A Toolkit for Constructing Type- and Constraint-Based Program Analyses

Alexander Aiken, Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su

University of California, Berkeley* **

Abstract. BANE (the *Berkeley Analysis Engine*) is a publicly available toolkit for constructing type- and constraint-based program analyses.[1] We describe the goals of the project, the rationale for BANE's overall design, some examples coded in BANE, and briefly compare BANE with other program analysis frameworks.

1 Introduction

Automatic program analysis is central to contemporary compilers and software engineering tools. Program analyses are also arguably the most difficult components of such systems to develop, as significant theoretical and practical issues must be addressed in even relatively straightforward analyses.

Program analysis poses difficult semantic problems, and considerable effort has been devoted to understanding what it means for an analysis to be correct [CC77]. However, designing a theoretically well-founded analysis is necessary but not sufficient for obtaining a useful analysis. Demonstrating utility requires implementation and experimentation, preferably with large programs. Many plausible analyses are not beneficial in practice, and others require substantial modification and tuning before producing useful information at reasonable cost.

It is important to prototype and realistically test analysis ideas, usually in several variations, to judge the cost/performance trade-offs of multiple design points. We know of no practical analytical method for showing utility, because the set of programs that occur in practice is a very special, and not easily modeled, subset of all programs. Unfortunately, experiments are relatively rare because of the substantial effort involved.

BANE (for the *Berkeley ANalysis Engine*) is a toolkit for constructing type- and constraint-based program analyses. A goal of the project is to dramatically lower the barriers to experimentation and to make it relatively easy for researchers to realistically prototype and test new program analysis ideas (at

* Authors' address: Electrical Engineering and Computer Science Department, University of California, Berkeley, 387 Soda Hall #1776, Berkeley, CA 94720-1776 Email: {aiken,manuel,jfoster,zhendong}@cs.berkeley.edu

** Supported in part by an NDSEG fellowship, NSF National Young Investigator Award CCR-9457812, NSF Grant CCR-9416973, and gifts from Microsoft and Rockwell.

[1] The distribution may be obtained from the BANE homepage at `http://bane.cs.berkeley.edu`.

least type- and constraint-based ideas). To this end, in addition to providing constraint specification and resolution components, the BANE distribution also provides parsers and interfaces for popular languages (currently C and ML) as well as test suites of programs ranging in size from a few hundred to tens of thousands of lines of code.

BANE has been used to implement several realistic program analyses, including an uncaught exception inference system for ML programs [FA97,FFA98], points-to analyses for C [FFA97,FFSA98], and a race condition analysis for a factory control language [AFS98]. Each of these analyses also scales to large programs—respectively at least 20,000 lines of ML, 100,000 lines of C, and production factory control programs. These are the largest programs we have available (the peculiar syntax of the control language precludes counting lines of code).

2 System Architecture

Constraint-based analysis is appealing because elaborate analyses can be expressed with a concise and simple set of constraint generation rules. These rules separate analysis specification (constraint generation) from implementation (constraint resolution). Implementing an analysis using BANE involves only writing code to (1) generate the appropriate constraints from the program text and (2) interpret the solutions of the constraints. Part (1) is usually a simple recursive walk of the abstract-syntax tree, and part (2) is usually testing for straightforward properties of the constraint solutions. The system takes care of constraint representation, resolution, and transformation. Thus, BANE frees the analysis designer from writing a constraint solver, usually the most difficult portion of a constraint-based analysis to design and engineer.

In designing a program analysis toolkit one soon realizes that no single formalism covers both a large fraction of interesting analyses and provides uniformly good performance in an implementation. BANE provides a number of different constraint *sorts*: constraint languages and associated resolution engines that can be reused as appropriate for different applications. Each sort is characterized by a language of expressions, a constraint relation, a solution space, and an implementation strategy. In some cases BANE provides multiple implementations of the same constraint language as distinct sorts because the different implementations provide different engineering trade-offs to the user. Extending BANE with new sorts is straightforward.

An innovation in BANE is support for *mixed constraints*: the use of multiple sorts of constraints in a single application [FA97]. In addition to supporting naturally multi-sorted applications, we believe the ability to change constraint languages allows analysis designers to explore fine-grain engineering decisions, targeting subproblems of an analysis with the constraint system that gives the best efficiency/precision properties for the task at hand. Section 3 provides an example of successively refining an analysis through mixed constraints.

$$
\begin{array}{rl}
\cup : \texttt{Set}\ \texttt{Set} \to \texttt{Set} & \\
\cap : \texttt{Set}\ \texttt{Set} \to \texttt{Set} & \\
\neg\{c_1, \ldots, c_n\} : \texttt{Set} & \text{for any set of } \texttt{Set}\text{-constructors } c_i \in \Sigma_{Set} \\
0 : \texttt{Set} & \\
1 : \texttt{Set} &
\end{array}
$$

Fig. 1. Operations in the sort `Set`.

Mixed constraint systems are formalized using a many-sorted algebra of expressions. Each sort s includes a set of variables V_s, a set of constructors Σ_s, and possibly some other operations. Each sort has a constraint relation $\subseteq_s$. Constraints and resolution rules observe sorts; that is, a constraint $X \subseteq_s Y$ implies X and Y are s-expressions.

The user selects the appropriate mixture of constraints by providing constructor signatures. If S is the set of sorts, each n-ary constructor c is given a signature

$$c : \iota_1 \ldots \iota_n \to S$$

where ι_i is s or $\overline{s}$ for some $s \in S$. Overlined sorts mark contravariant arguments of c; the rest are covariant arguments. For example, let sort `Term` be a set of constructors $\Sigma_{\texttt{Term}}$ and variables $V_{\texttt{Term}}$ with no additional operations. Pure terms over $\Sigma_{\texttt{Term}}$ and $V_{\texttt{Term}}$ are defined by giving constructor signatures

$$c : \underbrace{\texttt{Term} \ldots \texttt{Term}}_{arity(c)} \to \texttt{Term} \quad c \in \Sigma_{\texttt{Term}}$$

As another example, let `Set` be a sort with the set operators in Figure 1 (the set operations plus least and greatest sets). Pure set expressions are defined by the signatures

$$c : \underbrace{\texttt{Set} \ldots \texttt{Set}}_{arity(c)} \to \texttt{Set} \quad c \in \Sigma_{\texttt{Set}}$$

There are many examples of program analyses based on equations between `Terms` (e.g., [DM82,Hen92,Ste96]) and based on inclusion constraints between `Set` expressions (e.g., [And94,AWL94,EST95,FFK+96,Hei94]). The literature also has natural examples of mixed constraint systems, although they have not been recognized previously as a distinct category. For example, many *effect systems* [GJSO92] use a function space constructor

$$\cdot \dot{\to} \cdot : \ \overline{\texttt{Term}}\ \texttt{Set}\ \texttt{Term} \to \texttt{Term}$$

where the `Set` expressions are used only to carry the set of latent effects of the function.

