Efficient Type Inference for Higher-Order Binding-Time Analysis*

Fritz Henglein
DIKU
University of Copenhagen
Universitetsparken 1
2100 Copenhagen Ø
Denmark
Internet: henglein@diku.dk

May 29, 1991

Abstract

Binding-time analysis determines when variables and expressions in a program can be bound to their values, distinguishing between *early* (compile-time) and *late* (run-time) binding. Binding-time information can be used by compilers to produce more efficient target programs by *partially evaluating* programs at compile-time. Binding-time analysis has been formulated in abstract interpretation contexts and more recently in a type-theoretic setting.

In a type-theoretic setting binding-time analysis is a *type inference problem*: the problem of inferring a *completion* of a λ-term e with binding-time annotations such that e satisfies the typing rules. Nielson and Nielson and Schmidt have shown that every simply typed λ-term has a unique completion $\hat{e}$ that minimizes late binding in TML, a monomorphic type system with explicit binding-time annotations, and they present exponential time algorithms for computing such minimal completions.[1] Gomard proves the same results for a variant of his two-level λ-calculus without a so-called "lifting" rule. He presents another algorithm for inferring completions in this somewhat restricted type system and states that it can be implemented in time $O(n^3)$. He conjectures that the completions computed are minimal.

In this paper we expand and improve on Gomard's work in the following ways.

- We identify the combinatorial core of type inference for binding-time analysis in Gomard's type system with "lifting" by effectively characterizing it as solving a specific class of constraints on type expressions.
- We present normalizing transformations on these constraints that preserve their solution sets, and we use the resultant normal forms to prove constructively the existence of *minimal solutions*, which yield *minimal completions*; this sharpens the minimal completion result of Gomard and extends it to the full type system with "lifting".
- We devise a very efficient algorithm for computing minimal completions. It is a refinement of a fast unification algorithm, and an amortization argument shows that a fast union/find-based implementation executes in almost-linear time, $O(n\alpha(n,n))$, where α is an inverse of Ackermann's function.

*In Proc. Conference on Functional Programming Languages and Computer Architecture (FPCA), Cambridge, Massachusetts, August 1991, pp. 448-472, Springer-Verlag, Lecture Notes in Computer Science, Vol. 523. This research has been supported by Esprit BRA 3124, Semantique.

[1] A (minimal) completion is called a (best) decoration in the work of Nielson and Nielson and Schmidt.

Our results are for the two-level type system of Gomard, but we believe they are also adaptable to the Nielsons' TML. Our algorithm improves the computational complexity of computing minimal completions from exponential time to almost-linear time. It also improves on Gomard's polynomial time completion algorithm by a quadratic factor and as such appears to be the first efficient algorithm that provably computes minimal completions.

1 Introduction

Given information on the static or dynamic availability of data, binding-time analysis determines which operations can safely be evaluated "early" (statically, at compile-time) and which should be deferred until "late" (to be executed dynamically, at run-time). Instead of 'early' and 'late' we also use the common terms 'static' and 'dynamic', respectively. Binding-time information can be used to optimize the implementation of programming languages by scheduling operations with statically bound data at compile time (e.g., [JM76,JM78,JS80,Kro81,NN86] *etc*). Automatic binding-time analysis is usually beneficial for guiding the actions of program specializers, but it plays an especially important role in the generation of compilers from self-applicable partial evaluators [JSS85,JSS89,Con88,Rom87,JGB+90,Bon90a,GJ91] as a preprocessing phase to program specialization [BJMS88].

In the early work on partial evaluators a simple dichotomy static/dynamic was used (e.g., [Ses85]). This was generally too conservative for compound data types such as lists, and rewriting of a program was required to recover more static binding information. This was remedied by using more refined binding-time values [Lau87,Mog87], which Mogensen calls partially static structures. The relevance of this refinement is not limited to first-order data types. For example, if y is dynamic the function $(\lambda x.x@y)$ is partially static: we can evaluate *statically* the application of $\lambda x.x@y$ to a (completely) dynamic argument z; yet we cannot evaluate statically the resulting application of x to y since x will be bound to z, which is completely dynamic.

Every function application can of course be evaluated late, which amounts to deferring all computation to run-time. Partial evaluation seeks to do as much computation at compile-time as is correct, possible and sensible. For this purpose binding-time analysis is used to find as many operations that can be evaluated ("bound") statically.

Early binding-time analysis work limited itself to first-order languages [JSS85,JSS89,Jon87, Mog89,Rom87,Lau87,Lau90], but, more recently, higher-order languages have also been analyzed: Mogensen [Mog89], Bondorf [Bon90a,Bon90b], Consel [Con90] and Hunt and Sands [HS91] describe higher-order binding time analysis in an abstract interpretation framework; Nielson and Nielson [NN88a], Schmidt [Sch87] and Gomard [Gom89,Gom90,GJ91] formalize it as a *type inference* (or *type annotation*) problem within two-level typed λ-calculus.

A two-level type system has an *early-binding* and a *late-binding* variant of every operator, which are distinguished by *binding-time annotations*. In Gomard's two-level typed λ-calculus there is a distinguished type Λ that represents unevaluated, *untyped* λ-terms[2], which are the result of late-binding operators at partial evaluation time. A late-binding syntactic operator (such as @, λ) is indicated by *underscoring* it (e.g., $\underline{@}$, $\underline{\lambda}$) whereas its early-binding counterpart has *no* underscore.

The main difference to the original two-level λ-calculus TML of Nielson [NN88b] is that in Nielson's work the represented λ-terms are also *typed*, which is reflected in a more complex type structure for (two-level) λ-terms denoting such unevaluated "object terms". Nielson and

[2] In [Gom90] the type Λ type is actually called "untyped"; and in [Gom89] it is referred to as "code" because of its intended interpretation in the two-level λ-calculus.

Nielson [NN88a] and Schmidt [Sch87] have shown that every *simply typed* λ-term has a *minimal* completion $\hat{e}$ in TML in the sense that all late-binding annotations (underscores in Gomard's system) in $\hat{e}$ occur in every completion of e with binding-time annotations, and they present exponential time algorithms for computing such minimal completions. Gomard proves the same results for *untyped* λ-terms in his two-level λ-calculus without a "lifting" rule [Gom89]. He presents a more efficient completion algorithm [Gom90] and states that this algorithm can be implemented in time $O(n^3)$ using Huet's fast unification algorithm [Gom89]. He conjectures that his algorithm computes minimal completions.

In this paper we expand and improve on Gomard's work in the following ways.

- We identify the combinatorial core of type inference for binding-time analysis in Gomard's type system with "lifting" by effectively characterizing it as solving a specific class of constraints on type expressions.
- We present normalizing transformations on these constraints that preserve their solution sets, and we use the resultant normal forms to prove constructively the existence of *minimal solutions*, which yield *minimal completions* for the type inference system; this sharpens the minimal completion result of Gomard and extends it to the full type system with "lifting".
- We devise a very efficient algorithm for computing minimal completions. It is a refinement of a fast unification algorithm, and an amortization argument shows that a fast union/find-based implementation executes in almost-linear time, $O(n\alpha(n,n))$, where α is an inverse of Ackermann's function.

Our results are for the two-level type system of Gomard (with "lifting"), but we believe they are also adaptable to the Nielsons' TML. Our algorithm improves the computational complexity of computing minimal completions from exponential time to almost-linear time. It also improves on Gomard's polynomial time algorithm by a quadratic factor and as such appears to be the first efficient algorithm that provably computes minimal completions.

We believe that the constraints used to characterize typability may be usable in describing other, similar type inference problems, thus providing a common "back-end" for a whole class of type inference problems. Furthermore, we expect the union/find-based implementation of our type inference algorithm to be implementable, adaptable and responsive in practice. This may well lay the foundation of a practical implementation technology for type-based program analyses.

In Section 2 we recall Gomard's type inference approach to binding time analysis. We present a characterization of completions by solutions of type constraint systems in Section 3. In Section 4 we show how these constraint systems can be normalized while preserving their solution sets. Section 5 shows how minimal completions are constructed. An efficient implementation of the constraint transformations is the basis of our almost-linear time constraint normalization in Section 6. In Section 7 we show how it can be used to implement an almost-linear time binding-time analysis algorithm. Finally, in Section 8 we briefly summarize our results and propose directions in which this kind of type-based analysis might be extended and generalized.

