Constraint Abstractions

Jörgen Gustavsson and Josef Svenningsson

Chalmers University of Technology and Göteborg University
{gustavss,josefs}@cs.chalmers.se

Abstract

Many type based program analyses with subtyping, such as flow analysis, are based on inequality constraints over a lattice. When inequality constraints are combined with polymorphism it is often hard to scale the analysis up to large programs. A major source of inefficiency in conventional implementations stems from computing substitution instances of constraints. In this paper we extend the constraint language with *constraint abstractions* so that instantiation can be expressed directly in the constraint language and we give a cubic-time algorithm for constraint solving. As an application, we illustrate how a flow analysis with flow subtyping, flow polymorphism and flow-polymorphic recursion can be implemented in $O(n^3)$ time where n is the size of the explicitly typed program.

1 Introduction

Constraints are at the heart of many modern program analyses. These analyses are often implemented by two stages. The first stage collects constraints in an appropriate constraint language and the second stage finds a solution (usually the least) to the constraints. If the constraints are collected through a simple linear time traversal over the program yielding a linear amount of constraints the first phase can hardly constitute a bottleneck. But often the constraints for a program point are computed by performing a non constant-time operation on constraints collected for another part of the program. Notable examples, and the motivation for this work, are analyses which combine subtyping and polymorphism. There, typically, the constraints for a call to a polymorphic function f are a *substitution instance* of the constraints for the body of f. For these analyses, to naïvely collect constraints typically leads to unacceptable performance. Consider, for example, how we naïvely could collect the constraints for a program of the following form.

$$
\begin{array}{l}
\texttt{let}\ f_0 = \ldots \\
\texttt{in}\ \texttt{let}\ f_1 = \ldots f_0 \ldots f_0 \\
\quad \texttt{in}\ \texttt{let}\ \ldots \\
\quad\quad \texttt{in}\ \texttt{let}\ f_n = \ldots f_{n-1} \ldots f_{n-1} \ldots \\
\quad\quad\quad \texttt{in}\ \ldots f_n \ldots f_n \ldots
\end{array}
$$

We first collect the constraints for the polymorphic function f_0. Then for the two calls to f_0 in the body of f_1, we compute two different substitution instances of

the constraints from the body of f_0. As a result the number of constraints for f_1 will be at least twice as many as those for f_0. Thus, the number of resulting constraints grows exponentially in the call depth n (even if the underlying types are small). In analyses which combine subtyping and polymorphic recursion, and rely on a fixed point iteration, this effect may show up in every step of the iteration and thus the constraints may grow exponentially in the number of required iterations. We can drastically reduce the number of constraints if we can simplify the constraints to fewer but equivalent constraints. It is therefore no surprise that lots of work has been put into techniques for how to simplify constraints [FM89,Cur90,Kae92,Smi94,EST95,Pot96,TS96,FA96,AWP97,Reh97,FF97].

Another approach is to make the constraint language more powerful so that constraints can be generated by a simple linear time traversal over the program. This can be achieved by *making substitution instantiation a syntactic construct in the constraint language*. But when we make the constraint language more powerful we also make constraint solving more difficult. So is this a tractable approach? The constraint solver could of course just perform the delayed operations and then proceed as before. But can one do better? The answer, of course, depends on the constraint language in question.

In this paper we consider a constraint language with simple inequality constraints over a lattice. Such constraints show up in several type based program analyses such as flow analyses, e.g., [Mos97], binding time analyses, e.g., [DHM95], usage analyses, e.g., [TWM95], points-to-analyses, e.g., [FFA00] and uniqueness type systems [BS96]. We extend this simple constraint language with *constraint abstractions* which allow the constraints to compactly express substitution instantiation.

The main result of this paper is a constraint solving algorithm which computes least solutions to the extended form of constraints in cubic time. We have used this expressive constraint language to formulate usage-polymorphic usage analyses with usage subtyping [Sve00,GS00] and an algorithm closely related to the one in this paper is presented in the second author's Master's thesis [Sve00] ([GS00] focuses on the usage type system and no constraint solving is presented). In this paper, as another example, we show how the constraint language can be used to yield a cubic algorithm for Mossin's polymorphic flow analysis with flow subtyping and flow-polymorphic recursion [Mos97]. This is a significant result – the previously published algorithm, by Mossin, is $O(n^8)$. Independently, Fähndrich and Rehof [RF01] have given an algorithm for Mossin's flow analysis based on *instantiation constraints* which is also $O(n^3)$. We will take a closer look an the relationship of their algorithm and ours in section 4.

1.1 Outline

The rest of this article is organised as follows. In section 2 we introduce our constraint language and give the semantics. In section 3 we present our constraint solving algorithm, its implementation and computational complexity. Section 4 discusses related work and section 5 concludes. In appendix A we illustrate how

the constraint language can be used in a flow analysis. In appendix B we give the proof of Theorem 1.

2 Constraints

In this section we will first introduce the underlying constraints language that we consider in this paper, and then extend the constraint language with constraint abstractions which can express substitution instantiation. The *atomic* constraints we consider are inequality constraints of the form

$$a \leq b$$

where a and b are taken from an countably infinite set of variables. The constraint language also contains the trivially true constraint, conjunction of constraints and existential quantification as given by the following grammar.

$$\begin{array}{ll} \text{Atomic Constraints} & A ::= a \leq b \\ \text{Constraint Terms} & M, N ::= A \mid \top \mid M \wedge N \mid \exists a.M \end{array}$$

These kinds of constraints show up in several different type based program analyses such as, for example, flow analysis, e.g., [Mos97] which we will use as our running example. The constraints arise from the use of subtyping between flow types - i.e., types annotated with flow information.

Depending on the application, the constraints can be interpreted in different domains. For example, for flow analysis we can interpret the constraints in a lattice of finite sets of labels with subset as the ordering.

Definition 1. *We interpret a constraint term in a lattice $\mathcal{L}$, with a bottom element and the ordering $\sqsubseteq$, by defining the notion of a model of a constraint term. Let θ range over mappings from variables into $\mathcal{L}$. Then $\theta \models M$, read as θ is a model of M, is defined inductively by the following rules.*

$$\frac{\theta(a) \sqsubseteq \theta(b)}{\theta \models a \leq b} \qquad \frac{}{\theta \models \top} \qquad \frac{\theta \models M \qquad \theta \models N}{\theta \models M \wedge N} \qquad \frac{\theta[a := d] \models M}{\theta \models \exists a.M}\ d \in \mathcal{L}$$

Given a constraint term one is usually interested in finding its optimal model (usually the least) given a fixed assignment of some of the variables. For example, in flow analysis some of the variables in the constraint term correspond to points in the program where values are produced, often referred to as the *sources* of flow. Other variables correspond to points in the program where values are consumed, often referred to as the *targets* of flow. The existentially quantified variables correspond to the flow annotations on intermediate flow types. To find the flow from the sources to the targets we can fix an assignment for the source variables (usually by associating a unique label l to each source and interpret it as the singleton set $\{l\}$) and compute the least model which respects this assignment. For this simple constraint language it is easy to compute least solutions (it can

be seen as a transitive closure problem) in $O(n^3)$ time, where n is the number of variables. [1]

2.1 Constraint abstractions

When subtyping is combined with polymorphism the need to compute substitution instances of constraint terms arise. We will build this operation into our constraint language through the means of constraint abstractions.

$$\begin{array}{ll} \text{Constraint Abstraction Variables} & f, g, h \\ \text{Constraint Abstractions} & F ::= f\ \vec{a} = M \end{array}$$

