Making Set-Constraint Program Analyses Scale

Manuel Fähndrich* Alexander Aiken*

EECS Department
University of California, Berkeley
Berkeley, CA 94720-1776
{manuel,aiken}@cs.berkeley.edu

September 30, 1996

1 Introduction

Constraint-based program analyses are appealing because elaborate analyses can be described with a concise and simple set of constraint generation rules. Constraint resolution algorithms have been developed for many kinds of constraints, conceptually allowing an implementation of a constraint-based program analysis to reuse large pieces of existing code. In practice, however, new analyses often involve re-implementing new, complex constraint solving frameworks, tuned for the particular analysis in question. This approach wastes development time and interferes with the desire to experiment quickly with a number of different analyses.

We believe that implementing an analysis should require writing only the code to generate the constraints, and that a well engineered-library can take care of constraint representation, resolution, and transformation. Writing such a library capable of handling small programs is not too difficult, but scaling to large programs is hard. Toward this goal, we are developing a scalable, expressive framework for solving a class of set constraints. Scalability is achieved through four techniques: *polymorphism*, *simplification*, *separation*, and *sparse representation* of constraints.

Our ultimate goal is to demonstrate constraint-based analysis on programs of at least 100,000 lines of code. Currently, we evaluate our design on an application inferring types and exceptions for Standard ML [MTH90] with subtyping. Our implementation analyzes SML programs in the lambda intermediate representation produced by the SML/NJ compiler [App92]. The largest program analyzed thus far is the parser generator `sml-yacc`, containing 6017 non-comment lines of source, which translate into 66120 abstract syntax tree nodes in the SML/NJ intermediate representation. To the best of our knowledge, this is currently the largest program analyzed by a set constraint implementation (including results reported in [AWL94, Hei94]). Full type and exception inference for `sml-yacc` currently takes less than 10 minutes on an HP9000/715 running at 64MHz equipped with 64MB of main memory. Even though the analysis time is still far from practical, we improve upon a similar analysis done previously by Yi [Yi94] by a factor of 50. His abstract interpretation for estimating uncaught exceptions runs about 617 minutes for `sml-yacc` on an SGI Challenger.

Our empirical results show that our system can almost certainly handle programs larger than `sml-yacc`; it just happens that `sml-yacc` is the largest example we currently have. Though our implementation scales nicely, it is currently much slower on medium-size programs than the system described in [Hei94].

The remainder of the paper is organized as follows: Section 2 gives a short overview of the set expressions and constraints used in our framework. Section 3 presents the techniques we find useful in building a scalable system. Section 4 discusses our implementation and preliminary empirical results. Throughout

*This material is based in part upon work supported by NSF Young Investigator Award No. CCR-9457812 and NSF Infrastructure Grant No. CDA-9401156. The content of the information does not necessarily reflect the position or the policy of the Government.

the paper we point out open theoretical issues where progress would, we believe, aid the engineering of implementations similar to ours.

2 Types and Constraints

In our framework, set expressions represent types, and we use the terms *set expression* and *type expression* interchangeably. We describe the type language and the meaning of types informally. A formal development of an ideal model for this type language can be found in [AW93].

The full type language consists of type variables, a least type $\bot$, a greatest type $\top$, constructed types $\mathrm{c}(\tau_1, \ldots, \tau_n)$ where c is a constructors of arity n drawn from an infinite set of constructors C, function types $\tau_1 \to \tau_2$, intersections, unions, and conditional types.

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \top \mid \bot \mid \mathrm{c}(\tau_1, \ldots, \tau_n) \mid \tau_1 \to \tau_2 \\ & \mid & \tau_1 \cap \tau_2 \mid \tau_1 \cup \tau_2 \mid \tau_1 \Rightarrow \tau_2 \end{array}$$

We use a meaning function $\mathcal{I}$ to give meaning to free variables and extend it in the obvious way to type expressions. There is a subtype relation $\subseteq$ on meanings which we omit for brevity. The relation $\tau_1 \subseteq \tau_2$ holds between expressions τ_1 and τ_2 if for all assignments $\mathcal{I}$ to variables, we have $\mathcal{I}(\tau_1) \subseteq \mathcal{I}(\tau_2)$.

Unlike "standard" set expressions, our expression language includes sets of functions $\tau_1 \to \tau_2$. For a given assignment $\mathcal{I}$, the meaning of this expression is

$$\{f \mid x \in \mathcal{I}(\tau_1) \Rightarrow f(x) \in \mathcal{I}(\tau_2)\}$$

in an appropriate domain. From the point of view of constraint resolution, the key property of function types is that they are anti-monotonic in the domain; that is $\tau_1 \subseteq \tau_2$ implies $\tau_2 \to \tau \subseteq \tau_1 \to \tau$. We say that a subexpression τ' of a type expression τ appears *monotonically* (resp. *anti-monotonically*) if τ' appears to the left of an even (resp. odd) number of $\to$'s within τ.