2 Binding-time analysis by type inference

We use a small, but paradigmatic higher-order language of *untyped λ-terms* for which we perform binding-time analysis; they are the terms e generated by the production

$$e ::= x \mid \lambda x.e' \mid e'@e'' \mid \mathbf{fix}\ e \mid \ \mathbf{if}\ e'\ \mathbf{then}\ e''\ \mathbf{else}\ e''' \mid c \mid e'\ op\ e''$$

where x ranges over a class of variables, c is a class of given first-order constants, *op* stands for a set of given binary first-order operations, **fix** is a recursion (fixed-point) operator, and **if** e' **then** e'' **else** e''' is the usual conditional form.

In the type inference approach to binding-time analysis binding-time values are represented by *type expressions (types)*. Our types τ are generated by the production

$$\tau ::= B \mid \Lambda \mid \alpha \mid \tau' \to \tau''$$

The type constant B represents the binding-time value "static"; it denotes the set of (first-order) *base values*, such as the Boolean truth values and integers. The type constant Λ represents the binding-time value "dynamic"; extensionally it stands for the dynamic values: the set of unevaluated untyped λ-terms. A *type variable* α represents an arbitrary (fixed, but unknown) binding-time value. The function type $\tau_1 \to \tau_2$ represents a higher-order binding-time value; e.g., $\Lambda \to \Lambda$ represents a partially static function such as $(\lambda x.x@y)$ for dynamic y.

Binding-time information is represented explicitly by *operator, type*, and *lifting annotations* in λ-terms: a late-bound operator is distinguished from an early-bound operator by an underscore; an early-bound λ-abstraction carries an explicit type; and application of a lift operator represents turning a base value into a dynamic value. A λ-term with such annotations is called an *annotated* λ-term. Binding-time annotations are used to guide the actions of a partial evaluator: an early-bound operator is directly executed on its arguments, a late-bound operator is left uninterpreted (i.e., it results in constructing a piece of code from its (code) arguments), and a lift operator is interpreted by generating code representing the (static) value of its argument. We assume that base values, but not functions, can be made dynamic by application of the lift operator. As a consequence we distinguish in our type system between $B \to B$ and B even though both represent fully static binding-time values.

A binding-time proposition "expression e has binding-time value τ" is written as a *typing* $e : \tau$. Such typings are derived by a formal inference system.[3]

Figure 1 contains the inference rules for our *(two-level) type system* in natural deduction style. In it we use the following conventions: e, e', e'' range over λ-terms; x, x' over variables, τ, τ' over type expressions. A hypothesis that may be discharged at a rule application is written in square brackets; i.e., $[x : \tau']$. The introduction of a typing hypothesis for variable x "hides" all other typing hypotheses for x until it is discharged. In the following we write $A \vdash e : \tau$ if $e : \tau$ can be derived from the rules in Figure 1 and exactly one hypothesis for each free variable in e, the set of which is denoted by A. In this case we call e a *well-annotated* λ-term (w.r.t. A). For example, $(\lambda x : \Lambda.x\underline{@}y)@z$ is well-annotated w.r.t. $\{y : \Lambda, z : \Lambda\}$.

This type system is monomorphic and corresponds in expressive power roughly to what has been termed a "sticky" analysis in abstract interpretation. The inference rules consist of one rule each for every early-binding and late-binding operator, a rule for constants and the lifting rule. There are no compound (first-order) data types such as pairs and lists in our language, but they and their binding-time properties — formalized with or without partially static structures — could be added easily in the form of additional type inference rules.

Apart from inessential differences this type system is the same as Gomard's [Gom89,GJ91]. (Note that the type inference algorithm in [Gom90] is for the same type system, but without

[3] Binding-time properties are semantic properties, and there is thus an arbitrary number of concrete conceivable "static" (recursive) analyses to approximate these quintessentially dynamic properties. The inference rules specify exactly the expressiveness of an analysis and thus determine the particular binding-time analysis problem to be solved without predetermining how to compute binding-time properties.

rule (LIFT).) The corresponding two-level type system for simply typed λ-terms is described in Nielson and Nielson [NN88a].

The *erasure* of an annotated λ-term e is the untyped λ-term in which all operator, type and lifting annotations are eliminated, but which is otherwise identical to e. A *completion* of an unannotated λ-term e w.r.t. typing assumptions A is a well-annotated λ-term $\bar{e}$ (w.r.t. A) whose erasure is e.

A set of typing assumptions is a *(first-order) binding-time assumption* for λ-term e if it contains exactly one typing assumption for every free variable in e, and all typing assumptions are of the form $x : B$ or $x : \Lambda$. In this context binding-time analysis is the problem of computing a *minimal* completion of an unannotated λ-term w.r.t. a given binding-time assumption in the intuitive sense that it has a minimum of late-binding operators (see Section 5 for a definition of minimality).

By using the erasures of annotated λ-terms in the "explicit" type system of Figure 1 we obtain an "implicit" type inference system for untyped λ-terms with the properties that: if $A \vdash e : \tau$ in the explicit system then $A \vdash e' : \tau$ in the implicit system where e' is the erasure of e; if $A \vdash e : \tau$ in the implicit system then there exists a completion $\bar{e}$ such that $A \vdash \bar{e} : \tau$ in the explicit system; and, in particular, for every type derivation for an untyped λ-term e there is a *unique* completion of e with an isomorphic derivation in the explicit type inference system. In other words, implicit type derivations, explicit type derivations and well-annotated λ-terms are pairwise in natural one-to-one correspondences.

Since the lift operator and every variant — underscored and not underscored — of every syntactic operator have exactly one inference rule scheme (without side conditions) it follows that type *checking* for an annotated λ-term is (RAM-)linear-time equivalent to solving a unification problem [Wan87,Hen88a] and thus can be done efficiently in linear [PW78,MM82] or almost-linear time [Hue76,ASU86]. We will show that type *inference* for unannotated (or partially annotated) λ-terms and computation of minimal completions can actually be done in essentially the *same* time.

3 Type constraint characterization

Recall that the universe of type expressions is the class of terms $T(A, V)$ generated from the ranked alphabet $A = \{B, \rightarrow, \Lambda\}$ where B and Λ have arity (rank) 0 and $\rightarrow$ has arity 2; V is the set of type variables $\alpha, \alpha', \ldots, \beta, \gamma$ *etc.* We write $\tau = \tau'$ if τ and τ' are the same type expressions.

We define $\leq_b, \leq_f$ to be the "flat" partial orders that contain only the strict inequalities

$$B \quad <_b \quad \Lambda \tag{1}$$

$$\Lambda \rightarrow \Lambda \quad <_f \quad \Lambda \tag{2}$$

A *constraint system* C is a multiset of formal constraints of the form

- $\alpha' \rightarrow \alpha'' \stackrel{?}{\leq}_f \alpha$,
- $\beta \stackrel{?}{\leq}_b \alpha$,
- $\alpha \stackrel{?}{=} \alpha'$, and
- $\alpha \rhd \alpha'$.

