# Type-Based Analysis and Applications

Jens Palsberg

Purdue University
Dept. of Computer Science
West Lafayette, IN 47907
palsberg@cs.purdue.edu

## ABSTRACT

Type-based analysis is an approach to static analysis of programs that has been studied for more than a decade. A type-based analysis assumes that the program type checks, and the analysis takes advantage of that. This paper examines the state of the art of type-based analysis, and it surveys some of the many software tools that use type-based analysis. Most of the surveyed tools use types as discriminators, while most of the theoretical studies use type and effect systems. We conclude that type-based analysis is a promising approach to achieving both provable correctness and good performance with a reasonable effort.

## 1. INTRODUCTION

This paper is a survey of the theory and practice of type-based analysis. It tries to answer the following questions:

- What is a type-based analysis?
- What are the advantages of type-based analysis?
- Is type-based analysis competitive with other approaches to static analysis?
- Which tools use type-based analysis?
- What is the current spectrum of type-based analyses?

As background for examining these questions, let us begin with a brief overview of some of the past successes and future challenges of the broader field of static analysis.

Traditionally, optimizing compilers were the main consumers of static analyses. Classical examples of static analyses are liveness analysis (for doing, e.g., register allocation) and data-flow analysis (for doing, e.g., common-subexpression elimination). Many textbooks on compiler design, including [2, 4, 44], contain substantial coverage of how to define, implement, and use static analyses in compilers. There are books devoted entirely to static analysis, including [48], and there are annual international conferences, such as the Static Analysis Symposium, for presentation and discussion of advances in the area.

Nowadays, static analysis is also used by tools for various software engineering tasks such as program understanding [74, 28], debugging [14], testing [57], and reverse engineering [22]. Among the international conferences that cover the application of static analysis to the area of software engineering is the International Symposium on Software Testing and Analysis.

In the coming years, there are new challenges for static analysis. For example, there is an increasing need for verifying key properties of software, including real-time properties, security-related behavior, and power consumption. The emerging paradigm [12] of combining model extraction (based on static analysis) and model checking is promising for this purpose. Moreover, the notion of dynamic class loading that has been popularized by Java has led to increased interest in run-time compilation and therefore also in highly efficient static analyses. Finally, work on scalable static analysis is needed to enable the application of static analysis to larger and larger programs.

Does the field of static analysis have a realistic hope of being able to help address the software problems of today and tomorrow? The properties that need to be analyzed are increasingly complex, and the programs being analyzed are ever larger. When push comes to shove, will static analysis measure up? I believe the answer to both questions is Yes.

One of the key reasons for optimism comes from the field of programming language design and the growing popularity of static type checking [10]. In the 1990s, most new software was written in languages such as C [33], C++ [17], and Java [23] which all feature varying degrees of static type checking. In particular, the type system of Java has received considerable attention, and for substantial subsets of Java, there are automatically-checked proofs of type soundness, e.g., [50]. The trend of typeful programming seems to continue, and types are now also being used in the intermediate languages of compilers, including the Java VML, and even in assembly languages [41, 40]. Traditionally, compilers would apply static analyses to untyped intermediate representations of programs, and so these analyses worked for all programs, whether typable or not. Now, an increasing number of static analyses are defined on statically-typed representations of programs, such as Java bytecodes. Of course, a static analysis can simply ignore the types, and some of them do. However, the presence of types has led researchers


*PASTE'01,* June 18–19, 2001, Snowbird, Utah, USA..

to ask:

> **Question:** Can a static analysis take advantage of that the program type checks?

In particular, can the types help with defining more complicated analyses, can they help with reasoning about the correctness of an analysis, and can they help with making the static analyses more efficient? These questions have been asked since, at least, the 1980s, and after the work of many researchers, it can be concluded that the answer to each of the questions is Yes. As a result, there is an emerging field of type-based analysis:

> **Terminology:** A *type-based analysis* assumes that the program type checks, and the analysis takes advantage of that.

This paper examines the state of the art of type-based analysis. We start with an example that illustrates what type-based analysis is and isn't, and we then discuss the advantages of type-based analysis, survey some of the many software tools that use type-based analysis, and map out the landscape of type-based analysis.

# 2. EXAMPLE

Let us consider a classical static-analysis problem: flow analysis for the $\lambda$-calculus. We will present four well-known static analyses for solving this problem: one that does not rely on types, and three type-based analyses. The goal is to illustrate various advantages of type-based analysis.

We use $x$ to range over program variables, we use $l$ to range over labels, and we use the following grammar to define a language of $\lambda$-terms:

$$e \quad ::= \quad x \mid \lambda^l x.e \mid e_1 e_2.$$

The goal of a flow analysis of a program $E$ is to approximate, for each subexpression $e$ of $E$, the set of labels (called *flow set*) of the abstractions $\lambda^l x.e'$ that are the possible values of $e$. The flow analysis must be conservative, that is, if $\lambda^l x.e'$ is a possible value of $e$, then $l$ must be in the flow set for $e$.

