Control-Flow Analysis of Functional Programs

JAN MIDTGAARD, Aarhus University

We present a survey of control-flow analysis of functional programs, which has been the subject of extensive investigation throughout the past 30 years. Analyses of the control flow of functional programs have been formulated in multiple settings and have led to many different approximations, starting with the seminal works of Jones, Shivers, and Sestoft. In this article, we survey control-flow analysis of functional programs by structuring the multitude of formulations and approximations and comparing them.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Applicative functional languages*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Control-flow analysis, higher-order functions

ACM Reference Format:
Midtgaard, J. 2012. Control-flow analysis of functional programs. ACM Comput. Surv. 44, 3, Article 10 (June 2012), 33 pages.
DOI = 10.1145/2187671.2187672 http://doi.acm.org/10.1145/2187671.2187672

1. INTRODUCTION

Since the introduction of high-level languages and compilers, much work has been devoted to approximating, at compile time, which values the variables of a given program may denote at runtime. This problem has been named *data-flow analysis* or *flow analysis*.

In a language without higher-order functions, the operator of a function call is apparent from the text of the program: it is a lexically visible identifier, and therefore, the called function is available at compile time. One can thus base an analysis for such a language on the textual structure of the program, since it determines the exact *control flow* of the program, for example, as a flow chart. On the other hand, in a language with higher-order functions, the operator of a function call may not be apparent from the text of the program: it can be the result of a computation and therefore, the called function may not be available until runtime. A *control-flow analysis* approximates at compile time which functions may be applied at runtime, that is, it determines an approximate control flow of a given program.

Prerequisites. We assume some familiarity with program analysis in general and with control-flow analysis in particular. For a tutorial or an introduction to the area, we refer to Nielson et al. [1999]. We also assume familiarity with functional programming and a basic acquaintance with continuation-passing style (CPS) [Steele Jr. 1978] and

Part of this work was done with support of the Carlsberg Foundation ad an INRIA post-doc grant.
Authors' addresses: J. Midtgaard, Department of Computer Science, Aarhus University, Aabogade 34, DK-8200 Aarhus N., Denmark; email: jmi@cs.au.dk.

DOI 10.1145/2187671.2187672 http://doi.acm.org/10.1145/2187671.2187672

with recursive equations [O'Donnell 1977]. We furthermore assume some knowledge about closures for representing functional values at runtime [Landin 1964] and with Reynolds's defunctionalization [Reynolds 1972; Danvy and Nielsen 2001].

1.1. History

Historically, Reynolds was the first to analyze LISP programs [Reynolds 1969]. Independently of Reynolds, Schwartz [1975] developed *value-flow analysis* for SETL, an untyped, first-order language based on set theory [Schwartz et al. 1986]. Also independently of Reynolds, Jones and Muchnick analyzed programs with LISP-like structured data [Jones and Muchnick 1979, 1982]. Jones was the first to analyze lambda expressions and to use the term control-flow analysis [Jones 1981a, 1981b] for the problem of approximating the control flow of higher-order programs. Rozas [1984] since developed a flow analysis for Scheme as part of the Liar Scheme compiler. Independently, Shivers formulated control-flow analysis for Scheme, proved it sound, and suggested a number of variations, improvements, and applications [Shivers 1988, 1991a]. Sestoft then developed a *closure analysis* for programs in direct style [Sestoft 1988, 1989]. The latter was reformulated first by Bondorf [1991] and later by Palsberg [1994], whose account is closest to how control-flow analysis is presented in textbooks today [Nielson et al. 1999].

1.2. Terminology

1.2.1. Flow vs. Closure Analysis. Jones and Shivers named their analyses control-flow analysis [Jones 1981a, 1981b; Shivers 1991a], whereas Sestoft [1988] named his analysis closure analysis. Even though they are presented with different terminology, all three analyses compute flow information, that is, they approximate where a given first-class function is applied and which first-class functions are applied at a given call site. The term value-flow analysis of Schwartz [1975] is also used to refer to analyses computing such flow information [Henglein et al. 2005]. The term control-flow analysis was originally used by Allen [1970] to refer to the extraction of properties of already given control-flow graphs.

A different line of analysis introduced by Steele in his M.S. thesis [Steele Jr. 1978] is also referred to as closure analysis [Clinger and Hansen 1994; Serrano 1995]. These analyses, on the other hand, are concerned with approximating which function calls are *known* and which functions need to be closed because they *escape* their scope. A call to a known procedure can be implemented more efficiently than the closure-based procedure-call convention, and a non-escaping function does not require a heap-allocated closure [Kranz 1988].

1.2.2. Approximating Allocation. In control-flow analysis, one typically approximates a dynamically allocated closure by its code component, that is, the lambda term, representing its place of creation. The idea is well known from analyses approximating other heap-allocated structures [Jones and Muchnick 1979, 1982; Andersen 1994; Cejtin et al. 2000], where it is named the *token* or *birth-place* approach. Consel [1990], for example, uses the approach in his work on binding-time analysis.

More generally, dynamic allocated storage can be represented by the (approximate) state of the computation at allocation time [Jones and Muchnick 1982; Hudak 1987; Deutsch 1990]—an idea which has been named the *birth-time*, *birth-date*, or *time-stamp* approach [Harrison III 1989; Harrison III and Ammarguellat 1993; Venet 2002]. The state of the computation can be represented by the (approximate) paths or traces leading to it. One such example is *contours* [Shivers 1991a], which are finite string encodings approximating the calling context, that is, the history of function calls in the computation leading up to the current state. The term contour was originally used to model block structure in programming languages [Johnston 1971].

1.2.3. Sensitivity. Established terminology from static analysis has been used to characterize and compare the precision of analyses [Hind 2001]. Much of this terminology has its roots in data-flow analysis, where one distinguishes *intra-procedural* analyses, that is, local analyses operating within procedure boundaries, from *inter-procedural* analyses, that is, global analyses operating across procedure calls and returns. In a functional language based on expressions, such as Scheme or ML, function calls and returns are omnipresent. As a consequence, the data-flow analysis terminology does not fit as well. Throughout the rest of this article, we will use the established terminology where appropriate.

A *context-sensitive* analysis distinguishes different calling contexts when analyzing expressions, whereas a *context-insensitive* analysis does not. Within the area of control-flow analysis the terms *polyvariant* analysis and *monovariant* analysis are used for the same distinction [Nielson et al. 1999]. A *flow-sensitive* analysis follows the control-flow of the source program whereas a *flow-insensitive* analysis more crudely approximates the control-flow of the source program by assuming that any expression can be evaluated immediately after any other expression.

2. CONTEXT-INSENSITIVE FLOW ANALYSIS

In this section, we consider context-insensitive control-flow analyses. Starting from the most crude approximation, we list increasingly precise approximations.

2.1. All Functions

The initial, somewhat naive, approximation is that all lambda expressions in a program can potentially occur at each application site. In his M.S. thesis [Sestoft 1988], Sestoft suggested this approximation as safe but too approximate, which motivates his introduction of a more precise closure analysis. This rough flow approximation also underlies the polymorphic defunctionalization suggested by Pottier and Gauthier [2006]. Their transformation enumerates all source lambda expressions (of varying type and arity) and represents them by injection into a single global data type. The values of the data type are consumed by a single global *apply* function. This approach requires a heavier type machinery than is available in ML. Their work illustrates that a resulting program can be type-checked using "guarded algebraic data types".

2.2. All Functions of Correct Arity

Warren independently discovered defunctionalization in the context of logic programming [Warren 1982]. He outlines how to extend Prolog to higher-order predicates. The extension works by defunctionalizing the predicates-as-parameters, with one *apply* function per predicate-arity. The transformation effectively relies on an underlying flow approximation, which approximates an unknown function (predicate) by all functions (predicates) of the correct arity.

This approximation is not safe for dynamically typed languages, such as Scheme, where arity mismatches can occur at runtime. On the other hand, the approximation is safe for languages where arity checks are performed at compile time. Ironically, the latter can be achieved by another control-flow analysis or a static type system.

2.3. Escape Analysis

A lightweight approach to compiling higher-order functions is the so-called *escape analysis* [Appel 1992; Appel and Jim 1989]. The approach is based on a rough flow approximation originally due to Steele [1978]. In its simplest formulation, the analysis partitions the source lambda expressions into two groups: *escaping functions*, that is, functions that (potentially) escape their lexical scope by being returned, passed as a parameter, stored in a pair, etc., and *known functions*, that is, functions that do

not escape [Shao and Appel 1994]. The categorization can be formulated as a simple mapping from source lambdas to a binary domain. In essence, this analysis categorizes higher-order functions as 'escaping', whereas first-order functions are categorized as 'known'. In the Rabbit Scheme compiler [Steele Jr. 1978], Steele used the analysis to decide whether to *close* lambda expressions, that is, create a *closure* over their free variables or not.

A slightly better approximation supplements the preceding partition of source lambdas with a similar partition of function calls. Function calls are also partitioned into two groups: *known calls* and *unknown calls* [Appel and Jim 1989]. This categorization can also be formulated as a simple mapping from call sites to a binary domain. As a consequence, a source lambda can both escape and also be the operator of a known call. In the Orbit compiler [Kranz et al. 1986; Kranz 1988], Kranz further distinguished between *upward escaping* and *downward escaping* variables and lambda expressions, because closures in the latter category could be stack allocated. Garbage collection was considered relatively expensive at the time, and Kranz's motivation [Kranz 1988] was to show that Scheme could be compiled as efficiently as, for example, Pascal, which was designed to be stack-implementable.

Escape analysis is a modular flow approximation, that is, separate modules can be analyzed independently, as opposed to a whole-program flow analysis. The flow approximation is useful for both closure-conversion [Steele Jr. 1978; Appel 1992] and defunctionalization [Tolmach 1997]. Different terminology has been used to name the approach, sometimes with unfortunate overlap. Steele used the term *binding analysis* for the corresponding pass in his compiler [Steele Jr. 1978]. Kranz refers to the distinction as *escape analysis* [Kranz 1988]. Both Steele and Kranz refer to their decision procedure for choosing closure representation and layout as *closure analysis* [Steele Jr. 1978; Kranz 1988]. Clinger and Hansen [1994] characterize escape analysis as a *first-order closure analysis* to stress that it detects first-order use of functions. Serrano referred to escape analysis as *closure analysis* [Serrano 1995]. Cejtin et al. [2000] refer to escape analysis as a *syntactic heuristic*.

2.4. Simple Closure Analysis

Henglein first introduced *simple closure analysis* in an influential (though not often credited) technical report [Henglein 1992]. The analysis is inspired by type inference, and as such, it is based on emitting equality constraints that are then solved by *unification*. The latter can be performed efficiently in almost linear time using a *union-find data structure*, as is well-known from type inference [Pierce 2002]. Another inspiration was a similar but somewhat more advanced binding-time analysis [Henglein 1991] performing two rounds of unification. Bondorf and Jørgensen [1993] later documented an extension of Henglein's approach in the context of a partial evaluator, and Heintze gave a simple formulation in terms of equality constraints [Heintze 1995]. Tolmach and Oliva [1997, 1998] as well as Mossin [1997] have since formulated (typed) variants of the approach.

Van Horn and Mairson [2008b] have recently reinvestigated simple closure analysis. For linear lambda-calculus terms, that is, terms in which variables occur exactly once, they proved that simple closure analysis and evaluation are equivalent. They furthermore proved that simple closure analysis is complete for polynomial time.

Henglein titled the approach simple closure analysis, but the analysis has since been named *equality-based flow analysis* [Bondorf and Jørgensen 1993; Palsberg 1998], as well as *control-flow analysis via equality* [Heintze 1995]. The idea is also referred to as *unification-based* [Cejtin et al. 2000; Das 2000; Hind 2001] or *bi-directional* [Rehof and Fähndrich 2001; Hind 2001]. Aiken refers to the analysis as based on *term*

equations [Aiken 1999]. We shall sometimes refer to it as unification-based analysis when contrasting it with other analyses.

2.5. 0-CFA

The textbook presentation [Nielson et al. 1999] of a *zeroth-order control-flow analysis* (0-CFA) computes a fixed point over two global maps: (a) an abstract environment that anticipates all the actual bindings that can occur at runtime and (b) an abstract *cache* that anticipates all the functional values to which expressions can evaluate at runtime. In the words of Jones [1987], the analysis "neglects all coordination among the variable bindings, and merges all bindings of the same variable. (Nonetheless it seems to work well in practice.)"