(ABSTR) $\dfrac{[x : \tau'] \quad e : \tau}{\lambda x : \tau'.e : \tau' \to \tau}$

(APPL) $\dfrac{e : \tau' \to \tau \quad e' : \tau'}{e@e' : \tau}$

(FIX) $\dfrac{e : \tau \to \tau}{\mathbf{fix}\, e : \tau}$

(IF) $\dfrac{e : B \quad e' : \tau \quad e'' : \tau}{\mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e'' : \tau}$

(OP) $\dfrac{e : B \quad e' : B}{e\ op\ e' : B}$

(CONST) $c : B$

(LIFT) $\dfrac{e : B}{lift\, e : \Lambda}$

(ABSTR-DYN) $\dfrac{[x : \Lambda] \quad e : \Lambda}{\underline{\lambda} x.e : \Lambda}$

(APPL-DYN) $\dfrac{e : \Lambda \quad e' : \Lambda}{e\underline{@}e' : \Lambda}$

(FIX-DYN) $\dfrac{e : \Lambda}{\underline{\mathbf{fix}}\, e : \Lambda}$

(IF-DYN) $\dfrac{e : \Lambda \quad e' : \Lambda \quad e'' : \Lambda}{\underline{\mathbf{if}}\, e\, \underline{\mathbf{then}}\, e'\, \underline{\mathbf{else}}\, e'' : \Lambda}$

(OP-DYN) $\dfrac{e : \Lambda \quad e' : \Lambda}{e\ \underline{op}\ e' : \Lambda}$

Figure 1: Type inference system with type Λ

where $\alpha, \alpha', \alpha_1, \ldots, \alpha_k, \beta$ are type variables or the type constant Λ; β can also be the type constant B. A substitution S (of type expressions for type variables) is a *solution* of C if

- for every constraint of the form $\alpha' \to \alpha'' \stackrel{?}{\leq}_f \alpha$ we have $S(\alpha' \to \alpha'') \leq_f S(\alpha)$;
- for every constraint of the form $\beta \stackrel{?}{\leq}_b \alpha$ we have $S(\beta) \leq_b S(\alpha)$;
- for every constraint of the form $\alpha \stackrel{?}{=} \alpha'$ we have $S(\alpha) = S(\alpha')$;
- for every constraint of the form $\alpha \rhd \alpha'$ we have that if $S(\alpha) = \Lambda$ then $S(\alpha') = \Lambda$;
- for every type variable α not occurring in C we have $S(\alpha) = \alpha$.[4]

We write $\mathrm{Sol}(C)$ for the set of all solutions of C.

Let e be a λ-term and A a binding-time assumption for e. We associate a type variable $\alpha_{x'}$ with every λ-bound variable x' occurring in e and unique type variables $\alpha_{e'}, \bar{\alpha}_{e'}$ for every λ-term occurrence e' in e (for an occurrence of a λ-bound variable x we take $\alpha_e = \alpha_x$). W.l.o.g. we assume that there are no two λ-bindings for any variable in e. We define the constraint system $C_A(e)$ by induction as follows.

1. If $e = \lambda x.e'$ then $C_A(e) = \{\alpha_x \to \bar{\alpha}_{e'} \stackrel{?}{\leq}_f \alpha_e, \alpha_e \stackrel{?}{\leq}_b \bar{\alpha}_e\} \cup C_A(e')$;
2. if $e = e'@e''$ then $C_A(e) = \{\bar{\alpha}_{e''} \to \alpha_e \stackrel{?}{\leq}_f \bar{\alpha}_{e'}, \alpha_e \stackrel{?}{\leq}_b \bar{\alpha}_e\} \cup C_A(e') \cup C_A(e'')$;
3. if $e = \mathbf{fix}\, e'$ then $C_A(e) = \{\alpha_e \to \alpha_e \stackrel{?}{\leq}_f \bar{\alpha}_{e'}, \alpha_e \stackrel{?}{\leq}_b \bar{\alpha}_e\} \cup C_A(e')$;
4. if $e = \mathbf{if}\, e' \,\mathbf{then}\, e'' \,\mathbf{else}\, e'''$ then $C_A(e) = \{B \stackrel{?}{\leq}_b \bar{\alpha}_{e'}, \alpha_e \stackrel{?}{=} \bar{\alpha}_{e''}, \alpha_e \stackrel{?}{=} \bar{\alpha}_{e'''}, \bar{\alpha}_{e'} \rhd \alpha_e, \alpha_e \stackrel{?}{\leq}_b \bar{\alpha}_e\} \cup C_A(e') \cup C_A(e'') \cup C_A(e''')$;
5. if $e = c$ then $C_A(e) = \{B \stackrel{?}{\leq}_b \alpha_e, \alpha_e \stackrel{?}{\leq}_b \bar{\alpha}_e\}$;
6. if $e = e'\, op\, e''$ then $C_A(e) = \{B \stackrel{?}{\leq}_b \bar{\alpha}_{e'}, \bar{\alpha}_{e'} \stackrel{?}{=} \bar{\alpha}_{e''}, \bar{\alpha}_{e''} \stackrel{?}{=} \alpha_e, \alpha_e \stackrel{?}{\leq}_b \bar{\alpha}_e\}$;
7. if $e = x$ (x a λ-bound variable) then $C_A(e) = \{\alpha_x \stackrel{?}{\leq}_b \bar{\alpha}_e\}$.
8. if $e = x$ (x a free variable with typing assumption $x : \tau$) then $C_A(e) = \{\tau \stackrel{?}{\leq}_b \bar{\alpha}_e\}$.

Every type derivation for an untyped λ-term e (in the implicit type inference system) corresponds uniquely to a *type labeling* of the syntax tree of e; that is, to a mapping of (λ-term) occurrences in e into type expressions. A type labeling that arises from a type derivation in this fashion, however, can equally well be viewed as a mapping from the canonical type variables associated above with the occurrences in e to type expressions. Consequently, every (implicit) type derivation for a λ-term e determines uniquely a *substitution* on these type variables by mapping every other type variable to itself. By induction on the syntax of λ-terms e it can be shown that such a substitution is a solution of the constraint system $C_A(e)$ and, vice versa, every solution of $C_A(e)$ is a substitution determined by a type derivation for e. Since every implicit type derivation of e corresponds to a unique completion of e we have the following theorem.

[4]The last condition is a technical condition to guarantee that solutions S and S' are considered equal for a constraint system C whenever their restrictions to the variables actually occurring in C are equal.

Theorem 1 *For every λ-term e and binding-time assumption A for e there is a one-to-one correspondence between the completions of e and the solutions of $C_A(e)$.*

Let us get an intuitive idea of why $C_A(e)$ captures exactly the type constraints necessary and sufficient for any completion of e w.r.t. A. First of all, the implicit version of our type inference system is useful since it describes well-typedness in terms of a given *unannotated* λ-term, which is all we have initially. In the implicit version of the type inference system of Figure 1 there are two rule schemes for every syntactic constructor (with the exception of constants and variables). Our goal is to translate the implicit system into a deterministic syntax-directed type system in which every construct has exactly one typing rule (scheme). The idea is to factor well-typedness into syntactic well-formedness plus satisfaction of (nonsyntactic) type constraints. Since syntactic well-formedness is presupposed in the input the solutions to the resulting type constraints characterize all the typings. For example, consider the implicit versions of the typing rules for λ-abstraction, (ABSTR) and (ABSTR-DYN), in Figure 1. They can be combined into the single rule

$$\text{(ABSTR-COMB)}\quad \frac{\begin{array}{l}[x:\tau'] \\ e:\tau \\ \tau' \to \tau = \tau'' \text{ or } \tau' = \tau = \tau'' = \Lambda\end{array}}{\lambda x.e : \tau''}$$

It is obvious that rule (ABSTR) is applicable if and only if the first disjunct in the side condition on types holds, and rule (ABSTR-DYN) is applicable if and only if the second disjunct holds. Since the disjunction is exclusive any rule application of (ABSTR-COMB) corresponds to an application of exactly one of the rules (ABSTR), (ABSTR-DYN), and it is easy to figure out to which. Note that $\tau' = \tau = \tau'' = \Lambda$ if and only if $\tau' \to \tau <_f \tau''$. Consequently we can write the disjunction as $\tau' \to \tau = \tau'' \vee \tau' \to \tau <_f \tau''$, viz. $\tau' \to \tau \leq_f \tau''$. Using the type variables associated with subterms in the construction of $C_A(e)$, rule (ABSTR-COMB) can be applied to a λ-abstraction $e = \lambda x.e'$ if and only if $\alpha_x \to \alpha_{e'} \stackrel{?}{\leq}_f \alpha_e$ is solvable; c.f. the constraints in $C_A(e)$ above.[5] Generally it is possible that a subterm is "lifted" using rule (LIFT). For occurrence e' the type variable $\alpha_{e'}$ represents the "immediate" type of e', and $\bar{\alpha}_{e'}$ represents its type after possible lifting. This explains the type constraints of the form $\alpha_{e'} \stackrel{?}{\leq}_b \bar{\alpha}_{e'}$ in the definition of $C_A(e)$.

