be defined for sums and some recursive types (e.g., lists).

When generating the phase constraints we refer to the normalized translation rules given in Fig. 6. In all cases where the binding-time annotations must coincide, we equate the annotations of two automata. Wherever coercions are allowed, we relate the annotations as prescribed in Fig. 4. Both result in obvious algorithms which traverse two automata simultaneously and generate constraints. The number of constraints is bounded by the product of the number of states of the automata. Each of these is bounded by the number of states of the global automaton, which is bounded by the size of the program.

Definition 8 Local Equational Binding-Time Analysis.

– Construct $\mathcal{A}$ by type reconstruction for simple types with recursion.

$$\Gamma\{x:\rho\} \vdash x \rightsquigarrow x : \rho \qquad \Gamma \vdash 0 \rightsquigarrow 0 : \mathrm{int}^0 \qquad \frac{\Gamma \vdash e \rightsquigarrow E : \upsilon^{\beta}}{\Gamma \vdash \mathrm{succ}\ e \rightsquigarrow \mathrm{succ}^{\beta}\ E : \mathrm{int}^{\beta}}$$

$$\frac{\Gamma\{x:\rho_2\} \vdash e \rightsquigarrow E : \rho_1}{\Gamma \vdash \lambda x.e \rightsquigarrow \lambda^0 x.E : \rho_2 \to^0 \rho_1}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow E_1 : \rho_2' \to^{\beta} \rho_1 \quad \Gamma \vdash e_2 \rightsquigarrow E_2 : \rho_2 \quad \rho_2 \rightsquigarrow \rho_2'}{\Gamma \vdash e_1 @ e_2 \rightsquigarrow E_1 @^{\beta} (\langle \rho_2 \rightsquigarrow \rho_2' \rangle E_2) : \rho_1}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow E_1 : \rho_1 \quad \Gamma \vdash e_2 \rightsquigarrow E_2 : \rho_2}{\Gamma \vdash (e_1, e_2) \rightsquigarrow (E_1, E_2)^0 : \rho_1 \times^0 \rho_2} \qquad \frac{\Gamma \vdash e \rightsquigarrow E : \rho_1 \times^{\beta} \rho_2}{\Gamma \vdash \pi_i e \rightsquigarrow \pi_i^{\beta} E : \rho_i}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow E_1 : \upsilon^{\beta} \quad \Gamma \vdash e_2 \rightsquigarrow E_2 : \rho_2 \quad \Gamma \vdash e_3 \rightsquigarrow E_3 : \rho_3 \quad \rho_2 \rightsquigarrow \rho \quad \rho_3 \rightsquigarrow \rho}{\Gamma \vdash \mathrm{if0}\ e_1\ e_2\ e_3 \rightsquigarrow \mathrm{if0}^{\beta}\ E_1\ (\langle \rho_2 \rightsquigarrow \rho \rangle E_2)\ (\langle \rho_3 \rightsquigarrow \rho \rangle E_3) : \rho}$$

Fig. 6. Normalized Translation Rules for Local Equational BTA

- For each subexpression e^ℓ of the program build a local automaton $\mathcal{A}_e$ as the subautomaton of $\mathcal{A}$ with initial state $M(\ell)$.
- Build a BTC set from the well-formedness constraints derived from all $\mathcal{A}_e$, the phase constraints, and the initial binding times, i.e., $b_i \leq \beta_{x_i}$.
- Solve the BTC set giving well-formed binding-time annotations for each $\mathcal{A}_e$. The binding-time annotations also satisfy the phase constraints.

The complexity of the algorithm is dominated by the cost of generating the phase constraints. Let s be the size of the program. Flow type inference takes $O(s \cdot \alpha(s))$ time [18], generating the well-formedness BTCs takes $O(s^2)$ time, generating the phase BTCs takes $O(s^2)$ time for each program point resulting in a total of $O(s^3)$. Solving the BTC set is linear in the size of the constraint set. Hence, the overall time complexity is $O(s^3)$.

