Binding-Time Analysis for Standard ML

Lars Birkedal & Morten Welinder
DIKU, Department of Computer Science
University of Copenhagen
DK–2100 Copenhagen Ø, Denmark
Email: `birkedal@diku.dk` & `terra@diku.dk`

Abstract

We present an efficient base algorithm for binding-time analysis based on constraint solving and the union-find algorithm. In practice it has been used to handle all of Standard ML except modules and we show the principles of how constraints can be used for binding-time analysis of Standard ML; in particular we show how to binding-time analyse nested pattern matching. To the best of our knowledge no previous binding-time analysis has treated nested pattern matching.

This article is partly based on [BW93] in which we document the development of a semantics-based compiler generator, ML-MIX, for Standard ML programs.

1 Introduction

There are two common ways of doing binding-time analysis: by fixed-point iteration and by constraint solving. Analyses based on the former method tend to be stronger [PS94], but have so far been rather slow (cubic complexity for higher-order languages) and thus impractical for large programs. Analyses based on the latter method are as fast as we can expect — essentially linear time complexity — and have been treated in [Hen91] (untyped lambda calculus), [And92] (a subset of C), [BJ93] (a subset of Scheme), and [BW93] (a subset of Standard ML).

Except for [BW93], constraint based binding-time analyses have either been for dynamically typed languages or have not used the available type information. We show that the use of type information is advantageous because it leads to an algorithm that is simple, proven correct, very efficient, and implemented.

We have used our algorithm to handle all of Core Standard ML [MTH90] which clearly demonstrates that usefulness has not been sacrificed for simplicity.

1.1 Constraint Based Binding-Time Analysis

Figure 1 outlines the stages of constraint based binding-time analysis. First, the abstract syntax tree of the source program is annotated with one or more variables for every

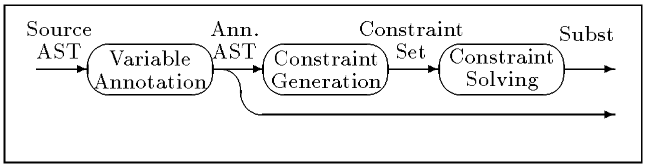

Figure 1: The structure of a BTA based on constraints.

node in the tree. These variables are placeholders for the binding times that will be inferred and for certain control information to be described in Section 3.

In the next phase a set of constraints on the variables is generated from the annotated syntax tree. The set of constraints is called a constraint system and the constraints describe the necessary relations between the binding-time types that the variables will take.

We present our constraint systems in Section 2 and we shall see that we our constraint systems are more general than those used by Henglein [Hen91]; in particular, we allow more expressive logical side conditions in the constraint systems.

In Section 3 we explain how the constraints can be used in a binding-time analysis for Standard ML by describing which constraints are generated for typical constructs in Stardard ML.

After the constraint generation, the set of constraints is solved by assigning a binding-time type to every variable. Constraint solving is done by rewriting constraint systems until a so-called normal form is reached. From the normal form it is easy to obtain the solution mapping the binding-time variables to binding-time types. In Section 4 we define the normalization process and how to extract a solution from a normal form and we prove the correctness of the constraint solving algorithm. Moreover, we describe and solve a serious problem with Henglein's constraint solving algorithm [Hen91].

A naive implementation of the constraint solving algorithm would be extremely slow. Therefore we show how it can be implemented efficiently using union-find techniques in two levels. One of these union-find levels is specifically used to handle the above mentioned problem with Henglein's constraint solving algorithm. We determine the complexity of the algorithm with respect to the size of the constraint system. The complexity turns out to be essentially linear as is also the case for Henglein's algorithm [Hen91]. Thus, the extra union-find level used to solve the above mentioned problem has not added to the time complexity of the al-

Accepted for ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM–94.

gorithm. The efficient implementation and the complexity analysis is presented in Section 5. A concrete Standard ML implementation of the algorithm is given in AppendixA.

The result of the binding-time analysis is the solution mapping the binding-time variables to binding-time types together with the annotated syntax tree. Often, these two parts are post-processed into another annotated abstract syntax tree called a two-level syntax tree [NN88,BW93] but we have chosen not to include this step here for simplicity.

2 Constraint Systems

Definition 1 (Binding-time types) *The binding-time types are generated by the following grammar:*

$$\tau ::= \mathtt{D} \mid \mathtt{S} \mid [\tau_1, \ldots, \tau_n]$$

where $n \geq 0$.

These will be used respectively to describe dynamic entities, static first-order values, and structured values. (The square brackets do *not* imply any connection with lists.)

We assume the existence of an infinite set of variables called binding-time variables, and introduce the convention that α's range over binding-time variables, that β's range over binding-time variables or the binding-time type D, and that τ's range over binding-time types.

Definition 2 (Constraint systems) *A constraint system is a finite set of formal constraints, each of which must take one of the following forms:*

Constraint	Name
$\beta_1 = \beta_2$	equality constraint
$(\beta_1, \ldots, \beta_n) \triangleright \beta$, where $n \geq 0$	dependency constraint
$[\beta_1, \ldots, \beta_n] \leq \beta$, where $n \geq 0$	structure constraint
$\beta_1 \rightsquigarrow \beta_2$	lift constraint

Note that the notation here is just syntax and β's. There is no implied connection between say $[\beta_1, \ldots, \beta_n]$ and structured binding-time types; the interpretation is given below. Constraints of the form $(\beta_1) \triangleright \beta$ will be written as $\beta_1 \triangleright \beta$ for short. The binding-time type S is not allowed in any constraint system.

To the above formal constraints we attach semantics[1] Each constraint is either satisfied or not, so they are predicates in the binding-time variables. The semantics is relative to a substitution, S, that maps binding-time variables to binding-time types and D to D.

Definition 3 (Constraint semantics) *Let τ_i stand for $S(\beta_i)$ (and so on). The constraints have the following semantics with respect to S:*

- *The equality constraint $\beta_1 = \beta_2$ is satisfied if and only if τ_1 and τ_2 consist of the same string of symbols.*
- *The dependency constraint $(\beta_1, \ldots, \beta_n) \triangleright \beta$ is satisfied if $\tau = \mathtt{D}$ or, if $\tau_i \neq \mathtt{D}$ for some i. The informal meaning is that "if τ_1 and τ_2, ..., and τ_n all are D, then τ must also be D."*
- *The structure constraint $[\beta_1, \ldots, \beta_n] \leq \beta$ is satisfied either by equality or if τ and all τ_i's are D. In words, structured values that are dynamic have dynamic components.*
- *The lift constraint $\beta \rightsquigarrow \beta'$ is satisfied either by equality, or if $\tau = \mathtt{S}$ and $\tau' = \mathtt{D}$.*

[1] Following mathematical tradition we do not distinguish notationally between formal and semantic constraints.