Note that the inequalities $<_b, <_f$ do *not* induce inequalities on higher-order types because there is no analogue for "induced" lifting in the type inference system.

The constraint system $C_A(e)$ depends only on the language of type expressions, not the actual syntax of λ-terms. When adding additional language primitives or additional type expressions (such as a list-type and appropriate list manipulation primitives) it may be possible to modify the construction of the constraint system correspondingly without actually changing the class of constraints that are to be solved in an essential way. Thus any efficient method for solving constraint systems can conceivably be reused directly for (slightly) different program analyses. In the following two sections we show how constraint systems arising here can be solved efficiently. We view the initial translation of (a type inference problem for) a program term e into the "intermediate language" of constraints as the "front end" of our analysis and the algorithm(s)

[5] Similar considerations can be applied to other type inference problems to derive systematically reductions to solvability of type constraint systems; in particular, the reductions of Mitchell [Mit84], Wand [Wan87], Fuh and Mishra [FM88], Stansifer [Sta88], Henglein [Hen88b] can be derived in this fashion.

for solving these constraints as its "back end". We hope that, possibly after some suitable generalization, our constraints are both expressive enough to capture many interesting program analyses and still constrained enough to admit efficient computation of solutions.

4 Normalization of type constraints

In Section 3 we have seen that the type derivations for a λ-term e under binding-time assumption A — and thus its completions — can be characterized by the solutions of a constraint system $C_A(e)$. In this section we present transformations that preserve the set of solutions of such a constraint system. A constraint system that is in normal form with respect to these transformations will have the further property that it defines directly a solution with "minimality" property.

Our transformation rules define a labeled reduction relation $C \stackrel{S}{\Rightarrow} C'$ where C and C' are constraint systems and S is a substitution. If the substitution is the identity substitution we simply write $C \Rightarrow C'$. For substitution S and constraint system C, we denote applying S to all type expressions in C by $S(C)$. Let $G(C)$ be the directed graph on variables in constraint system C that contains an edge (α, β) if and only if there is an inequality constraint of the form $\alpha \to \alpha' \stackrel{?}{\leq}_f \beta$ or $\alpha' \to \alpha \stackrel{?}{\leq}_f \beta$ in C. If $G(C)$ contains a cycle we say C is *cyclic*; *acyclic* otherwise.[6] The transformation rules are given in Figure 2. The first two inequality constraint rules show how inequality constraints with *identical* right-hand sides are eliminated: If the left-hand sides have the *same* type constructor then these left-hand sides are equated in the "reduced" system (Rule 1a); if the left-hand sides have *different* left-hand side type constructors then the right-hand side is equated with Λ (Rule 1b) and the inequalities are eventually eliminated by Rules 1f and 1g.

The transitive closure of the transformation rules is defined by: $C \stackrel{S}{\Rightarrow}\!+\, C'$ if $C \stackrel{S}{\Rightarrow} C'$ and $C \stackrel{S;S'}{\Rightarrow}\!+\, C''$ if $C \stackrel{S}{\Rightarrow}\!+\, C', C' \stackrel{S'}{\Rightarrow}\!+\, C''$ for some C', where $S; S'$ denotes the left-to-right composition of S and S'. We say C is a *normal form* (or *normalized*) constraint system if there is no C' such that $C \stackrel{S}{\Rightarrow} C'$ for any S, and C *has* a normal form if there is a normal form C' such that $C \stackrel{S}{\Rightarrow}\!+\, C'$ for some substitution S. The correctness of the transformations is captured in the following theorem, which is easily proved by induction on the length of transformation sequences and by case analysis of the individual rules using elementary properties of $\leq_b, \leq_f$.

Theorem 2 *(Soundness and completeness of transformations)*
Let $C \stackrel{S}{\Rightarrow}\!+\, C'$. *Then* $Sol(C) = \{(S; S') \mid S' \in Sol(C')\}$.

The transformations can be used to derive an algorithm for normalizing constraint systems based on the following theorem.

Theorem 3 *(Normalization of constraint systems)*

1. *The transformations of Figure 2 are (weakly) normalizing; that is, every* C *has a normal form.*

[6] Constraints of the form $\alpha \stackrel{?}{=} \alpha'$ and $\alpha \stackrel{?}{\leq}_b \alpha'$ need not be considered in the definition of cyclicity since our transformation rules eliminate all equational constraints and $\leq_b$-inequality constraints remaining in a normal form constraint system are irrelevant.

1. (inequality constraint rules)

   (a) $C \cup \{\alpha \to \alpha' \overset{?}{\leq}_f \gamma, \beta \to \beta' \overset{?}{\leq}_f \gamma\} \Rightarrow C \cup \{\alpha \to \alpha' \overset{?}{\leq}_f \gamma, \alpha \overset{?}{=} \beta, \alpha' \overset{?}{=} \beta'\}$;

   (b) $C \cup \{\alpha \to \alpha' \overset{?}{\leq}_f \gamma, B \overset{?}{\leq}_b \gamma\} \Rightarrow C \cup \{\alpha \to \alpha' \overset{?}{\leq}_f \gamma, B \overset{?}{\leq}_b \gamma, \gamma \overset{?}{=} \Lambda\}$;

   (c) $C \cup \{\alpha \to \alpha' \overset{?}{\leq}_f \beta, \beta \overset{?}{\leq}_b \beta'\} \Rightarrow C \cup \{\alpha \to \alpha' \overset{?}{\leq} \beta, \beta \overset{?}{=} \beta'\}$;

   (d) $C \cup \{B \overset{?}{\leq}_b \alpha, \alpha \overset{?}{\leq}_b \alpha'\} \Rightarrow C \cup \{B \overset{?}{\leq}_b \alpha, B \overset{?}{\leq}_b \alpha', \alpha \rhd \alpha'\}$;

   (e) $C \cup \{B \overset{?}{\leq}_b \alpha', \alpha \overset{?}{\leq}_b \alpha'\} \Rightarrow C \cup \{B \overset{?}{\leq}_b \alpha, B \overset{?}{\leq}_b \alpha', \alpha \rhd \alpha'\}$;

   (f) $C \cup \{\alpha \to \alpha' \overset{?}{\leq}_f \Lambda\} \Rightarrow C \cup \{\alpha \overset{?}{=} \Lambda, \alpha' \overset{?}{=} \Lambda\}$;

   (g) $C \cup \{B \overset{?}{\leq}_b \Lambda\} \Rightarrow C$;

   (h) $C \cup \{\Lambda \overset{?}{\leq}_b \alpha\} \Rightarrow C \cup \{\Lambda \overset{?}{=} \alpha\}$.

   (i) $C \cup \{\alpha \overset{?}{\leq}_b \Lambda\} \Rightarrow C \cup \{B \overset{?}{\leq}_b \alpha\}$ if α is a type variable.

2. (equational constraint rules)

   (a) $C \cup \{\Lambda \overset{?}{=} \alpha\} \Rightarrow C \cup \{\alpha \overset{?}{=} \Lambda\}$ if α is a type variable;

   (b) $C \cup \{\Lambda \overset{?}{=} \Lambda\} \Rightarrow C$;

   (c) $C \cup \{\alpha \overset{?}{=} \alpha'\} \overset{S}{\Rightarrow} S(C)$ if α is a type variable and $S = \{\alpha \mapsto \alpha'\}$;

3. (dependency constraint rules)

   (a) $C \cup \{\alpha \rhd \Lambda\} \Rightarrow C$;

   (b) $C \cup \{\Lambda \rhd \alpha\} \Rightarrow C \cup \{\alpha \overset{?}{=} \Lambda\}$;

4. (occurs check rule)

   (a) $C \Rightarrow C \cup \{\alpha \overset{?}{=} \Lambda\}$ if C is cyclic and α is on a cycle in $G(C)$.