As a running example, we will use the $\lambda$-term

$$F = ((\lambda^1 f.\lambda^2 x.fx)(\lambda^3 a.a))(\lambda^4 b.b).$$

If we do $\beta$-reduction of $F$, then we get:

$$\begin{array}{lll} F & \rightarrow_\beta & (\lambda^2 x.((\lambda^3 a.a)\ x))\ (\lambda^4 b.b) \\ & \rightarrow_\beta & (\lambda^3 a.a)\ (\lambda^4 b.b) \\ & \rightarrow_\beta & \lambda^4 b.b. \end{array}$$

Hence, $\lambda^4 b.b$ is a possible value of $F$, so any sound flow analysis of $F$ must produce a flow set for $F$ that contains the label 4.

## 2.1 0-CFA

One can define a flow analysis for a $\lambda$-term $E$ by using a flow graph in which the nodes are the expressions occurring in $E$. The edges in the flow graph are generated from the following four rules (taken from [27]):

$$\lambda^l x.e \rightarrow \lambda^l x.e \tag{1}$$

$$\frac{e_1 \rightarrow \lambda^l x.e}{x \rightarrow e_2} \quad (e_1 e_2 \text{ occurs in } E) \tag{2}$$

$$\frac{e_1 \rightarrow \lambda^l x.e}{e_1 e_2 \rightarrow e} \quad (e_1 e_2 \text{ occurs in } E) \tag{3}$$

$$\frac{e_1 \rightarrow e_2 \qquad e_2 \rightarrow e_3}{e_1 \rightarrow e_3} \tag{4}$$

This analysis is conservative and works for all $\lambda$-terms. The idea is that if we analyze an expression $E$, then, for any subexpression $e$ of $E$, the flow set for $e$ is the set of abstractions $\lambda^l x.e'$ such that there is an edge $e \rightarrow \lambda^l x.e'$ in the flow graph. For the running example $F$, we can use Rules (1)–(4) to generate the edges:

$$\begin{array}{l} f \rightarrow \lambda^3 a.a \\ (\lambda^1 f.\lambda^2 x.fx)(\lambda^3 a.a) \rightarrow \lambda^2 x.fx \\ F \rightarrow fx \rightarrow a \rightarrow x \rightarrow \lambda^4 b.b, \end{array}$$

so by transitivity (Rule (4)), we have $F \rightarrow \lambda^4 b.b$.

The above analysis uses a style that is widely known as 0-CFA [59]. It can be executed in $O(n^3)$ time, where $n$ is the size of the program [55], and can be proved correct with respect to arbitrary $\beta$-reduction [52].

## 2.2 A Simple Type System

Below we present three type-based analyses that all assume that the program being analyzed is simply typed, that is, it obeys the following type discipline. We use $\alpha$ to range over type variables, and we use the following grammar to define types:

$$t \quad ::= \quad \alpha \mid t \rightarrow t.$$

A type environment is a partial function from program variables to types, and we use the notation $A[x : t]$ to denote a type environment which maps $x$ to $t$, and otherwise maps $y$ to $A(y)$ when $x \neq y$. The type rules are:

$$A \vdash x : t \qquad (A(x) = t) \tag{5}$$

$$\frac{A[x : s] \vdash e : t}{A \vdash \lambda^l x.e : s \rightarrow t} \tag{6}$$

$$\frac{A \vdash e_1 : s \rightarrow t \qquad A \vdash e_2 : s}{A \vdash e_1 e_2 : t} \tag{7}$$

For the running example $F$, we can use Rules (5)–(7) to construct a type derivation which contains the judgments:

$$\begin{array}{l} \emptyset \vdash \lambda^1 f.\lambda^2 x.fx : ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \rightarrow \\ \qquad\qquad ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \\ \emptyset[f : (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)] \vdash \lambda^2 x.fx : \\ \qquad\qquad (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \\ \emptyset \vdash \lambda^3 a.a : (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \\ \emptyset \vdash \lambda^4 b.b : \alpha \rightarrow \alpha \\ \emptyset \vdash F : \alpha \rightarrow \alpha. \end{array}$$

## 2.3 A Type and Effect System

The first type-based analysis is a so-called type and effect system. It uses the types and the type rules in a rather direct way (much like in [26]). The idea is to annotate the function types with a flow set $\varphi$. Thus, annotated types are defined by the grammar:

$$t \quad ::= \quad \alpha \mid t \xrightarrow{\varphi} t.$$

The revised type rules are:

$$A \vdash x : t \qquad (A(x) = t) \tag{8}$$

$$\frac{A[x : s] \vdash e : t}{A \vdash \lambda^l x.e : s \xrightarrow{\varphi} t} \qquad (l \in \varphi) \tag{9}$$

$$\frac{A \vdash e_1 : s \xrightarrow{\varphi} t \qquad A \vdash e_2 : s}{A \vdash e_1 e_2 : t} \tag{10}$$