Example 1 All the following constraints are satisfied under the substitution $\{\beta_1 \mapsto \mathtt{S}, \beta_2 \mapsto \mathtt{D}, \beta_3 \mapsto [\mathtt{S},\mathtt{D}], \mathtt{D} \mapsto \mathtt{D}\}$:

$$\beta_1 \rightsquigarrow \mathtt{D} \quad \beta_1 \rightsquigarrow \beta_2 \quad [\beta_1, \beta_2] \leq \beta_3 \quad (\beta_2, \beta_3) \triangleright \beta_1 \quad (\mathtt{D}, \beta_2) \triangleright \beta_2$$

whereas none of the following is

$$\beta_2 \rightsquigarrow \beta_1 \quad (\mathtt{D}, \beta_2) \triangleright \beta_1 \quad [\beta_2, \beta_1] \leq \beta_3 \quad [\mathtt{S}, \mathtt{D}, \mathtt{D}] \leq \beta_2 \quad \beta_1 = \beta_2$$

Note that the order of components is important in structure constraints. □

2.1 Well-Typedness

To make the solving of constraint systems efficient and simple we impose a well-typedness condition on constraint systems. The intention is that constraint systems generated from well-typed programs should be well-typed.

Definition 4 (Well-typed constraint system) *Let C be a constraint system and let $R(C)$ be the subset of C consisting of all constraints not having the form $(\ldots) \triangleright \beta$. Let $R'(C)$ be $R(C)$ with all occurrences of D replaced by different fresh binding-time variables.*

We define C to be well-typed *if $R'(C)$ can be solved equationally (i.e., if each structure and lift constraint can be replaced by a corresponding equality constraint, and a most general unifier, $U(C)$, exists for the resulting equational system). The solution may be circular to handle recursive data types.*

The exact purpose of this (admittedly rather obscure) definition is to make sure that whenever we have two structure constraints with the same right-hand side then they have the same number of components on the left-hand side.

Example 2 Consider the following set C of constraints

$$\{[\beta_1, \beta_2] \leq \beta_3, \quad [\beta_4, \beta_5, \beta_6] \leq \beta_3\}$$

Here $R'(C)$ cannot be solved by equality and thus C is not well-typed. □

2.2 Solutions

A constraint system is a kind of inequality system to be solved by replacing binding-time variables with binding-time types. To be specific

Definition 5 (Solutions to constraint systems) *Let C be a well-typed constraint system and let S be a substitution mapping C's binding-time variables into binding-time types. Let S' be the extension of S to all binding-time variables by mapping variables not occurring in C to themselves and let S'' be the component wise extension of S' to constraints. We say that S is a solution of C if for every constraint $c \in C$ it holds that $S''(c)$ is satisfied.*

It is easy to see that any constraint system has a solution, $S_{\mathtt{D}}$, which maps all variables present to the constant D. This property does not even depend on well-typedness — every constraint will simply be satisfied by $S_{\mathtt{D}}$ (see Definitions 2 and 3). We need more interesting solutions so we introduce the following partial ordering on solutions.

Definition 6 (Minimal solutions) *We define a partial order, $\preceq$, on binding-time types by $\tau \preceq \mathtt{D}$, $\tau \preceq \tau$, and $[\tau_1, \ldots, \tau_n] \preceq [\tau_1', \ldots, \tau_n']$ if $\forall i : \tau_i \preceq \tau_i'$. We extend this to variables by defining $\alpha \preceq \mathtt{D}$ and $\alpha \preceq \alpha$. We finally extend it to solutions by defining $S \preceq S'$ if $\forall \alpha : S(\alpha) \preceq S'(\alpha)$. We then define*

$$S \prec S' \iff S \preceq S' \wedge S \neq S'$$

A solution S of a constraint system C is said to be minimal if there is no solution S' of C that satisfies $S' \prec S$.

Minimality for a solution means that it maps as few variables as possible to D. We consider a minimal solution rather than a least solution. This is because our constraint systems may have solutions that are incommensurable as is the case for $\{\alpha_1 = \alpha_2\}$. Minimal solutions may map α_1 and α_2 to S, $[\mathtt{S}, \mathtt{S}]$ et cetera. The minimal solution we find in Theorem 7 maps α_1 and α_2 to S.

2.3 Logical Conditions

If we model truth by D and falsehood by any other binding-time type we see that it is possible to express logical conditions in the constraint systems. These logical conditions can use both conjunctions (modelled by having more than one component on the left-hand side of dependency constraints) and disjunctions (modelled by multiple dependency constraints with the same right-hand side). Negation cannot be modelled (a corollary of Theorem 7) so the logic is in no sense complete.

Example 3 Below the constraints on the left expresses the formulas on the right.

$$\begin{array}{cc} (\beta_1, \beta_2) \triangleright \beta_3 & \beta_1 \wedge \beta_2 \Rightarrow \beta_3 \\ \beta_1 \triangleright \beta_3 \quad \beta_2 \triangleright \beta_3 & \beta_1 \vee \beta_2 \Rightarrow \beta_3 \end{array}$$

□

In the following section we show how such logical conditions can be used.

3 Binding-Time Analysis for Core SML

We have used the constraint systems in a binding-time analysis for programs written in Standard ML (without modules) [BW93]. In this section we describe, by way of example, the main ideas of how the constraints are to be used in such a binding-time analysis. In particular we argue that the small and fixed set of formal constraints we have introduced is expressive enough to make a non-trivial binding-time analysis of a full-scale, realistic, statically typed, higher-order language like Standard ML. The reader is referred to [BW93] for details on exactly which constraints are generated for Standard ML.

The reader should notice that our constraint systems are slightly more general than those used by Henglein [Hen91]. The enhancements can be seen in two places: 1) we allow multiple elements on the left-hand side in dependency constraints (allowing us to express conjunction, c.f. Section 2.3), and 2) structure constraints are not limited to two components. Despite this, our method is simpler than Henglein's in the sense we only have half as many rewriting rules. Furthermore, our sets of constraint types and rewriting rules are *fixed* whereas the introduction of, e.g., tuples in Henglein's language would require a special constraint and extra rewriting rules.

In the case of Standard ML [MTH90,MT91] as treated in [BW93], three kinds of structured values are considered: functions, tuples, and constructed values. In all three cases structure binding-time types are used and well-typedness of the source program implies that *binding-time variables associated with different kinds of structured values are never mixed.* This includes the situation of tuples having different lengths or constructed values from different Standard ML types. To emphasize: from a well-typed Standard ML program a well-typed constraint system in the sense of Definition 4 is generated [BW93].

In the sequel we describe how the constraints can be used to express binding-time requirements of typical constructs in Standard ML.