These three examples—terms, set expressions, and a mixed language with set and term components—illustrate that by altering the signatures of constructors

a range of analysis domains can be realized. For example, a flow-based analysis using set expressions can be coarsened to a unification-based analysis using terms. Similarly, a term-based analysis can be refined to an effect analysis by adding a `Set` component to the $\rightarrow$ constructor.

2.1 The Framework

From the user's point of view, our framework consists of a number of sorts of expressions together with resolution rules for constraints over those expressions. In addition, the user must provide constructor signatures specifying how the different sorts are combined. In this section we focus on the three sorts `Term`, `FlowTerm`, and `Set`. The distributed implementation also supports a `Row` sort [Rém89] for modeling records.

Besides constructors and variables a sort may have arbitrary operations peculiar to that sort; for example, sort `Set` includes set operations. Each sort s has a constraint relation $\subseteq_s$ and resolution rules. Constraints and resolution rules preserve sorts, so that $X \subseteq_s Y$ implies X and Y are s-expressions. For example, for the `Term` sort, the constraint relation $\subseteq_{\texttt{Term}}$ is equality, and the resolution rules implement term unification for constructors with signatures $\texttt{Term} \ldots \texttt{Term} \rightarrow \texttt{Term}$. For clarity we write the constraint relation of term unification as "$=_{\texttt{t}}$" instead of $\subseteq_{\texttt{Term}}$.

The resolution rules in Figure 2 are read as left-to-right rewrite rules. The left- and right-hand sides of rules are conjunctions of constraints. Sort `FlowTerm` has the expressions of sort `Term` but a different set of resolution rules (see Figure 2b). `FlowTerm` uses inclusion instead of equality constraints. The inclusion constraints are more precise, but also more expensive to resolve, requiring exponential time in the worst case. For certain applications, however, `FlowTerm` is very efficient [HM97]. We write $\subseteq_{\texttt{ft}}$ for the `FlowTerm` constraint relation.

The constructor rules connect constraints of different sorts. For example, in sort `FlowTerm` the rule

$$S \wedge c(T_1, \ldots, T_n) \subseteq_{\texttt{ft}} c(T'_1, \ldots, T'_n) \equiv S \wedge T_1 \subseteq_{\iota_1} T'_1 \wedge \cdots \wedge T_n \subseteq_{\iota_n} T'_n \quad \text{if } c : \iota_1 \cdots \iota_n \rightarrow \texttt{FlowTerm}$$

says constraints propagate structurally to constructor arguments; this is where `FlowTerm` has a precision advantage over `Term` (see below). Note this rule preserves sorts. The rule for constructors of sort `Term` (Figure 2a) is slightly different because $\subseteq_{\texttt{Term}}$ is equality, a symmetric relation. Thus, constraints on constructor arguments are also symmetric:

$$S \wedge f(T_1, \ldots, T_n) =_{\texttt{t}} f(T'_1, \ldots, T'_n) \equiv S \wedge T_1 \subseteq_{\iota_1} T'_1 \wedge T'_1 \subseteq_{\iota_1} T_1 \wedge \cdots \wedge T_n \subseteq_{\iota_n} T'_n \wedge T'_n \subseteq_{\iota_n} T_n \quad \text{if } f : \iota_1 \cdots \iota_n \rightarrow \texttt{Term}$$

Figure 2c shows the rules for the `Set` sort. In addition to the standard rules [AW93], `Set` includes special rules for set complement, which is problematic in the presence of contravariant constructors. We deal with set complement using

$$
\begin{aligned}
S \wedge f(T_1, \ldots, T_n) =_{\mathtt{t}} f(T'_1, \ldots, T'_n) &\equiv S \wedge T_1 \subseteq_{\iota_1} T'_1 \wedge T'_1 \subseteq_{\iota_1} T_1 \wedge \cdots \wedge \\
&\quad\; T_n \subseteq_{\iota_n} T'_n \wedge T'_n \subseteq_{\iota_n} T_n \quad \text{if } f : \iota_1 \cdots \iota_n \to \mathtt{Term} \\
S \wedge f(\ldots) =_{\mathtt{t}} g(\ldots) &\equiv \text{inconsistent} \qquad\qquad\quad \text{if } f \neq g
\end{aligned}
$$

(a) Resolution rules for sort `Term`.

$$
\begin{aligned}
S \wedge c(T_1, \ldots, T_n) \subseteq_{\mathtt{ft}} c(T'_1, \ldots, T'_n) &\equiv S \wedge T_1 \subseteq_{\iota_1} T'_1 \wedge \cdots \wedge T_n \subseteq_{\iota_n} T'_n \\
&\qquad\qquad \text{if } c : \iota_1 \cdots \iota_n \to \mathtt{FlowTerm} \\
S \wedge c(\ldots) \subseteq_{\mathtt{ft}} d(\ldots) &\equiv \text{inconsistent} \quad \text{if } c \neq d \\
S \wedge \alpha \subseteq_{\mathtt{ft}} c(T_1, \ldots, T_n) &\equiv S \wedge \alpha = c(\alpha_1, \ldots, \alpha_n) \wedge \alpha_i \subseteq_{\iota_i} T_i \\
&\qquad\qquad \alpha_i \text{ fresh}, \quad c : \iota_1 \cdots \iota_n \to \mathtt{FlowTerm} \\
S \wedge c(T_1, \ldots, T_n) \subseteq_{\mathtt{ft}} \alpha &\equiv S \wedge \alpha = c(\alpha_1, \ldots, \alpha_n) \wedge T_i \subseteq_{\iota_i} \alpha_i \\
&\qquad\qquad \alpha_i \text{ fresh}, \quad c : \iota_1 \cdots \iota_n \to \mathtt{FlowTerm}
\end{aligned}
$$