Notice that Rule (9) enforces that the function type $s \xrightarrow{\varphi} t$ "remembers" the label $l$ by having the side condition $l \in \varphi$. The idea is that if we have the judgment $A \vdash e : s \xrightarrow{\varphi} t$, then the flow set for $e$ is $\varphi$. For the running example $F$, we can use Rules (8)–(10) to construct a type derivation which contains the judgments:

$$\begin{aligned}
&\emptyset \vdash \lambda^1 f.\lambda^2 x.fx : ((\alpha \xrightarrow{\{4\}} \alpha) \xrightarrow{\{3\}} (\alpha \xrightarrow{\{4\}} \alpha)) \xrightarrow{\{1\}} \\
&\qquad\qquad ((\alpha \xrightarrow{\{4\}} \alpha) \xrightarrow{\{2\}} (\alpha \xrightarrow{\{4\}} \alpha)) \\
&\emptyset[f : (\alpha \xrightarrow{\{4\}} \alpha) \xrightarrow{\{3\}} (\alpha \xrightarrow{\{4\}} \alpha)] \vdash \lambda^2 x.fx : \\
&\qquad\qquad (\alpha \xrightarrow{\{4\}} \alpha) \xrightarrow{\{2\}} (\alpha \xrightarrow{\{4\}} \alpha) \\
&\emptyset \vdash \lambda^3 a.a : (\alpha \xrightarrow{\{4\}} \alpha) \xrightarrow{\{3\}} (\alpha \xrightarrow{\{4\}} \alpha) \\
&\emptyset \vdash \lambda^4 b.b : \alpha \xrightarrow{\{4\}} \alpha \\
&\emptyset \vdash F : \alpha \xrightarrow{\{4\}} \alpha,
\end{aligned}$$

so the flow set for $F$ is $\{4\}$.

## 2.4 A Sparse-Flow-Graph Approach

The second type-based analysis for a $\lambda$-term $E$ uses a *sparse* flow graph and avoids transitive closure [27]. All potential nodes in the flow graph are defined by the grammar:

$$n \quad ::= \quad e \mid \mathrm{dom}(n) \mid \mathrm{ran}(n),$$

where $e$ occurs in the program being analyzed. The edges in the flow graph are generated from the rules (taken from [27]):

$$x \to \mathrm{dom}(\lambda^l x.e) \qquad (\lambda^l x.e \text{ occurs in } E) \tag{11}$$

$$\mathrm{ran}(\lambda^l x.e) \to e \qquad (\lambda^l x.e \text{ occurs in } E) \tag{12}$$

$$e_1 e_2 \to \mathrm{ran}(e_1) \qquad (e_1 e_2 \text{ occurs in } E) \tag{13}$$

$$\mathrm{dom}(e_1) \to e_2 \qquad (e_1 e_2 \text{ occurs in } E) \tag{14}$$

$$\frac{n_1 \to n_2 \qquad n \to \mathrm{ran}(n_1)}{\mathrm{ran}(n_1) \to \mathrm{ran}(n_2)} \tag{15}$$

$$\frac{n_1 \to n_2 \qquad n \to \mathrm{dom}(n_2)}{\mathrm{dom}(n_2) \to \mathrm{dom}(n_1)} \tag{16}$$

The idea is that if we analyze an expression $E$, then, for any subexpression $e$ of $E$, the flow set for $e$ is the set of abstractions $\lambda^l x.e'$ such that there is a path $e \to^* \lambda^l x.e'$ in the flow graph. For the running example $F$, we can use Rules (11)–(16) to generate the edges:

$$\begin{aligned}
&f \to \mathrm{dom}(\lambda^1 f.\lambda^2 x.fx) \to \lambda^3 a.a \\
&(\lambda^1 f.\lambda^2 x.fx)(\lambda^3 a.a) \to \mathrm{ran}(\lambda^1 f.\lambda^2 x.fx) \to \lambda^2 x.fx \\
&F \to \mathrm{ran}((\lambda^1 f.\lambda^2 x.fx)(\lambda^3 a.a)) \to \mathrm{ran}(\mathrm{ran}(\lambda^1 f.\lambda^2 x.fx)) \\
&\quad \to \mathrm{ran}(\lambda^2 x.fx) \to fx \to \mathrm{ran}(f) \\
&\quad \to \mathrm{ran}(\mathrm{dom}(\lambda^1 f.\lambda^2 x.fx)) \to \mathrm{ran}(\lambda^3 a.a) \to a \\
&\quad \to \mathrm{dom}(\lambda^3 a.a) \to \mathrm{dom}(\mathrm{dom}(\lambda^1 f.\lambda^2 x.fx)) \to \mathrm{dom}(f) \\
&\quad \to x \to \mathrm{dom}(\lambda^2 x.fx) \to \mathrm{dom}(\mathrm{ran}(\lambda^1 f.\lambda^2 x.fx)) \\
&\quad \to \mathrm{dom}((\lambda^1 f.\lambda^2 x.fx)(\lambda^3 a.a)) \\
&\quad \to \lambda^4 b.b,
\end{aligned}$$

so the flow set for $F$ is $\{4\}$.