Figure 2: Transformation rules for constraint systems

2. *If C' is a normal form constraint system then*
   (a) *it has no equational constraints;*
   (b) *it is acyclic;*
   (c) *its constraints are of the form $\beta \to \beta' \overset{?}{\leq}_f \alpha, \gamma \overset{?}{\leq}_b \alpha$ or $\alpha \rhd \alpha'$ where: α, α' are type variables; β is a type variable or the type constant Λ; and γ is a type variable or the type constant B.*
   (d) *for every inequality constraint of the form $\beta \to \beta' \overset{?}{\leq}_f \alpha$ the type variable α does not occur on the right-hand side of other $\leq_f$-inequalities or on the left-hand side of $\leq_b$-inequalities;*
   (e) *for every inequality constraint of the form $B \overset{?}{\leq}_b \alpha$ the type variable α does not occur on the right-hand side of $\leq_f$-inequalities or on either side of $\leq_b$-inequalities.*
3. *If C contains no constraints of the form $\alpha \overset{?}{\leq}_b \alpha'$ where α is a type variable and $C \overset{S}{\Rightarrow}+ C'$ then C' contains no constraint of that form either.*

Proof: *1.* Define a megastep as follows: apply any applicable rule and then apply the equational constraint transformation rules exhaustively. It is easy to see that every megastep terminates and that after it terminates all equational constraints have been eliminated. Let c be the number of constraints; n the number of variables occurring in them; and v the number of inequality constraints with a variable on the left-hand side. It is easy to check that every megastep decreases the sum $c+n+2v$ by at least one. Consequently every sequence of megasteps terminates.
2. By definition of normal form.
3. None of the rules introduce $\leq_b$-inequalities with a variable on the left-hand side. (End of proof) ■

Normal forms are unique modulo type variable renamings. A solution S of a constraint system C is a *ground solution* if it maps all types into ground types; that is, types that contain no type variables. We say that a ground solution S of a constraint system C is *minimal* if: it solves all $\leq_f$-inequality constraints equationally; and for every type variable α, if there is *any* (ground) solution S' such that $S'(\alpha) = B$ and S' solves all $\leq_f$-inequality constraints equationally then $S(\alpha) = B$. Clearly, a minimal solution is unique if it exists at all.

Theorem 4 *Every normal form constraint system C has a minimal solution.*

Proof: Interpret all inequalities in C as equations. Since C is a normal form constraint system, by Theorem 3, part 2, these equations have a most general unifier U [LMM87] . Let BS be the substitution that maps every type variable occurring in $U(C)$ to B. Since neither U nor BS substitutes Λ for any type variable, all the dependency constraints in C are trivially satisfied. Consequently, the substitution $U; BS$ is a ground solution that solves all $\leq_f$-constraints equationally. Let S' be any other solution with $S'(\alpha) = B$ that solves the $\leq_f$-constraints equationally. Since S' is a solution α cannot be the right-hand side of an $\leq_f$-constraint, and since it solves $\leq_f$-constraints equationally there is no sequence of type variables $\alpha_0, \ldots, \alpha_k$ such that $\alpha_0 = \alpha$, α_k is the right-hand side of an $\leq_f$-constraint, and $\alpha_{i-1} \overset{?}{\leq}_b \alpha_i$ or $\alpha_i \overset{?}{\leq}_b \alpha_{i-1}$ for $0 < i \leq k$. As a consequence U maps α to B or to a type variable. In either case $U; BS$ maps α to B. This shows that S is a minimal solution. (End of proof) ■

By extension we define as the minimal solution of an arbitrary constraint system C the substitution $S;U$ if $C \stackrel{S}{\Rightarrow}+ C'$, C' is a normal form, and U is the minimal solution of C'. This proposition shows that every constraint system is solvable, and in particular that the $\leq_f$-inequality constraints in a normal form constraint system can be interpreted as equational constraints without losing solvability. Note that this property holds for normal forms, but not for general constraint systems.

5 Minimal Completions

The minimal solution of $C_A(e)$ corresponds to a canonical "minimal" completion of e that has a minimum of late-binding operators and the "most static" type annotations possible. Formally, let $\sqsubseteq$ be the smallest partial order such that $B \sqsubseteq \tau, \tau' \sqsubseteq \Lambda$ for all τ, τ' and $\sigma_1 \to \tau_1 \sqsubseteq \sigma_2 \to \tau_2$ if $\sigma_1 \sqsubseteq \sigma_2, \tau_1 \sqsubseteq \tau_2$ (c.f., [Gom89]), and extend it point-wise to lists of type assumptions. A completion of e is minimal if the operator annotations occurring in it occur in *every* completion of e and every type annotation τ in any other completion with the *same* operator annotations has a type τ' in the corresponding position with $\tau \sqsubseteq \tau'$ (w.r.t. to a binding-time assumption A). For example, both $(\underline{\lambda}x.x\underline{@}y)\underline{@}z$ and $(\lambda x:\Lambda.x\underline{@}y)@z$ are completions of $(\lambda x.x@y)@z$ w.r.t. $\{y:\Lambda, z:\Lambda\}$, but only $(\lambda x:\Lambda.x\underline{@}y)@z$ is minimal.

Since the "="-part of an inequality constraint $\alpha \to \alpha' \stackrel{?}{\leq}_f \alpha''$ in constraint system $C_A(e)$ for λ-term e arises from applying an unannotated (early-binding) typing rule in Figure 1 whereas its "$<_f$"-part derives from the corresponding annotated (late-binding) type rule, the minimal solution of $C_A(e)$ translates into a completion of e (via Theorem 1) that has only operator annotations where every completion of e has one. This satisfies the first half of the minimal completion definition. From Theorem 4 it can be shown that it also "minimizes" the type annotations since every other completion with the same operator annotations also solves the $\leq_f$-inequality constraints equationally. Thus we have the following theorem.

Theorem 5 *(Minimal typing)*
Every λ-term e has a unique minimal completion w.r.t. any binding-time assumption A for e.

This theorem extends and sharpens Gomard's minimal completion result since his definition of minimality does not take type annotations into account and his result is for his type system without rule (LIFT) [Gom89].

6 Efficient constraint normalization

Since the transformations of Section 4 are normalizing they can be used to design an algorithm that first normalizes a constraint system and then extracts the minimal solution from it. Instead of using the naive normalization strategy described in the proof of Theorem 3 we present a fast algorithm using efficient data structures whose actions can be interpreted as implementations of (sequences of) transformation steps.

This algorithm only works on constraint systems with constraints of the form $\alpha \to \alpha' \stackrel{?}{\leq}_f \alpha'', B \stackrel{?}{\leq}_b \alpha, \alpha \stackrel{?}{=} \alpha'$ and $\alpha \rhd \alpha'$ where $\alpha, \alpha', \alpha''$ are type variables or the type constant Λ. No constraints of the form $\alpha \stackrel{?}{\leq}_b \alpha'$ are permitted where α is a type variable. Since the type

constructor ($\rightarrow$ or B) on the left-hand side of an inequality constraint identifies uniquely whether it is a $\leq_b$- or $\leq_f$-inequality we shall drop the subscripts in this section. At the end of this section it is indicated how our efficient constraint normalization algorithm algorithm can be refined to accommodate constraints of the form $\alpha \stackrel{?}{\leq}_b \alpha'$.

Term graphs with equivalence classes have been used for fast implementations of unification. We shall not go into details, but refer the reader to the literature; e.g. [HK71,AHU74,Hue76, PW78,MM82,ASU86]. The equivalence classes are represented by a system of equivalence class representatives (*ecr's*), and there are two operations available on ecr's: find(n) for node n is a function that returns the ecr of the equivalence class to which n belongs; union(n, n'), which can only be applied to distinct ecr's n, n', is a procedure that merges the equivalence classes of n and n' and returns an ecr of the merged equivalence class. An analysis of efficient union/find data structures can be found in [Tar83]. Our notation uses standard control structure dictions and some special operations, which are explained in Figure 3. All primitive operations, apart from union and find, can easily be implemented to execute in constant time.