A constraint abstraction $f\ \vec{a} = M$ can be seen simply as a function which when applied to some variables $\vec{b}$ returns $M[\vec{a} := \vec{b}]$. Constraint abstractions are introduced by a let-construct reminiscent of let-constructs in functional languages, and are also called in the same way. The complete grammar of the extended constraint language is as follows.

$$\begin{array}{lll} \text{Atomic Constraints} & A & ::= a \leq b \\ \text{Constraint Terms} & M, N & ::= A \mid \top \mid M \wedge N \mid \exists a.M \mid \texttt{let}\ \{\vec{F}\}\ \texttt{in}\ M \mid f\ \vec{a} \\ \text{Constraint Abstractions} & F & ::= f\ \vec{a} = M \end{array}$$

We will write $\mathrm{FV}(M)$ for the free variables of M and $\mathrm{FAV}(M)$ for the free abstraction variables of M. We will identify constraint terms up to α-equivalence, that is the renaming of bound variables and bound abstraction variables. In $\texttt{let}\ \{\vec{F}\}\ \texttt{in}\ M$ the constraint abstraction variables defined by $\vec{F}$ are bound both in M and in the bodies of $\vec{F}$ so our lets are mutually recursive. Consequently the variables defined by $\vec{F}$ must be distinct. We will use Γ to range over sets of constraint abstractions where the defined variables are distinct, and we will denote the addition of a group of distinct constraint abstractions $\vec{F}$ to Γ by juxtaposition: $\Gamma\{\vec{F}\}$. We will say that a group of constraint abstractions $\vec{F}$ is *garbage* in $\texttt{let}\ \Gamma\{\vec{F}\}\ \texttt{in}\ M$ if we can remove the abstractions without causing bound abstraction variables to become free. Recursive constraint abstractions goes beyond just expressing a delayed substitution instantiation. It also allows us to express a fixed-point calculation in a very convenient way. We will make use of this in the flow analysis in appendix A to express flow-polymorphic recursion.

To give a semantics to the extended constraint language we need to define the notion of a model of a constraint term in the context of a set of constraint abstractions Γ.

Definition 2. *In a lattice $\mathcal{L}$, with a bottom element and with the ordering $\sqsubseteq$, we define $\theta; \Gamma \models M$ coinductively by the following rules (we follow the notational convention of Cousot and Cousot [CC92] to mark the rules with a "–" to indicate*

[1] For a lattice where binary least upper bounds can be computed in constant time (for example a two point lattice) the least solution can be computed in $O(n^2)$ time.

that it is a coinductive definition).

$$-\frac{}{\theta;\Gamma \models a \leq b} \quad \theta(a) \sqsubseteq \theta(b) \qquad -\frac{\theta;\Gamma \models M \qquad \theta;\Gamma \models N}{\theta;\Gamma \models M \wedge N}$$

$$-\frac{}{\theta;\Gamma \models \top} \qquad -\frac{\theta[a := d];\Gamma \models M \quad d \in \mathcal{L}}{\theta;\Gamma \models \exists a.M \quad a \notin FV(\Gamma)}$$

$$-\frac{\theta;\Gamma\{\vec{F}\} \models M}{\theta;\Gamma \models \texttt{let}\ \{\vec{F}\}\ \texttt{in}\ M} \qquad -\frac{\theta;\Gamma\{f\ \vec{a} = M\} \models M[\vec{a} := \vec{b}]}{\theta;\Gamma\{f\ \vec{a} = M\} \models f\ \vec{b}}$$

The definition needs to be coinductive to cope with recursive constraint abstractions. The coinductive definition expresses the intuitive concept that such constraint abstractions should be "unfolded infinitely". When it is not clear from the context we will write $\theta;\Gamma \models_{\mathcal{L}} M$ to make explicit which lattice we consider. We will say that N is a *consequence* of M, written $M \models N$, iff for every $\mathcal{L}$, θ, Γ, if $\theta;\Gamma \models_{\mathcal{L}} M$ then $\theta;\Gamma \models_{\mathcal{L}} N$. We will write $M \Leftrightarrow N$ iff $M \models N$ and $N \models M$.

In definitions throughout this paper we will find it convenient to work with *constraint term contexts*. A constraint term context is simply a constraint term with a "hole" analogous to term contexts used extensively in operational semantics.

$$\begin{aligned}\text{Constraint Term Contexts } C \ ::= {} & [\cdot] \mid C \wedge M \mid M \wedge C \mid \exists a.C \mid \\ & \texttt{let}\ \Gamma\ \texttt{in}\ C \mid \texttt{let}\ \Gamma\{f\ \vec{a} = C\}\ \texttt{in}\ M\end{aligned}$$

We will write $C[M]$ to denote the filling of the hole in C with M. Hole filling may capture variables. We will write $\mathrm{CV}(C)$ for the variables that may be captured when filling the hole. We will say that the hole in C is *live* if the hole does not occur in a constraint abstraction which is garbage. Our first use of constraint term contexts is in the definition of the *free live atomic constraints* of a constraint term.

Definition 3. *The set of free live atomic constraints of a constraint term M, denoted $LIVE(M)$, is defined as follows.*

$$LIVE(M) = \{A \mid M \equiv C[A], FV(A) \cap CV(C) = \emptyset \text{ and the hole in } C \text{ is live.}\}$$

We will use $\mathrm{LIVE}(M)$ in definitions where we need to refer to the atomic subterms of M but want to exclude those which occur in constraint abstractions which are garbage and thus never will be "called" by the models relation. Note that all syntactically live constraint abstractions are semantically live since they are all "called" by the models relation.

Another use of constraint term contexts is in the statement of the following unwinding lemma.

Lemma 1. *If $FV(M) \cap CV(C) = \emptyset$ then*

$$\texttt{let}\ \Gamma\{f\ \vec{a} = M\}\ \texttt{in}\ C[f\ \vec{b}] \Leftrightarrow \texttt{let}\ \Gamma\{f\ \vec{a} = M\}\ \texttt{in}\ C[M[\vec{a} := \vec{b}]]$$

1. if $a \leq b, b \leq c \in \mathrm{LIVE}(M)$ then

   $\exists b.M \;\mapsto\; \exists b.M \wedge a \leq c$

2. if $A \in \mathrm{LIVE}(M)$, and, for some i, $a_i \in \mathrm{FV}(A)$ then

   $\begin{array}{ll} \mathtt{let}\ \Gamma\{f\ \vec{a} = M\} & \mapsto\ \mathtt{let}\ \Gamma\{f\ \vec{a} = M\} \\ \mathtt{in}\ C[f\ \vec{b}] & \quad\ \mathtt{in}\ C[f\ \vec{b} \wedge A[\vec{a} := \vec{b}]] \end{array}$

3. if $A \in \mathrm{LIVE}(C[f\ \vec{b}])$, and, for some i, $a_i \in \mathrm{FV}(A)$ then

   $\begin{array}{ll} \mathtt{let}\ \Gamma\{f\ \vec{a} = C[f\ \vec{b}]\} & \mapsto\ \mathtt{let}\ \Gamma\{f\ \vec{a} = C[f\ \vec{b} \wedge A[\vec{a} := \vec{b}]]\} \\ \mathtt{in}\ M & \quad\ \mathtt{in}\ M \end{array}$

4. if $A \in \mathrm{LIVE}(M)$, and for some i, $a_i \in \mathrm{FV}(A)$ then

   $\begin{array}{ll} \mathtt{let}\ \Gamma\{f\ \vec{a} = M\}\{g\ \vec{c} = C[f\ \vec{b}]\} & \mapsto\ \mathtt{let}\ \Gamma\{f\ \vec{a} = M\}\{g\ \vec{c} = C[f\ \vec{b} \wedge A[\vec{a} := \vec{b}]]\} \\ \mathtt{in}\ M & \quad\ \mathtt{in}\ M \end{array}$

Fig. 1. Rewrite rules

This lemma is necessary, and is the only difficulty, when proving the subject reduction property of the usage analysis in [GS00] and the flow analysis in appendix A. The premise $\mathrm{FV}(M) \cap \mathrm{CV}(C) = \emptyset$ is there to ensure that no inadvertent name capture takes place and it can always be fulfilled by an α-conversion. In the remainder of this paper we will leave this condition on unwindings implicit.