The other non-standard expression is conditional types $\tau_1 \Rightarrow \tau_2$ (formerly $\tau_2?\tau_1$ in [AWL94]). The meaning of a conditional type $\mathcal{I}(\tau_1 \Rightarrow \tau_2)$ is $\mathcal{I}(\tau_2)$ if $\mathcal{I}(\tau_1) \neq \bot$, and $\bot$ otherwise. (We use $\bot$ to denote both the syntactic least type and the semantic bottom element.) Conditional types are useful for expressing computations involving case analysis. For example, the predicate `null`, which tests whether a list is empty or not, can be assigned a very accurate type using conditions. In the following type, parentheses are used to show association:

$$\texttt{null} : \alpha \to (\alpha \cap \texttt{nil} \Rightarrow \texttt{true}) \cup (\alpha \cap \texttt{cons}(\top, \top) \Rightarrow \texttt{false})$$

To see that this type makes sense, note that the instance $\alpha = \texttt{nil}$ simplifies to $\texttt{nil} \to \texttt{true}$ (using the identity $\texttt{nil} \cap \texttt{cons}(\top, \top) = \bot$). An instance with $\alpha = \texttt{cons}(\beta, \gamma)$ yields $\texttt{cons}(\beta, \gamma) \to \texttt{false}$.

Solving constraints involving general intersection, union, and function types is still an open problem [AW93, Dam94]. Our system restricts the forms of constraints involving intersections on the left-hand side and unions on the right-hand side of constraints in order to solve them. For the purpose of this paper, it is sufficient to consider the *core type language* defined below, which avoids these irregularities. We still use the full type language in examples, however. The core type language distinguishes between "left" types τ^L and "right" types τ^R. Left types appear in monotonic positions and right types appear in anti-monotonic positions.

$$\begin{array}{rcl} \tau^L & ::= & \alpha \mid \bot \mid \top \mid \mathrm{c}(\tau_1^L, \ldots, \tau_n^L) \mid \tau_1^R \to \tau_2^L \mid \tau_1^L \cup \tau_2^L \\ \tau^R & ::= & \alpha \mid \bot \mid \top \mid \mathrm{c}(\tau_1^R, \ldots, \tau_n^R) \mid \tau_1^L \to \tau_2^R \mid \tau_1^R \cap \tau_2^R \end{array}$$

Constraints between types express containment of sets of values. We use κ to denote a set of constraints:

$$\kappa \quad ::= \{\tau_1^L \subseteq \tau_1^R, \ldots, \tau_n^L \subseteq \tau_n^R\}$$

A solution to a set of constraints $\{\tau_1^L \subseteq \tau_1^R, \ldots, \tau_n^L \subseteq \tau_n^R\}$ is a meaning function $\mathcal{I}$ s.t.

$$\mathcal{I}(\tau_i^L) \subseteq \mathcal{I}(\tau_i^R) \qquad \text{for } i = 1, \ldots, n$$

$$
\begin{aligned}
\kappa \cup \{\bot \subseteq \tau^R\} &\equiv \kappa && (1)\\
\kappa \cup \{\tau^L \subseteq \top\} &\equiv \kappa && (2)\\
\kappa \cup \{\alpha \subseteq \alpha\} &\equiv \kappa && (3)\\
\kappa \cup \{\mathrm{c}(\tau_1^L, \ldots, \tau_n^L) \subseteq \mathrm{c}(\tau_1^R, \ldots, \tau_n^R)\} &\equiv \kappa \cup \{\tau_i^L \subseteq \tau_i^R \mid 1 \leq i \leq n\} && (4)\\
\kappa \cup \{\tau_1^R \to \tau_1^L \subseteq \tau_2^L \to \tau_2^R\} &\equiv \kappa \cup \{\tau_1^L \subseteq \tau_2^R, \tau_2^L \subseteq \tau_1^R\} && (5)\\
\kappa \cup \{\tau_1^L \cup \tau_2^L \subseteq \tau^R\} &\equiv \kappa \cup \{\tau_1^L \subseteq \tau^R, \tau_2^L \subseteq \tau^R\} && (6)\\
\kappa \cup \{\tau^L \subseteq \tau_1^R \cap \tau_2^R\} &\equiv \kappa \cup \{\tau^L \subseteq \tau_1^R, \tau^L \subseteq \tau_2^R\} && (7)\\
\kappa \cup \{\tau^L \subseteq \alpha, \alpha \subseteq \tau^R\} &\equiv \kappa \cup \{\tau^L \subseteq \alpha, \alpha \subseteq \tau^R, \tau^L \subseteq \tau^R\} && (8)
\end{aligned}
$$