A *term graph representation* of a constraint system C consists of

- a term graph, representing all terms in C;
- an equivalence relation on its nodes, representing a substitution;
- a *dependency* list dps(n) of nodes $[n_1, \ldots, n_k]$ associated with every variable-labeled equivalence class representative n, representing dependencies $n \rhd n_i$;
- a nonvariable node leq(n) associated with every equivalence class representative n, representing the inequality constraints leq(n) $\stackrel{?}{\leq}_f$ n or leq(n) $\stackrel{?}{\leq}_b$ n;
- a worklist consisting of
  - inequality pairs (n <= n') with n non-variable-non-Λ-labeled, representing inequality constraints n $\stackrel{?}{\leq}_f$ n' or n $\stackrel{?}{\leq}_b$ n';
  - equality pairs (n = n'), representing equational constraints n $\stackrel{?}{=}$ n';
- a set dynned, implemented by a Boolean-valued map on nodes, for keeping track of those function-labeled nodes n whose children have already been set to Λ ("dynned"); that is, dynned(n) is set to true at the point when for all children c of n the equational constraints $c \stackrel{?}{=} \Lambda$ are added to the worklist.

Our algorithm delays checking for the conditions of the occurs check (see Figure 2) as long as possible since its applicability hinges upon a global condition of the constraint system; i.e., one that cannot be checked by simply looking at one or two constraints at a time. As a consequence it operates in four phases.

Phase 1: For constraint system C a term graph representation is constructed. It consists of: a term graph representing all terms occurring in C; the equivalence classes initialized to consist of one node each; all dependencies n $\rhd$ n', n $\rhd$ n", ...for n represented by initializing dps(n) to [n', n", ...]; leq(n) undefined for every node n; the worklist initialized to all inequality and equality constraints; dynned(n) set to false for every n.

(General operations)	
e = e'	returns true if e is equal to e'; false otherwise
e <> e'	returns true if e is not equal to e'; false otherwise
(Operations on lists)	
remove e **from** W	removes an element from (nonempty) list W and assigns it to variable e
add v **to** W	adds element v to the list W
[]	denotes the empty list
L ++ L'	concatenates lists L and L' destructively and returns the result
for x **in** L **do** <statement> **end for**;	executes <statement> for every element x in list L where L is traversed from "left to right"
for x **in** L || x' **in** L' **do** <statement> **end for**;	executes <statement> for every pair of elements x, x' while simultaneously and synchronously iterating over lists L and L' (L and L' have the same length)
(Operations on nodes)	
constr(n)	return the constructor of the nonvariable term represented by node n
children(n)	returns the list of children nodes of node n
dps(n)	returns a list of nodes dependent on ecr n (see dependency constraints)
leq(n)	if defined, for ecr n returns a node that must be $\leq_f$ n or $\leq_b$ n
dynned(n)	returns true if the children c of ecr n have already been "dynned"; i.e., pairs (c = dyn) added to the worklist
(Operations on equiv. classes)	
find(n)	returns the ecr of the equivalence class to which n belongs
union(n, n')	merges the equivalence classes represented by the distinct ecrs n, n' and returns one of n, n' as the new ecr of the merged class; it returns a nonvariable-label node or a node with leq defined whenever possible

Figure 3: Operations used in fast constraint system normalization algorithm

```
while worklist <> [] do
   remove e from worklist;
   case e of
      (n <= n'):
         m := find(n); m' := find(n');
         if leq(m') undefined then
            leq(m') := m;
         else (* leq(m') is defined *)
            m'' := find(leq(m'));
            if constr(m) = constr(m'') then
               if m <> m'' then
                  _ := union(m, m'');
                  for c in children(m) || c' in children(m'') do
                     add (c = c') to worklist;
                  end for;
               end if;
            else (* constr(m) <> constr(m') *)
               add (m' = dyn) to worklist;
               if not dynned(m) then
                  dynned(m) := true;
                  for c in children(m) do
                     add (c = dyn) to worklist;
                  end for;
               end if;
               if not dynned(m'') then
                  dynned(m'') := true;
                  for c' in children(m'') do
                     add (c' = dyn) to worklist;
                  end for;
               end if;
            end if;
         end if;
      (n = n'):
         m := find(n); m' := find(n');
         if m <> m' then
            m'' := union(m, m');
            if m = dyn then
               (m, m') := (m', m);
            end if;
            if m' = dyn then
               for l in dps(m) do
                  add (l = dyn) to worklist;
               end for;
            else (* m' is variable *)
               dps(m'') := dps(m) ++ dps(m');
               if leq(m) defined and leq(m') defined then
                  (* assume w.l.o.g. m = m'' *)
                  add (leq(m') <= m'') to worklist;
               end if
            end if;
         end if;
   end case;
end while;
```

Figure 4: Phase 2 of efficient constraint system normalization algorithm

Phase 2: Using the term graph representation constructed in Phase 1 a (term graph representation of a) normal form C' for the transformation rules *without* the occurs check rule is computed. See Figure 4.

Phase 3: A maximal strongly connected component analysis of (the representation of) $G(C')$ is performed and for every nontrivial strongly connected component an element α is taken and the pair $(\alpha = \Lambda)$ added to the worklist. (Not described here; see, e.g., [AHU74].)

Phase 4: Finally, the normalization of Phase 2 is performed again until a normal form is reached.

The possibly cyclic "quasi-normal" form reached after Phase 2 could be interpreted, analogous to Theorem 1, with rational (regular recursive) type expressions. Phase 3 implements a single application of the occurs-check rule 4a in Figure 2. Inserting the equations $\alpha \stackrel{?}{=} \Lambda$ for variables α that are on cycles in $G(C')$ will break all cycles since *all* variables on cycles will be substituted by Λ in Phase 4. The final result is guaranteed to be acyclic and thus (the representation of) a normal form constraint system. The proof of Theorem 5 shows how a minimal solution can be extracted from the normal form constraint system.

We analyze the asymptotic (time) complexity of our algorithm by amortization; that is, by averaging the complexity of each primitive operation over the whole sequence of operations executed by the algorithm for a given input [Tar83]. To accomplish this we associate a *computational potential* P with a data structure and calculate the amortized cost c_a of a primitive operation on that data structure by $c_a = c + \Delta P$ where c is the actual running time of the operation and ΔP is the difference in potential before and after the operation, which may in general be positive or negative. The cumulative amortized cost of a sequence of primitive operations is then the sum of their running times plus the difference in potential before and after applying them. As a special case we get that if the amortized cost of every primitive operation is zero and the potential of the data structure *after* executing them is nonnegative, then the total running time is bounded from above by the *initial* potential. In the following we define a potential on the data structure manipulated in Phase 2 of our constraint normalization algorithm such that every operation (in Phase 2) has an amortized cost of zero.

The data structure at the beginning of Phase 2 consists of:

1. n nodes, with n' variable-labeled nodes, n'' non-variable-non-Λ-labeled nodes, and a node labeled by Λ; i.e., $n = n' + n'' + 1$;
2. m term edges represented by the set of children lists children(n) for non-variable-non-Λ-labeled nodes n;
3. i inequalities (n <= n');
4. e equations (n = n'); and
5. d dependencies, represented by the set of dependency lists dps(n).

The size of the input to the algorithm is $N = n + m + i + e + d$. We define the potential of the data structure manipulated in Phase 2 by associating a computational "chunk" with the following components: equalities (n = n') and inequalities (n <= n') in the worklist, and variable-labeled and non-variable-non-Λ-labeled ecr's in the term graph. The potential of the whole term graph representation is then the sum of all its associated chunks. We denote these

chunks by $c_e, c_i, c_v(m), c_n(m)$, respectively, and we write c_{uf} for the cost of one union- or find-operation.

$$\begin{aligned} c_e &= 3c_{uf} + O(1) \\ c_i &= 3c_{uf} + O(1) \\ c_v(n) &= |dps(n)|(c_e + O(1)) + c_i + O(1) \\ c_n(n) &= |children(n)|(2c_e + O(1)) + O(1) \end{aligned}$$

The chunks for equalities and inequalities are constant whereas the chunks for ecr's depend on the ecr itself. The total potential at the beginning of Phase 2 is

$$\begin{aligned} P_0 &= \Sigma_{n'} c_v + \Sigma_{n''} c_n + ic_i + ec_e \\ &= d(c_e + O(1)) + n'c_i + O(n') + m(2c_e + O(1)) + O(n'') + ic_i + ec_e \\ &= (d + n' + 2m + i + e)c_e + O(d) + O(n') + O(m) + O(n'') \\ &\leq 6Nc_{uf} + O(N) \end{aligned}$$

It can be verified that the amortized cost of every operation in Phase 2 is zero. Under common assumptions on the presentation of the input Phase 1 takes linear time. Since the potential is never negative, the combined running times of Phases 2 and 4 is bounded by P_0 by our considerations above. Phase 3 can be implemented in linear time [Tar72]. Finally, using *path compression* and *union by rank* (or similar union/find heuristics; see [Tar83]) for Phase 2 we get the following theorem.