2.3 Comparison

In this section we compare the power of the global and local variants of equational BTA. As any global annotation can be considered a local annotation it is clear that the local variant cannot yield worse results than the global variant (see Sec. 1.3). The following example term shows that the inclusion is proper:

$$\text{let } f = \lambda z.z \text{ in } f@((\text{if0 } 0\ f\ g)@0) \tag{1}$$

is analyzed with $g : \text{int}^D \to^D \text{int}^D$, a dynamic function. Such a situation arises, for example, if g is a dynamic parameter of the goal function. The global annotation scheme translates this term to

$$\text{let}^D\ f = \lambda^D z.z \text{ in } f@^D((\text{if0}^0\ 0\ f\ g)@^D(\text{lift } 0)) \tag{2}$$

where everything except the conditional is dynamic. The specialized term is

$$\text{let } f = \lambda z.z \text{ in } f@(f@0). \tag{3}$$

The local annotation scheme translates the same term to

$$\text{let}^0\ f = \lambda^0 z.z \text{ in } f@^0((\text{if0}^0\ 0\ (\lambda^D w.f@^0 w)\ g)@^D(\text{lift } 0)) \tag{4}$$

which specializes to

$$(\lambda w.w)@0. \tag{5}$$

Hence, we have the following lemma.

Lemma 9. *The local variant of the equational BTA classifies strictly more program points as static than the global variant.*

$$(\text{t-const})\ \Gamma \vdash 0 : \top \qquad (\text{t-succ1})\ \frac{\Gamma \vdash e : \top}{\Gamma \vdash \text{succ}\ e : \top} \qquad (\text{t-succ2})\ \frac{\Gamma \vdash e : \text{int}}{\Gamma \vdash \text{succ}\ e : \top}$$

$$(\text{t-abs})\ \frac{\Gamma\{x : \top\} \vdash e : \top}{\Gamma \vdash \lambda x.e : \top} \qquad (\text{t-app})\ \frac{\Gamma \vdash e_1 : \top \qquad \Gamma \vdash e_2 : \top}{\Gamma \vdash e_1 @ e_2 : \top}$$

$$(\text{t-pair})\ \frac{\Gamma \vdash e_1 : \top \qquad \Gamma \vdash e_2 : \top}{\Gamma \vdash (e_1, e_2) : \top} \qquad (\text{t-proj})\ \frac{\Gamma \vdash e :}{\Gamma \vdash \pi_i e : \top}$$

$$(\text{t-if})\ \frac{\Gamma \vdash e_1 : \top \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if0}\ e_1\ e_2\ e_3 : \tau}$$

Fig. 7. Additional Partial Typing Rules

2.4 More Precision

The example from the preceding subsection can be improved by eta-expanding g before placing the annotations [9]:

$$\text{let}^0\ f = \lambda^0 z.z \text{ in } f @^0 ((\text{if0}^0\ 0\ (\lambda^0 w.f @^0 w)\ (\lambda^0 w.g @^D w)) @^0 (\text{lift}\ 0))$$

This completion reduces to 0. Below we sketch an algorithm to produce this result. First, we need a definition.

Definition 10 Eta-Expansors [11].

$$\Delta_\tau = \begin{cases} \lambda z.z & \tau = \text{int} \\ \lambda z.\lambda w.\Delta_{\tau_2} @ (z @ (\Delta_{\tau_1} @ w)) & \tau = \tau_1 \to \tau_2 \\ \lambda z.(\pi_1(\Delta_{\tau_1} @ z), \pi_2(\Delta_{\tau_2} @ z)) & \tau = \tau_1 \times \tau_2 \end{cases}$$

Definition 11 Improved Local Equational Binding-Time Analysis.

- Perform equational flow-type reconstruction.
- For all expressions e appearing as arguments of function calls, branches of conditionals, or arguments of primitive operations do simultaneously:
  - Replace e of type $\tau = \tau_1 \to \tau_2$ which is not a lambda expression by $\Delta_\tau @ e$.
  - Replace e of type $\tau = \tau_1 \times \tau_2$ that is not a pair construction by $\Delta_\tau @ e$.
- Continue as in Local Equational Binding-Time Analysis.