Shivers developed the context-insensitive 0-CFA for Scheme [Shivers 1991a, 1988] and suggested several context-sensitive flow analyses (see the following). During his work on globalization, that is, statically determining which function parameters can be turned into global variables, Sestoft had developed a similar flow analysis [Sestoft 1988, 1989] in order to handle higher-order programs.

Bondorf later simplified the equations of Sestoft's analysis [Bondorf 1991] in order to extend the Similix [Bondorf 1993] self-applicable partial evaluator to higher-order functions. Palsberg then limited Bondorf's equations to pure lambda calculus [Palsberg 1994, 1995]. He presented a simplified analysis, as well as a constraint-based formulation, and proved the equivalence of the three analyses.

Initially, Shivers [1988, 1991a] did not analyze the time complexity of his approach. In his Ph.D. dissertation, Sestoft [1991] gave an $O(n^4)$ upper bound, which was later refined to $O(n^3)$ [Oxhøj et al. 1992; Ayers 1992; Heintze 1994; Palsberg and Schwartzbach 1995]. In Section 5.3.2, we discuss a recent $O(n^3/\log n)$ algorithm for the unit cost random-access memory model machine. Van Horn and Mairson [2007] recently formulated 0-CFA as a decision problem, which they proved complete for polynomial time. Their proof is based on linear lambda calculus programs for which analysis and evaluation coincides.

Serrano [1995] describes a variant of Shivers's 0-CFA used in the Bigloo optimizing Scheme compiler. Serrano's description is given as a functional program with side effects (assignments). Reppy [2006] later describes a refinement of Serrano's algorithm reformulated as a pure functional program. The analyses of Serrano and Reppy do not infer control-flow information for all expressions; they infer control-flow information only for variables, that is, they compute an approximation of the runtime environment.

As opposed to a unification-based control-flow analysis, 0-CFA is sometimes referred to as an *inclusion-based* [Bondorf and Jørgensen 1993] or *subset-based* [Palsberg 1998] control-flow analysis. In the terminology of pointer analysis, it is a (*uni-*) *directional* flow analysis [Rehof and Fähndrich 2001; Hind 2001]. Variants of 0-CFA are used within the Bigloo optimizing Scheme compiler [Serrano 1995] and within the MLton whole-program optimizing SML compiler [Cejtin et al. 2000].

The term control-flow analysis by itself has since become synonymous with 0-CFA [Nielson et al. 1999].

2.6. Flow-Sensitive 0-CFA

Though Shivers's and Sestoft's analyses were thought to be equivalent, Mossin [1997] showed that Shivers's analysis is *flow-sensitive* (or evaluation-order dependent), contrary to Sestoft's closure analysis. Indeed, both Sestoft's analysis and the textbook presentation of 0-CFA [Nielson et al. 1999] compute a fixed point over a global abstract environment, whereas a *flow-sensitive 0-CFA* [Ashley and Dybvig 1998] computes an abstract environment for each expression in the program. As such, Shivers's

analysis depends on the order in which expressions occur in the input program. By this terminology, Shivers's original 0-CFA is not a *0-CFA* but, rather, a *flow-sensitive* 0-CFA.

In a language without references and assignment, computing an abstract environment per expression is excessive. Consider, for example, an application which is always evaluated in the same environment as its two direct subexpressions: the operator and the operand. A less excessive alternative was suggested by Steckler and Wand [1997] who developed a flow-sensitive 0-CFA with an abstract environment for each distinct lexical scope of the program. Their flow analysis is only one part of a larger analysis (further described in Section 6.2). Somewhat surprisingly, the worst-case time complexity of their flow-sensitive analysis remains cubic.

Ashley and Dybvig [1998] note that adding flow-sensitivity to 0-CFA in itself does not provide significantly more precision in a functional language, such as Scheme, where assignments are relatively rare. However, the combination of flow-sensitivity and other abstractions can provide such a precision boost (see Section 6.2).

3. CONTEXT-SENSITIVE FLOW ANALYSIS

In this section, we consider context-sensitive control-flow analyses. Starting from polymorphic splitting, we describe a number of increasingly precise analyses.

3.1. Polymorphic Splitting

Polymorphic splitting is a context-sensitive flow analysis suggested by Wright and Jagannathan [1998]. The analysis is inspired by type systems—in particular, Hindley-Milner (Damas-Milner) let-bound polymorphism [Milner 1978], where each occurrence of a let-bound variable is type-checked separately. In the same spirit, polymorphic splitting analyzes each occurrence of a let-bound variable separately. The analysis has an exponential worst-case time complexity, like that of the polymorphic type inference that inspired it. However, as with the corresponding type inference, the worst-case rarely seems to occur in practice [Wright and Jagannathan 1998].

One can view polymorphic splitting as a refinement of 0-CFA that partitions the flow of values to expressions and variables according to their static context (scope) in the program text. Polymorphic splitting is therefore referred to as approximating the *static link* of a stack-based implementation [Nielson and Nielson 1997].

3.2. k-CFA

Call strings and their approximation up to a fixed maximum length have their roots in data-flow analysis. Call strings were originally suggested by Sharir and Pnueli [1981] as a means for improving the precision of interprocedural data-flow analyses. Inspired by call strings, Shivers [1991a] formulated the context-sensitive *first-order control-flow analysis* (1-CFA) and suggested the extension to *kth-order control-flow analysis* (*k-CFA*) [Shivers 1991a] as a refined choice of contours. Since then, Jagannathan and Weeks [1995] have suggested a *polynomial-time 1-CFA*, a more approximate 1-CFA variant with better worst-case time complexity. Jagannathan and Weeks achieve the speedup by restricting the environment component of an abstract closure to a constant function, mapping all variables to a contour representing the most recent call-site. The *uniform k-CFA* is another k-CFA variant suggested by Nielson and Nielson [1997; 1999]. It uses a uniform contour distinction, that is, abstract caches and abstract environments partition the flow of values to expressions and variables identically. The resulting analysis has a better worst-case time complexity than the canonical k-CFA.

Van Horn and Mairson [2008a] recently proved k-CFA complete for EXPTIME for any constant $k > 0$. The proof works by encoding an exponential time Turing machine

as a lambda term, for which the analysis will decide acceptance or rejection. The proof is developed for the uniform k-CFA variant. Might et al. [2010] have recently investigated the apparent paradox in (1) the preceding completeness result and (2) the existence of well-known, polynomial-time k-CFA algorithms for object-oriented (OO) languages. In comparing a functional and an OO k-CFA, they conclude that the subtle language differences between objects and closures allow for the formulation of an equivalent, polynomial-time OO analysis with no loss of precision. Based on this OO analysis, they extract *m-CFA*: a hierarchy of functional, polynomial-time, context-sensitive analyses. Their benchmarks show that m-CFA (for $m = 1$) approaches the precision of 1-CFA, however with runtimes comparable to the polynomial time 1-CFA of Jagannathan and Weeks [1995].

One can view k-CFA as a refinement of 0-CFA that partitions the flow of values to expressions and variables according to their (approximate) dynamic calling context in a program execution. Call strings are therefore referred to as approximating the *dynamic link* of a stack-based implementation [Nielson and Nielson 1997].

3.3. Beyond the k-CFA Hierarchy

An alternative context-sensitive flow analysis suggested by Agesen [1995] takes argument types of the calling context into account. The original formulation of his *Cartesian-product algorithm* (CPA) was given as a type inference algorithm for a dynamically typed object-oriented programming language. As with much other work within type systems, the basic idea extends to control-flow analysis. Jagannathan and Weeks [1995] outline a control-flow analysis variant thereof, as do Nielson et al. [1999]. One can view the resulting analysis as a refinement of 0-CFA that partitions the flow of values to expressions and variables according to the actual argument types in their dynamic calling context.

The call string approach later inspired Harrison to suggest *procedure strings* [Harrison III 1989; Harrison III and Ammarguellat 1993] to capture both *procedure calls* and *returns* in a compact format. Might and Shivers recently suggested a new context-sensitive control-flow analysis [Might and Shivers 2006a] based on a variant of procedure strings called *frame strings*. Frame strings represent *stack frame operations*, which are more informative in a functional language where proper tail calls do not push a stack frame. Might and Shivers then approximate the frame strings by regular expressions. From the result of running their analysis, they finally extract an *environment analysis* [Shivers 1991a], that is, an analysis which statically detects when two runtime environments agree on a variable.

An alternative path was taken by Sereni [2006, 2007] who developed a CFA based on *tree automata* for programs in the form of curried, recursive equations. Tree automata can capture the potentially unbounded tree structure of environments, which is beyond the capabilities of the traditional depth k technique. The time complexity of this analysis is exponential in the square of the program's size. Sereni applied his analysis to predict call graphs that drive a size-change termination analysis of higher-order functional programs.

Recently, Vardoulakis and Shivers [2010] have developed *a context-free approach to control-flow analysis*. Their analysis combines two orthogonal ideas: first, they separate variables into stack-allocated variables and heap-allocated variables. The separation allows them to precisely model bindings to stack-allocated variables in the top-stack frame. Second, they develop a work-list algorithm extending the summarization technique of Sharir and Pnueli [1981] to handle both higher-order functions and tail calls. The resulting analysis provides precise, matching *call-return flows*, thereby avoiding flow from one call-site to return through a different call-site.

4. TYPE-BASED FLOW ANALYSIS

The range of untyped analyses mentioned so far are all applicable to typed programs as well. A parallel line of work has investigated control-flow analysis of typed higher-order programs. The extra static information provided by types suggests natural control-flow approximations. Alternatively, known type systems operating on types enriched with flow information suggests new control-flow analyses. In this section, we consider both kinds of type-based approximations. Starting from the simplest approximation, we consider increasingly precise type-based approximations.

4.1. Per Function Space (Typed)

A naive approach approximates the application of a function by all the functions of the same type. This context-insensitive approximation underlies Reynolds's initial presentation of defunctionalization [Reynolds 1972], where the function space of the environment and the function space of expressible values in his definitional interpreter were defunctionalized separately. Indeed, Reynolds recognized that his defining language was typed in a later commentary [Reynolds 1998].

Tolmach [1997] realized that Reynolds's defunctionalization was based on an underlying control-flow approximation induced by the types, noting that "typing obviously provides a good first cut at the analysis 'for free'" [Tolmach 1997]. Tolmach and Oliva [1997, 1998] furthermore pointed out that unification-based analysis can be viewed as a refinement to the function-space approximation: the function-space approximation places two functions in the same partition when the types of their argument and result match. On the other hand, a unification-based analysis places functions in the same partition when the type unifier unifies their types.

4.2. Linear-Time Subtransitive CFA (Typed)

Heintze and McAllester [1997a] presented a linear-time algorithm for bounded-type programs that can answer a number of context-insensitive CFA-related questions. Their algorithm builds a graph whose full transitive closure can list all callees for each call site in (optimal) quadratic time. By avoiding the computation of the full transitive closure, they are able to answer some questions in linear time, for example, list up to k functions for each call site, otherwise *many*. Their linear time complexity depends on bounded-type programs; for untyped or recursively typed programs, their algorithm may not terminate [Heintze and McAllester 1997a]. Independently, Mossin arrived at a quadratic-time analysis for simply typed programs with bounded types [Mossin 1997, 1998]. His analysis is based on flow graphs. It is furthermore *modular*, that is, different parts of the program may be analyzed separately.

Hankin et al. [2002] pointed out that both Mossin's and Heintze and McAllester's sub-cubic algorithms operate on type structures and perform implicit or explicit eta-expansions of the program. Since the latter can incur an exponential blowup of the program, both analyses need to assume bounds on the type of the program. Recently, Van Horn and Mairson [2007] showed that 0-CFA for simply typed, fully eta-expanded programs is complete for LOGSPACE, thereby giving a complexity-theoretic explanation of the sub-cubic breakthroughs.

Unpublished work by Saha et al. [1998] indicated that these preceding analyses do not scale, since real-world (functorized) programs do not always exhibit bounded types. They therefore suggested a hybrid approach, where Heintze and McAllester's algorithm is combined with a complementary demand-driven algorithm.

4.3. Context-Sensitive Type-Based Analysis (Typed)

Mossin gave two context-sensitive analyses for a simply typed programming language [Mossin 1997]: one inspired by let-polymorphism and one inspired by

polymorphic recursion. Rehof and Fähndrich [2001] later gave $O(n^3)$ algorithms for the two, improving their earlier complexity bounds on $O(n^7)$ and $O(n^8)$, respectively [Mossin 1997], with n being the size of the explicitly typed program.[1] Rehof and Fähndrich achieve the speedup by avoiding copying constraints when they are instantiated—instead, they remember the substitution in a separate *instantiation constraint*, leaving the original constraints unmodified. The algorithm further reduces the constraints to a *context-free language reachability problem* (see Section 5.3.2) over a flow graph for which cubic-time algorithms exist. Gustavsson and Svenningsson [2001] independently formulated a constraint-based version of Mossin's polymorphic recursion-based analysis with identical time complexity. Their analysis is developed as an application of a general constraint system with *constraint abstractions*. Analysis then works by successive constraint rewritings into *atomic* inclusion constraints from which flow can be read off using standard reachability methods.