The building of the flow graph may diverge for some $\lambda$-terms. In can be shown that if a $\lambda$-term is simply typed, then the flow graph will be finite, sparse, and built in finite time, and the produced flow information will be the same as that produced by 0-CFA. Moreover, if the sizes of the types are independent of the size of the program, as is often the case in practice, then the flow information can be computed in $O(n^2)$ time, which is an improvement over the $O(n^3)$ time spent by 0-CFA.

## 2.5 A Types-as-Discriminators Approach

The third type-based analysis uses the types as discriminators. To compute a flow set for an expression $e$ in a program $E$, the analysis concentrates on the *type* of $e$, and asks which abstractions in $E$ have the same type as $e$. The set of labels of those abstractions is the flow set for $e$.

For the running example $F$, the type of $F$ itself is $\alpha \to \alpha$. There is exactly one abstraction in $F$ which has type $\alpha \to \alpha$, namely $\lambda^4 b.b$, so the flow set for $F$ is $\{4\}$.

# 3. ADVANTAGES OF TYPE-BASED ANALYSIS

The example above illustrates the main advantages of type-based analysis, all revolving around the issues of simplicity, efficiency, and correctness. We discuss these advantages in more detail here and we consider whether type-based analysis is competitive with other approaches to static analysis.

## 3.1 Simplicity

Types provide an infrastructure on top of which analyses can be built. For example, the type and effect system in Section 2 illustrates the idea of *annotated types*, that is, the decoration of types with static information. Conceptually, the starting point for a type-based analysis is a type derivation for a program, not just a syntax tree. Such a type derivation is a convenient basis for designing static analyses. If the goal is to design a type and effect system, then each type rule provides a localized setting for thinking about the analysis of a single language construct. If the goal is to design an analysis using types as discriminators, then the type derivation contains the needed types.

## 3.2 Efficiency

Many researchers have observed that executing a static analysis on a typical statically-typed program tends to be faster and give qualitatively better results than executing the same style of analysis on a typical dynamically-typed

program. The reason seems to be that statically-typed programs are inherently more structured and therefore easier to analyze. The field of type-based analysis goes further by trying to get additional benefits from the types. The sparse-flow-graph approach above is an example of how the mere existence of types can help with computing static information faster. The types-as-discriminators approach is particularly efficient for a language with declared types because the needed type information is readily available in the program text.

### 3.3 Correctness

The correctness of a type system with respect to a semantics is usually phrased as a type soundness theorem: well-typed programs cannot go wrong [38]. The correctness of a type and effect system can similarly be phrased as a type soundness theorem; the correctness of the analysis is subsumed by the correctness of the annotated-type system. There is a well-understood method for proving type soundness [46, 77] based on proving type preservation and progress, and this method usually carries over to type and effect systems.

### 3.4 Competitiveness

Among the main approaches to static analysis are data flow analysis, constraint-based analysis, and abstract interpretation [48]. Many researchers, including Nielson, Nielson, and Hankin [48], have observed that there are important similarities between these approaches. Many type inference problems that are specified using type rules can be turned into equivalent constraint problems that are suitable for algorithmic considerations. Similarly, one can view a type and effect system as a specification of an analysis, which in turn can be transformed into a constraint problem. The type and effect system may be easier to formulate and reason about, and the constraint problem may be more appropriate when designing an algorithm to carry out the analysis.

One inherent advantage of type-based analysis is that it enables the definition of abstract domains in terms of types. The types-as-discriminators approach can be viewed as doing that by dividing the abstractions into equivalence classes based on the types.

The types can be of help when comparing two type-based analyses for the same language. The types are a *lingua franca* that is "spoken" by both analyses, and this may be of help when trying to identify similarities and differences.

## 4. TOOLS THAT USE TYPE-BASED ANALYSIS

We now survey some of the tools that successfully use type-based analysis. The tools work on programs written in C++ [17], Java [23], Modula 3 [11, 45], and Standard ML [39].

### 4.1 Method Inlining

In an object-oriented program we may have a virtual call site $e.m(\ldots)$. If a static analysis can determine a conservative approximation of the set of methods that can be invoked, then a compiler may be able to inline the call. One of the fundamental type-based analyses of object-oriented programs for doing that is the Class Hierarchy Analysis (CHA) of Dean, Grove, and Chambers [13]. In the terminology of Section 2, CHA uses types as discriminators to achieve good precision. We will use the notation $\mathit{StaticType}(e)$ to denote the static type of the expression $e$, $\mathit{SubTypes}(t)$ to denote the set of declared subtypes of type $t$, and the notation $\mathit{StaticLookup}(C, m)$ to denote the definition (if any) of a method with name $m$ that one finds when starting a static method lookup in the class $C$. For the virtual call site $e.m(\ldots)$, and each class $C \in \mathit{SubTypes}(\mathit{StaticType}(e))$ where $\mathit{StaticLookup}(C, m) = M'$, CHA determines that $M'$ is a method that can be invoked. Notice how the static type of $e$ is used to restrict attention to only some of the classes in the program.