3.1 Function Application

Consider an application expression $e \equiv e'e''$ of one expression e' to another expression e''. In our setting, functional values such as e' either have binding-time type $[\tau_1, \tau_2]$ (which is our notation for Henglein's $\tau_1 \rightarrow \tau_2$) or type D. The structured binding-time type is used if the function can be applied during specialization whereas D is used if the function is unknown during specialization and thus cannot be applied. If the binding time of the functional expression e' is D then the argument expression must be residualized (as the application cannot be performed during specialization) and, moreover, the binding time of the whole expression must also be dynamic. On the other hand, if the application can be performed during specialization then the binding-time types of e'' and e must agree with the binding-time types of τ_1 and τ_2 respectively. These requirements can be specified by the constraint $[\beta'', \beta] \leq \beta'$ where the binding-time variable β holds the binding-time type of the application expression e, β' the binding-time type of the functional expression e', and β'' the binding-time type of the argument expression e''. For every application expression such a constraint is generated.

3.2 Tuples and Records

Consider the tuple[2] expression $(e_1, \ldots, e_n)$ and assume that e_i has binding-time type τ_i. We will deal with partially static tuples to the extent that we will allow the tuple to have binding-time value $[\tau_1, \ldots, \tau_n]$ no matter what the τ_i's are, or to have binding-time type D if and if only $\tau_i = \mathtt{D}$ for all i. In words, we will only build code for a tuple when all components are code. These requirements are specified by the constraint $[\beta_1, \ldots, \beta_n] \leq \beta$ where β_i is a binding-time variable holding the binding-time type for e_i and β holds the binding-time type for the entire expression.

[2] SML records can be treated as tuples by sorting the record fields by labels.

3.3 User-Defined Algebraic Data Types

Now consider the case of Standard ML user-defined algebraic data types. We will allow constructors of a data type to be either known during specialization or not; the arguments of constructors may have any binding-time type. Thus we allow partially static structures where, for instance, the spine of a list may be known and the elements may be dynamic.[3] As an example we consider the following Standard ML data type of binary trees.

```
datatype bt  =  L of int  |  N of int * bt * bt
```

For this data type we use six binding-time variables: $\beta_{\mathtt{bt}}$ is used for the whole data type; it is D if the constructors are not all known during specialization; $\beta_{\mathtt{L}}$ and $\beta_{\mathtt{N}}$ are used for the L and N constructors and express the binding-time type of the arguments of the constructors, and β_1, β_2, β_3 are used for the binding-times of the components of the argument of the N constructor. The following constraints are used to express the requirements between the binding-time variables:

$$[\beta_{\mathtt{L}}, \beta_{\mathtt{N}}] \leq \beta_{\mathtt{bt}} \quad [\beta_1, \beta_2, \beta_3] \leq \beta_{\mathtt{N}} \quad \beta_2 = \beta_{\mathtt{bt}} \quad \beta_3 = \beta_{\mathtt{bt}}$$

Notice how cycles in the constraint system are used to express recursiveness of data types. The binding-time types of the data type depends, of course, on how the constructors are used in the analyzed program. Constructors occur in expressions and patterns in Standard ML. For every occurrence of the constructor N as an expression e we generate the following two constraints

$$[\beta_{\mathtt{N}}, \beta_{\mathtt{bt}}] \leq \beta_e \quad \beta_{\mathtt{bt}} \triangleright \beta_e$$

Intuitively, the first constraint expresses that the constructor N is in essence a function, i.e., the expression N takes something of binding-time type $\beta_{\mathtt{N}}$ as argument and returns something of binding-time type $\beta_{\mathtt{bt}}$. The second constraint expresses that if the constructors of the data type are not known during specialization, then the constructor application cannot be performed during specialization and thus the functional expression e must be residualized. For every occurrence of the constructor N in a pattern[4] Nx we generate the following constraints

$$\beta_{\mathtt{bt}} = \beta_p \quad \beta_{\mathtt{N}} = \beta_x$$

where β_p is used to hold the binding time of the pattern (corresponds to the binding time of the expression the pattern will be matched against) and β_x holds the binding-time of the variable x.

We have now seen how we can use the *fixed*, small set of formal constraints to express binding-time properties of *user-defined* data types in well-typed programs. In fact, the constraints can also be used to express other more advanced aspects of binding-time analysis as we shall see now.

[3] In [BW93] we have not allowed different binding times for different occurrences of a constructor as in [BJ93]. This is not because the binding times cannot be expressed using our constraint systems but because of the partial evaluation is performed using a handwritten cogen. See [BW93] for details.

[4] Recall that N as a pattern must always be "fully applied."

3.4 Pattern Matching

3.4.1 Choice between Branches

Pattern matching in Standard ML is combined with lambda abstraction [MTH90]. Thus, given a lambda abstraction with several branches it is not trivial statically to analyze whether one during specialization can determine which of the branches to choose. This problem is part of the analysis of where to insert memoization points in [BW93] — if one cannot choose between the branches a memoization point is to be inserted to avoid infinite unfolding. We now consider this problem.

For each lambda abstraction we will use a binding-time variable, β, as a flag describing whether we can choose a single branch during specialization. If β becomes D it is supposed to mean that we cannot choose between the branches. Let us assume that the branches are always exhaustive; this can always be ensured by preprocessing. If in one of these branches there is a constant and the corresponding binding-time type is D, then we cannot determine which branch to choose and the flag β must be set to D. Consider the following example where we have three branches:

```
fn (1, x)      => ...
 | (2, (1,3)) => ...
 | _           => ...
```

Assume that the binding time for the patterns is [S,[S,D]]. Then we cannot determine which branch to choose as the binding time corresponding to the constant 3 is D.

The idea will is to decompose the binding time for the patterns according to the patterns and generate $(\cdot) \triangleright \beta$ constraints for all constants. As it can be seen from the example, it is not obvious how to decompose the binding time for the patterns — the second pattern would intuitively yield a more fine-grained decomposition than the first. The crucial insight is, however, that it is not needed to find *one* most fine-grained decomposition. We can instead consider each pattern in turn, decomposing the binding time each time! It is not only for constants that $(\cdot) \triangleright \beta$ constraints should be generated. If the pattern is a constructed pattern then such a constraint should also be generated with the binding-time variable for the constructed pattern substituted for the dot; because if we do not know the constructor, we cannot determine the branch (constructors correspond here to ordinary constants).

Assume that β_p holds the binding time of the pattern in our example under consideration. Then the following constraints will be generated:

$$[\beta_1, \beta_2] \leq \beta_p \quad \beta_1 \triangleright \beta$$

$$[\beta_3, \beta_4] \leq \beta_p \quad \beta_3 \triangleright \beta$$

$$[\beta_5, \beta_6] \leq \beta_4 \quad \beta_5 \triangleright \beta \quad \beta_6 \triangleright \beta$$

Notice how $\leq$ constraints are used to decompose the binding time of the pattern — this is safe due to well-typedness. The binding time for the whole lambda abstraction will be $[\beta_p, \beta_e]$ for some β_e. If one compares with the constraints generated for a function application expression (see above), one sees that if the lambda abstraction is applied to something with binding-time type [S,[S,D]], then the constraints generated do express that we cannot choose between the branches. In this case, β will be D after solving the constraints.