3 Solving Constraints

As we discussed in the previous section we are interested in finding the least model of a constraint term given a fixed assignment of some of the variables. In this section we will present an algorithm for this purpose for our constraint language. The algorithm is based on a rewrite system which rewrites constraint terms to equivalent but more informative ones. Every rewrite step adds an atomic constraint to the constraint term and the idea is that when the rules have been applied exhaustively then enough information is explicit in the term so that the models can be constructed easily.

Definition 4. *We define the rewrite relation $\rightarrow$ as the compatible closure of the relation $\mapsto$ defined by the clauses in figure 1.*

Here we provide some explanation of the rewrite rules. The first rule,

1. if $a \leq b, b \leq c \in \mathrm{LIVE}(M)$ then

   $\exists b.M \;\mapsto\; \exists b.M \wedge a \leq c$

is a simple transitivity rule. If $a \leq b$ and $b \leq c$ are free live atomic subterms of M we may simply add the constraint $a \leq c$. Note that the rule requires a and c to be in scope at the binding occurrence of b. As a result we cannot, for example, perform the rewrite

$$\exists a.\exists b.(a \leq b) \wedge (\exists c.b \leq c) \;\rightarrow\; \exists a.\exists b.(a \leq b) \wedge (\exists c.b \leq c \wedge a \leq c)$$

which adds $a \leq c$ although it would make perfect sense. The reason is simply that at the binding occurrence of b, c is not in scope. The purpose of the restriction on the transitivity rule is an important one. It reduces the number of rewrite steps due to transitivity by taking advantage of scoping information. The second rule

2. if $A \in \mathrm{LIVE}(M)$, and, for some i, $a_i \in \mathrm{FV}(A)$ then

$$\begin{array}{ll} \texttt{let}\ \Gamma\{f\,\vec{a} = M\} & \mapsto\ \texttt{let}\ \Gamma\{f\,\vec{a} = M\} \\ \texttt{in}\ C[f\,\vec{b}] & \quad\ \texttt{in}\ C[f\,\vec{b} \wedge A[\vec{a} := \vec{b}]] \end{array}$$

allows us to unwind an atomic constraint. Note that at least one of the variables in A must be bound by the abstraction. The restriction is there to prevent rewrite steps which would not be useful anyway. The two last rules are similar to the second rule but deal with unwinding in mutually recursive constraint abstractions. A key property of the rewrite rules is that they lead to equivalent constraint terms.

Lemma 2. *If $M \mapsto N$ then $M \Leftrightarrow N$*

The property is easy to argue for the transitivity rule. For the second rule it follows from the unwinding property (Lemma 1). The two last rules rely on similar unwinding properties for unwinding in mutually recursive constraint abstractions.

3.1 Normal forms

Intuitively a constraint term is in normal form when the rules in figure 1 have been applied exhaustively. But nothing stops us from performing rewrite steps which just add new copies of atomic constraints which are already in the constraint term. We can of course do this an arbitrary number of times creating a sequence of terms which are different but "essentially the same". To capture this notion of essentially the same we define a congruence which equates terms which are equal up to copies of atomic constraints.

Definition 5. *We define $\sim$ as the reflexive, transitive, symmetric and compatible closure of the following clauses.*

$$\textit{(i)}\ A \wedge A \sim A \qquad \textit{(ii)}\ M \wedge \top \sim M \qquad \textit{(iii)}\ \top \wedge M \sim M$$
$$\textit{(iv) if}\ FV(A) \cap CV(C) = \emptyset\ \textit{and the hole in}\ C\ \textit{is live then}\ C[A] \sim C[\top] \wedge A$$

Rewriting commutes with $\sim$ so we can naturally extend $\rightarrow$ to equivalence classes of $\sim$. With the help of $\sim$ we can define the notion of a *productive* rewrite step $M \rightsquigarrow N$ which is a rewrite step which adds a new atomic constraint.

Definition 6. *$M \rightsquigarrow N$ iff $M \rightarrow N$ and $M \not\sim N$.*

Finally we arrive at our definition of *normal form* up to productive rewrite steps.

Definition 7. *M is in normal form iff $M \not\rightsquigarrow$.*

The main technical theorem in this paper is that when a constraint term with no free constraint abstraction variables is in normal form then the models of the constraint term are exactly characterised by the free live atomic constraints of the constraint term.

Theorem 1. *If M is in normal form and $FAV(M) = \emptyset$ then $\theta; \emptyset \models M$ iff $\theta \models LIVE(M)$*

Given a constraint term M and a fixed assignment of some of the variables we can find its least model as follows. First we find an equivalent constraint term N in normal form. Then we extract the free live atomic constraints of the normal form which exactly characterises the models of N and M. Since LIVE(N) is just a set of atomic constraints we can then proceed with any standard method, such as computing the transitive closure. The proof of Theorem 1 can be found in appendix B. The key component of the proof is the application of two key properties of unwindings of normal forms. The first property is that normal forms are preserved by unwindings.

Lemma 3. *If $\texttt{let}\ \Gamma\{f\ \vec{a} = M\}\ \texttt{in}\ C[f\ \vec{b}]$ is in normal form then the unwinding $\texttt{let}\ \Gamma\{f\ \vec{a} = M\}\ \texttt{in}\ C[M[\vec{a} := \vec{b}]]$ is in normal form.*

The lemma guarantees normal forms of arbitrary unwindings of a normal form which we need because of the coinductive definition of $\theta; \Gamma \models M$. The second property is that unwinding of a normal form does not change the free live atomic constraints of the constraint term.

Lemma 4. *If $\texttt{let}\ \Gamma\{f\ \vec{a} = M\}\ \texttt{in}\ C[f\ \vec{b}]$ is in normal form then*

$$LIVE(\texttt{let}\ \Gamma\{f\ \vec{a} = M\}\ \texttt{in}\ C[f\ \vec{b}]) = LIVE(\texttt{let}\ \Gamma\{f\ \vec{a} = M\}\ \texttt{in}\ C[M[\vec{a} := \vec{b}]])$$

3.2 Computing Normal Forms

Given a constraint term M, we need to compute an equivalent term in normal form. Our algorithm relies on a representation of equivalence classes of terms with respect to $\sim$ and computes sequences of the form

$$M_0 \rightsquigarrow M_1 \rightsquigarrow M_2 \rightsquigarrow \ldots.$$

The termination of the algorithm is ensured by the following result.

Lemma 5. *There is no infinite sequence of the form given above.*

Proof (Sketch). Let n be the number of variables (free and bound) in M_0. Note that the number of variables remain constant in each step. Thus the number of unique atomic constraints that can be added to M is bounded by n^2. Since every productive rewrite step introduces a new atomic constraint the number of steps is bounded by n^2.