(b) Resolution rules for sort `FlowTerm`.

$$
\begin{aligned}
S \wedge 0 \subseteq_{\mathtt{s}} T &\equiv S \\
S \wedge T \subseteq_{\mathtt{s}} 1 &\equiv S \\
S \wedge c(T_1, \ldots, T_n) \subseteq_{\mathtt{s}} c(T'_1, \ldots, T'_n) &\equiv S \wedge T_1 \subseteq_{\iota_1} T'_1 \wedge \cdots \wedge T_n \subseteq_{\iota_n} T'_n \\
&\qquad\qquad \text{if } c : \iota_1 \cdots \iota_n \to \mathtt{Set} \\
S \wedge c(\ldots) \subseteq_{\mathtt{s}} d(\ldots) &\equiv \text{inconsistent} \quad \text{if } c \neq d \\
S \wedge T_1 \cup T_2 \subseteq_{\mathtt{s}} T &\equiv S \wedge T_1 \subseteq_{\mathtt{s}} T \wedge T_2 \subseteq_{\mathtt{s}} T \\
S \wedge T \subseteq_{\mathtt{s}} T_1 \cap T_2 &\equiv S \wedge T \subseteq_{\mathtt{s}} T_1 \wedge T \subseteq_{\mathtt{s}} T_2 \\
S \wedge \alpha \subseteq_{\mathtt{s}} \alpha &\equiv S \\
S \wedge \alpha \cap T \subseteq_{\mathtt{s}} \alpha &\equiv S \\
S \wedge T_1 \subseteq_{\mathtt{s}} Pat(T_2, T_3) &\equiv S \wedge T_1 \cap T_3 \subseteq_{\mathtt{s}} T_2 \\
S \wedge \alpha \cap T_1 \subseteq_{\mathtt{s}} T_2 &\equiv S \wedge \alpha \subseteq_{\mathtt{s}} Pat(T_2, T_1) \\
S \wedge \neg\{c_1, \ldots, c_n\} \subseteq_{\mathtt{s}} \neg\{d_1, \ldots, d_m\} &\equiv S \qquad \text{if } \{d_1, \ldots, d_m\} \subseteq \{c_1, \ldots, c_n\} \\
S \wedge c(\ldots) \subseteq_{\mathtt{s}} \neg\{d_1, \ldots, d_m\} &\equiv S \qquad \text{if } c \notin \{d_1, \ldots, d_m\}
\end{aligned}
$$

(c) Resolution rules for sort `Set`.

$$
\begin{aligned}
S \wedge X \subseteq_\iota \alpha \wedge \alpha \subseteq_\iota Y &\equiv S \wedge X \subseteq_\iota \alpha \wedge \alpha \subseteq_\iota Y \wedge X \subseteq_\iota Y \\
S \wedge T_1 \subseteq_\tau T_2 &\equiv S \wedge T_2 \subseteq_\iota T_1
\end{aligned}
$$

(d) General rules.

Fig. 2. Resolution rules for constraints.

two mechanisms. First, explicit complements have the form $\neg\{c_1, \ldots, c_n\}$, which has all values of sort `Set` except those with head constructor $c_1, \ldots, c_n$. Second, more general complements are represented implicitly. Define $\neg R$ to be the set such that $R \cap \neg R = 0$ and $R \cup \neg R = 1$ (in all solutions). Now define

$$Pat(T, R) = (T \cap R) \cup \neg R$$

The operator Pat[2] encapsulates a disjoint union involving a complement. Pat is equivalent to in power to disjoint union, but constraint resolution involving Pat does not require computing complements. Of course, wherever $Pat(T, R)$ is used the set $\neg R$ must exist; this is an obligation of the analysis designer (see [FA97] for details). Given the definitions of Pat and $\neg\{c_1, \ldots, c_n\}$, basic set theory shows the rules in Figure 2c are sound.

Our specification of sort `Set` is incomplete. We have omitted some rules for simplifying intersections and some restrictions on the form of solvable constraints. The details may be found in [AW93,FA97].

Figure 2d gives two general rules that apply to all sorts. The first rule expresses that $\subseteq_\iota$ is transitive. The second flips constraints that arise from contravariant constructor arguments.

We now present a small example of a mixed constraint system. Consider an effect system where each function type carries a set of atomic effects (e.g., the set of globally visible variables that may be modified by invoking the function). Let the constructors have signatures

$$\begin{aligned} \cdot \xrightarrow{\cdot} \cdot &: \overline{\texttt{FlowTerm}}\ \texttt{Set}\ \texttt{FlowTerm} \to \texttt{FlowTerm} \\ \texttt{int} &: \texttt{FlowTerm} \\ \texttt{a}_1, \ldots, \texttt{a}_n &: \texttt{Set} \qquad \text{(the atomic effects)} \end{aligned}$$

The following constraint

$$\alpha \xrightarrow{\texttt{a}_1 \cup \texttt{a}_2} \beta \subseteq_{\texttt{ft}} \texttt{int} \xrightarrow{\gamma} \texttt{int}$$

is resolved as follows:

$$\begin{aligned} & \alpha \xrightarrow{\texttt{a}_1 \cup \texttt{a}_2} \beta \subseteq_{\texttt{ft}} \texttt{int} \xrightarrow{\gamma} \texttt{int} \\ \Rightarrow\ & \alpha \subseteq_{\overline{\texttt{tf}}} \texttt{int} \wedge \texttt{a}_1 \cup \texttt{a}_2 \subseteq_{\texttt{s}} \gamma \wedge \beta \subseteq_{\texttt{ft}} \texttt{int} \\ \Rightarrow\ & \texttt{int} \subseteq_{\texttt{tf}} \alpha \wedge \texttt{a}_1 \cup \texttt{a}_2 \subseteq_{\texttt{s}} \gamma \wedge \beta \subseteq_{\texttt{ft}} \texttt{int} \\ \Rightarrow\ & \alpha = \texttt{int} \wedge \texttt{a}_1 \cup \texttt{a}_2 \subseteq_{\texttt{s}} \gamma \wedge \beta = \texttt{int} \end{aligned}$$

Thus in all solutions α and β are both `int` and γ is a superset of $\texttt{a}_1 \cup \texttt{a}_2$.