2.5 Another Variation of Equational BTA

We can also express Gomard's BTA, which is based on partial types, in our framework. It adds the type $\top$ to the current system and the rules in Fig. 7. The motivation behind these rules is the desire to type all terms regardless whether their execution yields an error or not. Using this system as a basis for a BTA intends to defer all program parts that are possibly erroneous (have type $\top$) to run time. Additionally, Gomard's BTA allows for first-order bt coercions of the form $\text{int}^\beta \overset{b}{\leadsto} \text{int}^{\beta'}$ for $\beta \leq \beta'$. So we can say that Gomard's BTA consists of partial (equational) type inference with a local annotation scheme restricted to first-order bt coercions. Henglein's algorithm [18] performs the entire reconstruction for this system in almost-linear time. Strictly speaking, we are discussing Mogensen's system [25] because Gomard and Henglein disallow recursive types.

$$\bot \preceq^t \tau \qquad \tau \preceq^t \tau \qquad \tau \preceq^t \top \qquad \frac{\tau_1 \preceq^t \tau_2 \quad \tau_2 \preceq^t \tau_3}{\tau_1 \preceq^t \tau_3}$$

$$\frac{\tau_1' \preceq^t \tau_1 \quad \tau_2 \preceq^t \tau_2'}{\tau_1 \to \tau_2 \preceq^t \tau_1' \to \tau_2'} \qquad \frac{\tau_1 \preceq^t \tau_1' \quad \tau_2 \preceq^t \tau_2'}{\tau_1 \times \tau_2 \preceq^t \tau_1' \times \tau_2'}$$

Fig. 8. Type Coercions

$$\bot^0 \preceq^g \rho \qquad \rho \preceq^g \rho \qquad \upsilon^D \preceq^g \top^D \qquad \frac{\rho_1 \preceq^g \rho_2 \quad \rho_2 \preceq^g \rho_3}{\rho_1 \preceq^g \rho_3}$$

$$\frac{\beta \leq \beta'}{\mathrm{int}^\beta \preceq^g \mathrm{int}^{\beta'}} \qquad \frac{\rho_1' \preceq^g \rho_1 \quad \rho_2 \preceq^g \rho_2'}{\rho_1 \to^\beta \rho_2 \preceq^g \rho_1' \to^\beta \rho_2'} \qquad \frac{\rho_1 \preceq^g \rho_1' \quad \rho_2 \preceq^g \rho_2'}{\rho_1 \times^\beta \rho_2 \preceq^g \rho_1' \times^\beta \rho_2'}$$

Fig. 9. Coercion Relation for the Global Variant

3 Inclusion-Based Binding-Time Analysis

Equational flow analysis and BTA ignore the direction in which values flow. This sometimes deteriorates binding-time annotations because the analysis can equate program points that never flow together.

Hence, the inclusion-based BTA builds on an extension of the equational flow-type system with subtyping, $\bot$, and $\top$ types. The resulting system of simple types with recursion and subtyping originates from work by Amadio and Cardelli [1]. It only adds the subsumption rule to the typing rules of Fig. 2.

$$(\mathrm{sub})\ \frac{\Gamma \vdash e : \tau \quad \tau \preceq^t \tau'}{\Gamma \vdash e : \tau'}$$

The coercion relation for types $\preceq^t$ shown in Fig. 8 is standard. Typability for that system can be decided in polynomial time [17, 29], however type reconstruction requires exponential time [29]. The result of that algorithm is a global type automaton as in Sec. 2. On top of that automaton, we define our BTA.

As in Sec. 2, there are two ways to add binding-time annotations to the automaton: the global and the local strategy. For both strategies, we reuse Def. 3 for well-formed binding-time annotations. Only the phase constraints differ.

3.1 Global Variant

In the global setting, we can reuse all annotated translation rules from the equational setting. We only have to define an annotated translation rule for the rule (sub) of type subsumption. It is based on the coercion relation $\preceq^g$ on annotated types defined in Fig. 9. It leads to the following annotated translation rule.

$$\frac{\Gamma \vdash e \rightsquigarrow E : \rho \quad \rho \preceq^g \rho'}{\Gamma \vdash e \rightsquigarrow \langle \rho \overset{b}{\rightsquigarrow} \rho' \rangle E : \rho'}$$

This system has two problems. First, although we can coerce types we cannot coerce their binding-time annotations (except for type int). Second, the axiom

$\upsilon^D \preceq^g \top^D$ means if we want to forget the structure of a type, the binding time of the whole structure must be dynamic. Taken together, we find that a value which is used at type $\top$ anywhere in the program is annotated as dynamic in the whole program, even in places were its type is known. In consequence, we gain nothing compared to the global equational analysis.