When given a constraint term as input, our algorithm first marks all atomic constraints. These marked constraints can be thought of as a work list of atomic constraints to consider. The algorithm then unmarks the constraints one by one and performs all productive rewrite steps which only involve atomic constraints which are not marked. The new atomic constraints which are produced by a rewrite step are initially marked. The algorithm maintains the following invariant: the term obtained by replacing the marked terms with $\top$ is in normal form. The algorithm terminates with a normal form when no atomic constraints remain marked. The pseudo code for this algorithm is given below.

Algorithm 1 *1. Mark all atomic constraints.*
2. If there are no remaining marked constraints then stop otherwise pick a marked atomic constraint and unmark it.
3. Find all productive redexes which involve the unmarked constraint and perform the corresponding rewrite steps. Let the added atomic constraints be marked.
4. Go to step 2.

3.3 Data Structures

The efficiency of the algorithm relies on maintaining certain data structures. In step 3 of the algorithm we use data structures such that we can solve the following two problems:

1. find all redexes we need to consider in time proportional to the number of such, and
2. decide in constant time whether a redex is productive.

We can solve the first problem if we maintain, for every existentially bound variable b,

- a list of all a in scope at the point where b is bound, such that $a \leq b$ is an unmarked atomic constraint in the term.
- a list of all c in scope at the point where b is bound, such that $b \leq c$ is an unmarked atomic constraint in the term.

With this information we can easily list all transitivity-redexes we need to consider in step 3, in time proportional to the number of redexes. When we unmark a constraint we can update the data structure in constant time.

For the second problem, to decide in constant time whether a redex is productive, we need to decide, in constant time, whether the atomic constraint to be added already exists in the term. We can achieve this by a n times n bit-matrix

where n is the number of variables (free and bound) in the constraint term. If $a \leq b$ is in the term then the entry in the matrix for (a, b) is 1 and 0 otherwise. This is sufficient for the complexity argument in the next section but in practice we use a refined data structure which we describe in section 3.5.

3.4 Complexity

The cost of the algorithm is dominated by the operations performed by step 3, which searches for productive redexes. The cost is proportional to the number of redexes (productive or non-productive) considered and each redex in the final normal form is considered exactly once in step 3. Thus the cost of step 3 is proportional to the number of redexes in the final normal form. An analysis of the maximum number of redexes gives the following.

- The maximum number of transitivity-redexes is, for each existentially quantified variable a, the square of the number of variables in scope at the point where a is bound.
- The maximum number of unwind-redexes is, for each variable a bound in a constraint abstraction f, two times the number of variables in scope at the point where a is bound times the number of calls to f.

A consequence of this analysis is the complexity result we are about to state. Let the *skeleton* of a constraint term be the term where all occurrences of atomic constraints, and the trivially true constraint have been removed. What remains are the binding occurrences of variables and all calls to constraint abstractions. Now, for a constraint term M, let n be the size of the skeleton of M plus the number of free variables of M. The complexity of the algorithm can be expressed in terms of n as follows.

Theorem 2. *The normal form can be computed $O(n^3)$ time.*

3.5 Refined Data Structure

The cost of initialising the bit-matrix described in section 3.3 is dominated by the cost of step 3 in the algorithm but we believe that in practice the cost of initialising the matrix may be significant. Also the amount of memory required for the matrix is quite substantial and many entries in the matrix would be redundant since the corresponding variables have no overlapping scope. Below we sketch a refined approach based on this observation which we believe will be important in practice. We associate a natural number, index(a), with every variable a. We assign the natural number as follows. First we choose an arbitrary order for all the free variables and bind them existentially, in this order, at top level. Then we assign to each variable the lexical binding level of the variable. For example, in $\exists a.(\exists b.M) \wedge (\exists c.N)$ we assign 0 to a, 1 to b and c, and so on. Note that the number we assign to each variable is unique within the scope of the variable. Given this we have the following data structures. For every variable b,

- a set of all a such that index(a) $\leq$ index(b) and $a \leq b$ is an atomic constraint (marked or unmarked) in the term.
- a set of all c such that index(c) $\leq$ index(b) and $b \leq c$ is an atomic constraint (marked or unmarked) in the term.

The sets have, due to scoping, the property that, for any two distinct elements a and b, index(a) is distinct from index(b). Thus the sets can be represented by bit-arrays, indexed by index(a) so that set membership can be decided in constant time. Now, to decide whether an atomic constraint $a \leq b$ is in the constraint becomes just set membership in the appropriate set.

4 Related Work

The motivation for this paper is to reduce the cost of the combination of subtyping and polymorphism and in this respect it is related to numerous papers on constraint simplification techniques [FM89,Cur90,Kae92,Smi94,EST95,Pot96,TS96,FA96,AWP97,Reh97,FF97]. Our work is particularly related to the work by Dussart, Henglein and Mossin on binding-time analysis with binding-time-polymorphic recursion [DHM95] where they use constraint simplification techniques in combination with a clever fixed-point iteration to obtain a polynomial time algorithm. In his thesis Mossin applied these ideas to show that a flow analysis with flow-polymorphic recursion can be implemented in polynomial time [Mos97]. Our flow analysis in appendix A, that we give as an example of how constraint abstractions can be used, is based on this flow analysis. A consequence of the complexity of our constraint solving algorithm is that the analysis can be implemented in $O(n^3)$ time where n is the size of the explicitly type program. This is a substantial improvement over the algorithm by Mossin which is $O(n^8)$ [2] [Mos97].

To represent instantiation in the constraint language is not a new idea. It goes back at least to Henglein's work on type-polymorphic recursion [Hen93] where he uses *semiunification constraints* to represent instantiation. Although constraint abstractions and semiunification constraints may have similar applications they are inherently different: Semiunification constraints are inequality constraints of the form $A \leq B$ which constrains the (type) term B to be an instance of A by an *unknown* substitution. In contrast, a call to a constraint abstraction denotes a *given instance* of *the constraints* in the body of the abstraction.

Closely related to our work is the recent work by Rehof and Fähndrich [RF01] where they also give an $O(n^3)$ algorithm for Mossin's flow analysis. The key idea in their and our work is the same – to represent substitution instantiation in the constraints by extending the constraint language. However, the means are not the same. Where we use constraint abstractions they use *instantiation constraints*, a form of inequality constraints similar to semiunification constraints

[2] In his thesis Mossin states that he believes that the given algorithm can be improved. In fact an early version of [DHM95] contained a $O(n^3)$ algorithm for binding-time analysis but it was removed from the final version since its correctness turned out to be non-trivial (personal communication with Fritz Henglein).

but labelled with an instantiation site and a polarity. They compute the flow information from the constraints through an algorithm for *Context-Free Language (CFL) reachability* [Rep97,MR00]. A key difference between constraint abstractions and instantiation constraints is that constraint abstractions offer more structure and a notion of local scope whilst in the work by Rehof and Fähndrich all variables scope over the entire set of constraints. Our algorithm takes advantage of the scoping in an essential way. Firstly, we do not add any edges between variables that have no common scope and secondly the scoping comes into the restriction of our transitivity rule and the unwind rules. Although the scoping does not improve the asymptotic complexity in terms of the size of the explicitly typed program it shows up in the more fine-grained complexity argument leading to the cubic bound (see section 3.4) and it is essential for the refined data structures we sketch in section 3.5. Constraint abstractions also offer a more subjective advantage – the additional structure of constraint abstractions enforces many useful properties. As a result we think it will be easy to use constraint abstractions in a wide range of type based analyses and we think that constraint abstractions will not lead to any additional difficulties when establishing the soundness of the analyses.

We have previously used constraint abstraction to formulate usage-polymorphic usage analyses with usage subtyping [Sve00,GS00] and an algorithm closely related to the one in this paper is presented in the second authors masters thesis [Sve00] ([GS00] focuses on the usage type system and no constraint solving is presented).