We can extend CHA to take class-instantiation information into account. The result is known as *rapid type analysis* (RTA), and was first described by Bacon and Sweeney [5, 6]. The idea is to first collect the set $S$ of all classes $C$ for which there is an occurrence of "new C()" in the program. Then, for the virtual call site $e.m(\ldots)$, and each class $C \in \mathit{SubTypes}(\mathit{StaticType}(e))$ where $\mathit{StaticLookup}(C, m) = M'$ and $C \in S$, RTA determines that $M'$ is a method that can be invoked. Notice that a class is only taken into account if at least one object of that class may exist at run time.

One can go further and associate a single distinct set (like $S$) with each class, method, and/or field in an application [71]. If one associates a set with each expression, then the result is 0-CFA [54].

Sundaresan et al. [64] use a combination of a type-based analysis (either CHA or RTA) and a more traditional 0-CFA-like analysis in the following way. First they perform, say, RTA to determine a call graph approximation, and then they use a 0-CFA-like technique to propagate class information along the edges of that call graph. This turns out to be fast and give good results.

A weakness of RTA and other whole-program analyses is that they have problems with library-based applications for which the code of the library is not available at the time of the analysis. To overcome that, the application extractor Jax features a specification language that allows users to specify, at a high level, how to extract a library-based application [65]. The idea is that a specification tells Jax what to expect from the library.

The Swift compiler by Ghemawat, Randall, and Scales [21] has a front end which compiles Java to an typed intermediate representation that uses annotated Java types. The annotations can express such things as: the value is known to be an object of exactly a particular class (and not a subclass), the value is an array with a particular constant size, and the value is known to be non-null. The backend of the compiler uses the annotations for method inlining and other optimizations.

### 4.2 Application Extraction

For the purpose of extraction applications, the goal is to compute a conservative approximation of the set of methods that are reachable from the main method. It is straightforward to extend the basic formulations of CHA and RTA with a form of reachability analysis. The following set-constraint formulation of a version of CHA, borrowed from [71], uses a single set variable $R$ (for "reachable methods") that ranges of sets of methods. The constraints are derived from the program text in the following way:

1. $\mathit{main} \in R$ ($\mathit{main}$ denotes the main method)

2. For each method $M$, each virtual call site $e.m(\ldots)$ occurring in $M$, and each class $C \in SubTypes(StaticType(e))$ where $StaticLookup(C, m) = M'$:
$(M \in R) \Rightarrow (M' \in R)$.

Intuitively, the first constraint reads "the main method is reachable," and the second constraint reads: "if a method is reachable, and a virtual method call $e.m(\ldots)$ occurs in the body of that method, then every method with name $m$ that is inherited by a subtype of the static type of $e$ is also reachable." It is straightforward to show that there is a least set $R$ that satisfies the constraints, and a solution procedure that computes that set. The reason for computing the least $R$ that satisfies the constraints is that this maximizes the complement of $R$, i.e., the set of unreachable methods that can be removed safely.

RTA extended with reachability analysis uses both a set variable $R$ ranging over sets of methods, and a set variable $S$ which ranges over sets of class names. The variable $S$ approximates the set of classes for which objects are created during a run of the program. The constraints:

1. $main \in R$ ($main$ denotes the main method)
2. For each method $M$, each virtual call site $e.m(\ldots)$ occurring in $M$, and each class $C \in SubTypes(StaticType(e))$ where $StaticLookup(C, m) = M'$:
$(M \in R) \wedge (C \in S) \Rightarrow (M' \in R)$.
3. For each method $M$, and for each "new $C$()" occurring in $M$:
$(M \in R) \Rightarrow (C \in S)$.

Intuitively, the second constraint refines the corresponding constraint of CHA by insisting that $C \in S$, and the third constraint reads: "$S$ contains the classes that are instantiated in a reachable method."

RTA is easy to implement, scales well, and has been shown to compute call graphs that are significantly more precise than those computed by CHA [5]. There are several whole-program analysis systems that rely on RTA to compute call graphs (e.g., the Jax application extractor of [70].)

It turns out that for detecting unreachable methods, the inexpensive RTA does almost as well as when using a distinct set with each class, method, and/or field in an application [71].