In addition to these two type-based analyses, Mossin developed a context-sensitive control-flow analysis based on intersection types [Mossin 1997, 2003] which he called *exact*: if the analysis predicts a redex, there exists a reduction sequence that reduces it [Mossin 2003]. He showed the analysis to be decidable; it is, however, non-elementary recursive. The following year, Møller Neergaard and Mairson [2004] showed that type inference and normalization for a type system with intersection types are equivalent and discussed the need for type inference (and static analysis) to be faster and hence less precise than running the program.

5. FORMULATIONS

Control-flow analysis comes in numerous formulations. In this section, we describe the many formulations encountered, as well as the known equivalences and relationships between them.

5.1. Grammar-Based, Constraint-Based, and Set-Based Analysis

A constraint-based analysis is a two-phase algorithm. The first phase emits constraints that a solution to an intended analysis needs to satisfy. The second phase solves the constraints. Type inference [Wand 1987] is an example of a constraint-based analysis that has inspired many later analyses [Henglein 1991; Tang and Jouvelot 1992; Henglein 1992; Steensgaard 1996b; Hannan 1998; Mossin 1997]. The idea of formulating program analyses in terms of constraints has its advantages: the analysis presents itself in an intuitive form, and it allows for reusable constraint solving software, independent of a particular analysis. Aiken gives an introduction to set-constraint-based analysis [Aiken 1999]. Next, we outline the developments of constraint-based analysis relevant to CFA. For a more detailed account we refer to Pacholski and Podelski [1997].

Initially, Reynolds conceived the idea of formulating analyses in terms of *recursive set definitions* [Reynolds 1969], which resembled context-free grammars [Reynolds 1969]. He extended them with suitable list constructors (e.g., *cons*) and selectors (e.g., *car* and *cdr*) operating over sets. The analysis then eliminated the selectors from the definitions. Independently, Jones and Muchnick later used *extended regular tree grammars*, that is, *regular tree grammars* extended with selectors, to analyse programs with LISP-like structures [Jones and Muchnick 1979].

Heintze and Jaffar extended the idea of *grammar-based analysis* to handle projection (selectors) and intersection [Heintze and Jaffar 1990b, 1990a], originally in the context of analyzing logic programs and introduced the term *set constraints*. Aiken and Murphy [1991b] formulated a type-inference algorithm with types implemented as *regular tree*

[1]However, n itself may be exponential in the size of the implicitly typed program [Rehof and Fähndrich 2001].

expressions, and described their implementation [Aiken and Murphy 1991a]. Aiken and Wimmers [1992] later gave an algorithm for solving constraint systems over regular tree expressions, now under the name *set constraints*. As a follow-up, Aiken and Wimmers [1993] investigated set constraints further extended with *function types* and utilized them to formulate an inclusion-based type-inference system for an extended lambda calculus.

Heintze coined the term set-based analysis [Heintze 1992] for the intuitive formalism of formulating program analyses as a series of constraints over set expressions (extended with intersection and projection/selectors). He later formulated a *set-based analysis* for ML [Heintze 1994]. Independently, Palsberg reformulated Bondorf's simplification of Sestoft's control-flow analysis in terms of conditional set constraints [Palsberg 1994, 1995].[2] Heintze and McAllester [1997b] later gave a unified presentation of set/constraint-based analysis. Furthermore, they investigated the computational complexity of flow analysis for a language with pattern matching over undeclared data constructors.

Cousot and Cousot [1995] clarified how grammar-based, constraint-based, and set-based analyses are instances of the general theory of *abstract interpretation*. They furthermore suggested advantages of an abstract interpretation formulation: use of *widening* and easy integration with other non-grammar domains. Gallagher and Peralta [2001] investigated such a regular tree language domain in the context of partial evaluation.

As an extension to Heintze's set-based analysis [Heintze 1992], Flanagan and Felleisen [1997, 1999] suggested *componential set-based analysis*. Their analysis works by extracting, simplifying, and serializing constraints separately for each source program file. A later pass combines the serialized constraints into a global solution. One advantage of the approach is the avoidance of the re-extraction of constraints from an unmodified file upon later re-analysis. Flanagan used the analysis for a static debugger [Flanagan 1997]. Meunier et al. [2005] later identified that selectors complicated the analysis and suggested using conditional constraints in the style of Palsberg [1995] instead.

Henglein's simple closure analysis [Henglein 1992] and Bondorf and Jørgensen's efficient closure analysis for partial evaluation [Bondorf and Jørgensen 1993] are also based on constraints. However, they use a different form of constraints, namely *equality constraints*, that can be solved by unification in almost linear time [Aiken 1999].

5.2. Type-Based Analysis

Type-based analysis is an ambiguous term. It is used to refer to analyses of typed programs, as well as analyses expressed as *enriched type-systems*. Mossin [2003] distinguishes the two, by referring to them as Church-style analysis and Curry-style analysis, respectively. Mossin's Ph.D. dissertation [Mossin 1997] provides a comprehensive overview of type-based control-flow analysis, including typed versions of simple closure analysis and 0-CFA in addition to the already mentioned contributions. The field of type-based analysis is big enough to deserve a separate treatment. We refer to Palsberg [2001] and Jensen [2002] for general surveys of type-based analysis.

Banerjee et al. [2001] proved the correctness of two program transformations based on control-flow analysis. Their analysis operates on a simply typed language. It is a type-based analysis with a subtype relation on control-flow types. The analysis resembles one of Heintze's [1995] systems modulo Heintze's super type, for handling otherwise untypable programs.

[2]Conditional constraints have since been shown to be equally expressive to a constraint system with selectors [Aiken 1999] such as Heintze's [1992].

Recently, Reppy [2006] presented an analysis that utilizes the type abstraction of ML to increase precision, by approximating arguments of an abstract type with earlier computed results of the same abstract type. Whereas other analyses have relied on the typing of programs, e.g., for simple approximations [Tolmach 1997; Tolmach and Oliva 1998], or for termination or time complexity [Heintze and McAllester 1997a; Mossin 1997], Reppy exploits the static guarantees offered by the type system to boost the precision of an existing analysis.

5.3. Equivalences

A line of work has investigated equivalences between type systems, analyses, parsing techniques, and data-flow and context-free grammar reachability.

5.3.1. Equivalences between Type Systems and Analyses. Palsberg and O'Keefe [1995] showed that a 0-CFA-based safety analysis (cf. Section 8.2) is equivalent to a type system due to Amadio and Cardelli [1993] with subtyping and recursive types. In a followup paper, Palsberg and Smith [1996] proved the same type system (and consequently also the safety analysis) equivalent to a constrained type system without universal quantification. Independently, Heintze showed a number of similar equivalences [Heintze 1995] between equality-based and subset-based control-flow analyses and their counterpart type systems with simple types and subtyping, including the preceding.

Palsberg later gave counterexamples refuting Heintze's claim that equality-based flow analysis is equivalent to a type system with recursive types [Palsberg 1998]. He then showed a type system with recursive types and a very limited form of subtyping, which is equivalent to equality-based flow analysis. His type system furthermore includes top and bottom types to enable typability of otherwise untypable terms.

Palsberg and Pavlopoulou [2001] have since formulated a framework for studying equivalences between polyvariant flow analyses and type systems and used it to develop a flow-type system in the style of the Church group [Wells et al. 2002]. In a followup paper, Amtoft and Turbak [2000] formulated a framework for studying translations between polyvariant flow analyses and type systems. The translations between the two are *faithful* in that no information is lost by translating types to flows and back again (and vice versa). The framework is furthermore able to express both call-string polyvariance (such as Shivers's k-CFA) as well as argument-based polyvariance (such as Agesen's CPA).

5.3.2. Equivalences and Reductions between Control-Flow Analysis Problems and Other Problems. The context-free language (CFL) reachability problems seek to determine reachability between nodes of a directed, edge-labeled graph under the condition that the labels on a path must belong to a given context-free grammar. The problems come in variants with single or multiple source and target nodes.

Melski and Reps [2000] have shown how to convert, in linear time, a class of set constraints into a corresponding CFL reachability problem and vice versa. They also showed how to extend the result to Heintze's class of *ML set constraints* [Heintze 1994], which can express closure analysis. In doing so, they proved both set constraint solving problems to be complete for polynomial time. Kodumal and Aiken [2004] have shown a particularly simple reduction from a CFL reachability problem to set constraints in the special case of Dyck context-free languages, that is, languages of matching parentheses.

Recently, Chaudhuri [2008] gave $O(n^3/\log n)$ unit cost RAM algorithms for CFL reachability and for the related problem of *recursive state machine reachability*. The improvement is based on a known set compression technique [Rytter 1985]: instead of operating over individual values, the analysis operates over words each representing a subset of values for which the standard set operations can be precomputed. When

combined with the equivalence result of Melski and Reps [2000], the two constitute an (indirect) sub-cubic set-based analysis algorithm. The improved complexity of CFL-reachability furthermore benefits the two type-based polymorphic analysis algorithms of Rehof and Fähndrich [2001]. Midtgaard and van Horn [2009] provide a detailed explanation of this algorithm and apply Rytter's technique directly to derive sub-cubic algorithms for three increasingly precise 0-CFA variants incorporating reachability.

Heintze and McAllester [1997c] proved a number of problems to be 2NPDA-complete, including *control-flow reachability*, which is another formulation of 0-CFA as a decision problem. Surprisingly, Rytter [1985] showed that every 2NPDA language can be recognized in $O(n^3/\log n)$ time on a unit cost random-access machine. However, Heintze and McAllester's completeness results combined with Rytter's improvement did not break the cubic-time boundary of 0-CFA, as their reduction of control-flow reachability to 2NPDA incurs a logarithmic blow-up of the input program.

In addition to the control-flow reachability problem, Heintze and McAllester [1997c] proved two other problems to be 2NPDA-complete: *data-flow reachability* (in a formulation equivalent to the set constraints of Melski and Reps) and the complement of *Amadio-Cardelli typability* [Amadio and Cardelli 1993].

5.3.3. Equivalences between First-Order CFA and LR Parsing. After decades of research, the expression control-flow analysis has become synonymous with statically predicting the operators of call sites. In the presence of tail-call optimization, functions may not return to their callers at runtime. As a consequence, the *return-flow* may not be immediately apparent from the program text.

Debray and Proebsting [1997] have investigated control-flow analysis for a first-order language with tail-call optimization. Their analysis determines to which point control is returned upon procedure completion. They showed how to construct a program-specific context-free grammar on which traditional parsing techniques can be used to extract this preceding property. In particular, they showed how *follow sets* correspond to a first-order 0-CFA and how LR(1) items correspond to a first-order 1-CFA.

Midtgaard and Jensen [2009] recently derived a higher-order counterpart to the preceding first-order 0-CFA by abstract interpretation of a simple stack-based abstract machine. Their resulting CFA determines where function calls and returns transfer control to in the presence of first-class functions and tail-call optimization.

5.4. Linear Logic, Game Semantics, and Proof Nets

A line of papers have investigated analyses rooted in linear logic. Jensen and Mackie [1996] develop analyses based on Girard's Geometry of Interaction semantics. Starting with a graph representation of lambda-terms and a path algebra, they derive an abstract machine from which they extract a set of analysis equations. From the equations, they illustrate how to derive three analyses: strictness analysis, usage analysis (an analysis that detects unused variables), and control-flow analysis.

Malacaria and Hankin [1998] presented a cubic-time flow analysis based on game semantics. The analysis approximates *linear head reduction* of (simply typed) PCF terms. Their algorithm, like Heintze and McAllester's [1997a] sub-cubic algorithm, is based on graph structure and requires an amount of eta-expansion. In a later paper, Hankin et al. [2002] developed another cubic-time 0-CFA based on game semantics in which terms are represented as proof nets. Based on the semantics, their analysis algorithm first extracts a graph, for which the dynamic transitive closure is computed as a second step.

As mentioned in Section 2.5, Van Horn and Mairson [2007, 2008b] have used linear lambda calculus terms repeatedly to prove lower bounds on the time complexity of

control-flow analyses. For an introduction to linear lambda calculus, proof nets, context semantics, and their relations to linear logic and game semantics, we refer to Mairson [2002].