Figure 1: Constraint resolution for core types

Figure 1 shows the constraint resolution rules for the core types. See [AWL94] for the resolution rules of the full type language. In Figure 1, the relation $\kappa_1 \equiv \kappa_2$ means that the constraint systems κ_1 and κ_2 have the same solutions. The resolution rule for constructors is sound and complete for a domain of lazy constructors (i.e. $\mathcal{I}(\mathrm{c}(\ldots, \bot, \ldots)) \neq \bot$) and this rule is sound (but not complete) for a strict language like ML. The lazy interpretation has the advantage that the resolution time complexity is polynomial instead of exponential. We write $\overline{\kappa}$ for the *solved form* of κ, which is the set of constraints obtained by applying the resolution rules until closure.

Polymorphic constrained types (or simply polymorphic types) are written

$$\forall(\alpha_1, \ldots, \alpha_n).\tau \setminus \kappa$$

The meaning of a polymorphic type depends on the meaning of any free variables and is only defined if the constraints κ have a solution for at least one choice of values for the quantified variables $(\alpha_1, \ldots, \alpha_n)$.

$$\mathcal{I}(\forall(\alpha_1, \ldots, \alpha_n).\tau \setminus \kappa) = \bigcap_{\mathcal{I}'} \mathcal{I}'(\tau)$$

where $\mathcal{I}'$ ranges over solutions of κ, and $\mathcal{I}'(\alpha) = \mathcal{I}(\alpha)$ for all $\alpha \notin \{\alpha_1, \ldots, \alpha_n\}$.

Using polymorphic constrained types we can refine the type of `null` introduced above so that the type is undefined for arguments other than `nil` and `cons`.

$$\mathtt{null} : \forall(\alpha).\alpha \to (\alpha \cap \mathtt{nil} \Rightarrow \mathtt{true}) \cup (\alpha \cap \mathtt{cons}(\top, \top) \Rightarrow \mathtt{false}) \setminus \{\alpha \subseteq \mathtt{nil} \cup \mathtt{cons}(\top, \top)\}$$

3 Scalability

Our goal is to analyze very large programs (>100,000 lines) using constraint resolution. Designing a system to scale is not necessarily the same as designing a system to run fast. The primary engineering concern for a scalable system is space consumption. For example, a scalable system cannot assume that the entire program is available at one time, because it may not fit in machine memory. Thus the program must be analyzed in small pieces. This single observation leads to a radically different overall design than *whole program analyses* systems that analyze only complete programs (e.g., see [Hei94]).

To achieve a scalable system, we analyze parts of programs separately and combine the results later. We use four techniques to achieve this goal: polymorphism, simplification, separation, and sparse constraint representation.

Computing a polymorphic type for an expression allows a certain degree of abstraction of the properties of that expression with respect to its context. This abstraction from context makes the separate analysis and combination that we seek possible. Furthermore, polymorphism helps yield precise results without the need for repeatedly analyzing the same expression in different contexts.

Unfortunately, instances of polymorphic constrained types produce copies of the constraints associated with the type. Polymorphism by itself does not give us scalability, because the more polymorphism we use, the more constraints we have to carry around. The key to this problem is type and constraint *simplification*. By type and constraint simplification we mean replacing a polymorphic constrained type by an equivalent polymorphic type containing fewer variables. The number of variables is the main contributor to the size of types and constraints and consequently to running time, thus our focus is to eliminate as many variables as possible.

The need for simplification can be understood by drawing an analogy with implementations of the Hindley-Milner type system. Unification, which is used to solve equality constraints between types generated during Hindley-Milner type inference, automatically removes all redundant type variables and leaves no residual constraints. Constraint resolution systems based on inequalities do not have this property, which limits efficiency because of the accumulation of type variables generated during inference, unless steps are taken to eliminate redundant variables. Section 3.2 details the simplifications performed by our system.

Our third technique, *separation*, deals with the problem of maintaining and merging separate systems of constraints. Consider the problem of analyzing a subtree of a large abstract syntax tree (AST). The analysis of each subtree of size n should consume roughly the same time and space, no matter where the subtree is located within the full AST. Each subtree must thus be analyzed independently of any other subtree. As a consequence, instead of accumulating constraints into a global system, constraints for each subtree are generated, solved, and simplified independently. The constraints of independent subtrees are merged only at common ancestors. Keeping constraint systems of independent subtrees completely separate also allows more aggressive variable simplification.