Define $\overline{\rho}^t$ for an annotated type ρ by

$$\begin{array}{llll} \overline{\upsilon^D}^t & = \top^D & \overline{\rho_1 \to^0 \rho_2}^t & = \overline{\rho_1}^t \to^0 \overline{\rho_2}^t \\ \overline{\mathrm{int}^0}^t & = \mathrm{int}^0 & \overline{\rho_1 \times^0 \rho_2}^t & = \overline{\rho_1}^t \times^0 \overline{\rho_2}^t. \end{array}$$

Lemma 12. *Any annotated translation $\Gamma \vdash e \rightsquigarrow E : \rho$ in the global inclusion-based system gives rise to a translation $\overline{\Delta}^t, \overline{\Gamma}^t \vdash e \rightsquigarrow E : \overline{\rho}^t$ in the global equational system with partial types.*

Proof. Induction on the structure of a translation.

Theorem 13. *The translations induced by global inclusion-based BTA and by global equational BTA with partial types are identical.*

Proof. Lemma 12 gives us for each inclusion-based translation an equational translation which achieves the same effect. The other implication is obvious as each derivation in the system of simple recursive types without subtyping is trivially also a derivation in the system with subtyping.

$$\bot^0 \preceq^l \rho \qquad \rho \preceq^l \rho \qquad \upsilon^D \preceq^l \top^D \qquad \frac{\rho_1 \preceq^l \rho_2 \quad \rho_2 \preceq^l \rho_3}{\rho_1 \preceq^l \rho_3} \qquad \frac{\beta \leq \beta'}{\mathrm{int}^\beta \preceq^l \mathrm{int}^{\beta'}}$$

$$\frac{\rho_1' \preceq^l \rho_1 \quad \rho_2 \preceq^l \rho_2' \quad \beta \leq \beta' \quad \mathrm{wft}\ \rho_1 \to^\beta \rho_2 \quad \mathrm{wft}\ \rho_1' \to^{\beta'} \rho_2'}{\rho_1 \to^\beta \rho_2 \preceq^l \rho_1' \to^{\beta'} \rho_2'}$$

$$\frac{\rho_1 \preceq^l \rho_1' \quad \rho_2 \preceq^l \rho_2' \quad \beta \leq \beta' \quad \mathrm{wft}\ \rho_1 \times^\beta \rho_2 \quad \mathrm{wft}\ \rho_1' \times^{\beta'} \rho_2'}{\rho_1 \times^\beta \rho_2 \preceq^l \rho_1' \times^{\beta'} \rho_2'}$$

Fig. 10. Combined Type and Binding-Time Coercion Relation

3.2 Local Variant

In the local view, we adopt the position that every subexpression has its own type automaton and its own annotation. The annotations of neighboring types are related using bt coercions on top of the type coercions. In fact, the only change is in the coercion rules for type constructors. Now they can increase the binding time of the constructed value. Figure 10 defines the combined bt and type coercion relation $\preceq^l$. The additional power with respect to the global equational system stems from coercions like $\rho \preceq^l \top^D$, which coerces a value of a sensible type to a dynamic type error. In the equational system, the $\top$ type would have spread its dynamic binding time beyond the cause of the error.

3.3 Comparison of the Inclusion-Based BTAs

In this section we compare the power of the global and local variants of inclusion-based BTA. We have already proved that global inclusion-based BTA is equivalent to global equational BTA (see Theorem 13). Again, in the inclusion-based framework, any global annotation can be considered a local annotation. Hence, it is clear that the local variant cannot yield worse results than the global variant. Our previous example term (1) demonstrates that the inclusion is proper.

$$\text{let } f = \lambda z.z \text{ in } f@((\text{if0 } 0\ f\ g)@0). \tag{6}$$

For the assumption $g : \top^D$ the results of global and local equational BTA coincide (cf. (2)):

$$\text{let}^D\ f = \lambda^D z.z \text{ in } f@^D((\text{if0}^0\ 0\ f\ g)@^D(\text{lift } 0)) \tag{7}$$

However, the local inclusion based variant produces (cf. 4)

$$\text{let}^0\ f = \lambda^0 z.z \text{ in } f@^0((\text{if0}^0\ 0\ (\lambda^D w.f@^0 w)\ g)@^D(\text{lift } 0)), \tag{8}$$

where we can perform more reductions statically.

$$(\lambda w.w)@0 \tag{9}$$

Hence the following lemma.