3.4.2 Nested Pattern Matching

We have now seen how we can generate constraints expressing whether we can choose between several branches. We now consider the problem of statically analyzing which parts of a nested pattern matching can be performed during specialization and which parts must be residualized. To the best of our knowledge, no other binding-time analysis have dealt with complicated nested pattern matching — other binding-time analyses have only dealt with non-nested pattern matching, e.g. [Lau91]. Of course, nested pattern matching can be compiled into non-nested pattern matching, but we conjecture that valuable information may be lost in that process. As an example, consider the following lambda abstraction.

```
fn (1, 1, u) => ...
 | (1, y, z) => ...
 | _         => ...
```

In case we *can* choose between the branches during specialization, i.e., if the first two components of the patterns are both static, then the whole pattern matching can be performed during specialization, no matter what the binding time of the third component is.

The real problem occurs when we *cannot* choose between the branches during specialization. Thus assume that the first component has binding-time type D, the second component has binding-time type S, and that the third component has binding-time type D. Then we cannot choose between the branches and

1. we may perform the matching on the second component during specialization (since this component is static), and
2. no matching, beta-reduction, should be performed on the third component (since a memorization point is to be inserted as mentioned above).

In other words, the pattern matching of a component should be residualized only if we cannot choose between the branches *and* the pattern component is dynamic. The generalized dependency constraint $(\beta_1, \beta_2) \triangleright \beta$ is used to express this conjunction. Note that it is not expressible in Henglein's system [Hen91]. For our example under consideration, the following constraints are generated, assuming that β is used as a flag saying whether we can choose between the branches (c.f. above), and β_1', β_2', β_3' are used for each component respectively to say whether the matching should be residualized, and β_p is used to hold the binding time for the whole pattern.

$$[\beta_1, \beta_2, \beta_3] \leq \beta_p \quad [\beta, \beta_1] \triangleright \beta_1' \quad [\beta, \beta_2] \triangleright \beta_2' \quad [\beta, \beta_3] \triangleright \beta_3'$$

The issue of how the binding-time information is used to actually stage the pattern matching in partial evaluation is beyond the scope of this paper; the interested reader is referred to [BW93].

4 Normalization

This section describes the process of normalizing a set of constraints. The purpose is to reduce the set to a form that can be solved essentially by "treating the structure and lift constraints as equalities." For this to succeed, (equality-conflicting constraints must be rewritten or eliminated.

The normalization process also eliminates all non-trivial equality constraints by substitution. For example $\alpha = \mathtt{D}$ is eliminated by substituting D for α throughout the constraint system.

All reductions that we will make on constraint systems preserve solutions. This will be proved in Theorem 6.

Figure 2 outlines how normalization works. The initial constraint system C_0 undergoes a finite number of rewritings spawning off a substitution for each step. The final constraint system C_n is in normal form and can easily be solved in a best-possible way. ("Best-possible" will be defined later.) Finally, the solution to C_n is combined with all the substitutions from the rewritings to form a best-possible solution to C_0.

4.1 Variable Equivalence

The constraint solving algorithm presented in [Hen91] handles constraint systems similar to those presented here except that the only structures considered there are functions. Our algorithm solves some problems with the algorithm in [Hen91]. Using our notation, a constraint system like

$$\{[\beta_1, \beta_2] \leq \beta_3,\ [\mathtt{D}, \mathtt{D}] \leq \beta_4,\ \beta_5 \rightsquigarrow \beta_3,\ \beta_5 \rightsquigarrow \beta_4,\ \beta_1 \triangleright \beta_3\}$$

corresponds to the following in Henglein's notation

$$\{\beta_1 \to \beta_2 \leq_f \beta_3, \mathtt{D} \to \mathtt{D} \leq_f \beta_4, \beta_5 \leq_b \beta_3, \beta_5 \leq_b \beta_4, \beta_1 \triangleright \beta_3\}$$

which is a normal form constraint system in the sense defined in [Hen91]; none of the rewriting rules apply. The first four constraints in the system can be solved by equality but contrary to the claim in [Hen91, Proof 4] some variables (β_1,β_2) are mapped to D. This invalidates the claim that dependency constraints are trivially satisfied since if a variable β on the left hand side of a dependency constraint $\beta \triangleright \beta'$ is mapped to D, then the dependency constraint is only satisfied if β' is also mapped to D. The same problem can be found in [JGS93, Section 8.7], [And92, Chapter 5.2], and [And93, Section 5.4]. A binding-time analysis based on an algorithm with this problem can yield incorrect results leading to a crash of the partial evaluator.

The problem is that the rewritings performed on the constraint systems in [Hen91] only propagate the information that β_3 occurs on the right-hand side of a $\leq_f$-constraint through lift constraints *from left to right,* so the analysis does not discover that β_3 and β_4 are related. Unfortunately a purely local rewriting rule propagating information the other way through lift constraints would not be solution preserving.

To overcome the problem we[5] introduce the notion of variable equivalence, which can be conceived of as weakened equality. The concept of equivalence is global in the sense that it takes the entire constraint system into account.

Definition 7 (Variable equivalence) *For a constraint system C we define $\approx$ to be the smallest equivalence relation on the set of binding-time variables and* D *satisfying*

$$\beta_1 \approx \beta_2 \qquad \textit{if} \qquad \beta_1 = \beta_2 \in C \textit{ or } \beta_1 \rightsquigarrow \beta_2 \in C$$

[5] Henglein solved the problem independently using a similar method though his solution has remained unpublished.

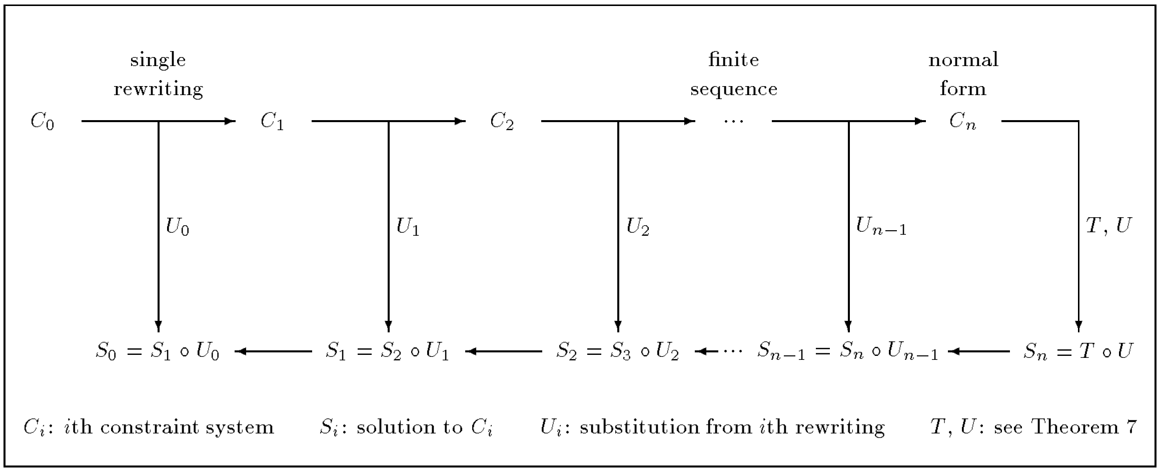

Figure 2: Solving a constraint system by normalization.