One issue raised by inferring separate independent constraint systems for independent expressions is detection of inconsistent constraints. Consider an example with two subtrees A and B, both referring to a free variable γ in the environment. If A adds the constraint $\texttt{int} \subseteq \gamma$, and B adds the constraint $\gamma \subseteq \texttt{float}$, then the inconsistency is detected at the point where the systems of A and B are merged. If the constraint added by A were visible during the inference of B, the inconsistency could be detected earlier. The latter approach is what typically happens in Hindley-Milner type inference based on unification. Type errors are reported as soon as an inconsistent constraint is added to the global constraint system. Though perhaps more intuitive, this approach does not pinpoint type errors very accurately in general. There is no reason to favor tree B over tree A as the source of the inconsistency. Instead, both locations can be flagged.

Our final technique concerns a sparse representation of constraints in solved form, which is discussed in Section 3.1.

3.1 Sparse Constraint Representation

Given a constraint set κ of size n, the worst case storage requirement for the solved form $\overline{\kappa}$ is $O(n^2)$ (for core expressions). We have observed this worst case requirement in practice. If constraints consume space quadratic in the size of the program analyzed, the size of the largest program that can be analyzed is unfortunately not very large. Furthermore, performance of constraint resolution and transformation suffers severely due to the need to traverse the large solved form.

The square order space requirement stems mainly from the generation of transitive constraints. Consider the following constraints:

$$\texttt{cons}(T_1,\texttt{nil}) \subseteq \alpha_1 \quad \alpha_1 \subseteq \alpha_2 \quad \alpha_2 \subseteq \alpha_3 \quad \alpha_3 \subseteq \texttt{cons}(T_2,\texttt{nil})$$

Besides adding transitive constraints between variables such as $\alpha_1 \subseteq \alpha_3$, the transitive closure of these constraints replicates the lower bound $\texttt{cons}(T_1,\texttt{nil})$ and upper bound $\texttt{cons}(T_2,\texttt{nil})$ on all variables. The representation used in our implementation avoids this replication. While computing the transitive constraints for the above system, we keep only the constraint $\texttt{cons}(T_1,\texttt{nil}) \subseteq \texttt{cons}(T_2,\texttt{nil})$. The missing transitive constraints are recomputed as needed; this sparse representation trades time for space.

Define a directed graph G as follows:

- The nodes of G are the expressions and subexpressions of κ.
- For each constraint $\tau_1 \subseteq \tau_2$ in $\overline{\kappa}$, there is an edge $\tau_1 \longrightarrow \tau_2$. (Note that the rules of Figure 1 only add constraints between subexpressions in the original system κ.)

Let $G\downarrow$ be the transitive reduction of G. Our representation has at least the edges of $G\downarrow$, but no more than G. Our representation differs from $G\downarrow$ only in that we do not eliminate transitive constraints introduced by rules other than rule (8) of Figure 1.

3.2 Simplifications

Simplifications reduce the number of variables in types and constraints. More precisely, simplifying a type means replacing it with an *equivalent* type containing fewer variables:

Definition 3.1 Consider two polymorphic constrained types $\sigma_a = \forall\overline{\alpha}.\tau_a \setminus \kappa_a$ and $\sigma_b = \forall\overline{\beta}.\tau_b \setminus \kappa_b$. Let S_a (resp. S_b) be the set of solutions of κ_a (resp. κ_b). Then $\sigma_a \subseteq \sigma_b$ if for every $\mathcal{I}_b \in S_b$ there exists $\mathcal{I}_a \in S_a$ such that $\mathcal{I}_a(\tau_a) \subseteq \mathcal{I}_b(\tau_b)$. The types σ_a and σ_b are equivalent $\sigma_a \equiv \sigma_b$ if $\sigma_a \subseteq \sigma_b$ and $\sigma_b \subseteq \sigma_a$.

(It is an open problem whether the subtype relation on quantified constrained types is decidable; no complete algorithm is known.)

Besides reducing the space requirement for types and constraints, eliminating variables is crucial for performance, since the computational complexity of many algorithms in a constraint framework grows super-linearly in the number of variables. Simplifications also make types easier for humans to read.

To every simplification, there are two aspects: First, a *pattern* identifying candidate types to be simplified, and second, a *set of conditions* to verify that a particular simplification is sound. Currently, our simplification suite simply consists of a set of equivalences we have found necessary to achieve scalability, readability, and to a lesser extent performance. A general, uniform simplification framework would clearly be preferable, but we know of none. Recent work by Pottier [Pot96] proposes a more uniform framework based on an entailment relation. The uniform part of his framework is a soundness test for simplifying substitutions. He does not propose a uniform framework for finding candidates for simplification—this part of the system is still heuristic. In [TS96], the authors develop a decidable, incomplete subtyping relation based on entailment which is more powerful than Pottier's, but still incomplete. In [Smi94], Smith describes a set of type simplifications similar to ours, but for a simpler type language.

Since our primary goal is to reduce the number of variables, the patterns we recognize for simplifications all involve variables. A common condition to all simplifications is that any variable occurring in the pattern must be a universally quantified variable. Thus variables free in the type are never part of a pattern and are never eliminated.