2.2 Scalability

The main technical challenge in BANE is to develop methods for scaling constraint-based analyses to large programs. Designing for scalability has led to a system

[2] *Pat* stands for "pattern," because it is used most often to express pattern matching.

with a significantly different organization than other program analysis systems [Hei94,AWL94].

To handle very large programs it is essential that the implementation be structured so that independent program components can be analyzed separately first and the results combined later. Consider the following generic inference rule where expressions are assigned types under some set of assumptions A and constraints C

$$\frac{A, C \vdash e_1 : \tau_1 \quad A, C \vdash e_2 : \tau_2}{A, C \vdash E[e_1, e_2] : \tau}$$

where $E[e_1, e_2]$ is a compound expression with subexpressions e_1 and e_2. In all other implementations we know of, such inference systems are realized by accumulating a set of global constraints C. In BANE one can write rules as above, but the following alternative is also provided:

$$\frac{A, C_1 \vdash e_1 : \tau_1 \quad A, C_2 \vdash e_2 : \tau_2}{A, C_1 \wedge C_2 \vdash E[e_1, e_2] : \tau}$$

C_1 contains only the constraints required to type e_1 (similarly for C_2 and e_2). This structure has advantages. First, separate analysis of program components is trivial by design rather than added as an afterthought. Second, the running time of algorithms that examine the constraints (e.g., *constraint simplification*, which replaces constraint systems by equivalent, and smaller, systems) is guaranteed to be a function only of the expression being analyzed; in particular, the running time is independent of the rest of the program. Note that this design changes the primitive operation for accumulating constraints from adding individual constraints to a global system to combining independent constraint systems. Because this latter operation is more expensive, BANE applications tend to use a mixture of the two forms of rules to obtain good overall performance and scalability.

Many other aspects of the BANE architecture have been engineered primarily for scalability [FA96]. The emphasis on scalability, plus the overhead of supporting general user-specified constructor signatures, has a cost in runtime performance, but this cost appears to be small. For example, a BANE implementation of the type inference system for core Standard ML performs within a factor of two of the hand-written implementation in the SML/NJ compiler.

In other cases a well-engineered constraint library can substantially outperform hand-written implementations. BANE implementations of a class of cubic-time flow analyses can be orders of magnitude faster than special-purpose systems because of optimizations implemented in the solver for BANE's set constraint sort [FFSA98].

3 The BANE Interface by Example

This section presents a simple analysis written in BANE. We show by example how an analysis can be successively refined using mixed constraints. BANE is

a library written in Standard ML of New Jersey [MTH90]. Writing a program analysis using BANE requires ML code to traverse abstract syntax while generating constraints and ML code to extract the desired information from the solutions of the constraints.

For reasons of efficiency, BANE's implementation is stateful. BANE provides the notion of a *current constraint system* (CCS) into which all constraints are added. Functionality to create new constraint systems and to change the CCS are provided, so one is not limited to a single global constraint system. For simplicity, the examples in this section use only a single constraint system.

3.1 A Trivial Example: Simple Type Inference for a Lambda Calculus

This example infers types for a lambda calculus with the following abstract syntax:

```
datatype ast =
    Var of string
  | Int of int
  | Fn of {formal:string, body:ast}
  | App of {function:ast, argument:ast}
```

The syntax includes identifiers (strings), primitive integers, abstraction, and application. The language of types consists of the primitive type `int`, a function type $\rightarrow$, as well as type variables v.

$$\tau ::= v \mid \texttt{int} \mid \tau \rightarrow \tau$$

The first choice is the sort of expressions and constraints to use for the type inference problem. All that is needed in this case are terms and term equality; the appropriate sort is Term (structure `Bane.Term`). To make the code more readable, we rebind this structure as structure `TypeSort`.

```
structure TypeSort = Bane.Term
```

BANE uses distinct ML types for expressions of distinct sort. In this case, type expressions have ML type

```
type ty = TypeSort.T Bane.expr
```

Next, we need the type constructors for integers and functions. The integer type constructor can be formed using a constant signature, and a standard function type constructor is predefined.

```
val int_tycon = Cons.new {name="int", signa=TypeSort.constSig}
val fun_tycon = TypeSort.funCon
```

The constant integer type is created by applying the integer constructor to an empty list of arguments. We also define a function to apply the function type constructor to the domain and range, using the generic function `Bane.Common.cons : 'a constructor * genE list -> 'a expr` that applies a constructor of sort

$$\frac{}{A \vdash x : A[x]}\ [\text{VAR}] \qquad \frac{}{A \vdash i : \texttt{int}}\ [\text{INT}]$$

$$\frac{\alpha \text{ fresh} \quad A[x \mapsto \alpha] \vdash e : \tau}{A \vdash \lambda x.e : \alpha \to \tau}\ [\text{ABS}] \qquad \frac{\begin{array}{l}A \vdash e_1 : \tau_1 \\ A \vdash e_2 : \tau_2 \\ \alpha \text{ fresh} \\ \tau_1 = \tau_2 \to \alpha\end{array}}{A \vdash e_1\ e_2 : \alpha}\ [\text{APP}]$$

Fig. 3. Type inference rules for example lambda calculus

`'a` to a list of arguments. In general, constructor arguments can have a variety of distinct sorts with distinct ML types. Since ML only allows homogeneously typed lists, BANE uses an ML type `genE` for expressions of any sort. The lack of subtyping in ML forces us to use conversion functions `TypeSort.toGenE` to convert the domain and range from `TypeSort.T Bane.expr` to `Bane.genE`.

```
val intTy = Bane.Common.cons (int_tycon, [])
fun funTy (domain,range) = Common.cons (fun_tycon,
                                        [TypeSort.toGenE domain,
                                         TypeSort.toGenE range])
```