Informally, this means that two binding-time variables are equivalent if they are connected by a chain of lift and equality constraints. In the example above β_3 and β_4 are equivalent. The symmetry of the equivalence relation ensures that information about structure can be propagated not only left-to-right but also right-to-left through lift constraints. Considering also equality constraints makes Theorem 1 (below) hold, which in turn is important for the efficient implementation.

The introduction of variable equivalence leads to an implementation that is rather different from that presented in [Hen91] and [JGS93]. The binding-time analysis in [BJ93] uses a separate analysis called a "flow analysis" to capture the equivalent of variable equivalence.

4.2 Normal Forms

Definition 8 (Normal form) *A normal form constraint system is a constraint system where only the following types of constraints occur*

$$\mathtt{D} = \mathtt{D}, \quad \alpha \rightsquigarrow \beta, \quad (\alpha_1, \ldots, \alpha_m) \triangleright \beta, \quad [\beta_1, \ldots, \beta_n] \leq \alpha$$

where $m \geq 1$, $n \geq 0$, and for which the following conditions hold

1. *If $\alpha \approx \alpha'$ then there are no two different $\leq$ constraints with α on the right-hand side of the first and α' on the right-hand side of the second.*
2. *If $\alpha \approx \mathtt{D}$ then α does not occur on the right-hand side of any $\leq$ constraint.*

We shall later see that a normal form constraint system can be solved by equality yielding a minimal solution, and that every well-typed constraint system can be "normalized" in a solution-preserving way. Allowing $\mathtt{D} = \mathtt{D}$ in a normal form may look strange, but it saves a rewriting rule below.

4.3 Rewriting Rules

We will turn constraint systems into normal form by repeatedly applying some rewriting rule until no rule applies. The rewriting rules are shown in Figure 3.

Every rewriting rule takes one or two constraints and replaces it/them by a number of other constraints. The column "Condition" contains a precondition that must be satisfied before a rule can be used, and the column "Substitution" contains what we will call the resulting substitution. I is the identity substitution.

There are three classes of rewriting rules: *Substitution rules* (SU–n), *Simplification rules* (SI–n), and *Combination rules* (C–n).

Definition 9 (The normalization process) *The normalization process is the process of exhaustive use of the above rewriting rules in any order.*

The result of the normalization process is a normal form constraint system (Theorem 4).

4.4 Properties

We are now ready to prove a number of properties of constraint systems. Some of these are almost trivial and others are not, but they are all needed to understand why the algorithm we will eventually present works.

Theorem 1 (Variable equivalence is preserved) *Let C be a well-typed constraint system, let C' be the resulting system after use of one of the rewriting rules, and let S be the resulting substitution. If $\beta_1 \approx \beta_2$ with respect to C, then $S(\beta_1) \approx S(\beta_2)$ with respect to C'.*

Proof Only the substitution rules and rule SI–3 remove any equality or lift constraint from C. In the substitution case the equivalence is moved into the substitution and in rule SI–3 one equivalence-causing constraint is replaced by another. In all other cases the theorem is trivially satisfied since the new relation is larger or equal. □

That we do not preserve equivalence by removing too much information by making too many variables equivalent in the rewriting process is a corollary of Theorem 6. Preservation of variable equivalence is important because it means that we need not recalculate the equivalence relation from scratch

#	Constraint(s)	Condition	Replacement	Substitution
SU–1	$\alpha = \mathtt{D}$ or $\mathtt{D} = \alpha$	-	-	$\{\alpha \mapsto \mathtt{D}\}$
SU–2	$\alpha_1 = \alpha_2$	-	-	$\{\alpha_2 \mapsto \alpha_1\}$
SI–1	$() \triangleright \beta$	-	$\beta = \mathtt{D}$	I
SI–2	$(\beta_1, \ldots, \beta_{i-1}, \mathtt{D}, \beta_{i+1}, \ldots, \beta_n) \triangleright \beta$	-	$(\beta_1, \ldots, \beta_{i-1}, \beta_{i+1}, \ldots, \beta_n) \triangleright \beta$	I
SI–3	$\mathtt{D} \rightsquigarrow \beta$	-	$\beta = \mathtt{D}$	I
C–1	$[\beta_1, \ldots, \beta_n] \leq \alpha$, $[\beta_1', \ldots, \beta_n'] \leq \alpha'$	$\alpha \approx \alpha' \not\approx \mathtt{D}$	$[\beta_1, \ldots, \beta_n] \leq \alpha$, $\beta_i = \beta_i'$, $\alpha = \alpha'$	I
C–2	$[\beta_1, \ldots, \beta_n] \leq \beta$	$\beta \approx \mathtt{D}$	$\beta_i = \mathtt{D}$, $\beta = \mathtt{D}$	I

Figure 3: Rewriting Rules

after each rewriting. Instead we can make do with taking unions of equivalence classes.

Theorem 2 (Well-typedness is preserved) *For every well-typed constraint system, C, any of the rewriting rules yields a well-typed constraint system, C'.*

Proof We prove this rule by rule:

For rule SU–1 we have a simpler unification problem for $R'(C')$ since different occurrences of β have been changed to different fresh variables. For SU–2 we have $\alpha_1 = \alpha_2 \in R'(C)$. This guarantees the existence of a $U(C')$.

Since all occurrences of D's are replaced by different variables at unification time it is easy to see that all simplification rules preserve well-typedness.

For combination rule C–1 we have that $U(\alpha) = U(\alpha')$ since $\alpha \approx \alpha'$ and $\alpha \not\approx \mathtt{D}$. We then have $U(\beta_i) = U(\beta_i')$ so U is also a unifier for C'. For rule C–2 the unification problem in C' is simpler since the ith D matches anything that β_i matched. □

Again it follows from Theorem 6 that we do not obtain preservation of well-typedness by removing too much information. That well-typedness is preserved means that the theorems below which depend on well-typedness can be used not only on the original constraint system but also on rewritten constraint systems.

Theorem 3 (Termination) *For any well-typed constraint system the normalization process terminates.*

Proof The follows from the fact that all rules lexicographically decreases the tuple (a, b), where a is the total number of symbols (which are "(", "]", variables, "D", ...) on left-hand sides of non-equality constraints, and b is the number of constraints. □

Theorem 4 (Soundness) *Let C be a well-typed constraint system, and let C' be the constraint system obtained by normalization. C' is a normal form constraint system.*

Proof First we prove that only the allowed types of constraints occur in C'. Concerning lift constraints: the claim is that D does not occur on the left-hand side; this holds by rule SI–3. Concerning inequality constraints: the claim is that D does not occur on the right-hand side; this holds by rule C–2. Concerning dependency constraints: the claim is that D does not occur on the left-hand side and that the left-hand side is not empty; this holds by rules SI–1 and SI–2. Concerning equality constraint: there are none with variables by rules SU–1 and SU–2.