We first motivate and illustrate the various simplifications by example before giving a more complete description of each simplification. Consider the type

$$\forall(\alpha,\beta).\alpha \to \beta \setminus \{\alpha \subseteq \texttt{int}, \texttt{float} \subseteq \beta\}$$

Because function types are anti-monotonic in the domain, it is easy to verify that this type is equivalent to $\forall().\texttt{int} \to \texttt{float}$ (or just $\texttt{int} \to \texttt{float}$) using Definition 3.1. Intuitively, because α occurs only anti-monotonically, it can be set to its upper bound; similarly, because β occurs only monotonically, it can be set to its lower bound. We call this kind of simplification *minimization/maximization* of variables. Section 3.2.1 describes minimization/maximization in more detail. Next, consider the type

$$\forall(\gamma).\alpha \to \beta \setminus \{\alpha \subseteq \gamma, \gamma \subseteq \beta\}$$

In all instances of this type, the variable γ lies between α and β. However, the type of the instance is not affected by the choice for γ, so long as the constraints are satisfied. An equivalent but simpler type is

$$\forall().\alpha \to \beta \setminus \{\alpha \subseteq \beta\}$$

Note that the transitive constraint through γ is made explicit. Section 3.2.2 contains more about this simplification. The constraints of the following type contain a cycle of dependent variables:

$$\forall(\alpha,\beta).\alpha \to \beta \setminus \{\alpha \subseteq \beta, \beta \subseteq \alpha\}$$

Here we have a cycle of length 2. Cycles can be collapsed to a single variable (Section 3.2.3). Clearly if $\alpha \subseteq \beta$ and $\beta \subseteq \alpha$, then $\alpha = \beta$, leading to the simpler type

$$\forall(\alpha).\alpha \to \alpha$$

The next simplification is subtler. Consider the type

$$\forall(\alpha,\beta).(\alpha \cap \beta) \to \texttt{cons}(\alpha,\beta)$$

Suppose function f has this type and we apply f to a value of type $T \subseteq \alpha \cap \beta$. Then $\texttt{cons}(\alpha,\beta) \supseteq \texttt{cons}(T,T)$, with equality if $\alpha = \beta$. This observation is true for every T, thus the distinct variables α and β can be merged into a single variable (Section 3.2.4). The equivalent type is

$$\forall(\alpha).\alpha \to \texttt{cons}(\alpha,\alpha)$$

3.2.1 Minimization/Maximization

Whether or not a variable $\alpha \in (\alpha_1,\ldots,\alpha_n)$ can be minimized or maximized depends on the occurrences of α within $\forall(\alpha_1,\ldots,\alpha_n).\tau \setminus \kappa$. Below is the definition of functions *Pos* and *Neg* that compute the set of monotonically, respectively anti-monotonically occurring variables within a type expression.

$$\begin{array}{rclrcl}
Pos(\alpha) & = & \{\alpha\} & Neg(\alpha) & = & \{\} \\
Pos(\mathrm{c}(\tau_1,\ldots,\tau_n)) & = & \bigcup_{i=1,\ldots,n} Pos(\tau_i) & Neg(\mathrm{c}(\tau_1,\ldots,\tau_n)) & = & \bigcup_{i=1,\ldots,n} Neg(\tau_i) \\
Pos(\tau_1 \to \tau_2) & = & Neg(\tau_1) \cup Pos(\tau_2) & Neg(\tau_1 \to \tau_2) & = & Pos(\tau_1) \cup Neg(\tau_2) \\
Pos(\top) & = & \{\} & Neg(\top) & = & \{\} \\
Pos(\bot) & = & \{\} & Neg(\bot) & = & \{\} \\
Pos(\tau_1 \cup \tau_2) & = & Pos(\tau_1) \cup Pos(\tau_2) & Neg(\tau_1 \cap \tau_2) & = & Neg(\tau_1) \cup Neg(\tau_2) \\
Pos(\tau_1 \cap \tau_2) & = & Pos(\tau_1) \cup Pos(\tau_2) & Neg(\tau_1 \cup \tau_2) & = & Neg(\tau_1) \cup Neg(\tau_2)
\end{array}$$

The function FV computes the set of variables free in a (constrained) polymorphic type. The set of variables occurring monotonically P and the set of variables occurring anti-monotonically N within a constrained type $\tau \setminus \kappa$ are the least sets satisfying:

$$\begin{array}{lll}
\multicolumn{3}{l}{FV(\forall(\alpha_1,\ldots,\alpha_n).\tau \setminus \kappa) \cup Pos(\tau) \subseteq P} \\
\multicolumn{3}{l}{FV(\forall(\alpha_1,\ldots,\alpha_n).\tau \setminus \kappa) \cup Neg(\tau) \subseteq N} \\
\text{if } \alpha \in P, & \tau' \subseteq \alpha \in \overline{\kappa}, \text{ then} & Pos(\tau') \subseteq P, Neg(\tau') \subseteq N \\
\text{if } \alpha \in N, & \alpha \subseteq \tau' \in \overline{\kappa}, \text{ then} & Pos(\tau') \subseteq N, Neg(\tau') \subseteq P
\end{array}$$