5 Conclusions and Future Work

In this paper we have shown how a constraint language with simple inequality constraints over a lattice can be extended with constraint abstractions which allow the constraints to compactly express substitution instantiation. The main result of this paper is a constraint solving algorithm which computes least solutions to the extended form of constraints in cubic time. In [GS00] we have used this expressive constraint language to formulate a usage-polymorphic usage analyses with usage subtyping and usage-polymorphic recursion and in an appendix to this paper we demonstrate how the extended constraint language can be used to yield a cubic algorithm for Mossin's polymorphic flow analysis with flow subtyping and flow polymorphic recursion [Mos97]. We believe that our approach can be applied to a number of other type based program analyses such as binding time analyses, e.g., [DHM95], points-to-analyses, e.g., [FFA00] and uniqueness type systems [BS96].

An interesting possibility for future work is to explore alternative constraint solving algorithms. The current algorithm has a rather compositional character in that, it rewrites the body of a constraint abstraction without considering how it is called. In [Sve00] we describe an algorithm where the different calls to a constraint abstraction lead to rewrites inside the abstraction. The algorithm can in this way take advantage of global information (it can be thought of as a form

of caching) which yields a interesting finer grained complexity characterisation. The algorithm in [Sve00] is however restricted to *non-recursive* constraint abstractions and it is not clear whether the algorithm can be extended to recursive constraint abstractions (although we believe so). Another opportunity for future work is to investigate whether constraint abstractions can be a useful extension for other underlying constraint languages. Constraint abstraction could also possibly be made more powerful by allowing constraint abstractions to be passed as parameters to constraint abstractions (i.e., making them higher order). Finally a practical comparison with Mossin's algorithm and the algorithm by Rehof and Fähndrich remains to be done. The outcome of such a comparison is not clear to us.

Acknowledgements. We would like to thank our supervisor David Sands for his support and the anonymous referees for their useful comments.

References

[AWP97] A. Aiken, E. Wimmers, and J. Palsberg. Optimal representation of polymorphic types with subtyping. In *Proceedings TACS'97 Theoretical Aspects of Computer Software*, pages 47–77. Springer Lecture Notes in Computer Science, vol. 1281, September 1997.

[BS96] E. Barendsen and S. Smetsers. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.

[CC92] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the Ninthteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94, Albuquerque, New Mexico, January 1992. ACM Press, New York, NY.

[Cur90] P. Curtis. Constrained qualification in polymorphic type analysis. Technical Report CSL-09-1, Xerox Parc, February 1990.

[DHM95] D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *proceedings of 2nd Static Analysis Symposium*, September 1995.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings OOPSLA'95*, 1995.

[FA96] M. Fändrich and A. Aiken. Making set-constraint program analyses scale. In *Workshop on Set Constraints*, 1996.

[Fax95] Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In *Proc. of SAS'95*, pages 136–153. Springer-Verlag, LNCS 983, September 1995.

[FF97] C. Flanagan and M. Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*. ACM, June 1997.

[FFA00] J. Foster, M. Fändrich, and A. Aiken. Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C. In *Proceedings of 2000 Static Analysis Symposium*, June 2000.

[FM89] Y. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Proceedings of Int'l J't Conf. on Theory and Practice of Software Development*, pages 167–183. Springer-Verlag, March 1989.

[GS00] J. Gustavsson and J. Svenningsson. A usage analysis with bounded usage polymorphism and subtyping. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages*. Springer-Verlag, LNCS, September 2000. To appear.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[Kae92] Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *1992 ACM conference on LISP and Functional Programming*, pages 193–204, San Francisco, CA, 1992. ACM Press.

[Mos97] C. Mossin. *Flow Analysis of Typed Higher-Order Programs (Revised Version)*. PhD thesis, University of Copenhagen, Denmark, August 1997.

[MR00] David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free language reachability. *Theoretical Computer Science*, 248, November 2000.

[Pot96] F. Pottier. Simplifying subtyping constraints. In *Proceedings ICFP'97, International Conference on Functional Programming*, pages 122–133. ACM Press, May 1996.

[Reh97] J. Rehof. Minimal typings in atomic subtyping. In *Proceedings POPL'97, 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 278–291, Paris, France, January 1997. ACM.

[Rep97] T. Reps. Program analysis via graph reachability. In *Proc. of ILPS '97*, pages 5–19. Springer-Verlag, October 1997.

[RF01] Jakob Rehof and Manuel Fändrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *Proceedings of 2001 Symposium on Principles of Programming Languages*, 2001.

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

[Sve00] Josef Svenningsson. An efficient algorithm for a sharing analysis with polymorphism and subtyping. Masters thesis, June 2000.

[TS96] V. Trifonov and S. Smith. Subtyping constrained types. In *Proceedings SAS'96, Static Analysis Symposium*, pages 349–365, Aachen, Germany, 1996. Springer Verlag.

[TWM95] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proc. of FPCA*, La Jolla, 1995.

A Flow Analysis

In this appendix we illustrate how constraint abstractions can be used in practice. As an example, we briefly present a flow-polymorphic type based flow analysis with flow-polymorphic recursion. For another example see [GS00] where constraint abstractions are used in usage analysis. The flow analysis is based on the flow analysis by Mossin [Mos97] but we use our extended constraint language with constraint abstractions. A similar analysis, but without polymorphic recursion, is given by Faxén [Fax95]. For simplicity we restrict ourself to a simply typed functional language. To extend the analysis to a language with a Hindley-Milner style type system is not difficult. See for example [Fax95]. A key result is that the analysis can be implemented in $O(n^3)$ time where n is the size of the

explicitly typed program which is a substantial improvement over the algorithm by Mossin which is $O(n^8)$ [Mos97].

The aim of flow analysis is to statically compute an approximation to the flow of values during the execution of a program. To be able to pose flow questions we will label subexpressions with unique flow variables. We will label expressions in two distinct ways, as a *source* of flow or as a *target* of flow. We will use e^a as our notation for labelling e (with flow variable a) as a source of flow and e_a as our notation for labelling e as a target of flow. If we are interested in the flow of values from *producers* to *consumers* then we label all program points where values are created as sources of flow, we label the points where values are destructed as targets of flow, and we leave all other subexpressions unlabelled. In the example below we have labelled all values as sources with flow variables a_0 through a_4 and we have labelled the arguments to plus as targets with a_5 and a_6. We have not labelled the others consumers (the applications) to keep the example less cluttered.

$$\begin{array}{l}\texttt{let}\ \mathit{apply} = (\lambda f.(\lambda y.f\ y)^{a_0})^{a_1}\\ \texttt{in}\ \texttt{let}\ \mathit{id} = (\lambda x.x)^{a_2}\\ \quad \texttt{in}\ (\mathit{apply}\ \mathit{id}\ 5^{a_3})_{a_5} + (\mathit{apply}\ \mathit{id}\ 7^{a_4})_{a_6}\end{array}$$

We may now ask the question "which values may show up as arguments to plus?". Our flow analysis will give the answer that the value labelled with a_3 (5) may flow to a_5 (the first argument) and a_4 (7) may flow to a_6 (the second argument). In this example the flow polymorphic types that we assign to id and *apply* plays a crucial role. A monomorphic system would conservatively say that both values could flow to both places. For some applications we might be interested in, not only the flow from producers to consumers, but also the flow to points on the way from a consumer to a producer. In our example we might be interested in the flow to x in the body of id. We then add a target label on x as in

$$\begin{array}{l}\texttt{let}\ \mathit{apply} = (\lambda f.(\lambda y.f\ y)^{a_0})^{a_1}\\ \texttt{in}\ \texttt{let}\ \mathit{id} = (\lambda x.x_{a_7})^{a_2}\\ \quad \texttt{in}\ (\mathit{apply}\ \mathit{id}\ 5^{a_3})_{a_5} + (\mathit{apply}\ \mathit{id}\ 7^{a_4})_{a_6}\end{array}$$

and then ask for the flow to a_7. Our analysis would answer with a_3 and a_4. An important property of the analysis is that the type of id remains polymorphic even though we tap off the flow passing through x. Thus our type system corresponds to the *sticky interpretation* of a type derivation in [Mos97]. The key to this property is to distinguish between source labels and target labels. If the label on x would serve as both a source and a target label the flow through id would be monomorphic. [3]

The language we consider is a lambda calculus extended with recursive let-expressions, integers, lists and case-expressions. The grammar of the language is

[3] We can achieve this degrading effect by annotating x both as a source and as a target but using the same flow variable, i.e., as $x_{a_7}^{a_7}$.

as follows.