Finally, we define a function for creating fresh type variables by specializing the generic function `Bane.Var.freshVar : 'a Bane.sort -> 'a Bane.expr`. We also bind operator `==` to the equality constraint of `TypeSort`.

```
fun freshTyVar () = Bane.Var.freshVar TypeSort.sort
infix ==
val op == = TypeSort.unify
```

With these auxiliary bindings, the standard type inference rules in Figure 3 are translated directly into a case analysis on the abstract syntax. Type environments are provided by a module with the following signature:

```
signature ENV =
  sig
      type name = string

      type 'a env

      val empty  : 'a env
      val insert : 'a env * name * 'a -> 'a env
      val find   : 'a env * name -> 'a option
  end
```

The type of identifiers is simply looked up in the environment. If the environment contains no assumption for an identifier, an error is reported.

```
fun elaborate env ast =
  case ast of
```

```
  Var x => (case Env.find (env, x) of
              SOME ty => ty
            | NONE => <report error: free variable>)
```

The integer case is even simpler:

```
  | Int i => intTy
```

Abstractions are typed by creating a fresh unconstrained type variable for the lambda bound formal, extending the environment with a binding for the formal, and typing the body in the extended environment.

```
  | Fn {formal,body} =>
      let val v = freshTyVar ()
          val env' = Env.insert (env,formal,v)
          val body_ty = elaborate env' body
      in
          funTy (v, body_ty)
      end
```

For applications we obtain the function type `ty1` and the argument type `ty2` via recursive calls. A fresh type variable `result` stands for the result of the application. Type `ty1` must be equal to a function type with domain `ty2` and range `result`. The handler around the equality constraint catches inconsistent constraints in the case where `ty1` is not a function, or the domain and argument don't agree.

```
  | App {function,argument} =>
      let val ty1 = elaborate env function
          val ty2 = elaborate env argument

          val result = freshTyVar ()
          val fty = funTy (ty2, result)
      in
          (ty1 == fty) handle exn =>
                        <report type error>;
          result
      end
```

We haven't specified whether our type language for lambda terms includes recursive types. The Term sort allows recursive solutions by default. If only non-recursive solutions are desired, an occurs check can be enabled via a BANE option:

```
Bane.Flags.set (SOME TypeSort.sort) "occursCheck";
```

As an example, consider the Y combinator

$$\mathsf{Y} = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

Its inferred type is

$$(\alpha \to \alpha) \to \alpha$$

where the type variable α is unconstrained. With the occurs check enabled, type inference for Y fails.

3.2 Type Inference with Flow Information

The simple type inference described above yields type information for each lambda term or fails if the equality constraints have no solution. Suppose we want to augment type inference to gather information about the set of lambda abstractions to which each lambda expression may evaluate. We assume the abstract syntax is modified so that lambda abstractions are labeled:

```
  | Fn of {formal:string, body:ast, label:string}
```

Our goal is to refine function types to include a label-set, so that the type of a lambda term not only describes the domain and the range, but also an approximation of the set of syntactic abstractions to which it may evaluate. The function type constructor thus becomes a ternary constructor $\mathrm{fun}(dom, rng, labels)$. The resulting analysis is similar to the flow analysis described in [Mos96]. The natural choice of constraint language for label-sets is obviously set constraints, and we bind the structure `LabelSet` to one particular implementation of set constraints:

```
structure LabelSet = Bane.SetIF
```

We define the new function type constructor containing an extra field for the label-set by building a signature with three argument sorts, the first two being Type sorts and the last being a LabelSet sort. Note how the variance of each constructor argument is specified in the signature through the use of functions `TypeSort.ctv_arg` (contravariance) and `TypeSort.cov_arg` (covariance). Resolution of equality constraints itself does not require variance annotations, but other aspects of BANE do.

```
val funSig = TypeSort.newSig {args=[TypeSort.ctv_arg TypeSort.genSort,
                                    TypeSort.cov_arg TypeSort.genSort,
                                    TypeSort.cov_arg LabelSet.genSort],
                              attributes=[]}

val fun_tycon = Bane.Cons.new {name="fun", signa=funSig}
```

We are now using a mixed constraint language: types are terms with embedded label-sets. Constraints between types are still equality constraints, and as a result, induced constraints between label sets are also equalities.

The type rules for abstraction and application are easily modified to include label information.

$$\frac{\begin{array}{l}\alpha \text{ fresh} \\ A[x \mapsto \alpha] \vdash e : \tau \\ \{l\} \subseteq \epsilon \qquad \epsilon \text{ fresh}\end{array}}{A \vdash \lambda^l x.e : \mathrm{fun}(\alpha, \tau, \epsilon)} \quad [\mathrm{ABS}] \qquad \frac{\begin{array}{l}A \vdash e_1 : \tau_1 \\ A \vdash e_2 : \tau_2 \\ \alpha, \epsilon \text{ fresh} \\ \tau_1 = \mathrm{fun}(\tau_2, \alpha, \epsilon)\end{array}}{A \vdash e_1\ e_2 : \alpha} \quad [\mathrm{APP}]$$

Because `Term` constraints generate equality constraints on the embedded `Sets`, the label-sets of distinct abstractions may be equated during type inference. As a result, the [ABS] rule introduces a fresh label-set variable ϵ along with a constraint $\{l\} \subseteq \epsilon$ to correctly model that the lambda abstraction evaluates to

itself. (Note that this inclusion constraint is between `Set` expressions.) Using a constrained variable rather than a constant set $\{l\}$ allows the label-set to be merged with other sets through equality constraints. The handling of arrow-effects in region inference is similar [TT94].

The label-set variable ϵ introduced by each use of the [APP] rule stands for the set of abstractions potentially flowing to that application site.