Any variable in $(\alpha_1,\ldots,\alpha_n) \setminus P$ can be maximized, and any variable in $(\alpha_1,\ldots,\alpha_n) \setminus N$ can be minimized. Minimizing α means replacing α in $\tau \setminus \kappa$ by the union of α's lower bounds $\bigcup\{T \mid (T \subseteq \alpha) \in \overline{\kappa}\}$. Maximization is done analogously, but the intersection of the upper bounds of α is used. Since constraints may be recursive, T can be substituted for α only if $\alpha \notin FV(T)$ (i.e., recursively constrained variables cannot be eliminated). A proof of soundness for this simplification is in a forthcoming paper.

3.2.2 Truly Intermediate Variables

Consider the type $\forall(\alpha,\beta).\alpha \to \beta \setminus \{\alpha \subseteq \beta\}$. Minimization/maximization does not eliminate α or β , because both appear monotonically and anti-monotonically. However, this type is equivalent to the type of the identity function $\alpha \to \alpha$, so we should be able to either minimize β or maximize α.

We observe that $\bot \subseteq \alpha \subseteq \beta \subseteq \top$, suggesting that either α or β is actually unconstrained. If we choose a particular type, say `int` for α, then β will be constrained from below, and the partial instance will be $\texttt{int} \to \beta \setminus \texttt{int} \subseteq \beta$. As we can see, in this partial instance, β only occurs positively, and the type can be simplified to $\texttt{int} \to \texttt{int}$ using minimization. A similar argument holds for fixing β.

The following refinement of minimization/maximization simplification computes two new sets, sP and sN, in function of P, and N and $\tau \setminus \kappa$.

$$\begin{array}{lll}
\multicolumn{3}{l}{FV(\forall(\alpha_1,\ldots,\alpha_n).\tau \setminus \kappa) \cup Pos(\tau) \subseteq sP} \\
\multicolumn{3}{l}{FV(\forall(\alpha_1,\ldots,\alpha_n).\tau \setminus \kappa) \cup Neg(\tau) \subseteq sN} \\
\text{if } \alpha \in P, & \tau' \subseteq \alpha \in \overline{\kappa}, \text{ and } \tau' \text{ is not a variable, then} & Pos(\tau') \subseteq sP, Neg(\tau') \subseteq sN \\
\text{if } \alpha \in N, & \alpha \subseteq \tau' \in \overline{\kappa}, \text{ and } \tau' \text{ is not a variable, then} & Pos(\tau') \subseteq sN, Neg(\tau') \subseteq sP
\end{array}$$

Instead of using P and N, variables not occurring in sP can be maximized, and variables not occurring in sN can be minimized.

3.2.3 Collapsing Cycles

A set of variables $(\beta_1, \ldots, \beta_n)$ are cyclicly dependent in κ, if $\overline{\kappa}$ contains the constraints $\{\beta_1 \subseteq \beta_2, \beta_2 \subseteq \beta_3, \ldots, \beta_n \subseteq \beta_1\}$. Clearly, any solution $\mathcal{I}$ for κ satisfies $\mathcal{I}(\alpha_1) = \mathcal{I}(\alpha_2) = \ldots = \mathcal{I}(\alpha_n)$.

Cyclic dependencies in $\overline{\kappa}$ can be found using standard graph algorithms on a graph representation of the constraints. The simplification is carried out by replacing each occurrence of any of the variables $(\beta_1, \ldots, \beta_n)$ within $\tau \setminus \kappa$ with a single new variable.

3.2.4 Intersection/Union Merging

Let α and β be two quantified variables appearing in a constrained type $\tau \setminus \kappa$. Further assume that wherever α appears monotonically, it is in a union $\alpha \cup \beta$. Similarly, assume that all monotonic uses of β are in unions $\alpha \cup \beta$. Finally, assume that α and β have the same upper bound(s). Then α and β can be unified, i.e. we can set $\alpha = \beta = \gamma$, where γ is a fresh variable.