Variables	x, y, z	
Flow Variables	a	
Expressions	e	$::= \lambda x.e \mid n \mid \texttt{nil} \mid \texttt{cons}\ e_0\ e_1 \mid x \mid e_0 + e_1 \mid e_0\ e_1 \mid$ $\texttt{let}\ \{\vec{b}\}\ \texttt{in}\ e \mid \texttt{case}\ e\ \texttt{of}\ alts \mid e^a \mid e_a$
Bindings	b	$::= x = e$
Alternatives	$alts$	$::= \{\texttt{nil} \Rightarrow e_0, \texttt{cons}\ x\ y \Rightarrow e_1\}$

The language is simply typed and for our complexity result we assume that the terms are explicitly typed by having type annotations attached to every subterm. For our flow analysis we label the types of the underlying type system with flow variables.

$$\text{Flow Types } \tau ::= \texttt{Int}^a \mid (\tau \to \tau')^a \mid (\texttt{List}\ \tau)^a$$

We will let ρ range over flow types without the outermost annotation. The subtype entailment relation which take the form $M \vdash \tau_0 \leq \tau_1$ is defined in Figure 2. Recall that M ranges over constraint terms as defined in Section 2.1. We read $M \vdash \tau_0 \leq \tau_1$ as "from the constraint term M it can be derived that $\tau_0 \leq \tau_1$". We will let σ range over type schemas.

$$\text{Type Schemas } \sigma ::= \forall \vec{a}.\ f\ \vec{a} \Rightarrow \tau$$

Since the underlying type system is monomorphic type schemas will only quantify over flow variables. A type schema contains a call $f\,\vec{a}$ to a constraint abstraction which may constrain the quantified variables. We will let Θ and Δ range over typing contexts which associates variables with types or type schemas depending on whether it is a let-bound variable or not. We will use juxtaposition as our notation for combining typing contexts. Our typing judgements take the form $\Theta; M \vdash e : \tau$ for terms, $\Theta; F \vdash b : (x : \sigma)$ for bindings and $\Theta; \Gamma \vdash \{\vec{b}\} : \Delta$ for groups of bindings. (Recall that F ranges over constraint abstractions and that Γ ranges over sets of constraint abstractions.) The typing rules of the analysis can be seen in Figure 3. The key difference to the type system in [Mos97] is in the rule Binding where generalisation takes place. Instead of putting the constraint term used to type the body of the binding into the type schema the constraint term is inserted into a new constraint abstraction and a call to this abstraction is included in the type schema.

To compute the flow in a program we can proceed as follows. First we compute a principal typing of the program which includes a constraint term where the free variables are the flow variables labelling the program. We then apply the algorithm from Section 3 and extract a set of atomic constraints which we can view as a graph. If there is a path from a_0 to a_1 then a_0 may flow to a_1. The typing rules as presented here are not syntax directed and cannot directly be interpreted as describing an algorithm for computing principal typings. Firstly, the subsumption rule (Sub) and the rule (Exist-intro) which introduces existential quantification in constraints can be applied everywhere in a typing derivation. This problem is solved by the standard approach to incorporate (Sub) and

$$\frac{}{\top \vdash \texttt{Int} \leq \texttt{Int}} \qquad \frac{M \vdash \tau \leq \tau'}{M \wedge (a \leq a') \vdash (\texttt{List}\ \tau)^a \leq (\texttt{List}\ \tau')^{a'}}$$

$$\frac{M \vdash \tau_0' \leq \tau_0 \qquad N \vdash \tau_1 \leq \tau_1'}{M \wedge N \vdash \tau_0 \to \tau_1 \leq \tau_0' \to \tau_1'} \qquad \frac{M \vdash \rho_0 \leq \rho_1}{M \wedge (a \leq a') \vdash \rho_0^a \leq \rho_1^{a'}}$$

Fig. 2. Subtyping rules

$$\text{Abs}\frac{\Theta\{x : \tau\}; M \vdash e : \tau'}{\Theta; M \vdash \lambda x.e : (\tau \to \tau')^a} \quad \text{Int}\frac{}{\Theta; \top \vdash n : \texttt{Int}^a} \quad \text{Nil}\frac{}{\Theta; \top \vdash \texttt{nil} : (\texttt{List}\ \tau)^a}$$

$$\text{Cons}\frac{\Theta; M \vdash e_0 : \tau \qquad \Theta; N \vdash e_1 : (\texttt{List}\ \tau)^a}{\Theta; M \wedge N \vdash \texttt{cons}\ e_0\ e_1 : (\texttt{List}\ \tau)^a}$$

$$\text{Var-}\sigma\frac{}{\Theta\{x : \forall \vec{a}.f\ \vec{a} \Rightarrow \tau\}; f\ \vec{b} \vdash x : \tau[\vec{a} := \vec{b}]} \quad \text{Var-}\tau\frac{}{\Theta\{x : \tau\}; \top \vdash x : \tau}$$

$$\text{Plus}\frac{\Theta; M \vdash e_0 : \texttt{Int}^{a_0} \qquad \Theta; N \vdash e_1 : \texttt{Int}^{a_1}}{\Theta; M \wedge N \vdash e_0 + e_1 : \texttt{Int}^a}$$

$$\text{App}\frac{\Theta; M \vdash e_0 : (\tau \to \tau')^a \qquad \Theta; N \vdash e_1 : \tau}{\Theta; M \wedge N \vdash e_0\ e_1 : \tau'}$$

$$\text{Let}\frac{\Theta\Delta; \Gamma \vdash \{\vec{b}\} : \Delta \qquad \Theta\Delta; M \vdash e : \tau}{\Theta; \texttt{let}\ \Gamma\ \texttt{in}\ M \vdash \texttt{let}\ \{\vec{b}\}\ \texttt{in}\ e : \tau} \qquad \text{Case}\frac{\Theta; M \vdash e : \tau \qquad \Theta; N \vdash alts : \tau \Rightarrow \tau'}{\Theta; M \wedge N \vdash \texttt{case}\ e\ \texttt{of}\ alts : \tau'}$$

$$\text{Alts}\frac{\Theta; M \vdash e_0 : \tau' \qquad \Theta\{x : \tau, y : (\texttt{List}\ \tau)^a\}; N \vdash e_1 : \tau'}{\Theta; M \wedge N \vdash \{\texttt{nil} \Rightarrow e_0; \texttt{cons}\ x\ y \Rightarrow e_1\} : (\texttt{List}\ \tau)^a \Rightarrow \tau'}$$