The code changes required to accommodate the new rules are minimal. For abstractions, the label is converted into a constant set constructor with the same name through `Cons.new`. A constant set expression is then built from the constructor and used to constrain the fresh label-set variable `labelvar`. Finally, the label-set variable is used along with the domain and range to build the function type of the abstraction.

```
| Fn {formal,body,label} =>
    let val v = freshTyVar ()
        val env' = Env.insert (env,formal,v)
        val body_ty = elaborate env' body
        (* create a new constant constructor *)
        val c = Cons.new {name=label, signa=LabelSet.constSig}
        val lab = Common.cons (c,[])
        val labelvar = freshLabelVar ()
    in
        (lab <= labelvar);
        funTy (v, body_ty, labelvar)
    end
```

The changes to the implementation of [APP] are even simpler, requiring only the introduction of a fresh label-set variable. The label-set variable may be stored in a map for later inspection of the set of abstractions flowing to particular application sites.

```
| App {function,argument} =>
      let val ty1 = elaborate env function
          val ty2 = elaborate env argument

          val result = freshTyVar ()
          val labels = freshLabelVar ()
          val fty = funTy (ty2, result, labels)
      in
          (ty1 == fty) handle exn =>
                        <report type error>;
          result
      end
```

We now provide a number of examples showing the information gathered by the flow analysis. Consider the standard lambda encodings for values true, false,

nil, and cons, and their inferred types.

$$
\begin{array}{ll}
\mathsf{true} = \lambda^{\mathsf{true}} x.\lambda^{\mathsf{true}_1} y.x & \alpha \xrightarrow{\epsilon_1} \beta \xrightarrow{\epsilon_2} \alpha \setminus \mathsf{true} \subseteq \epsilon_1 \wedge \mathsf{true}_1 \subseteq \epsilon_2 \\
\mathsf{false} = \lambda^{\mathsf{false}} x.\lambda^{\mathsf{false}_1} y.y & \alpha \xrightarrow{\epsilon_1} \beta \xrightarrow{\epsilon_2} \beta \setminus \mathsf{false} \subseteq \epsilon_1 \wedge \mathsf{false}_1 \subseteq \epsilon_2 \\
\mathsf{nil} = \lambda^{\mathsf{nil}} x.\lambda^{\mathsf{nil}_1} y.x & \alpha \xrightarrow{\epsilon_1} \beta \xrightarrow{\epsilon_2} \alpha \setminus \mathsf{nil} \subseteq \epsilon_1 \wedge \mathsf{nil}_1 \subseteq \epsilon_2 \\
\mathsf{cons} = \lambda^{\mathsf{cons}} hd.\lambda^{\mathsf{c}_1} tl.\lambda^{\mathsf{c}_2} x.\lambda^{\mathsf{c}_3} y.y\ hd\ tl & \alpha \xrightarrow{\epsilon_1} \beta \xrightarrow{\epsilon_2} \gamma \xrightarrow{\epsilon_3} (\alpha \xrightarrow{\epsilon_4} \beta \xrightarrow{\epsilon_5} \delta) \xrightarrow{\epsilon_6} \delta \setminus \\
 & \mathsf{cons} \subseteq \epsilon_1 \wedge \mathsf{c}_1 \subseteq \epsilon_2 \wedge \\
 & \mathsf{c}_2 \subseteq \epsilon_3 \wedge \mathsf{c}_3 \subseteq \epsilon_6
\end{array}
$$

The analysis yields constrained types $\tau \setminus C$, where the constraints C describe the label-set variables embedded in type τ. (To improve the readability of types, function types are written using the standard infix form with label-sets on the arrow.) For example, the type of nil

$$\alpha \xrightarrow{\epsilon_1} \beta \xrightarrow{\epsilon_2} \alpha \setminus \mathsf{nil} \subseteq \epsilon_1 \wedge \mathsf{nil}_1 \subseteq \epsilon_2$$

has the label-set ϵ_1 on the first arrow, and associated constraint $\mathsf{nil} \subseteq \epsilon_1$. The label-set is extracted from the final type using the following BANE code fragment:

```
val ty = elaborate error baseEnv e
val labels = case Common.deCons (fun_tycon, ty) of
               SOME [dom,rng,lab] =>
                   LabelSet.tlb (LabelSet.fromGenE lab)
             | NONE => []
```

The function `Common.deCons` is used to decompose constructed expressions. In this case we match the final type expression against the pattern fun(dom, rng, lab). If the match succeeds, `deCons` returns the list of arguments to the constructor. In this case we are interested in the least solution of the label component `lab`. We obtain this information via the function `LabelSet.tlb`, which returns the *transitive lower-bound* (TLB) of a given expression. The TLB is a list of constructed expressions $c(\ldots)$, in our case a list of constants corresponding to abstraction labels.

A slightly more complex example using the lambda expressions defined above is

$$
\begin{array}{ll}
\mathsf{head} = \lambda^{\mathsf{head}} l.l\ \mathsf{nil}\ (\lambda^{\mathsf{head}_1} x.\lambda^{\mathsf{head}_2} y.x) & ((\alpha \xrightarrow{\epsilon_1} \iota_1 \xrightarrow{\epsilon_2} \alpha) \xrightarrow{\epsilon_3} \\
 & (\beta \xrightarrow{\epsilon_4} \iota_2 \xrightarrow{\epsilon_5} \beta) \xrightarrow{\epsilon_6} \gamma) \\
 & \xrightarrow{\epsilon_7} \gamma \setminus \\
 & \mathsf{head} \subseteq \epsilon_7 \wedge \\
 & \mathsf{nil} \subseteq \epsilon_1 \wedge \\
 & \mathsf{nil}_1 \subseteq \epsilon_2 \wedge \\
 & \mathsf{head}_1 \subseteq \epsilon_4 \wedge \\
 & \mathsf{head}_2 \subseteq \epsilon_5
\end{array}
$$

$$\mathsf{head}\ (\mathsf{cons}\ \mathsf{true}\ \mathsf{nil}) : \alpha \xrightarrow{\epsilon_1} \beta \xrightarrow{\epsilon_2} \alpha \setminus \mathsf{true} \subseteq \epsilon_1 \wedge \mathsf{true}_1 \subseteq \epsilon_2$$

The expression head (cons true nil) takes the head of the list containing true. Even though the function head is defined to return nil if the argument is the empty list, the flow analysis correctly infers that the result in this case is true.