Theorem 6 *A constraint system can be normalized in time $O(N\alpha(N,N))$ on a pointer machine, where N is the size of the input.*

Since $G(C)$ may be cyclic it does not appear that any of the linear-time unification algorithms can be adapted to our problem. Consequently it remains to be seen whether there is a linear-time algorithm for solving our constraint-systems. Note, however, that the factor $\alpha(n,n)$ can be treated as a small constant in practice. We believe that our algorithm is not only asymptotically fast, but also implementable, efficient, adaptable and responsive in practice. In particular it appears to be well-suited for adaptation in an incremental environment (see following section).

We conclude this section by sketching an extension of the fast constraint normalization algorithm to handle constraints of the form $\alpha \overset{?}{\leq}_b \alpha'$, too. First we "prenormalize" a constraint system with respect to rules 1c, 1d, and 1e. This can be accomplished with a linear-time reachability algorithm. During full constraint normalization substituting Λ for a type variable α by rule 2c triggers the following actions: for all α', α'' with $\alpha' \overset{?}{\leq}_b \alpha$ and $\alpha \overset{?}{\leq}_b \alpha''$ we add $(B <= \alpha')$ and $(\Lambda = \alpha')$ to the worklist. The cost of adding these constraints to the worklist and subsequently processing them is dominated by the other operations of the constraint normalization algorithm. So Theorem 6 also holds for constraint systems with "pure variable" constraints $\alpha \overset{?}{\leq}_b \alpha'$.

7 Efficient binding-time analysis

The constraint system normalization algorithm of Section 6 forms the core of efficient binding-time analysis. Theorem 5 guarantees that every λ-term e has a unique minimal completion given any binding-time assumption A for e. Our binding-time algorithm computes a minimal completion as follows.

1. (Construction) Construct a type constraint system $C_A(e)$ as described in Section 3.

2. (Normalization) Normalize the constraint system according to the transformation rules of Figure 2 using the fast constraint normalization algorithm of Section 6.

3. (Minimal solution) Construct a minimal solution as described in Section 4.

4. (Annotation) Annotate (the syntax-tree of) e with operator, type and lifting annotations using the minimal solution to get the minimal completion of e as described in Section 5.

Every type variable in the constraint normalization system is associated with a node in the syntax-tree. The binding-time annotations of the minimal completion of e can be computed in a single pass over the syntax-tree of e. Thus all but the constraint normalization can be done in linear time with respect to the size of e. This proves the following theorem.

Theorem 7 *The minimal completion of e w.r.t. binding-time assumption A (for e) can be computed in time $O(n\alpha(n,n))$ where n is the size of e.*[7]

Our constraint normalization algorithm can easily be adapted to an algorithm that operates on-line under additions of constraints to a constraint system without loss in efficiency. As a consequence the construction of the constraint normalization system and its normalization can be interleaved efficiently. For example, it is possible to display immediately the annotations of an untyped λ-term in an interactive environment as it is being typed in at an "almost-constant" cost per character typed. (This does *not* hold for *deletions*!) The flexibility of the constraint normalization algorithm should make it also possible to devise a "toolbox" of binding-time analyses for different environments by modularizing such an analysis into input and output actions, constraint construction and constraint normalization. Notice specifically that constraint normalization is independent of the syntax of the λ-terms unlike syntax-oriented algorithms that have to be modified every time the syntax is modified or enhanced.

We conclude this section with an illustration of how our binding-time analysis algorithm proceeds on the λ-term

$$(\lambda x.x@y)@z$$

w.r.t. the binding-time assumption $\{y : \Lambda, z : \Lambda\}$. Its syntax tree with associated type variables is displayed in Figure 5.

The constraint system constructed from this syntax tree is[8]

$$\begin{array}{rcl}
t0 & \overset{?}{\leq}_b & \overline{t1} \\
\Lambda & \overset{?}{\leq}_b & \overline{t2} \\
\Lambda & \overset{?}{\leq}_b & \overline{t5} \\
\overline{t2} \to t3 & \overset{?}{\leq}_f & \overline{t1},\ t3\ \overset{?}{\leq}_b\ \overline{t3} \\
t0 \to \overline{t3} & \overset{?}{\leq}_f & t4,\ t4\ \overset{?}{\leq}_b\ \overline{t4} \\
\overline{t5} \to t6 & \overset{?}{\leq}_f & \overline{t4},\ t6\ \overset{?}{\leq}_b\ \overline{t6}
\end{array}$$

[7] The cost of displaying the type annotations in string form is not included since the string representation of the type expressions may be exponentially bigger than the original input. Note this is also the case for simple type inference and for unification!

[8] In Figure 5 we write "D" instead of Λ.

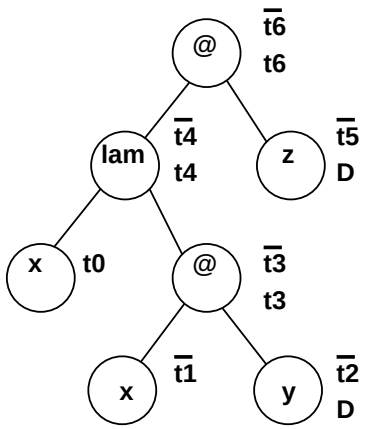

Figure 5: A syntax tree with type variables

Normalization produces the single constraint

$$\Lambda \to \Lambda \quad \stackrel{?}{\leq}_f \quad t4$$

and the substitution expressed by

$$\begin{array}{lll} t0, \overline{t1}, \overline{t2}, t3, \overline{t3}, \overline{t5}, t6, \overline{t6} & \mapsto & \Lambda \\ \overline{t4} & \mapsto & t4. \end{array}$$

The minimal solution of the remaining constraint is the substitution

$$t4 \quad \mapsto \quad \Lambda \to \Lambda.$$

Thus the minimal competion of $\lambda x.x@y)@z$ is the well-annotated λ-term

$$(\lambda x : \Lambda.x\underline{@}y)@z.$$

These annotations signal to a partial evaluator that the outer application can be evaluated statically whereas the inner application is deferred until the values for y and z become available.

8 Summary and directions for future research

Higher-order binding-time analysis can be viewed as a type inference problem for unannotated or partially annotated λ-terms in a two-level λ-calculus. We have shown that this type inference problem can be factored into: construction of type constraint systems with equational, conditional-equational, and specialized kinds of inequality constraints; transformation of this system into a normal form; and extraction of a "canonical" solution from the normal form. We have presented a higher-order binding-time analysis algorithm that can be implemented in almost-linear time. This improves on previous work in that our algorithm is guaranteed to produce minimal completions, yet its running time is almost linear; as such it is asymptotically more efficient than previous binding-time analysis algorithms. Since the data structures we use are known to behave well in practice, we expect our algorithm to be also very fast in practice.

The class of type constraint systems in this paper and our efficient constraint system normalization algorithm are of independent interest since they can be used to solve other analyses; in particular, we have applied it successfully to the description and design an efficient type inference algorithm for a dynamic typing discipline [Hen91].

Several problems might be addressed as a continuation of this work:

- Binding-time analysis has been cast in different conceptual frameworks: abstract interpretation, projection analysis, and type inference. What is their relative expressiveness for a given language? What is the computational complexity of these various problem formulations of binding-time analysis, and what is the complexity of already existing formulations?
- Can the type system with type Λ be generalized to an ML-style polymorphic system? What are its properties?
- In our system "lifting" is restricted to first-order values. But higher-order lifting is both possible and useful in practice, as exemplified in Similix [Bon90b]. What happens when lifting for static functions and "induced" lifting for arbitrary partially static structures and functions are also permitted?