## 4.3 Redundant-Load Elimination

Redundant-load elimination is a compile-time optimization that combines loop-invariant code motion and common-subexpression elimination. Since it is trying to reorder statements that may do pointer accesses, redundant-load elimination can benefit from alias information. Two access paths are said to be possibles aliases if they may refer to the same variable. Diwan, McKinley, and Moss [16] have presented three type-based alias analyses, all based on the idea of using types as discriminators. The most basic one, called TypeDecl, observes that two expressions $e_1$ and $e_2$ cannot be aliases if

$$SubTypes(StaticType(e_1)) \cap SubTypes(StaticType(e_2)) = \emptyset.$$

Their second analysis, called FieldTypeDecl, further distinguishes expressions based on observations such as: two expressions $e_1.f$ and $e_2.g$ cannot be aliases if $f \neq g$.

Their third analysis, called Selectively Merge Type References (SMTypeRefs), goes further by including a type-based flow analysis. The idea is that two expressions $e_1$ and $e_2$ cannot be aliases if the program never assigns an object of type $StaticType(e_1)$ to a reference of type $StaticType(e_2)$, or vice versa. Thus, the type-based flow analysis records the types involved in all assignments, parameter passings, and return of results, and computes an approximation of the possible flow between references of different types.

Experiments [16] show that both FieldTypeDecl and SMTypeRefs are good bases for doing redundant-load elimination, while TypeDecl seems to be too imprecise to get good results.

Fink, Knobe, and Sarkar [18] used a flow-sensitive version of FieldTypeDecl in their implementation of redundant-load and dead-store elimination.

Hosking, Nystrom, Whitlock, Cutts, and Diwan [29] have presented an approach to partial-redundancy elimination which uses the FieldTypeDecl approach to type-based alias analysis. Their experience with the approach is mixed, although they conclude that the main problem is not the alias analysis but the isolation between their optimizer and the underlying execution environment.

## 4.4 Encapsulation Checking

In a Java package, there may classes with the property that no object of those classes will escape the package. In other words, the objects of those classes are *encapsulated* in the package. Bokowski and Vitek [9] called such classes *confined*, and they presented an extension of Java in which one can specify that a class is confined. Grothoff, Palsberg, and Vitek [24] presented a type-based analysis for identifying confined classes in Java bytecode. Their analysis is defined using constraints, which, in turn, rely on a flow analysis to determine a call-graph approximation. They use a type-based flow analysis akin to the SMTypeRefs mentioned above. In effect, the combined analysis does a low-cost escape analysis which turns out to identify a high number of confined classes in a large benchmark suite.

## 4.5 Race Detection

In a multi-threaded program, a race condition occurs when two threads manipulate a shared data structure simultaneously, without synchronization. This can result in unexpected program behavior. To avoid it, one can use a programming discipline where each data structure is protected with a lock that can be held by at most one thread at a time. Flanagan and Freund [20] have presented a type-based analysis that detects race conditions in Java programs. The analysis is presented as a type and effect system. The current implementation requires adding some type annotations to the Java code. It remains open whether the type annotations can be computed by an analysis.

## 4.6 Memory Management

Tofte and Talpin [72] suggested that call-by-value functional languages can be implemented using *regions* for memory management. The idea is that, at run time, the store consists of a stack of regions. Region inference is a type-based analysis, presented as a type and effect system, which determines where regions can be allocated and deallocated. Birkedal, Tofte, and Vejlstrup [8] presented an implementation for Standard ML. which demonstrates that region

inference can result in significant space savings, in comparison with more traditional memory management based on garbage collection. Moreover, the region-based system can compete on speed with a garbage-collection-based system.

## 5. OTHER TYPE-BASED ANALYSES

There is a large number of published type and effect systems for such tasks as side-effect analysis [37, 32, 66, 76], binding-time analysis [49], strictness analysis [35, 36, 78, 3, 31], totality analysis, [62, 63, 61], callability analysis [67, 68], flow analysis [43, 42, 7, 30, 75, 73, 15, 56, 53], trust analysis [51], secure information flow analysis [60], closure conversion [25], resource allocation in compilers [69], continuation allocation [58], dependency analysis [1], communication analysis [48], and elimination of useless variables [34, 19]. Many of them have been proved correct, most have not yet been implemented for a full-fledged programming language, although some have been implemented for a toy language, and some still need an algorithm for performing the analysis. Nielson and Nielson [47] present the overall methodology behind type and effect systems, and they discuss the major design decisions, including whether or not to incorporate subtyping, subeffecting, polymorphism, and polymorphic recursion.

## 6. CONCLUSION

Most of the surveyed tools use types as discriminators, while most of the theoretical studies use type and effect systems. To enable a better comparison of the different approaches, future research may attempt a further formalization of the techniques used in current tools, and larger-scale implementations and experiments with published type and effect systems. Ideally, a static analysis should come with both a proof of correctness and convincing experimental results. Type-based analysis is a promising approach to achieving both with a reasonable effort.