5.5. Specification-Based

Nielson and Nielson have championed the specification approach to program analysis [Nielson and Nielson 1997, 1998, 1999; Gasser et al. 1997]. A specification is formulated as a series of declarative demands that a valid analysis result must fulfill. In effect, a specification-based analysis constitutes an *acceptability relation* that verifies a solution, as opposed to computing one. A corresponding analysis can typically be staged in two parts: first, the demands can be serialized into a set of constraints; second, the set of constraints can be solved iteratively.

Nielson and Nielson coined the term *flow logic* for such a tight declarative format describing analyses [Nielson and Nielson 1998]. They showed how such a specification can be gradually transformed into a more verbose constraint-based formulation [Nielson and Nielson 1998]. Their gradual transformation towards constraints involves formulations in terms of *(extended) attribute grammars*, which should be compared to the previously mentioned grammar/constraint correspondence.

Indeed, a specification-based analysis offers a constructive way of calculating a solution. Cachera et al. [2005] have illustrated this point by formalizing a specification-based analysis in constructive logic using the Coq proof assistant.

5.6. Abstract Interpretation-Based

As pointed out by Aiken [1999], the term *abstract interpretation* is used interchangeably to refer to both monotone analyses defined compositionally on the source program and to a formal program analysis methodology initiated by Cousot and Cousot [1977a, 1992a], which suggests that analyses should be derived systematically from a formal semantics, for example, through Galois connections. We refer here to abstract interpretation in the latter meaning.

In his Ph.D. dissertation, Harrison III [1989] used abstract interpretation of Scheme programs to automatically parallelize them. Harrison treats an imperative Scheme core language with first-class continuations. His starting point is a transition-system semantics based on procedure strings, in which functions in the core language are represented as functions at the meta level. A second refined semantics represents functions as closures. This semantics is then gradually transformed and abstracted into a computable analysis. The result serves as the starting point for a number of parallelizing program transformations.

Shivers's analysis [1991a] is based on abstract interpretation. His analysis is derived from a non-compositional denotational semantics based on closures. He does not use Galois connections. Instead, his soundness proofs are based on lower adjoints, that is, abstraction functions mapping concrete objects to abstract counterparts.

Ayers also treated higher-order flow analysis based on abstract interpretation in his Ph.D. dissertation [1993]. His work is similar to Shivers in that his analysis works on an untyped CPS-based core language. Ayers gradually transforms a denotational continuation semantics of Scheme into a state transition system based on closures, which is then approximated using Galois connections.

In a line of papers, Schmidt [1995, 1997, 1998], has investigated abstract interpretation in the context of operational semantics. Schmidt explains the traces of a computation as *paths* or *traces* in the tree induced by the inference rules of an operational semantics. A tree is then abstracted into an approximate *regular* tree that safely models its concrete counterpart and is finitely representable.

5.7. Minimal Function Graphs and Program-Dependent Domains

The *function graph* is a well-known formal characterization of a function as a set of argument-result pairs. Characterizing a function for all arguments in a program analysis can lead to a combinatorial explosion. The general idea of considering only necessary arguments in an analysis was initially suggested by Cousot and Cousot [1977b]. The idea of considering only necessary arguments in the context of function graphs was suggested and named *minimal function graphs* by Jones and Mycroft [1986].

Jones and Rosendahl [1997] formulated closure analysis in terms of minimal function graphs. Their analysis is formulated for a system of curried recursive equations, where all function abstractions are named and defined at the top level. Jones and Rosendahl can thereby represent an abstract procedural value by the name of its origin and a natural number indicating how many arguments the function has been partially applied to.

Few control-flow analyses defined as functions on an expression-based language give approximate characterizations for all possible expression arguments. Instead, analyses are often specified as finite partial functions or as total functions on a program-dependent domain [Palsberg 1995; Nielson et al. 1999], which is finite for any given (finite) program.

5.8. Frameworks

Jones provided the first CFA framework in his pioneering paper [1981b]. The framework approximates the reachable states of a call-by-name evaluating abstract machine. The analysis is parameterized by a token set representing *context stacks*. Jones illustrates the expressivity of the framework by providing three instantiations: an exact execution with an infinite set of tokens, a precise analysis reminiscent of a 1-CFA, and a less-precise analysis reminiscent of a flow-sensitive 0-CFA.

Following Shivers's presentation [1988, 1991a], a line of work has investigated control-flow analysis frameworks in order to better understand the commonalities between different analyses. Stefanescu and Zhou [1994] developed one such framework for expressing CFAs. Their framework is based on term-rewriting sequences of a small closure-based core language. The analysis is given in the form of a system of traditional data-flow equations, and their approximations are formulated as static partitions based on the call sites. They suggest two such partitions: the unit partition corresponding to 0-CFA and a finer partition corresponding to Shivers's 1-CFA [1991a].

Jagannathan and Weeks [1995] developed a framework based on flow graphs and instantiate it to 0-CFA, a polynomial-time 1-CFA, and an exponential-time 1-CFA. Furthermore, Jagannathan and Weeks proved their 0-CFA instantiation equivalent to Heintze's set-based analysis [Heintze 1994]. In a later paper, Jagannathan et al. [1997] developed a type-directed flow analysis framework for a subset of *System F*. The framework is parameterized over *instances*—an abstraction reminiscent of contours—each of which represents a set of evaluation contexts. Jagannathan et al. instantiated the framework with a 0-CFA and a polyvariant analysis based on explicit type applications. As the latter analysis is not guaranteed to terminate, they considered three variations: a variant based on bounded types, a restriction of the source language to core ML for which the analysis terminates, and a traditional let-polymorphic CFA. Finally, they defined a preorder on instantiations for formally ordering and comparing the individual analyses.

Nielson and Nielson [1997] developed a general non-compositional analysis framework formulated as a co-inductive definition. They instantiate the framework with a 0-CFA, k-CFA, a polymorphic splitting analysis, and a uniform k-CFA. In a later

paper, Nielson and Nielson [1999], develop a framework for control-flow analysis of a functional language with side effects. The approach incorporates ideas from interprocedural data-flow analysis. To illustrate the generality of the framework, they instantiate it with k-CFA in the style of Shivers [1991a], with call strings in the style of Sharir and Pnueli [1981], and with *assumption sets* [Nielson et al. 1999].

Ashley and Dybvig [1998] developed an experimental flow-analysis framework for the *Chez Scheme* compiler. Their framework is flow sensitive. It is furthermore parameterized by an abstract allocation function and a projection operator. By different instantiations, they obtain a number of flow-sensitive analyses: a 0-CFA, a 1-CFA, a Cartesian-product variant, and a sub-0-CFA, the latter analysis being a sub-cubic 0-CFA variant that allows only a limited number of updates to each cache entry. Their results showed that the sub-0-CFA instantiation enabled effectively the same optimizations in the underlying compiler as the 0-CFA.

Smith and Wang [2000] developed a polyvariant flow-analysis framework based on *constrained types*. Their framework differs from other frameworks in that a function is not repeatedly analyzed in different contexts. Instead, they achieve polyvariance by different instantiations of the function's type scheme. They illustrate the expressiveness of their framework by instantiating it with a k-CFA, a Cartesian-point algorithm in the style of Agesen [1995], and a new Cartesian-point algorithm incorporating data polymorphism.

6. EXTENSIONS

In this section, we consider a number of extensions and variations to the traditional analyses mentioned so far.

6.1. Reachability

A standard control-flow analysis analyses all terms of a source program, regardless of whether they will be reached during execution or not. A line of work therefore extends control-flow analysis with dead-code detection, which can be utilized to minimize the execution time of a full analysis. The technique has been named *reachability* [Ayers 1992, 1993] or *demand-driven analysis* [Biswas 1997].

Ayers [1992, 1993] illustrated how limiting the analysis to the live parts of the program can yield a speedup in analysis time. Palsberg and Schwartzbach [1995] extended a constraint-based CFA with dead-code detection in their formulation of a *safety analysis*. The abstract semantics of Jagannathan and Weeks's framework [1995] contains a reachability predicate to minimize the size of the generated flow graphs. Gasser et al. [1997] formulated a control-flow analysis for Concurrent ML [Reppy 1999]. Starting with an abstract specification, they incorporate reachability predicates and finally show how to generate constraints from the specified analysis. Reachability predicates were later used by Wand [2002] to extend a constraint-based analysis to more closely reflect call-by-name and call-by-value evaluation. Biswas [1997] augmented a set-based analysis in the style of Heintze [1992] with boolean constraints to formulate a demand-driven flow analysis for detecting dead code in higher-order functional programs. Midtgaard and Jensen [2008, 2009] recently calculated demand-driven 0-CFAs using abstract interpretation. Both their analyses approximate the reachable states of an abstract machine and are calculated by composing well-known Galois connections.

6.2. Must Analysis and Abstract Cardinality

Whereas much work in control-flow analysis has focused on inferring *may* information, Steckler and Wand [1997] developed a flow-sensitive *must analysis* [Nielson et al. 1999] based on which they prove the correctness of *lightweight closure conversion*. The first phase consists of a flow-sensitive CFA. In two subsequent phases, their algorithm infers

a common procedure invocation *protocol* between caller and callee, as well as *invariant sets*, that is, a set of variables whose value must agree across the caller's and callee's environments. The worst-case time complexity of the whole analysis remains cubic.

Jagannathan et al. [1998] formulated a constraint-based must analysis for a higher-order functional language. The algorithm repeatedly alternates between computing approximate control-flow and cardinality information when given approximate reachability information and vice versa. Since the involved control-flow analysis alone has cubic worst-case time complexity, the entire analysis is quartic. The analysis determines whether all bindings of a given variable reachable from each program point refer to the same value, thereby enabling lightweight closure conversion. Their analysis determines a related property for reference cells that enables other optimizing transformations.

Might and Shivers [2006b] formulated *abstract reachability* and *abstract cardinality* as separate extensions to off-the-shelf control-flow analyses. The former improves precision of analyses by performing an abstract garbage collection of any unreachable abstract bindings. The latter helps to infer equalities of concrete values, thereby enabling environment analysis, for example, lightweight closure conversion. They observe that the increased precision actually speeds up the running time of the analysis, but they do not report the time complexity of the analysis.

Inspired by advances in shape analysis of imperative programs, Might [2010] has recently developed a CFA enriched with *anodization*, which is an enhancement technique to boost precision of a coarse, collapsing analysis. Anodization works by separately tracking bindings that represent only one concrete binding. The resulting analysis is useful for determining equality among concrete bindings based on equality of their abstract counterparts. Might considers a range of use cases of such equalities, for example, inlining functions in the presence of (potentially differently bound) free variables. He further extends the analysis approach by incorporating *binding invariants*, that is, a form of relational analysis over predicates. The extended analysis can be used for *generalized environment analysis*, that is, determining when two different runtime environments agree on two different bindings.

6.3. Modular and Separate Analysis

Traditionally, control-flow analysis is considered a whole-program analysis, that is, it assumes the entire program text is available. A line of work has investigated modular or separate control-flow analyses. Shivers [1988, 1991a] initially addressed the issue by over-approximating the calls to *external procedures* as well as later *external calls* to any escaping procedures. A similar approach was taken by Ashley and Dybvig [1998].

Tang and Jouvelot [1994] combined a type and effect system with a control-flow analysis in the style of Shivers's 1-CFA [Shivers 1991a] to achieve separate abstract interpretation. Their approach is separated into two phases. First, the control-flow effect system approximates the initial contour and value environments. Second, the output is translated into starting points for re-analysis using the more precise 1-CFA. The approach extends earlier work that formulated a control-flow effect system [Tang and Jouvelot 1992]. In addition to Mossin's modular quadratic-time analysis, both of his type-based formulations of simple-closure analysis and 0-CFA enjoy the principal typing property of their corresponding type-system and are hence modular [Mossin 1997].

Faxén [1997] presented a polymorphic constraint-based analysis for untyped programs inspired by type systems and type inference—in particular, recursive types, union types, subtyping, and System-F polymorphism. As type inference for such a system is known to be undecidable, his analysis algorithm is sound but not complete. Contrary to standard CFA's, the analysis propagates flow variables and solves contraints

incrementally. As a consequence, the analysis is modular. The analysis is, however, preliminary, and as such, Faxén provides no correctness proof.