Acknowledgments

I would like to express my thanks: to Carsten Gomard for many hours of explaining the use of his type inference system in binding-time analysis, for providing me with a copy of his Master's thesis, for listening patiently to the ideas presented in this paper as they were evolving, and for stimulating comments and suggestions on this work and possible continuations; to Neil Jones for clarifying and rectifying my understanding of binding-time analysis, for detailed comments, critical suggestions and several corrections[9] on earlier drafts of this paper, and for explaining to me the importance of the lift-operator in partial evaluation; to Johnny Chang for fine-combing through the fast constraint normalization algorithm and finding several small errors in it; to Anders Bondorf for suggesting valuable improvements to the presentation of this paper and for sharing some of his extensive practical knowledge of self-applicable partial evaluators with me; and to the TOPPS group at DIKU as a whole for providing active feedback and a generally stimulating environment.

References

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986. Addison-Wesley, 1986, Reprinted with corrections, March 1988.

[BJMS88] A. Bondorf, N. Jones, T. Mogensen, and P. Sestoft. Binding time analysis and the taming of self-application. To appear in Transactions on Programming Languages and Systems (TOPLAS), Sep. 1988.

[9] Naturally, any remaining errors are completely my responsibility just as those that have already been corrected were since they shouldn't even have been in the paper in the first place.

[Bon90a] A. Bondorf. Automatic autoprojection of higher order recursive equations. In N. Jones, editor, *Proc. 3rd European Symp. on Programming (ESOP '90), Copenhagen, Denmark*, pages 70–87. Springer, May 1990. Lecture Notes in Computer Science, Vol. 432.

[Bon90b] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Dec. 1990.

[Con88] C. Consel. New insights into partial evaluation: The Schism experiment. In *Proc. 2nd European Symp. on Programming (ESOP), Nancy, France*, pages 236–246. Springer, 1988. Lecture Notes in Computer Science, Vol. 300.

[Con90] C. Consel. Binding time analysis for higher order untyped functional languages. In *Proc. LISP and Functional Programmin (LFP), Nice, France.* ACM, July 1990.

[FM88] Y. Fuh and P. Mishra. Type inference with subtypes. In *Proc. 2nd European Symp. on Programming*, pages 94–114. Springer-Verlag, 1988. Lecture Notes in Computer Science 300.

[GJ91] C. Gomard and N. Jones. A partial evaluator for the untyped lambda-calculus. *J. Functional Programming*, 1(1):21–69, Jan. 1991.

[Gom89] C. Gomard. Higher order partial evaluation – hope for the lambda calculus. Master's thesis, DIKU, University of Copenhagen, Denmark, September 1989.

[Gom90] C. Gomard. Partial type inference for untyped functional programs (extended abstract). In *Proc. LISP and Functional Programming (LFP), Nice, France*, July 1990.

[Hen88a] F. Henglein. Simple type inference and unification. Technical Report (SETL Newsletter) 232, Courant Institute of Mathematical Sciences, New York University, Oct. 1988.

[Hen88b] F. Henglein. Type inference and semi-unification. In *Proc. ACM Conf. on LISP and Functional Programming (LFP), Snowbird, Utah*, pages 184–197. ACM Press, July 1988.

[Hen91] F. Henglein. Dynamic typing. Semantique Note 90, DIKU, University of Copenhagen, 35 pp., March 1991.

[HK71] J. Hopcroft and R. Karp. An algorithm for testing the equivalence of finite automata. Technical Report TR-71-114, Dept. of Computer Science, Cornell U., 1971.

[HS91] S. Hunt and D. Sands. Binding time analysis: A new PERspective. In *Proc. ACM/IFIP Symp. on Partial Evaluation and Semantics Based Program Manipulation (PEPM), New Haven, Connecticut*, June 1991.

[Hue76] G. Huet. *Résolution d'equations dans des langages d'ordre 1, 2, ..., omega (thèse de Doctorat d'Etat).* PhD thesis, Univ. Paris VII, Sept. 1976.

[JGB+90] N. Jones, C. Gomard, A. Bondorf, O. Danvy, and T. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *1990 International Conference on Computer Languages, New Orleans, Louisiana, March 1990*, pages 49–58. IEEE Computer Society, 1990.

[JM76] N. Jones and S. Muchnick. Binding time optimization in programming languages: Some thoughts toward the design of the ideal language. In *Proc. 3rd ACM Symp. on Principles of Programming Languages*, pages 77–94. ACM, Jan. 1976.

[JM78] N. Jones and S. Muchnick. *TEMPO: A Unified Treatment of Binding Time and Parameter Passing Concepts in Programming Languages*, volume 66 of *Lecture Notes in Computer Science*. Springer, 1978.

[Jon87] N. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjorner, A. Ershov, and N. Jones, editors, *Proc. Partial Evaluation and Mixed Computation*, pages 225–282. IFIP, North-Holland, Oct. 1987.

[JS80] N. Jones and D. Schmidt. Compiler generation from denotational semantics. In N. Jones, editor, *Semantics-Directed Compiler Generation, Aarhus, Denmark*, pages 70–93. Springer, 1980. Lecture Notes in Computer Science, Vol. 94.

[JSS85] N. Jones, P. Sestoft, and H. Sondergard. An experiment in partial evaluation: The generation of a compiler generator. *SIGPLAN Notices*, 20(8), Aug. 1985.

[JSS89] N. Jones, P. Sestoft, and H. Sondergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2:9–50, 1989.

[Kro81] H. Kroeger. Static-Scope-Lisp: Splitting an interpreter into compiler and run-time system. In W. Brauer, editor, *GI — 11. Jahrestagung, Munich, Germany*, pages 20–31. Springer, Munich, Germany, 1981. Informatik-Fachberichte 50.

[Lau87] J. Launchbury. Projections for specialisation. In D. Bjorner, A. Ershov, and N. Jones, editors, *Proc. Workshop on Partial Evaluation and Mixed Computation, Denmark*, pages 465–483. North-Holland, Oct. 1987.

[Lau90] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, University of Glasgow, Jan. 1990.

[LMM87] J. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kauffman, 1987.

[Mit84] J. Mitchell. Coercion and type inference. In *Proc. 11th ACM Symp. on Principles of Programming Languages (POPL)*, 1984.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, Apr. 1982.

[Mog87] T. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjorner, A. Ershov, and N. Jones, editors, *Proc. Workshop on Partial Evaluation and Mixed Computation, Denmark*, pages 325–347. North-Holland, Oct. 1987.

[Mog89] T. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Diaz and F. Orejas, editors, *Proc. Int. Conf. Theory and Practice of Software Development (TAPSOFT), Barcelona, Spain*, pages 298–312. Springer, March 1989. Lecture Notes in Computer Science 352.

[NN86] H. Nielson and F. Nielson. Semantics directed compiling for functional languages. In *Proc. ACM Conf. on LISP and Functional Programming (LFP)*, 1986.

[NN88a] H. Nielson and F. Nielson. Automatic binding time analysis for a typed lambda calculus. *Science of Computer Programming*, 10:139–176, 1988.

[NN88b] H. Nielson and F. Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56, 1988.

[PW78] M. Paterson and M. Wegman. Linear unification. *J. Computer and System Sciences*, 16:158–167, 1978.

[Rom87] S. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In D. Bjorner, A. Ershov, and N. Jones, editors, *Proc. Workshop on Partial Evaluation and Mixed Computation, Denmark*, pages 465–483. North-Holland, Oct. 1987.

[Sch87] D. Schmidt. Static properties of partial evaluation. In D. Bjorner, A. Ershov, and N. Jones, editors, *Proc. Workshop on Partial Evaluation and Mixed Computation, Denmark*, pages 465–483. North-Holland, Oct. 1987.

[Ses85] P. Sestoft. The structure of a self-applicable partial partial evaluator. In H. Ganzinger and N. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark*, pages 236–256. Springer, 1985. published in 1986, Lecture Notices in Computer Science, Vol. 217.

[Sta88] R. Stansifer. Type inference with subtypes. In *Proc. 15th ACM Symp. on Principles of Programming Languages*, pages 88–97, San Diego, California, Jan. 1988. ACM.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2), June 1972.

[Tar83] R. Tarjan. *Data Structures and Network Flow Algorithms*, volume CMBS 44 of *Regional Conference Series in Applied Mathematics*. SIAM, 1983.

[Wan87] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, X:115–122, 1987.