Further information about type-based analysis and links to many of the cited papers are available from:

http://www.cs.purdue.edu/homes/palsberg/tba/

## Acknowledgments

Thanks to Tony Hosking for helpful discussions, and to John Field and the other PASTE 2001 organizers for encouragement. Palsberg was supported by a National Science Foundation Faculty Early Career Development Award, CCR–9734265, by CERIAS (Center for Education and Research in Information Assurance and Security), and by IBM.

## REFERENCES

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Proceedings of POPL'99, 26th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 147–160, 1999.

[2] Alfred V. Aho, Ravi I. Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, second edition, 1986.

[3] Torben Amtoft. Minimal thunkification. In *Proceedings of WSA'93, 3rd International Workshop on Static Analysis*, pages 218–229. Springer-Verlag (*LNCS* 724), 1993.

[4] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[5] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, pages 324–341, San Jose, CA, 1996. *SIGPLAN Notices* 31(10).

[6] David Francis Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, Computer Science Division, University of California, Berkeley, December 1997. Report No. UCB/CSD-98-1017.

[7] Anindya Banerjee. A modular, polyvariant and type-based closure analysis. In *Proceedings of ICFP'97, ACM International Conference on Functional Programming*, pages 1–10, 1997.

[8] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of POPL'96, 23nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 171–183, 1996.

[9] Boris Bokowski and Jan Vitek. Confined types. In *Proceedings of the Fourteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, pages 82–96, Denver, CO, 1999.

[10] Luca Cardelli. Type systems. In *CRC Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press, 1997.

[11] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *Sixteenth Symposium on Principles of Programming Languages*, pages 202–212, 1989.

[12] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of ICSE'00, 22nd International Conference on Software Engineering*, pages 439–448, 2000.

[13] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95)*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.

[14] David Detlefs, K. Rustan Leino, Greg Nelson, and James Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, 1998.

[15] Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In *Proceedings ICFP '97, International Conference on Functional Programming, ACM SIGPLAN Notices 32(8)*, pages 11–24, 1997.

[16] Amer Diwan, Kathryn McKinley, and Eliot Moss. Type-based alias analysis. In *Proceedings of PLDI'98, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 106–117, 1998.

[17] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[18] Stephen Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *Proceedings of SAS'00, 7th International Static Analysis Symposium*, pages 155–174. Springer-Verlag (*LNCS* 1824), 2000.

[19] Adam Fischbach and John Hannan. Type systems and algorithms for useless-variable elimination. In *Proceedings of PADO'01, Symposium on Programs as Data Objects*, 2001. To appear.

[20] Cormac Flanagan and Stephen Freund. Type-based race detection for Java. In *Proceedings of PLDI'00, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[21] Sanjay Ghemawat, Keith Randall, and Daniel Scales. Field analysis: Getting useful and low-cost interprocedural information. In *Proceedings of PLDI'00, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 334–344, 2000.

[22] Rajeev Gopal and Stephan R. Schach. Using automatic program decomposition techniques in software maintenance tools. In *Proceedings of ICSM'89, International Conference on Software Maintenance*, pages 132–141, 1989.

[23] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[24] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Proceedings of OOPSLA'01, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2001. To appear.

[25] John Hannan. Type systems for closure conversions. In *Proceedings of Workshop on Types for Program Analysis*, pages 48–62, 1995.

[26] Nevin Heintze. Control-flow analysis and type systems. In *Proceedings of SAS'95, International Static Analysis Symposium*, pages 189–206. Springer-Verlag (*LNCS* 983), Glasgow, Scotland, September 1995.

[27] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 261–272, 1997.

[28] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[29] Antony L. Hosking, Nathaniel Nystrom, David Whitlock, Quintin Cutts, and Amer Diwan. Partial redundancy elimination for access path expressions. *Software – Practice & Experience*, 31(6):577–600, 2001.

[30] Suresh Jagannathan, Andrew Wright, and Stephen Weeks. Type-directed flow analysis for typed intermediate languages. In *Proceedings of SAS'97, International Static Analysis Symposium*. Springer-Verlag, 1997.

[31] Thomas Jensen. Inference of polymorphic and conditional strictness properties. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 209–221, San Diego, California, January 1998.

[32] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Proceedings of POPL'91, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 303–310, 1991.

[33] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[34] Naoki Kobayashi. Type-based useless variable elimination. In *Proceedings of PEPM'00, ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 84–93, 2000.

[35] Tsung-Min Kuo and Prateek Mishra. On strictness and its analysis. In *Proceedings of POPL'87, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 144–155, 1987.

[36] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, pages 260–272, 1989.

[37] John Lucassen and David Gifford. Polymorphic effect systems. In *Proceedings of POPL'88, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[38] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

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

[40] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talx86: A realistic typed assembly language. ACM Workshop on Compiler Support for System Software, May 1999.

[41] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 85–97, 1998.

[42] Christian Mossin. Exact flow analysis. In *Proceedings of SAS'97, International Static Analysis Symposium*, pages 250–264. Springer-Verlag (*LNCS*), 1997.