Banerjee [1997] developed a modular and polyvariant control-flow and type-inference system that can type a superset of all ML programs. The analysis is formulated as an instrumentation of the rank 2 intersection type system, and it relies on the principal typing property hereof [Jim 1996]. It is therefore able to analyze code fragments containing free variables. The analysis works by generating and reducing flow constraints underway in a modular manner. In a followup paper, Banerjee and Jensen [2003] formulated a modular and polyvariant control-flow analysis based on rank 2 intersection types for simply typed programs with recursive function definitions. Both analyses are compositional and modular in that the analysis of an expression can be calculated by combining the analyses of its subexpressions using intersection types without re-analysis of any subexpressions. The already mentioned componential set-based analysis of Flanagan and Felleisen [1999] is not modular by this definition, as it involves computing a global analysis over preprocessed constraints. Flanagan and Felleisen's analysis furthermore differs from the preceding in that it is based on first-order constraints with selectors, whereas the analyses of Banerjee and Jensen are higher-order (with function types).

Lee et al. [2002] describe a modularized 0-CFA. The analysis is polyvariant in the modules of the program, for which the authors coin the term *module-variant*. Modules are analyzed bottom-up, for example, separately in topological order of their (acyclic) dependencies. The resulting analysis is more precise than a 0-CFA because of the module variance.

Meunier et al. [2006] presented a set-based analysis for modules with *contracts*, that is, a form of assertions regarding the input-output behavior of functions. The analysis is structured in three parts. The first part labels the subexpressions of the program. The second part lifts nested contracts out of subexpressions. Finally, the third part generates and solves conditional constraints in the style of Palsberg [1995]. By construction, the analysis is inherently a whole-program analysis, since conditional constraints consider all abstraction-application pairs in the program, but Meunier et al. argue that a module can be (whole-program) lifted and analyzed separately. The worst-case complexity of the analysis is $O(n^6)$, partly because the contract lifting may duplicate code in order to analyze each module reference in isolation. Confusingly, they use the term *modular analysis* to mean "analysis of a module," rather than separate or compositional analysis.

7. FORMULATION ISSUES

7.1. Evaluation-Order Dependence

Flow-sensitive analyses of functional languages can potentially model evaluation order and strategy, for example, a flow-sensitive analysis for a call-by-value language with left-to-right evaluation could potentially model the directed program flow through operator to operand for an application. Most often, the effect is achieved by first sequentializing the program. A flow-insensitive analysis approximates all evaluation orders and strategies.

Reynolds's seminal paper [1969] inspired Jones to develop control-flow analyses for lambda expressions under both call-by-value [Jones 1981a] and call-by-name [Jones 1981b] evaluation. Shivers formulated and proved his analysis sound for a CPS language, which by nature is evaluation order-independent. Sestoft proved his closure analysis sound with respect to a strict call-by-value semantics [Sestoft 1988] and a lazy call-by-name semantics [Sestoft 1991]. Palsberg [1995] then claimed the soundness of closure analysis with respect to general β-reduction. A flaw in his proof was later

pointed out and corrected by Wand and Williamson [2002]. Faxén [1995] took a more operational route by formulating a polymorphic type-based flow analysis of lazy functional programs with explicitly forced evaluation and delayed computation (thunks). In an unpublished report, Wand [2002], then compared prior soundness results with respect to different semantics.

7.2. Prior Term Transformation

A number of analyses operate on a normalized core language, such as CPS or recursive equations, in the same way as a number of algorithms over matrices or polynomials operate on normal forms.

Jones [1987] simplified his earlier analysis approach by limiting his input to recursive equations as obtained, for example, by lambda lifting [Johnsson 1985]. An extended version of Jones's chapter with correctness proofs has recently been published [Jones and Andersen 2007]. Sestoft's analysis was also specified for recursive equations [Sestoft 1988, 1989]. Shivers argued that in CPS, lambda expressions capture all control flow in one unifying construct. As a consequence, he formulated his original analyses for sequentialized source programs in CPS [Shivers 1988, 1991a] and continues to do so [Might and Shivers 2006a]. Ayers's analysis was also formulated for a core language in CPS [Ayers 1993]. The flow analysis of Ashley and Dybvig operates on sequentialized source programs in a variant of CPS [Ashley and Dybvig 1998].

Consel and Danvy [1991] pointed out that CPS transforming a program could improve the outcome of a binding-time analysis and Muylaert-Filho and Burn [1993] showed a similar result for strictness analysis. Sabry and Felleisen [1994] then gave examples showing that prior CPS transformation could either increase or decrease precision when comparing the output of two constant-propagation analyses. They attributed increased precision to the duplication of continuations and decreased precision to the confusion of return points. It was later pointed out [Damian and Danvy 2003b; Palsberg and Wand 2003], however, that Sabry and Felleisen were comparing a flow-sensitive analysis to a flow-insensitive analysis.

Damian and Danvy [2003b] proved that a non-duplicating CPS transformation does not affect the precision of a flow-insensitive textbook 0-CFA. They also proved that CPS transformation can improve and does not degrade the precision of binding-time analysis. Independently, Palsberg and Wand [2003] proved that a non-duplicating Plotkin-style CPS transformation [Plotkin 1975; Danvy and Filinski 1992] does not change the precision of a standard constraint-based 0-CFA, a result that Damian and Danvy [2003a] extended to a 'one-pass' CPS transformation that performs administrative reductions. Midtgaard and Jensen [2009] recently compared two CFAs with reachability: one operating over programs in CPS and one operating over programs in direct style. They relate reachability, calls, and returns across the two analyses and prove that the analysis of a direct-style program and the analysis of its (non-duplicating) CPS transformed counterpart operate in lock step. In conclusion, a duplicating CPS transformation may improve the precision of a 0-CFA, and a non-duplicating CPS transformation does not affect its precision.

7.3. Cache-Based Analysis and Iteration Order

Hudak and Young [1991] introduced the idea of *cache-based* collecting semantics, in which the domain of answers of the analysis equations is not an abstract answer but, rather, a function mapping (labeled) expressions to abstract answers. As a result, a cache is passed to and returned from all equations of the analysis, which yields an answer mapping all subexpressions to abstract values. The advantage of this approach is that the specification of the analysis itself is already close to an implementation.

Shivers's analysis is cache-based [Shivers 1991a]. His implementation [Shivers 1991a], however, has a global cache which is updated through assignments—a well-known alternative to threading a value through a program. The cache-based formulation has since influenced many subsequent analyses [Bondorf 1991; Palsberg 1995; Nielson and Nielson 1997].

In a cache-based analysis, the iteration strategy is mixed with the equations of the analysis. In the words of Schmidt, many closure analyses "mix implementation optimizations with specifications and leave unclear exactly what closures analysis is" [Schmidt 1995]. The alternative is to separate the equations of the analysis from the iteration strategy for solving them. The advantage of separating them is that one can develop and calculate an analysis focusing on soundness of the analysis and later experiment with different iteration strategies for calculating a solution.

Cousot and Cousot [1995] have noted that several analyses using regular tree grammars incorporate an implicit *widening operator* to ensure convergence. Their point also applies to the equivalent cache-based constraint analyses [Palsberg 1995]: the joining of consecutive *cache iterates* constitutes a widening. To keep an analysis as precise as possible, one should instead widen explicitly, placing a minimal number of widening operators to still ensure convergence [Bourdoncle 1993]. Deutsch [1997] and Blanchet [1998] have used this approach in the context of escape analyses. Bourdoncle [1993] suggested different iteration strategies, some of which are applicable to analyzing higher-order programs. He concluded more work is needed in the higher-order case.

7.4. Compositionality

Keeping an analysis compositional prevents it from diverging by recursively analyzing the same terms repeatedly (it may however still diverge for other reasons). Furthermore, one can reason about a compositional analysis by structural induction. Different means have been used to prevent non-compositional analyses from repeatedly analyzing the same terms: in an unpublished technical report, Young and Hudak [1986], invented *pending analysis*, of which Shivers's *time-stamps* are a variant [Shivers 1991b, 1991a]; and Ashley and Dybvig [1998] use a similar concept which they name *pending sets*. A related technique is the worklist algorithm from data-flow analysis [Kildall 1973; Nielson et al. 1999].

The original formulation of 0-CFA in Shivers's Ph.D. dissertation [Shivers 1991a] is not compositional. The formulation in the later paper proving the soundness of the approximation is however compositional [Shivers 1991b, p. 196]. Shivers's implementation [Shivers 1991a] used a time-stamping approach to ensure convergence on recursive programs. The formal correctness of time-stamping was later established by Damian [2001]. Neither Serrano's nor Reppy's 0-CFA formulations are compositional [Serrano 1995; Reppy 2006]. In order to avoid re-analyzing function bodies (or looping on recursive functions), Reppy's analysis passes around a cache of function-result approximations.

Initially, Nielson and Nielson's specifications were non-compositional and defined by co-induction [Nielson and Nielson 1997; Gasser et al. 1997], but they were later reformulated compositionally [Nielson and Nielson 1998, 1999] (in which case induction and co-induction coincide [Nielson et al. 1999]).

The context-sensitive analyses—the k-CFA formulation of Shivers [1991a], the polymorphic splitting formulation of Wright and Jagannathan [1998], and the uniform k-CFA formulation of Nielson et al. [1997, 1999] are non-compositional. The analysis framework of Nielson and Nielson's later paper on higher-order flow analysis-supporting side effects [Nielson and Nielson 1999] is, however, compositional, as is Rehof and Fähndrich's [2001] context-sensitive flow analysis of typed higher-order programs.

7.5. Abstract Compilation and Partial Evaluation of CFA

Boucher and Feeley [1996] illustrated two approaches for eliminating the interpretive overhead of an analysis. They group these two approaches under the name *abstract compilation*. Their first approach serializes the program-specific analysis textually in a file that is later interpreted, for example, using the Scheme *eval* function. Their second approach avoids the interpretive overhead and the I/O of the preceding serialization by utilizing the closures of the host language. Ashley and Dybvig [1998] noted that the prototype implementation of their analysis is staged. In their own words, "code is compiled to closures," that is, they are effectively performing abstract compilation.

Boucher and Feeley [1996] suggested two optimizations, namely η-reduction and static lookup of constants and lambda expressions. They noted abstract compilation can be seen as a form of *partial evaluation*, where the analysis is a curried function of two arguments, of which the static (known) argument is the source program to be analyzed.

Damian [1999] implemented an interpreter for a small imperative language in which he encodes a variant of Shivers's 0-CFA. He then specializes the interpreter with respect to the analysis and a source program and reports relative speedups on par with Boucher and Feeley's [1996] results.

In a related technical report, Amtoft [1999], partially evaluates two constraint interpreters with respect to a set of (program-specific) CFA constraints (on the same set of benchmarks [Boucher and Feeley 1996]). He compares the two to their unspecialized counterparts and reports unmanageable residual code size in the one case and smaller speedups in the other. When reading his results, one should keep in mind that a constraint-based analysis has already eliminated the (repeated) interpretive overhead of the original source program. As such, Amtoft's results do not contradict the results of Boucher, Feeley, and Damian.

An interesting question is how the effectiveness of abstract compilation using closures (and their suggested optimizations) compares to an off-the-shelf constraint-based analysis, as the latter also incurs a certain overhead due to the serialization into a list of constraints and their later iterative interpretation. Such a comparison would, however, be relative, as the outcome would depend heavily on the implementation of closures in the host language.

The choice between the compositional interpreting analysis, the serialized/constraint interpreting analysis, and the compiled program analysis strongly echoes the choice between standard approaches to implementing programming languages: the compositional interpreter, the serialized/byte-code interpreter, and the compiled program.

8. APPLICATIONS AND RELATED ANALYSES

8.1. CFA in Compilers

With his flow analysis for the Liar Scheme compiler, Rozas [1984, 1992], was among the very first to implement a higher-order flow analysis. Though developed independently of Shivers, his analysis is "essentially the same as Olin Shivers's 0-CFA" [Rozas 1992]. His analysis formulation is graph-based with edges representing inclusion between program variables, and the analysis consists of computing the (dynamic) transitive closure hereof.

In an unpublished report, Siskind [1999], documents a precise flow-analysis framework for his optimizing Scheme compiler, Stalin. The framework combines flow analysis with several other analyses, including reachability, must-alias analysis, and escape analysis. His results indicate that the combined analysis enables an impressive amount of optimization; however he does not report the time complexity of the approach.

The flow analysis of the MLton Standard ML compiler operates on simply typed programs [Cejtin et al. 2000], that is, after *functors* and *polymorphism* have been eliminated. Both eliminating transformations are realized through code duplication, thereby increasing the size of source programs. Cejtin et al. use a standard constraint-based CFA with a few modifications: inclusions in datatype elimination and inclusions in tuple introduction and elimination are substituted with equalities, which are then solved by unification [Weeks 2006]. Apparently, the resulting analysis does not exhibit cubic-time behavior [Weeks 2006], which seems consistent with linear-time CFA on bounded-type programs [Heintze and McAllester 1997a; Mossin 1998].