$$\text{Source}\frac{\Theta; M \vdash e : \rho^a}{\Theta; M \wedge (a \leq c) \wedge (b \leq c) \vdash e^b : \rho^c} \quad \text{Target}\frac{\Theta; M \vdash e : \rho^a}{\Theta; M \wedge (a \leq c) \wedge (a \leq b) \vdash e_b : \rho^c}$$

$$\text{Binding group-}\emptyset\frac{}{\Theta; \emptyset \vdash \emptyset : \emptyset} \qquad \text{Binding group}\frac{\Theta; \Gamma \vdash \{\vec{b}\} : \Delta \qquad \Theta; F \vdash b : (x : \sigma)}{\Theta; \Gamma\{F\} \vdash \{\vec{b}, b\} : (\Delta, x : \sigma)}$$

$$\text{Binding}\frac{\Theta; M \vdash e : \tau}{\Theta; f\ \vec{a} = M \vdash x = e : (x : \forall \vec{a}.f\ \vec{a} \Rightarrow \tau)}\ \{\vec{a}\} \cap \text{FV}(\Theta, f\ \vec{a} = M, e) = \emptyset$$

$$\text{Sub}\frac{\Theta; M \vdash e : \tau}{\Theta; M \wedge N \vdash e : \tau'}\ N \vdash \tau \leq \tau' \quad \text{Exist-intro}\frac{\Theta; M \vdash e : \tau}{\Theta; \exists \vec{a}.M \vdash e : \tau}\ \{\vec{a}\} \cap \text{FV}(\Theta, e, \tau) = \emptyset$$

Fig. 3. Typing rules for a flow analysis

(Exists-intro) into an appropriate subset of the other rules to obtain a syntax-directed set of rules. Secondly, in the rule (Let) an inference algorithm would have to come up with an appropriate Δ. However, this only amounts to coming up with fresh names: Clearly, Δ would have to contain one type associations of the form $x : \sigma$ for each variable defined by the let-expression. Recall that σ is of the form $\forall \vec{a}.f\ \vec{a} \Rightarrow \tau$. We obtain τ simply by annotating the underlying type with fresh flow variables. Since they are fresh we will be able to generalise over all of them so we can take $\vec{a}$ to be these variables in some order. Finally we generate the fresh name f for the constraint abstraction. Note that no fixed-point calculation is required which is possible because we have recursive constraint abstractions. Now let us apply the algorithm to our example program. We first compute the constraint term in the principal typing which yields the following.

$$\begin{array}{l}
\texttt{let}\ f_{apply}\ b_0\ b_1\ b_2\ b_3\ b_4\ b_5\ b_6 = \exists c_0.\exists c_1.\exists c_2.(b_3 \leq b_0) \wedge (b_1 \leq b_4) \wedge (b_2 \leq c_2) \wedge \\
\qquad\qquad (c_1 \leq b_5) \wedge (a_0 \leq b_5) \wedge (c_0 \leq b_6) \wedge (a_1 \leq b_6) \\
\texttt{in let}\ f_{id}\ b_0\ b_1\ b_2 = \exists c_0.\exists c_1.(b_0 \leq c_1) \wedge (c_1 \leq b_1) \wedge (c_1 \leq a_7) \wedge \\
\qquad\qquad (c_0 \leq b_2) \wedge (a_2 \leq b_2) \\
\quad \texttt{in}\ \exists c_0 \ldots \exists c_{18}.(f_{apply}\ c_0\ c_1\ c_2\ c_3\ c_4\ c_5\ c_6) \wedge (f_{id}\ c_0\ c_1\ c_2) \wedge \\
\qquad\qquad (c_7 \leq c_3) \wedge (a_3 \leq c_3) \wedge (c_4 \leq c_8) \wedge (c_4 \leq a_5) \wedge \\
\qquad\qquad (f_{apply}\ c_{10}\ c_{11}\ c_{12}\ c_{13}\ c_{14}\ c_{15}\ c_{16}) \wedge (f_{id}\ c_{10}\ c_{11}\ c_{12}) \wedge \\
\qquad\qquad (c_{17} \leq c_{13}) \wedge (a_3 \leq c_{13}) \wedge (c_{14} \leq c_{18}) \wedge (c_{14} \leq a_5)
\end{array}$$

Then we apply the algorithm from Section 3 and extract the set of free live atomic constraints which is $\{a_3 \leq a_5,\ a_4 \leq a_6,\ a_3 \leq a_7,\ a_4 \leq a_7\}$. The paths in this constraint set (viewed as a graph) is the result of the analysis.

Finally, by inspecting the rules we can see that the size of the skeleton of the constraint term required to type a program is proportional to the size of the explicitly typed program and that the number of free variables is the number of flow variables in the program. From this fact and theorem 2 we can conclude that the complexity of the flow analysis is $O(n^3)$ where n is the size of the typed program.

B Proof of Theorem 1

In this appendix we give a proof of Theorem 1. We first introduce a form of constraint term contexts, reminiscent of evaluation contexts used in operational semantics, where the hole may not occur under any binder.

$$\text{Evaluation Contexts } E ::= [\cdot] \mid E \wedge M \mid M \wedge E$$

Note that the hole in an evaluation context is always live. We have the following properties for evaluation contexts which we state without proof.

Lemma 6. *1.* $\texttt{let}\ \Gamma\ \texttt{in}\ E[\texttt{let}\ \Gamma'\ \texttt{in}\ M]$ *is in normal form iff* $\texttt{let}\ \Gamma\Gamma'\ \texttt{in}\ E[M]$ *is in normal form.*

2. $LIVE(\texttt{let}\ \Gamma\ \texttt{in}\ E[\texttt{let}\ \Gamma'\ \texttt{in}\ M]) = LIVE(\texttt{let}\ \Gamma\Gamma'\ \texttt{in}\ E[M])$.

3. *If* $a \notin FV(\Gamma, E)$, *and* $\texttt{let}\,\Gamma\,\texttt{in}\,E[\exists a.M]$ *is in normal form then* $\texttt{let}\,\Gamma\,\texttt{in}\,E[M]$ *is in normal form.*

The key to the proof of Theorem 1 is the following auxiliary relation.

Definition 8. *We define an auxiliary relation* $\theta; \Gamma \bullet\models M$ *as:*
$\theta; \Gamma \bullet\models M$ *iff there exists* E *such that*

1. $\texttt{let}\,\Gamma\,\texttt{in}\,E[M]$ *is in normal form,*
2. $\theta \models \mathit{LIVE}(\texttt{let}\,\Gamma\,\texttt{in}\,E[M])$,
3. $\mathit{FAV}(\texttt{let}\,\Gamma\,\texttt{in}\,E[M]) = \emptyset$.

The technical core of the proof now shows up in the proof of the following lemma.

Lemma 7. *if* $\theta; \Gamma \bullet\models M$ *then* $\theta; \Gamma \models M$.

Before we proceed with the proof of this lemma we will use it to establish Theorem 1.

Proof (Theorem 1). Assume the premise. The right way implication (if $\theta; \emptyset \models M$ then $\theta \models \mathrm{LIVE}(M)$) follows the fact that all syntactically live constraints are semantically live. To show the left way implication (if $\theta \models \mathrm{LIVE}(M)$ then $\theta; \emptyset \models M$) assume that $\theta \models \mathrm{LIVE}(M)$ which immediately gives $\theta; \emptyset \bullet\models M$. Thus, by Lemma 7, $\theta; \emptyset \models M$ as required.

Finally we prove Lemma 7.