Second we prove that the claims 1–2 in the definition of normal form holds in C'. (1) holds by well-typedness (only compatible entities meet) and by rule C–1. (2) holds by rule C–2. □

Theorems 3 and 4 tell us that the the specified normalization process is indeed an algorithm: it does what it is supposed to in a finite number of steps.

Lemma 5 *Let C be a well-typed constraint system, let C' be the result of applying one of the rewriting rules on C, and let U be the resulting substitution. Then*

$$\begin{array}{lcl} S \text{ is a solution for } C & \Rightarrow & \exists S' : S = S' \circ U \quad \text{and} \\ & & S' \text{ is a solution for } C' \\ S' \text{ is a solution for } C' & \Rightarrow & S \stackrel{\text{def}}{=} S' \circ U \text{ is a solution for } C. \end{array}$$

Proof Observe that the sets of variables used are unchanged except for the substitution rules and the requirement for substitutions to map variables not occurring in a constraint system to themselves is therefore preserved in those cases. In the same cases we further have that U is the identity substitution so that the above claims reduce to showing that solutions coincide.

We prove the two claims by examining the rewriting rules in turn.

- Substitution rule 1: The first claim is satisfied with $S' = V^{-1} \circ S \circ V$, where V maps α to a variable not used elsewhere and passes everything else. In words S' is S with the binding on α removed. S' maps α onto itself and no other variables disappear or arrive. The second claim is satisfied since S must substitute D for α.

- Substitution rule 2: The first claim is satisfied as for the preceding rule, but with α_2 instead of α. The second claim is satisfied since S must map α_1 and α_2 to the same thing.

- Simplification rules 1–3: these preserve solutions both ways as they are direct application of the semantics of constraints.

- Combination rule 1: The first claim holds since if $S(\alpha) = \mathtt{D}$ then also $S(\alpha') = \mathtt{D}$ and S clearly solves C'. Otherwise $S(\alpha) = [\tau_1, \ldots, \tau_n] = S(\alpha')$ since structured values cannot be lifted and because $\alpha \approx \alpha'$. The second claim holds obviously.

- Combination rule 2: The first claim holds since $\beta \approx \mathtt{D}$ and because structured values cannot be lifted so we have $S(\beta) = \mathtt{D}$ and thus $S(\beta_i) = \mathtt{D}$. The second claim holds obviously.

$\square$

Theorem 6 (Solutions are preserved) *Let C be a well-typed constraint system, let C' be the result of applying one of the rewriting rules on C. Then there is a one-to-one correspondence between solutions of C and solutions of C'.*

Proof For non-substitution rules this was proved in the previous lemma. For the substitution rules we have the correspondence $S' = V^{-1} \circ S \circ V$ and $S = V \circ S' \circ V^{-1}$ using notation from the lemma.[6] $\square$

Theorem 7 *Every normal form constraint system C has a minimal solution.*

Proof We will prove this constructively by exhibiting a minimal solution. Let C' be the subset of C consisting of constraints having the form $[\beta_1, \ldots, \beta_n] \leq \alpha$ or $\alpha \rightsquigarrow \alpha'$ (i.e., leave out dependency constraints and lift constraints with $\mathtt{D}$ on the right-hand side). Solve all constraints in C' by equality. This is possible due to well-typedness and the definition of normal form. Call the resulting substitution U and let T be the substitution that maps all variables in $U(C)$ to the constant value $\mathtt{S}$. We claim that $V = T \circ U$ solves the entire system C and that V is a minimal solution.

Notice first that neither U nor T maps any variable to $\mathtt{D}$. Every dependency constraint is therefore satisfied by V. For every $\alpha \rightsquigarrow \mathtt{D}$ in C we have by the second condition of normal form that $V(\alpha) = \mathtt{S}$ so the constraint is satisfied. All other remaining constraints are equality-solved by U and therefore satisfied by V. As variables not in C are mapped to themselves by both U and T we conclude that V is indeed a solution to C.

Now consider some other solution $V' \neq V$. If V' solves all the above mentioned constraints by equality it must map at least one of the remaining variables to something different from $\mathtt{S}$, and the two solutions are then either incommensurable or we have $V \preceq V'$. If on the other hand V' does not solve all the above mentioned constraints by equality then it maps some α to $\mathtt{D}$ where V does not, and we have $V' \not\preceq V$. We finally conclude that V is minimal. $\square$

Theorem 8 (Minimality is preserved) *Let C be a well-typed constraint system, let C' be the result of applying one of the rewriting rules on C with the resulting substitution U. If S' is a minimal solution to C' then $S \equiv S' \circ U$ is a minimal solution to C.*

Proof Assume that T is a solution to C and that $T \prec S$. It follows from Lemma 5 that S really is a solution and that we can choose a solution T' to C' such that $T = T' \circ U$.

For an α where $S(\alpha) \neq T(\alpha)$ we now have $T(\alpha) \neq \mathtt{D}$ and $S(\alpha) = \mathtt{D}$ from which it follows that $T'(\alpha) \neq \mathtt{D}$, $U(\alpha) \neq \mathtt{D}$, and $S'(\alpha) = \mathtt{D}$. Thus $T'(\alpha) \preceq S'(\alpha)$.

For an α where $S(\alpha) = T(\alpha)$ we have by choice of T' that $S'(U(\alpha)) = T'(U(\alpha))$. If $U(\alpha) = \alpha$ then this means that $T'(\alpha) = S'(\alpha)$. Otherwise, by the definition of a solution, we directly have $T'(\alpha) = S'(\alpha)$.

Since $S' \neq T'$ we have thus proved $T' \prec S'$ which is a contradiction to the minimallity of S'. $\square$

All in all, the previous theorems prove that every well-typed constraint system C has a minimal solution: since C is well-typed it can be normalized yielding a normal form constraint system C'. A minimal solution exists for C' and this gives rise to a minimal solution for C.

5 Efficient Implementation

A naïve implementation of the normalization process described above would be very slow since it would require searching for some constraints every time one of the combination rules should be used. This section presents an efficient implementation that runs in almost linear time.

The key to efficiency is to consider the constraints one by one and to record enough information about already-seen constraints to make it possible to decide which rules can be applied when a new constraint is considered.

Actual code In Appendix A the actual ML-code needed for normalization is shown. For readability we have used standard list notation for the dependency lists although our complexity analysis requires that the append operation on lists is $O(1)$ in time complexity. To ensure this a special data type with direct access to both ends of the list should be used.