Wells et al. [2002] have investigated a type-based intermediate language with intersection and union flow types. Their focus has been type-based compilation, rather than flow analysis. As such, they have inferred control-flow information using known flow analyses and afterwards decorated the flow types with the inferred flow information [Dimock et al. 2001; Banerjee and Jensen 2003].

8.2. Safety Analysis

Safety analysis is another analysis of untyped functional programs related to control-flow analysis. The basic goal is shared with that of type inference, that is, to statically guarantee the absence of runtime errors, such as applying the successor function to a lambda abstraction. Static type systems give such guarantees, however, at the price of ruling out otherwise useful untypable programs.

Palsberg and Schwartzbach [1995] coined the term *safety analysis* for such an analysis. Their analysis is based on a constraint-based CFA. It accepts strictly more programs than type inference (for simple types). Palsberg and Schwartzbach proved the analysis sound with respect to both call-by-value and call-by-name evaluation. Thiemann [1993] had earlier used the term safety analysis for an unrelated analysis for functional programs that detect when in-place updating is safe, that is, when it does not affect the outcome of programs.

8.3. Pointer Analysis

A related field of control-flow analysis is that of *pointer analysis*. However the body of research within pointer analysis is so big that it deserves an independent survey to do it justice. We refer to Hind [2001] for such a survey.

Pointer analysis in a language with function pointers shares some of the issues of higher-order functions in that the operator of a function call may not be apparent from the program text. As a consequence, such pointer analyses are sometimes said to support higher-order functions [Fähndrich et al. 2000]. However, one should note that even the formal semantics of a language with pointers, representing an ideal (uncomputable) analysis, already constitutes a crude approximation of the semantics of a higher-order language because it approximates closures with mere function pointers.

Two very significant contributions within the field bear a strong resemblance to control-flow analysis and deserve mentioning: Andersen's subset-based pointer analysis [Andersen 1994] and Steensgaard's equality-based pointer analysis [Steensgaard 1996b, 1996a]. Andersen's pointer analysis was formulated in terms of subset-inclusion constraints [1994], whereas Steensgaard's pointer analysis was formulated as a type system with a nonstandard set of types and unification [1996b].

Andersen's pointer analysis [1994] was conceived simultaneously with Palsberg's control-flow analysis in constraint form [1994] and Heintze's set-based analysis [1994]. On the other hand, Steensgaard's pointer analysis [1996b] postdates Henglein's [1992] technical report on closure analysis by type inference by four years, and indeed Steensgaard [1996b] cites Henglein [1991] as a source of inspiration for his unification-based pointer analysis.

More recently, Das [2000] has suggested a compromise between Andersen's and Steensgaard's algorithms. The pointer analysis is (like Steensgaard's) formulated as a type system. The type system allows only subtyping (containment) at the top level, as opposed to arbitrary subtyping (containment). Elsewhere, flow is propagated by unification. As a result, the algorithm has a quadratic worst-case time complexity.

8.4. Escape Analysis and Stackability

Control-flow analysis is concerned with *flows-from* information, that is, inferring the origin of function values that may occur at a given expression. Escape analysis, on the other hand, is concerned with *flows-to* information, that is, inferring where function values originating at a given lambda expression may occur.

The escape analysis of Section 2.3 provides a fast and practical static approximation that determines whether a function may escape its static scope. The analysis does so at the expense of crudely approximating higher-order programs. The basic idea applies to less crude approximations and to other data types as well, for example, a heap-allocated cons cell may be stack allocated if an analysis can infer that it will not escape its static scope.

Park and Goldberg [1992] devised an escape analysis for higher-order programs. Their initial analysis handled constants and procedural values [Goldberg and Park 1990]. It was later extended to handle lists [Park and Goldberg 1992]. The analysis was formulated as a forward analysis requiring exponential time even in the first-order case. Deutsch [1997] later gave an equally precise backwards analysis for first-order programs requiring only $O(n \log^2 n)$ time. Deutsch furthermore proved that any equally precise analysis on second-order functions is EXPTIME-hard, suggesting that an extension to higher-order functions would demand further approximation. Blanchet [1998] extended Deutsch's backwards escape analysis [Deutsch 1997] to a higher-order ML-like core language incorporating further approximation to ensure rapid termination.

Banerjee and Schmidt [1998] developed a static stackability criterion for simply typed call-by-value λ-calculus terms, that is, a static analysis that determines whether it is safe to evaluate a given λ-term with stack-allocated bindings. In order to do so, the analysis has to guarantee that bindings will not escape their static scope by being among the free variables of a returned closure. Their analysis is based on Sestoft's closure analysis. It is developed as a gradual transformation of an uncomputable specification into a computable specification.

Tang and Jouvelot [1992] formulated a control-flow effect system that infers control-flow information. The system infers both to which function a given expression may evaluate *and* which functions may be evaluated during the evaluation of a given expression. Tang and Jouvelot applied their analysis to infer escape information for procedures.

Hannan [1998] suggested a type-based escape analysis that detects whether variable bindings will escape their scope. The analysis is formulated as a type-directed translation from a simply typed source language into a target language where binding and look-up of stack variables are explicitly marked.

Serrano and Feeley [1996] presented a *storage use analysis*. Their analysis is an extension of Shivers's 0-CFA with modules and general data storage. They presented two applications of the analysis: *stack allocation* and *unboxing*.

Mohnen [1995] gave a (worst-case) quadratic-time algorithm for *inheritance analysis* for higher-order recursive equations with (monomorphic) data structures. His analysis can calculate whether functional arguments to a function, that is, closures, are inherited in the result, which is then encoded as a binary domain. When no inheritance is detected, the closures can be stack allocated. He also gave a measure for determining

whether a closure will only have one active activation at a time during execution, in which case he suggested static allocation. Mohnen's work extends earlier work by Hughes [1992], who formulated an inheritance analysis for lists in higher-order programs. Hughes's main application was compile-time garbage collection.

9. DISCUSSION

In this section, we discuss the systematic application of abstract interpretation techniques to control-flow analysis and their relevance to verification of higher-order programs.

9.1. Towards Abstract Interpretation Analyses

Most CFA approaches have been bottom-up in the sense that researchers have started with a given computable approximation and tried to improve it: Shivers refined 0-CFA into 1-CFA, 2-CFA and k-CFA [Shivers 1991a]. Wright and Jagannathan refined 0-CFA into polymorphic splitting [Wright and Jagannathan 1998], and Nielson and Nielson reformulated k-CFA into a uniform k-CFA [Nielson and Nielson 1997]. In contrast, the traditional abstract interpretation approach is top-down [Cousot and Cousot 1992a]. The starting point here is the (collecting) semantics, which is the most precise (and hence not computable) analysis. Through Galois connections or other approximations, the analysis is then gradually refined into something computable.

Much work in the field of semantics-based control-flow analysis has focused on ensuring that the proposed analyses compute safe approximations of the semantics [Palsberg 1995; Nielson and Nielson 1997]. In contrast, abstract interpretation offers best approximations [Cousot and Cousot 1992a] in the form of abstraction functions. Together with a companion concretization function, the two can form a Galois connection [Cousot and Cousot 1992a]. Few papers investigating control-flow analysis relate them by Galois connections [Ayers 1993; Stefanescu and Zhou 1994; Nielson and Nielson 1999]. Ayers's work on Galois connections is available only in his Ph.D. dissertation. Nielson and Nielson's work, on the other hand, focuses on proving three analyses correct with respect to a general specification (an uncomputable collecting semantics) in the context of a functional language with side effects, rather than relating the individual analyses. Nielson and Nielson earlier formulated the open question of how "to exploit Galois connections and widenings to systematically coarsen" [Nielson and Nielson 1997] control-flow analyses.

9.2. Finite and Infinite Domains

There continues to be some confusion about the applicability of infinite domains within the area of constraint-based analysis and the general area of abstract interpretation [Heintze 1992; Cousot and Cousot 1995; Aiken 1999]. The data representation of constraints (or the equivalent *regular-tree grammar* [Cousot and Cousot 1995]) is a finite representative on a potentially infinite domain. An abstract interpretation can always inherit that finite representation and their corresponding convergence guarantee [Aiken 1999, p.106] to yield a terminating analysis. To emphasize the point, Cousot and Cousot develop a finitary grammar domain [Cousot and Cousot 1995], thereby expressing constraint-based analysis as an instance of abstract interpretation. A lesson from abstract interpretation is that an infinite domain with widening and narrowing operators can offer more precision than a finite domain [Cousot and Cousot 1992b].

9.3. CFA with Widening

Few control-flow analyses have been formulated with an explicit widening operator. Steensgaard and Marquard [1994] included a dynamic widening operator in their (unpublished) analysis to ensure convergence in an infinite domain. Correspondingly,

Ashley and Dybvig [1998] included in their framework a projection operator similar to a widening operator to ensure rapid termination.

Schmidt [1998] outlined an alternative closure analysis that approximates environments less crudely. To still ensure termination of his analysis he suggested indexing environments by numbers: closure environments bound inside the environment of another closure have an index one less than their outer binding environment; and environments of index 0 are simply joined. Even though not completely formulated as such, Schmidt's approach can be interpreted as an indexed widening, as is well-known [Cousot and Cousot 1992a] in abstract interpretation. The k-bounded CFA of Sereni and Jones [2005, 2006, 2007] is based on the same idea of cutting off nested environments at depth k. The complexity of the analysis is however doubly exponential in k; hence, only practically feasible for $k \leq 2$ [Sereni 2006].

There is a clear line of research headed towards more precise modeling of contexts [Shivers 1991a; Nielson and Nielson 1997; Wright and Jagannathan 1998; Might and Shivers 2006a]. However, one will not get full benefit of a very precise context representation if code and environment components of closures are analyzed separately as independent attributes [Jones and Muchnick 1981]. The key to precise control-flow analysis may be to keep the code and its environment together in abstract closures, thereby obtaining a relational analysis [Jones and Muchnick 1981] as in the previously mentioned work by Steensgaard, Marquard, and Schmidt. Since closures can contain closures ad infinitum, one would need to introduce widening in order to ensure convergence of a fixed-point computation operating on such a domain.

9.4. Relevance

Serrano [1995] questioned the usefulness of the additional context component in a 1-CFA for an optimizing compiler, compared to a 0-CFA. A possible answer is as follows. One is not interested in context per se, that is, the analysis uses context as a refinement (to increase precision), but it is not essential in the result. Any compiler pass utilizing CFA information should therefore benefit from it, just as they would benefit from substituting an escape analysis with a 0-CFA. As a consequence, contexts should not necessarily be abstracted symbolically, as is traditional in CFA. Alternatively, contexts could be approximated numerically in order to distinguish them and still gain precision (as in the abstract interpretation analyses of Deutsch [1997], Blanchet [1998], and Venet [2002]).

Research by Waddell and Dybvig [1997] indicates that for a functional programming-language implementation, a rough CFA approximation backed up by a well-tuned inliner is sufficient for an effective compiler. However, with the advances in formal verification (and very precise analyses), for example, ASTRÉE [Cousot et al. 2005], one will still need precise control-flow analyses in order to bring the advances to verification of higher-order programs.

10. CONCLUSION

Over 30 years after Jones's initial flow analysis of lambda expressions [1981b], control-flow analysis has been the subject of a considerable amount of research. A range of useful analyses have been designed for programs with first-class functions, all of which differ in their precision and in their time and space complexity. As a result, analyses now come in many formulations. Some of them are available only as technical reports, and others not at all.

We have surveyed the field in an attempt to put structure to this body of research. In doing so, we have assembled context-sensitive and context-insensitive approximations

from both theory and practice, and we have classified analyzes according to their formulation.

As Nielson and Nielson pointed out [1997], a simple and systematic development of control-flow analyses utilizing the tools of abstract interpretation remains to be found. In two recent papers [Midtgaard and Jensen 2008, 2009], we have taken the first steps towards remedying this situation. Such a development may provide insight to extend recent developments in the verification of first-order programs to verifying higher-order programs.

ACKNOWLEDGMENTS

This article benefited from Olivier Danvy and Kevin Millikin's numerous comments and encouragement. We thank the anonymous reviewers for many helpful suggestions, insights, and improvements. Thanks are also due to Thomas P. Jensen and Janus Dam Nielsen for comments on an earlier version of this survey and to Fritz Henglein and David Van Horn for insightful discussions on control-flow analysis.