Let LB_α, LB_β (resp. UB_α, UB_β) be the set of lower bounds (resp. upper bounds) of α, β in $\overline{\kappa}$. Then $LB_\gamma = LB_\alpha \cup LB_\beta$, and $UB_\gamma = UB_\alpha = UB_\beta$. This simplification is sound because

- The type τ is preserved: all monotonic occurrences of $\alpha \cup \beta$ are constrained below by $LB_\alpha \cup LB_\beta$ before and after the simplification. All anti-monotonic occurrence of either α or β were constrained above by $UB_\alpha = UB_\beta = UB_\gamma$ before and after the simplification.
- The constraint system has the same solutions as before. Since $UB_\alpha = UB_\beta = UB_{\alpha,\beta}$, we have $LB_\alpha \subseteq UB_{\alpha,\beta} \in \overline{\kappa}$, and $LB_\beta \subseteq UB_{\alpha,\beta} \in \overline{\kappa}$. After the simplification, we have $LB_\alpha \cup LB_\beta = LB_\gamma \subseteq UB_\gamma = UB_{\alpha,\beta}$.

The simplification for intersections in anti-monotonic positions is analogous.

We conclude this section by noting that our simplifications are not normalizing, i.e. applying simplifications in different orders may yield different types with differing numbers of variables. How best to simplify constrained types remains an important open problem.

4 Empirical Results

We evaluate our framework with a type and effect inference analysis with subtypes for Standard ML. The analysis is performed on the lambda intermediate representation generated by SML/NJ.

We focus here on the two largest programs in our suite of examples. In the table below, the source line count (LOC) does not include comment lines, but includes interface specifications.

Program	LOC	AST nodes	Type vars	Analysis Time (best)
LexGen	1151	17609	8558	97 sec
SmlYacc	6017	66120	33286	552 sec

To show that our simplifications are effective, we ran our type inference with a range of simplification frequencies. When simplifications are rare, the number of variables in the types and constraints is large. We measure this indirectly by observing the time to recompute the transitive bounds in our sparse constraint representation. The more variables, the longer this recomputation takes.

The graphs in Figure 2 show three time components, total analysis time, time recomputing transitive bounds, and time spent doing simplification (not all time components included in the total analysis time are shown). The simplification interval I (x-axis) defines how often simplification is performed. In this experiment, simplifications were performed on AST nodes of depth $k*I, k = 1, 2, \ldots$ and always in the very end. For $I = 64$, only the final type was simplified. We note that for frequent simplification, the analysis time is dominated by the simplification process itself. For high simplification frequencies, the simplification cost is higher than the gain from the reduction of variables, thus reducing the frequency reduces the overall analysis time. But as the frequency is further reduced, the recomputation of transitive bounds eventually dominates the analysis time due to the many variables that need to be traversed. For `lexgen` this increase is gradual, whereas for `sml-yacc`, the increase is dramatic between intervals 8 and 10. This jump is likely due to the particular choice of where to apply simplification in this experiment. Given two large types

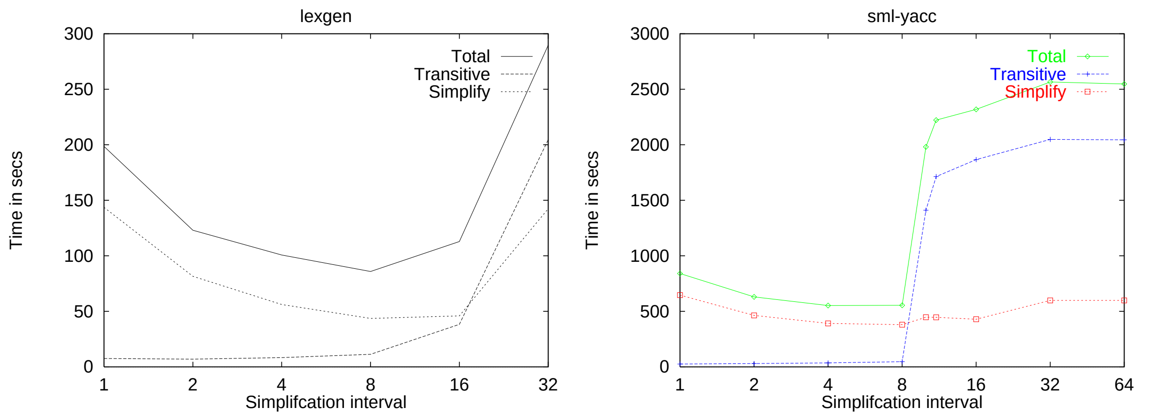

Figure 2: Analysis Time in function of simplification interval. The curve marked "Transitive" shows the time spent recomputing transitive bounds due to our sparse constraint representation. The curve marked "Simplify" shows the time spent doing simplification.

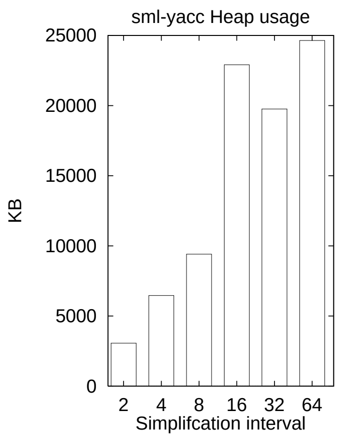

Figure 3: Maximum heap size used by the analysis in function of simplification interval.