Lemma 14. *The local inclusion-based BTA produces strictly better results than its global counterpart.*

3.4 Comparison of the Equational and Inclusion-Based BTAs

Finally, we need to compare the local variants of the equational and the inclusion-based frameworks. Every type derivation of the equational system is also a type derivation of the inclusion-based system, without type coercions. To show that the inclusion is proper we consider the term

$$\text{let } g = \lambda x.x \text{ in let } f = \lambda z.g \text{ in } (f@0)@f@g@0 \tag{10}$$

The local inclusion-based BTA constructs the completion

$$\text{let}^0\ g = \lambda^0 x.x \text{ in let}^0\ f = \lambda^0 z.g \text{ in } (f@^0 0)@^0 f@^0 g@^0 0$$

which reduces statically to 0. In contrast, the local equational BTA yields

$$\text{let}^D\ g = \lambda^D x.x \text{ in let}^0\ f = \lambda^0 z.g \text{ in } (f@^0 0)@^D f@^0 g@^D 0$$

which reduces statically to

$$\text{let } g = \lambda x.x \text{ in } g@(g@0)$$

4 Related Work

Here, we only discuss additional work that has not been discussed in the body of the paper.

Mogensen [25] was the first one to consider BTA based on recursive types. His motivation was typing Y in order to unfold fixpoints statically. Gomard's system assigns type $\top^D$ to Y, thus it defers all occurrences of Y to run time.

Palsberg and Schwartzbach [30] compare BTAs based on abstract interpretation (ai) with type-based BTAs. They show that their ai-based approach is more powerful than Mogensen's approach [25] which is more powerful than Gomard's [14] approach. In view of the current work, the latter is not surprising because more terms are typable in the presence of recursive types. Due to the result of Heintze, Palsberg, and O'Keefe [17,29] we conjecture that their ai-based algorithm is equivalent to the local inclusion-based system presented here.

The ML partial evaluator Pell-Mell [24] employs a BTA based on set-based analysis [16]. There are significant parallels between set-based analysis and the simple type system with subtyping and recursion [17]. These parallels again suggest that the BTA of Pell-Mell is also equivalent to the local inclusion-based system presented here.

Launchbury [22] considers BTA based on projections. This BTA has parallels with our equational system, but appears to be more restrictive because it insists on uniform properties of recursive types.

The present author [34] has considered a BTA augmented with representation analysis that removes some of the restrictions of the current BTA frameworks and keeps track of additional information. This work can be recast in the present framework by including additional layers of annotations.

Solberg, in her PhD thesis [33], gives a general overview of the use of annotated type systems in program analysis. She considers BTA in the style of Nielson and Nielson [27]. She shows how that particular analysis fits into the general framework, but does not compare different alternatives for BTA and her work is not geared towards partial evaluation.

Nielson and Nielson [28] give a systematic description of different multi-level lambda-calculi using algebraic methods. Their interest lies in the different well-formedness criteria for expressions. In our current work, we are concerned with combining different type systems with different annotation strategies. The well-formedness criterion of expressions remains fixed.

5 Conclusions

With the exception of the work of Palsberg and Schwartzbach [30], BTAs have lead fairly separate lives. They could not be compared because they relied on different frameworks (abstract interpretation, type inference, set-based analysis, projections, and so on). We have constructed a general BTA framework based on annotated type systems that allows such comparisons in a clean and simple way. Beyond that, our systematic approach has identified three novel BTAs.

Acknowledgements Thanks to Olivier Danvy and Dirk Dussart for discussions on higher-order coercions and type-based program analysis.