REFERENCES

AGESEN, O. 1995. The Cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*. W. G. Olthoff, Ed. Lecture Notes in Computer Science, vol. 952, Springer-Verlag, Berlin, 2–26.

AIKEN, A. 1999. Introduction to set constraint-based program analysis. *Sci. Comput. Program. 35,* 2-3, 79–111.

AIKEN, A. AND MURPHY, B. R. 1991a. Implementing regular tree expressions. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, 427–447.

AIKEN, A. AND MURPHY, B. R. 1991b. Static type inference in a dynamically typed language. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*. R. C. Cartwright, Ed., 279–290.

AIKEN, A. AND WIMMERS, E. L. 1992. Solving systems of set constraints (extended abstract). In *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science (LICS'92)*. A. Scedrov, Ed., 329–340.

AIKEN, A. AND WIMMERS, E. L. 1993. Type inclusion constraints and type inference. In *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture*, 31–41.

ALLEN, F. E. 1970. Control flow analysis. *SIGPLAN Not. 5,* 7, 1–19.

AMADIO, R. M. AND CARDELLI, L. 1993. Subtyping recursive types. *ACM Trans. Program. Lang. Syst. 15,* 4, 575–631.

AMTOFT, T. 1999. Partial evaluation for constraint-based program analyses. Tech. rep. BRICS-RS-99-45, BRICS, Department of Computer Science, University of Aarhus.

AMTOFT, T. AND TURBAK, F. A. 2000. Faithful translations between polyvariant flows and polymorphic types. In *Proceedings of the 9th European Symposium on Programming*. Lecture Notes in Computer Science, vol. 1782, Springer-Verlag, Berlin, 26–40.

ANDERSEN, L. O. 1994. Program analysis and specialization for the C programming language. Ph.D. dissertation, Computer Science Department, University of Copenhagen, Denmark.

APPEL, A. W. 1992. *Compiling with Continuations*. Cambridge University Press, Cambridge, UK.

APPEL, A. W. AND JIM, T. 1989. Continuation-passing, closure-passing style. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*. M. J. O'Donnell and S. Feldman, Eds., 293–302.

ASHLEY, J. M. AND DYBVIG, R. K. 1998. A practical and flexible flow analysis for higher-order languages. *ACM Trans. Program. Lang. Syst. 20,* 4, 845–868.

AYERS, A. E. 1992. Efficient closure analysis with reachability. In *Actes WSA Workshop on Static Analysis*. 126–134.

AYERS, A. E. 1993. Abstract analysis and optimization of Scheme. Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA.

BANERJEE, A. 1997. A modular, polyvariant, and type-based closure analysis. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*. 1–10.

BANERJEE, A., HEINTZE, N., AND RIECKE, J. G. 2001. Design and correctness of program transformations based on control-flow analysis. In *Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software (TACS'01)*, N. Kobayashi and B. C. Pierce, Eds., Lecture Notes in Computer Science, vol. 2215. Springer-Verlag, Berlin, 420–447.

BANERJEE, A. AND JENSEN, T. 2003. Modular control-flow analysis with rank 2 intersection types. *Math. Structures Comput. Sci. 13,* 1, 87–124.

BANERJEE, A. AND SCHMIDT, D. A. 1998. Stackability in the typed call-by-value lambda calculus. *Sci. Comput. Program. 31,* 1, 47–73.

BISWAS, S. K. 1997. A demand-driven set-based analysis. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*. 372–385.

BLANCHET, B. 1998. Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*. 25–37.

BONDORF, A. 1991. Automatic autoprojection of higher-order recursive equations. *Sci. Comput. Program. 17,* 1-3, 3–34.

BONDORF, A. 1993. Similix 5.1 manual. Tech. rep., DIKU, Computer Science Department, University of Copenhagen, Denmark. (Included in the Similix 5.1 distribution.)

BONDORF, A. AND JØRGENSEN, J. 1993. Efficient analyses for realistic off-line partial evaluation. *J. Funct. Program. 3,* 3, 315–346.

BOUCHER, D. AND FEELEY, M. 1996. Abstract compilation: A new implementation paradigm for static analysis. In *Proceedings of the 6th International Conference on Compiler Construction*. 192–207.

BOURDONCLE, F. 1993. Efficient chaotic iteration strategies with widenings. D. Bjørner, M. Broy, and I. V. Pottosin, Eds. Lecture Notes in Computer Science, vol. 735. Springer-Verlag, Berlin, 128–141.

CACHERA, D., JENSEN, T., PICHARDIE, D., AND RUSU, V. 2005. Extracting a data flow analyzer in constructive logic. *Theor. Comput. Sci. 342,* 1, 56–78.

CEJTIN, H., JAGANNATHAN, S., AND WEEKS, S. 2000. Flow-directed closure conversion for typed languages. In *Proceedings of the 9th European Symposium on Programming*. Lecture Notes in Computer Science, vol. 1782. Springer-Verlag, Berlin, 56–71.

CHAUDHURI, S. 2008. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th Annual ACM Symposium on Principles of Programming Languages*, G. C. Necula and P. Wadler, Eds. 159–169.

CLINGER, W. D. AND HANSEN, L. T. 1994. Lambda, the ultimate label or a simple optimizing compiler for Scheme. In *Proceedings of the ACM Conference on LISP and Functional Programming. LISP Pointers, 7*, 3. 128–139.

CONSEL, C. 1990. Binding time analysis for higher order untyped functional languages. In *Proceedings of the ACM Conference on LISP and Functional Programming*, M. Wand, Ed. 264–272.

CONSEL, C. AND DANVY, O. 1991. For a better support of static data flow. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture.* 496–519.

COUSOT, P. AND COUSOT, R. 1977a. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, R. Sethi, Ed. 238–252.

COUSOT, P. AND COUSOT, R. 1977b. Static determination of dynamic properties of recursive procedures. In *IFIP Conference on Formal Description of Programming Concepts*, E. J. Neuhold, Ed. 237–277.

COUSOT, P. AND COUSOT, R. 1992a. Abstract interpretation frameworks. *J. Logic Comput. 2,* 4, 511–547.

COUSOT, P. AND COUSOT, R. 1992b. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming (PLILP '92)*, M. Bruynooghe and M. Wirsing, Eds. Lecture Notes in Computer Science, vol. 631. Springer-Verlag, Berlin, 269–295.

COUSOT, P. AND COUSOT, R. 1995. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture*, S. Peyton Jones, Ed. 170–181.

COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. 2005. The ASTRÉE Analyser. In *Proceedings of the 14th European Symposium on Programming, Languages and Systems (ESOP '05)*. M. Sagiv, Ed. Lecture Notes in Computer Science, vol. 2618. Springer-Verlag, Berlin, 21–30.

DAMIAN, D. 1999. Partial evaluation for program analysis. Progress report, BRICS Ph.D. School, University of Aarhus. Available at http://www.brics.dk/~damian/.

DAMIAN, D. 2001. Time stamps for fixed-point approximation. In *Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics*, S. Brookes and M. Mislove, Eds. Electronic Notes in Theoretical Computer Science, vol. 45. Elsevier Science Publishers, Aarhus, Denmark, 43–54.

DAMIAN, D. AND DANVY, O. 2003a. CPS transformation of flow information, part II: Administrative reductions. *J. Funct. Program. 13,* 5, 925–934.

DAMIAN, D. AND DANVY, O. 2003b. Syntactic accidents in program analysis: On the impact of the CPS transformation. *J. Funct. Program. 13,* 5, 867–904. A preliminary version was presented at the ACM SIGPLAN International Conference on Functional Programming.

DANVY, O. AND FILINSKI, A. 1992. Representing control, a study of the CPS transformation. In *Special issue on the ACM Conference on LISP and Functional Programming*, M. Wand, Ed. Math. Structures Comput. Sci., *2*, 4. Cambridge University Press, Cambridge, U.K., 361–391.

DANVY, O. AND NIELSEN, L. R. 2001. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'01)*. H. Søndergaard, Ed. 162–174. Extended version available as the Tech. rep. BRICS RS-01-23.

DAS, M. 2000. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*. 35–46.

DEBRAY, S. K. AND PROEBSTING, T. A. 1997. Interprocedural control flow analysis of first-order programs with tail-call optimization. *ACM Trans. Program. Lang. Syst. 19,* 4, 568–585.

DEUTSCH, A. 1990. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*. 157–168.

DEUTSCH, A. 1997. On the complexity of escape analysis. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*. 358–371.

DIMOCK, A., WESTMACOTT, I., MULLER, R., TURBAK, F., AND WELLS, J. B. 2001. Functioning without closure: Type-safe customized function representations for Standard ML. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, X. Leroy, Ed. 14–25.

FÄHNDRICH, M., REHOF, J., AND DAS, M. 2000. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*. 253–263.

FAXÉN, K.-F. 1995. Optimizing lazy functional programs using flow inference. In *Proceedings of the Static Analysis 2nd International Symposium Static Analysis (SAS'95)*. Lecture Notes in Computer Science, vol. 983. Springer-Verlag, Berlin, 136–153.

FAXÉN, K.-F. 1997. Polyvariance, polymorphism and flow analysis. In *Selected Papers from the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*. Lecture Notes in Computer Science, vol. 1192. 260–278.

FLANAGAN, C. 1997. Effective static debugging via componential set-based analysis. Ph.D. dissertation, Rice University, Houston, Texas.

FLANAGAN, C. AND FELLEISEN, M. 1999. Componential set-based analysis. *ACM Trans. Program. Lang. Syst. 21,* 2, 370–416.

GALLAGHER, J. P. AND PERALTA, J. C. 2001. Regular tree languages as an abstract domain in program specialisation. *Higher-Order Symb. Comput. 14,* 2-3, 143–172.

GASSER, K. L. S., NIELSON, F., AND NIELSON, H. R. 1997. Systematic realization of control flow analyses for CML. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*. 38–51.

GOLDBERG, B. AND PARK, Y. G. 1990. Higher-order escape analysis: Optimizing stack allocation in functional program implementations. In *Proceedings of the 3rd European Symposium on Programming*, N. D. Jones, Ed. Lecture Notes in Computer Science, vol. 432. Springer-Verlag, Berlin, 152–160.

GUSTAVSSON, J. AND SVENNINGSSON, J. 2001. Constraint abstractions. In *Proceedings of the 2nd Symposium on Programs as Data Objects (PADO'01)*, O. Danvy and A. Filinski, Eds. Lecture Notes in Computer Science. Springer-Verlag, vol. 2053, Berlin, 63–83.

HANKIN, C., NAGARAJAN, R., AND SAMPATH, P. 2002. Flow analysis: Games and nets. In *The Essence of Computation: Complexity, Analysis, Transformations. Essays Dedicated to Neil D. Jones*. Lecture Notes in Computer Science, vol. 2566. 135–156.

HANNAN, J. 1998. A type-based escape analysis for functional languages. *J. Funct. Program. 8,* 3, 239–273.

HARRISON III, W. L. 1989. The interprocedural analysis and automatic parallelization of Scheme programs. *LISP Symbol. Comput. 2,* 3/4, 179–396.

HARRISON III, W. L. AND AMMARGUELLAT, Z. 1993. A program's eye view of Miprac. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, 512–537.

HEINTZE, N. 1992. Set-based program analysis. Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

HEINTZE, N. 1994. Set-based program analysis of ML programs. In *Proceedings of the ACM Conference on LISP and Functional Programming. LISP Pointers, 7*, 3. 306–317.

HEINTZE, N. 1995. Control-flow analysis and type systems. In *Proceedings of the 2nd International Symposium Static Analysis (SAS'95)*. Lecture Notes in Computer Science, vol. 983. 189–206.

HEINTZE, N. AND JAFFAR, J. 1990a. A decision procedure for a class of set constraints. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, J. Mitchell, Ed. 42–51.

HEINTZE, N. AND JAFFAR, J. 1990b. A finite presentation theorem for approximating logic programs. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*. 197–209.

HEINTZE, N. AND MCALLESTER, D. 1997a. Linear-time subtransitive control flow analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, R. K. Cytron, Ed. 261–272.

HEINTZE, N. AND MCALLESTER, D. 1997b. On the complexity of set-based analysis. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*. 150–163.

HEINTZE, N. AND MCALLESTER, D. 1997c. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS '97)*, G. Winskel, Ed. 342–351.

HENGLEIN, F. 1991. Efficient type inference for higher-order binding-time analysis. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture.* Lecture Notes in Computer Science, vol. 523. 448–472.