[43] Christian Mossin. *Flow Analysis of Typed Higher-Order Languages*. PhD thesis, DIKU, University of Copenhagen, 1997.

[44] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[45] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[46] Flemming Nielson. The typed lambda-calculus with first-class processes. In *Proceedings of PARLE*, pages 357–373, April 1989.

[47] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design*, pages 114–136, 1999.

[48] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[49] Hanne R. Nielson and Flemming Nielson. Automatic binding time analysis for a typed $\lambda$-calculus. *Science of Computer Programming*, 10:139–176, 1988.

[50] Tobias Nipkow and David von Oheimb. $\text{Java}_{\text{light}}$ is type-safe – definitely. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 161–170, San Diego, California, January 1998.

[51] Peter Ørbæk and Jens Palsberg. Trust in the $\lambda$-calculus. *Journal of Functional Programming*, 7(6):557–591, November 1997. Preliminary version in Proceedings of SAS'95, International Static Analysis Symposium, Springer-Verlag (*LNCS* 983), pages 314–330, Glasgow, Scotland, September 1995.

[52] Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, January 1995. Preliminary version in Proceedings of CAAP'94, Colloquium on Trees in Algebra and Programming, Springer-Verlag (*LNCS* 787), pages 276–290, Edinburgh, Scotland, April 1994.

[53] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *Journal of Functional Programming*, to appear. Preliminary version in Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 197–208, San Diego, California, January 1998.

[54] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings of OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, Phoenix, Arizona, October 1991.

[55] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[56] Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: From polymorphic subtyping to cfl-reachability. In *Proceedings of POPL'01, 28th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 54–66, 2001.

[57] Debra J. Richardson. TAOS: Testing with analysis and oracle support. In *International Symposium on Software Testing and Analysis*, pages 138–153, 1994.

[58] Zhong Shao and Valery Trifonov. Type-directed continuation allocation. In *ACM Workshop on Types in Compilation*, pages 116–136, Kyoto, Japan, March 1998.

[59] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, CMU, May 1991. CMU–CS–91–145.

[60] Geoffrey Smith and Dennis Volpano. Secure information flow in multi-threaded imperative language. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 355–364, 1998.

[61] Kirsten Solberg. *Annotated Type Systems for Program Analysis*. PhD thesis, University of Aarhus, 1995.

[62] Kirsten Solberg, Hanne Riis Nielson, and Flemming Nielson. Strictness and totality analysis. In *Proceedings of SAS'94, International Static Analysis Symposium*, pages 408–422. Springer-Verlag (*LNCS* 864), 1994.

[63] Kirsten Solberg, Hanne Riis Nielson, and Flemming Nielson. Strictness and totality analysis with conjunction. In *Proceedings of TAPSOFT'95, Theory and Practice of Software Development*, pages 501–515. Springer-Verlag (*LNCS* 915), Aarhus, Denmark, May 1995.

[64] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the Fifteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 264–280, Minneapolis, Minnesota), 2000.

[65] Peter F. Sweeney and Frank Tip. Extracting library-based object-oriented applications. In *Proceedings of the Eighth International Symposium on the Foundations of Software Engineering (FSE-8)*, pages 98–107, November 2000.

[66] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111:245–296, 1994. A preliminary version was presented at LICS'92.

[67] Yan Mei Tang and Pierre Jouvelot. Separate abstract interpretation for control-flow analysis. In *Proceedings of TACS'94, Theoretical Aspects of Computing Software*, pages 224–243. Springer-Verlag (*LNCS* 789), 1994.

[68] Yan Mei Tang and Pierre Jouvelot. Effect systems with subtyping. In *Proceedings of PEPM'95, ACM Symposium on Partial Evaluation and Sematics-Based Program Manipulation*, pages 45–53. ACM Press, 1995.

[69] Peter Thiemann. Formalizing resource allocation in a compiler. In *ACM Workshop on Types in Compilation*, pages 178–194, Kyoto, Japan, March 1998.

[70] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for Java. In *Proceedings of the Fourteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, pages 292–305, Denver, CO), 1999. *SIGPLAN Notices* 34(10).

[71] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of OOPSLA'00, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 281–293, Minneapolis, Minnesota, October 2000.

[72] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[73] Franklyn Turbak, Allyn Dimock, Robert Muller, and J. B. Wells. Compiling with polymorphic and polyvariant flow types. In *ACM SIGPLAN Workshop on Types in Compilation*, June 1997. http://www.cs.bc.edu/~muller/postscript/tic97.ps.Z.

[74] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[75] J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *Journal of Functional Programming*. To appear.

[76] Andrew Wright. Typing references by effect inference. In *Proceedings of ESOP'92, European Symposium on Programming*, pages 473–491. Springer-Verlag (*LNCS* 582), 1992.

[77] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[78] David A. Wright. A new technique for strictness analysis. In *Proceedings of TAPSOFT'91*, pages 235–258. Springer-Verlag (*LNCS* 494), 1991.