The use of equality constraints may cause undesired approximations in the flow information. Consider an example taken from Section 3.1 of Mossin's thesis [Mos96]

$$\mathsf{select} = \lambda^{\mathsf{select}} x.\lambda^{\mathsf{sel}_1} y.\lambda^{\mathsf{sel}_2} f.\texttt{if } x \texttt{ then } f\ x \texttt{ else } f\ y$$

The select function takes three arguments, x, y, and z, and depending on the truth value of x, returns the result of applying f to either x or y. The abbreviation `if` p `then` e_1 `else` e_2 stands for the application $p\ e_1\ e_2$. The type constraints for the two applications of f cause the flow information of x and y to be merged. As a result, the application

$$\mathsf{select}\ \mathsf{true}\ \mathsf{false}\ (\lambda z.z)$$

does not resolve the condition of the if-then-else to true. To observe the approximation directly in the result type, we modify the example slightly:

$$\mathsf{select'} = \lambda^{\mathsf{select}} x.\lambda^{\mathsf{sel}_1} y.\lambda^{\mathsf{sel}_2} f.\texttt{if } x \texttt{ then } f\ x\ x \texttt{ else } f\ y\ x$$

Now f is applied to two arguments, the first being either x or y, the second being x in both cases. We modify the example use of select such that f now ignores its first argument and simply returns the second, i.e. x. The expression thus evaluates to true.

$$\mathsf{select'}\ \mathsf{true}\ \mathsf{false}\ (\lambda z.\lambda w.w)$$

The inferred type for this application is

$$\tau \setminus \begin{array}{l} \tau = \tau \xrightarrow{\epsilon_1} \tau \xrightarrow{\epsilon_2} \tau \\ \mathsf{true} \cup \mathsf{false} \subseteq \epsilon_1 \\ \mathsf{true}_1 \cup \mathsf{false}_1 \subseteq \epsilon_2 \end{array}$$

where the label-set of the function type indicates that the result can be either true or false. This approximation can be overcome through the use of subtyping.

3.3 Type Inference with Flow Information and Subtyping

The inclusion relation on label-sets embedded within types can be lifted to a natural subtyping relation on structural types. This idea has been described in the context of control-flow analysis in [HM97], for a more general flow analysis in [Mos96], and for more general set expressions in [FA97]. A subtype-based analysis where sets are embedded within terms can be realized in BANE through the use of the `FlowTerm` sort. The `FlowTerm` sort provides inclusion constraints instead of equality for the same language and solution space as the `Term` sort. To take advantage of the extra precision of subtype inference in our example, we first change the `TypeSort` structure to use the `FlowTerm` sort.

```
structure TypeSort = Bane.FlowTerm
```

The definition of the function type constructor with labels remains the same, although the domain and range are now of sort `FlowTerm`.

```
val funSig = TypeSort.newSig {args=[TypeSort.ctv_arg TypeSort.genSort,
                                    TypeSort.cov_arg TypeSort.genSort,
                                    TypeSort.cov_arg LabelSet.genSort],
                              attributes=[]}

val fun_tycon = Bane.Cons.new {name="fun", signa=funSig}
```

The inference rules for abstraction and application change slightly. In the [ABS] rule, it is no longer necessary to introduce a fresh label-set variable, since label sets are no longer merged in the subtype approach. Instead the singleton set can be directly embedded within the function type. In the [APP] rule, we simply replace the equality constraint with an inclusion.

$$\frac{A[x \mapsto \alpha] \vdash e : \tau}{A \vdash \lambda^l x.e : \mathrm{fun}(\alpha, \tau, \{l\})} \quad [\mathrm{ABS}] \qquad \frac{\begin{array}{l} A \vdash e_1 : \tau_1 \\ A \vdash e_2 : \tau_2 \\ \tau_1 \subseteq \mathrm{fun}(\tau_2, \alpha, \epsilon) \end{array}}{A \vdash e_1\ e_2 : \alpha} \quad [\mathrm{APP}]$$

Note that the inclusion constraint in the [APP] rule allows subsumption not only on the label-set of the function, but also on the domain and the range, since

$$\mathrm{fun}(\mathit{dom}, \mathit{range}, \mathit{labels}) \subseteq \mathrm{fun}(\tau_2, \alpha, \epsilon) \Leftrightarrow \begin{array}{l} \tau_2 \subseteq \mathit{dom} \wedge \\ \mathit{range} \subseteq \alpha \wedge \\ \mathit{labels} \subseteq \epsilon \end{array}$$

We return to the example of the previous section where flow information was merged:

$$\mathsf{select'}\ \mathsf{true}\ \mathsf{false}\ (\lambda z.\lambda w.w)$$

Using subtype inference, the type of this expression is

$$\tau \setminus \begin{array}{l} \tau = \tau \xrightarrow{\epsilon_1} \tau \xrightarrow{\epsilon_2} \tau \\ \mathsf{true} \subseteq \epsilon_1 \\ \mathsf{true}_1 \subseteq \epsilon_2 \end{array}$$

The flow information now precisely models the fact that only true is passed as the second argument to $\lambda z.\lambda w.w$.

4 Analysis Frameworks

We conclude by comparing BANE with other program analysis frameworks. There have been many such frameworks in the past; see for example [ATGL96,AM95,Ass96,CDG96,DC96,HMCCR93,TH92,Ven89,YH93]. Most frameworks are based on standard dataflow analysis, as first proposed by Cocke [Coc70] and developed by Kildall [Kil73] and Kam and Ullman [KU76], while others are based on more general forms of abstract interpretation [Ven89,YH93].

In previous frameworks the user specifies a lattice and a set of transfer functions, either in a specialized language [AM95], in a Yacc-like system [TH92], or as a module conforming to a certain interface [ATGL96,CDG96,DC96,HMCCR93]. The framework traverses a program representation (usually a control flow graph) either forwards or backwards, calling user-defined transfer functions until the analysis reaches a fixed point.