Phase 1 — initialization

- Apply simplification rules SI–1 and SI–2 exhaustively. (As nobody in his/her right mind would generate constraints requiring such rewriting, typically no action is required.)
- For every $\beta_1 \rightsquigarrow \beta_2$ constraint add a $\beta_1 \triangleright \beta_2$ constraint. This handles rule SI–3 in a simple way.
- Using union-find techniques an evolving equivalence relation can be constructed on the set of binding-time variables and $\mathtt{D}$. This relation is used to trace variable substitutions (*not* variable equivalence!) The class in which β occurs is written $\overline{\beta}$ and we associate a set of dependencies, $\mathrm{dps}(\overline{\beta})$, with every class.
- The working list is a list of constraints that we have not dealt with yet. We initialize the working list to all non-dependency constraints. Note that a copy of this list must be kept for phase 3.
- For each constraint $(\beta_1, \ldots, \beta_k) \triangleright \beta$ create a pair (k, β). Insert a reference to this pair in all $\mathrm{dps}(\overline{\beta_i})$. Since k may be decremented later it is important that all the β_i's reference the same pair in memory. The meaning of such a pair is that k is decremented every time some β_i turns $\mathtt{D}$ and if k eventually gets decremented to zero then the constraint $\beta = \mathtt{D}$ will be inserted into the working list.

[6] For the formally inclined reader we note that the definition of a solution basically is the same in [Hen91] as in this paper; unused variables are required to be mapped to themselves. Strictly speaking this property together with Henglein's rewriting rule 3a invalidates the claim in [Hen91, Theorem 2] and its proof. To see this, one just has to consider the constraint system $\{\beta \triangleright \mathtt{D}\}$ which will be normalized to $\{\,\}$. The former system has a countable infinite number of solutions (because β can be mapped to any binding-time type), the latter has exactly one solution. The same problem can be found in [JGS93, Section 8.7], [And92, Chapter 5], and [And93, Figure 7].

- Construct a term graph for all members of the working list using nodes with labels S (arity 0), D (arity 0), = (arity 2), $\rightsquigarrow$ (arity 2), [ ] (some finite arity), and $\leq$ (arity 2). The binding-time variables are the other leaves of the graph.

- On the set of classes, $\overline{\beta}$, create another evolving equivalence relation for handling variable equivalence. Call the resulting classes $\overline{\overline{\beta}}$. With each class $\overline{\overline{\beta}}$ we associate a field, memory($\overline{\overline{\beta}}$), which is to hold at most one unprocessed inequality constraint with β on the right-hand side. Notice by inspection of the second combination rule that it does not matter which $\beta \in \overline{\beta}$ or $\overline{\beta} \in \overline{\overline{\beta}}$ is used. The memory field of $\overline{\mathtt{D}}$ will always be kept empty.

Phase 2 — main loop While the worklist is not empty remove a constraint, c, from it. If there is a lift or an equality constraint in the list, c must be such a constraint. (In other words we handle structure constraints only when no other constraints are available.)

If c is an equality or a lift constraint we now recognize the two sides to be variable equivalent and therefore unify the two $\overline{\overline{\beta}}$ classes if they are not already equivalent. This may lead to application of one of the combination rules since two conflicting memory fields may collide or one of c's sides may be variable equivalent to D.

If c is an equality constraint with different sides we also unify the two $\overline{\beta}$ classes thus recording the substitution. If both sides are variables then the dps-sets are concatenated. If on the other hand one of the sides is D we update the dps-sets as described above.

If c is a structural constraint with right-hand side β we have by the choice of c that the variable equivalence relation is up to date. If there is no remembered constraint in memory($\overline{\overline{\beta}}$) simply record c there. Otherwise we apply one of the combination rules.

Phase 3 — postprocessing

- For each $(\ldots) \leq \beta$ or $\cdot \rightsquigarrow \beta$ in the saved copy of the working list solve by equality if $\overline{\beta} \neq \overline{\mathtt{D}}$. This is done efficiently by changing β to a link to the left-hand side when the sides are different.

- Unify any remaining free variable with S.

After phase 3 all nodes representing binding-time variables have been unified with a node representing their binding-time types — the process is complete.

Complexity The algorithm described above is very efficient and our implementation confirms this. All in all the algorithm can be made to run in time $O(n \cdot \alpha(n, n))$, where n is the total number of components in the constraint system, and α is an inverse of Ackermann's function.

We now present the arguments for this time-complexity. We measure the size of the problem as the number of components in the constraint system. The number of components is defined as the number of constraints plus the number of occurrences of variables and constants.

We assume that removal (insertion) of constraints from (into) the working list, taking unions of dps-sets, and similar simple operations are performed in constant time. This can be achieved by suitable data structures. We also assume efficient implementation of the find primitives as in [Tar83]

Phase 1 clearly runs in linear time with respect to the number of components. It may double the number of constraints.

The total number of non-equality constraints processed in phase 2 is bounded by the number of constraints. The total number of equality constraints processed is bounded by the number of constraints plus the number of components on the left-hand sides of constraints, i.e., by twice the number of components.

The total number of "union" operations performed is bounded by the number of variables for each of the two relations, and thus by twice the number of constraints in total. The number of "find" operations is bounded by three times the number of constraints ever processed, and thus by twelve times the number of components.

Apart from the union/find operations already accounted for, phase 3 has linear time complexity with respect to the number of components.

In total we have a linear number of union-find operations and a linear number of primitive operations. Thus the stated time complexity.

6 Related Work

As already noted the algorithm described here is closely related to the one by Henglein in [Hen91]. The ideas in that paper have proven very useful as it can be seen from the fact that they play an important rôle in [And92,And93,BJ93, BW93]. These works also show how to use constraint systems for doing binding-time analysis.

Launchbury [Lau91] uses projections and abstract interpretation to perform binding-time analysis on a small strongly typed first-order language. His methods were later used in [dN93] for a similar language. The methods are well-suited for both polymorphic programs and for polyvariant specialization, but it seems to be difficult to use the methods for higher-order programs.

The constraint systems used in [BJ93] are significantly different from those in this article because, for historical reasons, a completely different set of binding-time types are used. Also, as noted in Section 4.1, a separate "flow analysis" is used instead of variable equivalence.

7 Conclusion and Future Work

We have presented and proved the correctness of a general framework for doing binding-time analysis of statically typed languages. We have also shown that the method presented can be used for languages with a variety of different data types, such a Standard ML's functions, records, user-defined algebraic data types, and pattern matching.

We believe that the fixed set of constraint types presented here is strong enough to express most needs for realistic binding-time analyses of statically typed languages. This belief is supported by the fact they are strong enough for (monomorphic) Standard ML with partially static data structures and nested pattern matching.

Future research should be conducted in two directions: the application of constraint based binding-time analysis to polymorphically typed programs, and polyvariant binding-time analysis by constraint solving.

Acknowledgements

Thanks to the DIKU Topps group for valuable discussions and to Robert Glück and the anonymous referees for comments on an earlier version of this paper.