Proof (Lemma 7). Recall that $\theta; \Gamma \models M$ is defined coinductively by the rules in Figure 1. That is, $\models$ is defined as the largest fixed point of the functional $\mathcal{F}$ expressed by the rules. By the coinduction principle we can show that $\bullet\models \subseteq \models$ if we can show that $\bullet\models \subseteq \mathcal{F}(\bullet\models)$. Thus we assume that $\theta; \Gamma \bullet\models M$ and proceed by case analysis on the structure of M.

case $M \equiv a \leq b$**:** By the definition of $\theta; \Gamma \bullet\models a \leq b$ there exists E which fulfils the requirements in Definition 8. In particular, $\theta \models \mathrm{LIVE}(\texttt{let}\,\Gamma\,\texttt{in}\,E[a \leq b])$. Since E cannot capture variables and the hole in E is live we know that $a \leq b \in \mathrm{LIVE}(\texttt{let}\,\Gamma\,\texttt{in}\,E[M])$ so $\theta; \Gamma\ \mathcal{F}(\bullet\models)\ a \leq b$.

case $M \equiv \top$**:** Trivial.

case $M \equiv K \wedge L$**:** To show that $\theta; \Gamma\ \mathcal{F}(\bullet\models)\ K \wedge L$ we need to show that $\theta; \Gamma \bullet\models K$ and $\theta; \Gamma \bullet\models L$. We will only show the former, the latter follows symmetrically. By the definition of $\theta; \Gamma \bullet\models K \wedge L$ there exists E which fulfils the requirements in Definition 8. Take E' to be $E[[\cdot] \wedge L]$. Then E' is a witness of $\theta; \Gamma \bullet\models K$.

case $M \equiv \texttt{let}\,\Gamma'\,\texttt{in}\,N$**:** We may without loss of generality (due to properties of α-conversion) assume that the constraint abstraction variables defined Γ and Γ' are disjoint. To show that $\theta; \Gamma\ \mathcal{F}(\bullet\models)\ \texttt{let}\,\Gamma'\,\texttt{in}\,N$ we need to show that $\theta; \Gamma\Gamma' \bullet\models N$. By the definition of $\theta; \Gamma \bullet\models \texttt{let}\Gamma'\texttt{in}M$ there exists E which fulfils the requirements in Definition 8. Floating of let bindings preserves normal forms (Lemma 6) so we can float out Γ' and obtain $\texttt{let}\,\Gamma\Gamma'\,\texttt{in}\,E[M]$ in normal form. Also, by Lemma 6, $\mathrm{LIVE}(\texttt{let}\,\Gamma\,\texttt{in}\,E[\texttt{let}\,\Gamma'\,\texttt{in}\,M]) = \mathrm{LIVE}(\texttt{let}\,\Gamma\Gamma'\,\texttt{in}\,E[M])$. Thus E is a witness of $\theta; \Gamma\Gamma' \bullet\models M$.

case $M \equiv f\ \vec{b}$**:** By the definition of $\theta; \Gamma \bullet\models f\ \vec{b}$ we know that f must bound by Γ, i.e., $\Gamma = \Gamma'\{f\ \vec{a} = N\}$ for some Γ' and some N. We are required to show that $\theta; \Gamma'\{f\ \vec{a} = N\} \bullet\models N[\vec{a} := \vec{b}]$. From $\theta; \Gamma \bullet\models f\ \vec{b}$ we know that there exists E which fulfils the requirements in Definition 8. Normal forms are closed under unwindings (Lemma 3) so $\texttt{let}\ \Gamma'\{f\ \vec{a} = N\}\ \texttt{in}\ E[N[\vec{a} := \vec{b}]]$ is in normal form. Also, by Lemma 4,

$$\text{LIVE}(\texttt{let}\ \Gamma'\{f\ \vec{a} = N\}\ \texttt{in}\ E[f\ \vec{b}]) = \text{LIVE}(\texttt{let}\ \Gamma'\{f\ \vec{a} = N\}\ \texttt{in}\ E[N[\vec{a} := \vec{b}]]).$$

Thus E is a witness of $\theta; \Gamma'\{f\ \vec{a} = N\} \bullet\models N[\vec{a} := \vec{b}]$.

case $M \equiv \exists a.N$ To show that $\theta; \Gamma\ \mathcal{F}(\bullet\models)\ \exists a.N$ we need to show that there exists $d \in \mathcal{L}$ such that $\theta[a := d]; \Gamma \bullet\models N$. Let

$$d = \bigsqcup\{\theta(a') \mid a' \neq a \text{ and } a' \leq a \in \text{LIVE}(N)\}.$$

By the definition of $\theta; \Gamma \bullet\models \exists a.N$ there exists E which fulfils the requirements in Definition 8. Without loss of generality (due to properties of α-conversion) we can assume that $a \notin \text{FV}(\Gamma, E)$. Since $\texttt{let}\ \Gamma\ \texttt{in}\ E[\exists a.N]$ is in normal form, and $a \notin \text{FV}(\Gamma, E)$ we know, by Lemma 6, that $\texttt{let}\ \Gamma\ \texttt{in}\ E[N]$ is in normal form. It remains to show that $\theta[a := d] \models \text{LIVE}(\texttt{let}\ \Gamma\ \texttt{in}\ E[N])$. Given $A \in \text{LIVE}(\texttt{let}\ \Gamma\ \texttt{in}\ E[N])$ we proceed by the following cases.

subcase $A \equiv a \leq a$**:** Trivial.

subcase $A \equiv b \leq c$ **where** $b \neq a$ **and** $c \neq a$**:**
In this case $A \in \text{LIVE}(\texttt{let}\ \Gamma\ \texttt{in}\ E[\exists a.N])$ so $\theta \models A$ and thus $\theta[a := d] \models A$.

subcase $A \equiv b \leq a$ **where** $b \neq a$ **:** In this case $A \in \text{LIVE}(N)$ and thus $\theta[a := d] \models A$ by the construction of d.

subcase $A \equiv a \leq b$ **and** $b \neq a$**:** In this case $a \leq b \in \text{LIVE}(N)$. We will show that $\theta(b)$ is an upper bound of

$$\{\theta(a') \mid a' \neq a \text{ and } a' \leq a \in \text{LIVE}(N)\}$$

and, since d is defined as the least upper upper bound, $\theta[a := d] \models a \leq b$ follows. Now given any a' such that $a' \neq a$ and $a' \leq a \in \text{LIVE}(N)$. Since $a' \leq a \in \text{LIVE}(N)$ and $a \leq b \in \text{LIVE}(N)$ we know that $\texttt{let}\ \Gamma\ \texttt{in}\ E[\exists a.N] \rightarrow \texttt{let}\ \Gamma\ \texttt{in}\ E[\exists a.N \wedge a' \leq b]$ and since $\texttt{let}\ \Gamma\ \texttt{in}\ E[\exists a.N]$ is in normal form we know that

$$\text{LIVE}(\texttt{let}\ \Gamma\ \texttt{in}\ E[\exists a.N \wedge a' \leq b]) = \text{LIVE}(\texttt{let}\ \Gamma\ \texttt{in}\ E[\exists a.N]).$$

Finally, since $a' \neq a$ it must be the case that $a' \leq b \in \text{LIVE}(\texttt{let}\ \Gamma\ \texttt{in}\ E[\exists a.N \wedge a' \leq b])$ and thus $a' \leq b \in \text{LIVE}(\texttt{let}\ \Gamma\ \texttt{in}\ E[\exists a.N])$. Hence $\theta \models a' \leq b$ so $\theta(a') \sqsubseteq \theta(b)$.