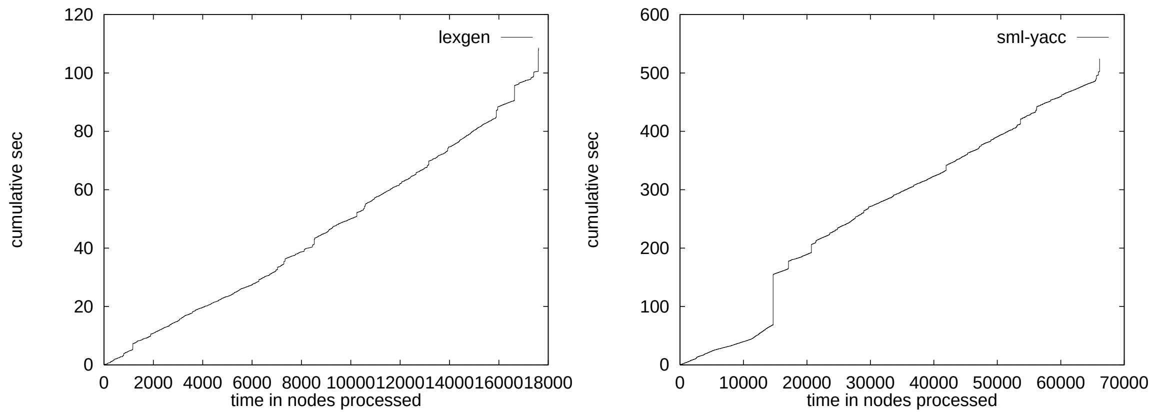

Figure 4: Cumulative time spent per AST node

and constraint systems it maybe much cheaper to simplify the systems separately before the merge, than to merge the large systems and to simplify afterwards. The time profile in Figure 4 shows cumulative time spent per node for `lexgen` and `sml-yacc`. Separation works well (every node takes roughly the same amount of time with a few exceptions) suggesting that we can scale to larger programs.

Figure 3 shows a conservative upper bound on the heap size used to store the types and constraints, again as a function of the simplification frequency. Simplifications reduce the amount of storage required for the constraint graph and the types dramatically. When simplifications are done at every node in the AST, at most 3MB of storage are required. If no simplifications are done, the required storage increases by a factor of more than 8 to 25MB.

Though our implementation seems to scale well, the absolute analysis times are still too long to be practical. We hope to further improve our performance through techniques similar to hash-consing and maybe selective caching of transitive bounds. Furthermore, heavy use of conditional types can blow up space and time requirements beyond reasonable bounds. We have not engineered this aspect of the system yet.

5 Conclusion

We have described the techniques used in our implementation of a expressive set constraint framework for program analysis. Our system scales linearly on an example analysis medium sized program of 6000 lines of non-comment code. Space requirements were one of the main obstacles to obtaining an analysis system that scales to large programs. The large space requirements come from two sources: 1) the number of variables, which has a direct impact on the number of constraints, 2) the representation of the constraints in solved form.

Simplifications help scaling by reducing the space and time requirements to store and traverse the constraint graph. However, simplifications are relatively expensive to compute and there is a point of diminishing returns. Frequent simplification keeps the constraint graph to a small size, thus the transitive bound computations are relatively fast. Fewer simplifications save time, but the space usage for the constraints is larger, and the traversal of the constraints takes more time.

Absolute analysis times are still high, in particular compared to the performance of the system implemented by Heintze [Hei94], which is based on different techniques. We hope to gain some insights into the speed discrepancy by studying his implementation.

6 Acknowledgments

We are grateful to Kwangkeun Yi for sharing his experience and his implementation with us, and to John Boyland for his comments on a draft of this paper.

References

[App92] Andrew Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[AW93] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[AWL94] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, January 1994.

[Dam94] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In *Proceedings of the '94 International Symposium on Theoretical Aspects of Computer Software*, pages 687–706, April 1994.

[Hei94] Nevin Heintze. Set Based Analysis of ML Programs. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 306–17, June 1994.

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

[Pot96] François Pottier. Simplifying subtyping constraints. In *Proceedings of the SIGPLAN '96 International Conference on Functional Programming*, May 1996. to appear.

[Smi94] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2–3):197–226, December 1994.

[TS96] Valery Trifonov and Scott Smith. Subtyping Constrained Types. In *Proceedings of the 3rd International Static Analysis Symposium*, page to be published, September 1996.

[Yi94] Kwangkeun Yi. Compile-time detection of uncaught exceptions for Standard ML programs. In *Proceedings of the First Annual Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*. Springer, 1994.