References

[And92] Lars Ole Andersen. C program specialization. Technical Report 92/14, DIKU, Department of Computer Science, University of Copenhagen, May 1992.

[And93] Lars Ole Andersen. Binding time analysis and the taming of C pointers. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 47–58. ACM Press, June 1993.

[BJ93] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming, special issue on partial evaluation*, 3, July 1993.

[BW93] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Technical Report 93/22, DIKU, October 1993.

[dN93] Anne de Niel. *Self-applicable Partial Evaluation of Polymorphically Typed Functional Languages.* PhD thesis, Katholieke Universiteit Leuven, January 1993.

[Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 448–472. Springer-Verlag, 1991.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Program Generation.* Prentice-Hall, 1993.

[Lau91] John Launchbury. *Projection Factorisations in Partial Evaluation.* Cambridge University Press, 1991.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[NN88] Hanne Riis Nielson and Flemming Nielson. Automatic binding time analysis for a typed λ-calculus. *Science of Computer Programming*, 10:139–176, 1988.

[PS94] Jens Palsberg and Michael I. Schwartzbach. Binding-time analysis: Abstract interpretation versus type inference. In *Proc. ICCL'94, Fith IEEE International Conference on Computer Languages, Toulouse, France*, May 1994. To Appear.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

A Normalization Code in Standard ML

The following Standard ML code can be used for solving constraint systems. The code assumes that phase 1 described earlier has been performed during constraint generation. The imperative aspects of Standard ML have been used a lot and we believe that this must be done for the complexity analysis to hold.

For readability we have used standard list notation for the dependency lists although our complexity analysis requires that the append operation on lists is $O(1)$ in time complexity.

The code for a couple of simple subroutines has been omitted for space reasons. The code has been replaced by an ellipsis and a comment describing what the code should do. The functions `find` and `findL` are standard, see [Tar83, Section 2.1].

```
datatype 'a Option = None | Some of 'a

datatype BType = (* Graph Nodes *)
  STATIC
| DYNAMIC of BLType ref
| CONSTR of BType ref list
| BTVAR of (int ref*Btype ref) list ref * BLType ref
| BTLINK of BType ref

and BLType = (* Equivalences on DYNAMIC/BTVAR *)
  BTLEQR of (BType ref*BType ref) Option ref
| BTLLINK of BLType ref

and Constraint = (* Constraints in worklist *)
  EQUAL of BType ref*BType ref
| LIFT of BType ref*BType ref
| LEQ of BType ref*BType ref

val Snode = ref STATIC
val DLnode = ref (BTLEQR (ref None))
val Dnode = ref (DYNAMIC DLnode)
val worklist = ref (nil : Constraint list)
val btvarlist = ref (nil : BType ref list)

(* Follow BTLINKs using transition compression: *)
fun find  n  = ...
(* Follow BTLLINKs using transition compression: *)
fun findL nl = ...
fun isD n   = (find n = Dnode)
fun isDL nl = (findL nl = DLnode)
fun isvar (BTVAR _) = true
  | isvar _ = false

fun geteqr (ref (DYNAMIC eqr)) = eqr
  | geteqr (ref (BTVAR (dps,eqr))) = findL eqr
  | geteqr _ = crash ()

fun getmem (ref (BTLEQR mem)) = mem
  | getmem _ = crash ()

fun apply f xs = (map f xs; ())
fun mk_eqconstraint c =
  worklist := EQUAL c :: (!worklist)
fun update_dps (k,n) =
  (k := !k - 1;
   if !k = 0 then mk_eqconstraint (n,Dnode) else ())

fun doC1 (l1,r1) (l2,r2) = (* Combination rule 1 *)
  case (!(find l1),!(find l2)) of
    (CONSTR c1,CONSTR c2) =>
      apply mk_eqconstraint (zip (r1::c1) (r2::c2))
  | _ => crash ()

fun doC2 (l,r) = (* Combination rule 2 *)
  case !(find l) of
    (CONSTR c) =>
      apply (fn n=>mk_eqconstraint (n,Dnode)) (r::c)
  | _ => crash ()

fun unionL n1 n2 = (* Make nodes var-equivalent *)
  let val (btl1,btl2) = (geteqr bt1,geteqr bt2)
  in if !btl1 = !btl2 then () else
    case (!btl1,!btl2) of
      (BTLEQR (ref None),BTLEQR (ref None)) =>
        if isDL(btl1) then btl2 := BTLLINK btl1
                      else btl1 := BTLLINK btl2
    | (BTLEQR (ref None),BTLEQR (ref (Some c))) =>
        if isDL btl1 then doC3 c; btl2 := BTLLINK btl1
                     else btl1 := BTLLINK btl2
    | (BTLEQR (ref (Some c)),BTLEQR (ref None)) =>
        if isDL btl2 then doC3 c; btl1 := BTLLINK btl2
                     else btl2 := BTLLINK btl1
    | (BTLEQR (ref (Some c1)),BTLEQR (ref(Some c2)))=>
        doC1 c1 c2; btl2 := BTLLINK btl1
    | _ => crash ()
  end

fun union n1 n2 = (* Make nodes `equal' *)
  if n1 = n2 then ()
  else (unionL n1 n2;
   case (!n1,!n2) of
     (BTVAR (dps1,_),BTVAR (dps2,_)) =>
       (dps2 := dps1 @ dps2; (* O(1) list append *)
        bt1 := BTLINK bt2)
   | (BTVAR (dps,_),DYNAMIC _) =>
       (apply update_dps (!dps); bt1 := BTLINK bt2)
   | (DYNAMIC _,BTVAR (dps,_)) =>
       (apply update_dps (!dps); bt2 := BTLINK bt1)
   | _ => crash ())

fun norm (EQUAL (l,r)) = union (find l,find r)
  | norm (LIFT (l,r)) = unionL (find l,find r)
  | norm (LEQ (l,r)) =
      let val c as (l,r) = (find l,find r)
          val eqr = geteqr r
          val mem = getmem eqr
      in case !mem of
           None => if isDL eqr then doC2 c
                   else mem := Some c
         | Some c' => doC1 c c'
      end

fun eq_solve (EQUAL _) = ()
  | eq_solve (LIFT (l,r)) =
      let val (l',r') = (find l,find r)
      in if isvar r' andalso l' <> r' then
            r' := BTLINK l else ()
      end
  | eq_solve (LEQ c) = eq_solve (LIFT c) (* ! *)

fun eq_S n = if isvar n then n := BTLINK Snode else ()

fun generate () = ... (* Generate constraints *)

(* Assumes phase 1 done during generation *)
fun normalize () =
  let val save = !worklist
  in worklist := radixsort (!worklist);
     (* Radixsort is O(n), LEQs are put last *)
     (* From now on LEQs are only chosen if
        there are no EQUALs/LIFTs *)
     while !worklist <> nil do
       let val (c :: rest) = !worklist
       in worklist := rest;
          norm c
       end
     apply eq_solve save;
     apply eq_S btvarlist
  end
```