References

1. R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. Prog. Lang. Syst.*, 15(4):575–631, 1993.
2. L. Birkedal and M. Welinder. Binding-time analysis for Standard-ML. In P. Sestoft and H. Søndergaard, editors, *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '94*, pages 61–71, Orlando, Fla., June 1994. ACM.
3. A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Programming*, 17:3–34, 1991.
4. A. Bondorf and J. Jørgensen. Efficient analysis for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.
5. C. Consel. Binding time analysis for higher order untyped functional languages. In LFP 1990 [23], pages 264–272.
6. C. Consel. Polyvariant binding-time analysis for applicative languages. In D. Schmidt, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '93*, pages 66–77, Copenhagen, Denmark, June 1993. ACM Press.
7. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, Jan. 1993. ACM Press.
8. O. Danvy. Type-directed partial evaluation. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg, Fla., Jan. 1996. ACM Press.
9. O. Danvy, K. Malmkjær, and J. Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, July 1995.
10. O. Danvy, K. Malmkjær, and J. Palsberg. Eta-expansion does The Trick. Technical Report BRICS RS-95-41, Computer Science Dept., Aarhus University, Denmark, Aug. 1995.
11. R. Di Cosmo and D. Kesner. Simulating expansions without expansions. *Mathematical Structures in Computer Science*, 4:1–48, 1994.
12. D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Mycroft [26], pages 118–136. LNCS 983.
13. R. Glück and J. Jørgensen. Fast multi-level binding-time analysis for multiple program specialization. In PSI '96 [31].
14. C. K. Gomard. Partial type inference for untyped functional programs. In LFP 1990 [23], pages 282–287.
15. C. K. Gomard and N. D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–70, Jan. 1991.
16. N. Heintze. Set-based analysis of ML-programs. In *Proc. 1994 ACM Conference on Lisp and Functional Programming*, pages 306–317, Orlando, Florida, USA, June 1994. ACM Press.
17. N. Heintze. Control-flow analysis and type systems. In Mycroft [26], pages 189–206. LNCS 983.

18. F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991*, pages 448–472, Cambridge, MA, 1991. Springer-Verlag. LNCS 523.
19. J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages and computation.* Addison-Wesley, 1979.
20. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
21. N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 124–140, Dijon, France, 1985. Springer-Verlag. LNCS 202.
22. J. Launchbury. *Projection Factorisations in Partial Evaluation*, volume 1 of *Distinguished Dissertations in Computer Science.* Cambridge University Press, 1991.
23. *Proc. 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, 1990. ACM Press.
24. K. Malmkjær, O. Danvy, and N. Heintze. ML partial evaluation using set-based analysis. In *Record of the ACM-SIGPLAN Workshop on ML and its Applications*, number 2265 in INRIA Research Report, pages 112–119, BP 105, 78153 Le Chesnay Cedex, France, June 1994.
25. T. Æ. Mogensen. Self-applicable partial evaluation for pure lambda calculus. In C. Consel, editor, *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '92*, pages 116–121, San Francisco, CA, June 1992. Yale University. Report YALEU/DCS/RR-909.
26. A. Mycroft, editor. *Proc. International Static Analysis Symposium, SAS'95*, Glasgow, Scotland, Sept. 1995. Springer-Verlag. LNCS 983.
27. F. Nielson and H. R. Nielson. *Two-Level Functional Languages.* Cambridge University Press, 1992.
28. F. Nielson and H. R. Nielson. Multi-level lambda-calculi: an algebraic description. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 338–354, Dagstuhl, Germany, Feb. 1996. Springer Verlag, Heidelberg.
29. J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *Proc. 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 367–378, San Francisco, CA, Jan. 1995. ACM Press.
30. J. Palsberg and M. I. Schwartzbach. Binding-time analysis: Abstract interpretation versus type inference. In *IEEE International Conference on Computer Languages 1994*, pages 289–298, Toulouse, France, 1994. IEEE Computer Society Press.
31. *PSI-96: Andrei Ershov Second International Memorial Conference, Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*, Novosibirsk, Russia, June 1996. Springer-Verlag.
32. H. Seidl. Least solutions of equations over $\mathcal{N}$. In *Proc. International Conference of Automata, Languages and Programming, ICALP '94*, volume 820 of *Lecture Notes in Computer Science*, pages 400–411. Springer-Verlag, 1994.
33. K. L. Solberg. *Annotated Type Systems for Program Analysis.* PhD thesis, Odense University, Denmark, July 1995. Also technical report DAIMI PB-498, Comp. Sci. Dept. Aarhus University.
34. P. Thiemann. Towards partial evaluation of full Scheme. In G. Kiczales, editor, *Reflection'96*, pages 95–106, San Francisco, CA, USA, Apr. 1996.