A fundamental distinction between BANE and these frameworks is the interface with a client analysis. In BANE, the interface is a system of constraints, which is an explicit data structure that the framework understands and can inspect and transform for best effect. In other frameworks the interface is the transfer and lattice functions, all of which are defined by the client. These functions are opaque—their effect is unknown to the framework—which in general means that the dataflow frameworks have less structure that can be exploited by the implementation. For example, reasoning about termination of the framework is impossible without knowledge of the client. Additionally, using transfer functions implies that information can flow conveniently only in one direction, which gives rise to the restriction in dataflow frameworks that analyses are either forwards or backwards. An analysis that is neither forwards nor backwards (e.g., most forms of type inference) is at best awkward to code in this model.

On the other hand, dataflow frameworks provide more support for the task of implementing traditional dataflow analyses than BANE, since they typically manage the control flow graph and its traversal as well as the computation of abstract values. With BANE the user must write any needed traversal of the program structure, although this is usually a simple recursive walk of the abstract syntax tree. Since BANE has no knowledge of the program from which constraints are generated, BANE cannot directly exploit any special properties of program structure that might make constraint solving more efficient.

While there is very little experimental evidence on which to base any conclusion, it is our impression that an analysis implemented using the more general frameworks with user-defined transfer functions suffers a significant performance penalty (perhaps an order of magnitude) compared with a special-purpose implementation of the same analysis. Note that the dataflow frameworks target a different class of applications than BANE, and we do not claim that BANE is particularly useful for traditional dataflow problems. However, as discussed in Section 2.2, we do believe for problems with a natural type or constraint formulation that BANE provides users with significant benefits in development time together with good scalability and good to excellent performance compared with hand-written implementations of the same analyses.

5 Conclusions

BANE is a toolkit for constructing type- and constraint-based program analyses. An explicit goal of the project is to make realistic experimentation with program analysis ideas much easier than is now the case. We hope that other researchers

find BANE useful in this way. The BANE distribution is available on the World Wide Web from `http://bane.cs.berkeley.edu`.

References

[AFS98] A. Aiken, M. Fähndrich, and Z. Su. Detecting Races in Relay Ladder Logic Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 4th International Conference, TACAS'98*, volume 1384 of *LNCS*, pages 184–200, Lisbon, Portugal, 1998. Springer.

[AM95] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. *Lecture Notes in Computer Science*, 983:33–50, 1995.

[And94] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Cophenhagen, May 1994.

[Ass96] U. Assmann. How to Uniformly Specify Program Analysis and Transformation with Graph Rewrite Systems. In *Proceedings of the Sixth International Conference on Compiler Construction (CC '96)*, pages 121–135. Springer-Verlag, April 1996.

[ATGL96] A. Adl-Tabatabai, T. Gross, and G. Lueh. Code Reuse in an Optimizing Compiler. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '96)*, pages 51–68, October 1996.

[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, January 1994.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Contruction or Approximation of Fixed Points. In *Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[CDG96] C. Chambers, J. Dean, and D. Grove. Frameworks for Intra- and Interprocedural Dataflow Analysis. Technical Report 96-11-02, Department of Computer Science and Engineering, University of Washington, November 1996.

[Coc70] J. Cocke. Global Common Subexpression Elimination. *ACM SIGPLAN Notices*, 5(7):20–24, July 1970.

[DC96] M. Dwyer and L. Clarke. A Flexible Architecture for Building Data Flow Analyzers. In *Proceedings of the 18th International Conference on Software Engineering (ICSE-18)*, Berlin, Germany, March 1996.

[DM82] L. Damas and R. Milner. Principle Type-Schemes for Functional Programs. In *Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound Polymorphic Type Inference for Objects. In *OOPSLA '95*, pages 169–184, 1995.

[FA96] M. Fähndrich and A. Aiken. Making Set-Constraint Based Program Analyses Scale. In *First Workshop on Set Constraints at CP'96*, Cambridge, MA, August 1996. Available as Technical Report CSD-TR-96-917, University of California at Berkeley.

[FA97] M. Fähndrich and A. Aiken. Program Analysis Using Mixed Term and Set Constraints. In *Proceedings of the 4th International Static Analysis Symposium*, pages 114–126, 1997.

[FFA97] J. Foster, M. Fähndrich, and A. Aiken. Flow-Insensitive Points-to Analysis with Term and Set Constraints. Technical Report UCB//CSD-97-964, University of California, Berkeley, July 1997.

[FFA98] M. Fähndrich, J. Foster, and A. Aiken. Tracking down Exceptions in Standard ML Programs. Technical Report UCB/CSD-98-996, EECS Department, UC Berkeley, February 1998.

[FFK+96] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching Bugs in the Web of Program Invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[FFSA98] M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, 1998.

[GJSO92] D. Gifford, P. Jouvelot, M. Sheldon, and J. O'Toole. Report on the FX-91 Programming Language. Technical Report MIT/LCS/TR-531, Massachusetts Institute of Technology, February 1992.

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

[Hen92] F. Henglein. Global Tagging Optimization by Type Inference. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 205–215, July 1992.

[HM97] N. Heintze and D. McAllester. Linear-Time Subtransitive Control Flow Analysis. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

[HMCCR93] M. Hall, J. Mellor-Crummey, A. Carle, and R. Rodríguez. FIAT: A Framework for Interprocedural Analysis and Transformation. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the 6th International Workshop on Parallel Languages and Compilers*, pages 522–545, Portland, Oregon, August 1993. Springer-Verlag.

[Kil73] G. A. Kildall. A Unified Approach to Global Program Optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, MA, October 1973. ACM, ACM.

[KU76] J. Kam and J. Ullman. Global Data Flow Analysis and Iterative Algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[Mos96] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996.

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

[Rém89] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas*, pages 60–76, January 1989.

[Ste96] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[TH92] S. Tjiang and J. Hennessy. Sharlit – A tool for building optimizers. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 82–93, July 1992.

[TT94] M. Tofte and J.-P. Talpin. Implementation of the Typed Call-by-Value λ-Calculus using a Stack of Regions. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 188–201, 1994.

[Ven89] G. A. Venkatesh. A framework for construction and evaluation of high-level specifications for program analysis techniques. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 1–12, 1989.

[YH93] K. Yi and W. Harrison, III. Automatic Generation and Management of Interprocedural Program Analyses. In *Proceedings of the Twnetieth Annual ACM Symposium on Principles of Programming Languages*, pages 246–259, January 1993.