HENGLEIN, F. 1992. Simple closure analysis. Tech. rep. Semantics Report D-193, DIKU, Computer Science Department, University of Copenhagen.

HENGLEIN, F., MAKHOLM, H., AND NISS, H. 2005. Effect type systems and region-based memory management. In *Advanced Topics in Types and Programming Languages*, B. Pierce, Ed. The MIT Press, Cambridge, MA.

HIND, M. 2001. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01).* 54–61.

HUDAK, P. 1987. A semantic model of reference counting and its abstraction. In *Abstract Interpretation of Declarative Languages*. Fellis Harwood, 45–62.

HUDAK, P. AND YOUNG, J. 1991. Collecting interpretations of expressions. *ACM Trans. Program. Lang. Syst. 13,* 2, 269–290.

HUGHES, S. 1992. Compile-time garbage collection for higher-order functional languages. *J. Logic Comput. 2,* 4, 483–509.

JAGANNATHAN, S., THIEMANN, P., WEEKS, S., AND WRIGHT, A. 1998. Single and loving it: Must-alias analysis for higher-order languages. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*. 329–341.

JAGANNATHAN, S. AND WEEKS, S. 1995. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages*, P. Lee, Ed. 393–407.

JAGANNATHAN, S., WEEKS, S., AND WRIGHT, A. K. 1997. Type-directed flow analysis for typed intermediate languages. In *Proceedings of the 4th International Symposium on Static Analysis (SAS'97)*. Lecture Notes in Computer Science, vol. 1302. 232–249.

JENSEN, T. 2002. Types in program analysis. In *The Essence of Computation: Complexity, Analysis, Transformations. Essays Dedicated to Neil D. Jones*. Lecture Notes in Computer Science, vol. 2566. Springer-Verlag, Berlin. 204–222.

JENSEN, T. P. AND MACKIE, I. 1996. Flow analysis in the geometry of interaction. In *Proceedings of the 6th European Symposium on Programming*, H. R. Nielson, Ed. Lecture Notes in Computer Science, vol. 1058. Springer-Verlag, Berlin. 188–203.

JIM, T. 1996. What are principal typings and what are they good for? In *Proceedings of the 23rd Annual Symposium on Principles of Programming Languages*. 42–53.

JOHNSSON, T. 1985. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, J.-P. Jouannaud, Ed. Lecture Notes in Computer Science, vol. 201. Springer-Verlag, Berlin, 190–203.

JOHNSTON, J. B. 1971. The contour model of block structured processes. In *Proceedings of the ACM Symposium on Data Structures in Programming Languages*. *SIGPLAN Not. 6*, 2, 55–82.

JONES, N. D. 1981a. Flow analysis of lambda expressions. Tech. rep. PB-128, University of Aarhus, Denmark.

JONES, N. D. 1981b. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming.*, S. Even and O. Kariv, Eds. Lecture Notes in Computer Science, vol. 115. Springer-Verlag, Berlin, 114–128.

JONES, N. D. 1987. Flow analysis of lazy higher-order functional programs. In *Abstract Interpretation of Declarative Languages*. Ellis Harwood. 103–122.

JONES, N. D. AND ANDERSEN, N. 2007. Flow analysis of lazy higher-order functional programs. *Theor. Comput. Sci. 375,* 1-3, 120–136.

JONES, N. D. AND MUCHNICK, S. S. 1979. Flow analysis and optimization of LISP-like structures. In *Proceedings of the 6th Annual ACM Symposium on Principles of Programming Languages*, B. K. Rosen, Ed. 244–256.

JONES, N. D. AND MUCHNICK, S. S. 1981. Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Upper Saddle River, NJ. 380–393.

JONES, N. D. AND MUCHNICK, S. S. 1982. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, R. DeMillo, Ed. 66–74.

JONES, N. D. AND MYCROFT, A. 1986. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages*, M. S. Johnson and R. Sethi, Eds. 296–306.

JONES, N. D. AND ROSENDAHL, M. 1997. Higher-order minimal function graphs. *J. Funct. Logic Program.* 2.

KILDALL, G. 1973. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM Symposium on Principles of Programming Languages*, J. D. Ullman, Ed. 194–206.

KODUMAL, J. AND AIKEN, A. 2004. The set constraint/CFL reachability connection in practice. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, C. Chambers, Ed. 207–218.

KRANZ, D., KESLEY, R., REES, J., HUDAK, P., PHILBIN, J., AND ADAMS, N. 1986. Orbit: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, S. I. Feldman, Ed. 219–233.

KRANZ, D. A. 1988. Orbit: An optimizing compiler for Scheme. Ph.D. dissertation, Computer Science Department, Yale University, New Haven, CT. Res. rep. 632.

LANDIN, P. J. 1964. The mechanical evaluation of expressions. *Comput. J. 6,* 4, 308–320.

LEE, O., YI, K., AND PAEK, Y. 2002. A proof method for the correctness of modularized 0CFA. *Inform. Process. Lett. 81,* 4, 179–185.

MAIRSON, H. G. 2002. From Hilbert spaces to Dilbert spaces: Context semantics made simple. In *Proceedings of the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science*. M. Agrawal and A. Seth, Eds. Lecture Notes in Computer Science, vol. 2556. Springer-Verlag, Berlin, 2–17.

MALACARIA, P. AND HANKIN, C. 1998. A new approach to control flow analysis. In *Proceedings of the 7th International Conference on Compiler Construction (CC'98)*. Lecture Notes in Computer Science, vol. 1383. Springer-Verlag, Berlin. 95–108.

MELSKI, D. AND REPS, T. 2000. Interconvertibility of a class of set constraints and context-free-language reachability. *Theor. Comput. Sci. 248,* 1-2, 29–98.

MEUNIER, P., FINDLER, R. B., AND FELLEISEN, M. 2006. Modular set-based analysis from contracts. In *Proceedings of the 33rd Annual ACM Symposium on Principles of Programming Languages*. 218–231.

MEUNIER, P., FINDLER, R. B., STECKLER, P., AND WAND, M. 2005. Selectors make set-based analysis too hard. *Higher-Order Symbol. Comput. 18,* 3-4, 245–269.

MIDTGAARD, J. AND VAN HORN, D. 2009. Subcubic control flow analysis algorithms. Computer science res. rep. 125, Roskilde University, Roskilde, Denmark. (Revised version to appear in *Higher-Order Symbol. Computat.*)

MIDTGAARD, J. AND JENSEN, T. 2008. A calculational approach to control-flow analysis by abstract interpretation. In *Proceedings of the 5th International Symposium on Static Analysis (SAS'08)*. Lecture Notes in Computer Science, vol. 5079. Springer-Verlag, Berlin. 347–362.

MIDTGAARD, J. AND JENSEN, T. P. 2009. Control-flow analysis of function calls and returns by abstract interpretation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, G. Hutton and A. P. Tolmach, Eds. 287–298.

MIGHT, M. 2010. Shape analysis in the absence of pointers and structure. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. G. Barthe and M. V. Hermenegildo, Eds. Lecture Notes in Computer Science, vol. 5944. Springer-Verlag, Berlin, 263–278.

MIGHT, M. AND SHIVERS, O. 2006a. Environmental analysis via ΔCFA. In *Proceedings of the 33rd Annual ACM Symposium on Principles of Programming Languages*. 127–140.

MIGHT, M. AND SHIVERS, O. 2006b. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, J. Lawall, Ed. 13–25.

MIGHT, M., SMARAGDAKIS, Y., AND VAN HORN, D. 2010. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, A. Aiken, Ed. 305–315.

MILNER, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 348–375.

MOHNEN, M. 1995. Efficient closure utilization by higher-order inheritance analysis. In *Proceedings of the 2nd International Symposium Static Analysis (SAS'95)*. Lecture Notes in Computer Science, vol. 983. Springer-Verlag, Berlin, 261–278.

MOSSIN, C. 1997. Flow analysis of typed higher-order programs. Ph.D. dissertation, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark. Tech, rep. DIKU-TR-97/1.

MOSSIN, C. 1998. Higher-order value flow graphs. *Nordic J. Comput. 5,* 3, 214–234.

MOSSIN, C. 2003. Exact flow analysis. *Math. Structures Comput. Sci. 13,* 1, 125–156.

MUYLAERT-FILHO, J. A. AND BURN, G. L. 1993. Continuation passing transformation and abstract interpretation. In *Proceedings of the 1st Imperial College Department of Computing Workshop on Theory and Formal Methods*, G. L. Burn, S. J. Gay, and M. D. Ryan, Eds. Workshops in Computing Series. Springer-Verlag, 247–259.

MØLLER NEERGAARD, P. AND MAIRSON, H. G. 2004. Types, potency, and idempotency: Why nonlinearity and amnesia make a type system work. In *Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, K. Fisher, Ed. 138–149.

NIELSON, F. AND NIELSON, H. R. 1997. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*. 332–345.

NIELSON, F. AND NIELSON, H. R. 1999. Interprocedural control flow analysis. In *Proceedings of the 8th European Symposium on Programming*, S. D. Swierstra, Ed. Lecture Notes in Computer Science, vol. 1576. Springer-Verlag, Berlin, 20–39.

NIELSON, F., NIELSON, H. R., AND HANKIN, C. 1999. *Principles of Program Analysis*. Springer-Verlag, Berlin.

NIELSON, H. R. AND NIELSON, F. 1998. Flow logics for constraint based analysis. In *Proceedings of the 7th International Conference on Compiler Construction*. Lecture Notes in Computer Science, vol. 1383. Springer-Verlag, Berlin. 109–127.

O'DONNELL, M. 1977. *Computing in Systems Described by Equations*. Lecture Notes in Computer Science, vol. 58. Springer-Verlag, Berlin.

OXHØJ, N., PALSBERG, J., AND SCHWARTZBACH, M. I. 1992. Making type inference practical. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*, O. L. Madsen, Ed. Lecture Notes in Computer Science, vol. 615. Springer-Verlag, Berlin, 329–349.

PACHOLSKI, L. AND PODELSKI, A. 1997. Set constraints: A pearl in research on constraints. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*. G. Smolka, Ed. Lecture Notes in Computer Science, vol. 1330. Springer-Verlag, Berlin, 549–562.

PALSBERG, J. 1994. Global program analysis in constraint form. In *Proceedings of the 19th Colloquium on Trees in Algebra and Programming (CAAP'94)*, S. Tison, Ed. Lecture Notes in Computer Science, vol. 787. Springer-Verlag, Berlin, 276–290.

PALSBERG, J. 1995. Closure analysis in constraint form. *ACM Trans. Program. Lang. Syst. 17,* 1, 47–62.

PALSBERG, J. 1998. Equality-based flow analysis versus recursive types. *ACM Trans. Program. Lang. Syst. 20,* 6, 1251–1264.

PALSBERG, J. 2001. Type-based analysis and applications. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*. 20–27.

PALSBERG, J. AND O'KEEFE, P. 1995. A type system equivalent to flow analysis. *ACM Trans. Program. Lang. Syst. 17,* 4, 576–599.

PALSBERG, J. AND PAVLOPOULOU, C. 2001. From polyvariant flow information to intersection and union types. *J. Funct. Program. 11,* 3, 263–317.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1995. Safety analysis versus type inference. *Inform. Comput. 118,* 1, 128–141.

PALSBERG, J. AND SMITH, S. 1996. Constrained types and their expressiveness. *ACM Trans. Program. Lang. Syst. 18,* 5, 519–527.

PALSBERG, J. AND WAND, M. 2003. CPS transformation of flow information. *J. Funct. Program. 13,* 5, 905–923.

PARK, Y. G. AND GOLDBERG, B. 1992. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, C. W. Fraser, Ed. 116–127.

PIERCE, B. C. 2002. *Types and Programming Languages*. The MIT Press, Cambridge, MA.

PLOTKIN, G. D. 1975. Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci. 1*, 125–159.

POTTIER, F. AND GAUTHIER, N. 2006. Polymorphic typed defunctionalization and concretization. *Higher-Order Symbol. Comput. 19,* 1, 125–162. A preliminary version was presented at the 31st Annual ACM Symposium on Principles of Programming Languages (POPL'04).

REHOF, J. AND FÄHNDRICH, M. 2001. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *Proceedings of the 28th Annual ACM Symposium on Principles of Programming Languages*, H. R. Nielson, Ed. 54–66.

REPPY, J. 1999. *Concurrent Programming in ML*. Cambridge University Press, Cambridge U.K.

REPPY, J. 2006. Type-sensitive control-flow analysis. In *Proceedings of the ACM SIGPLAN Workshop on ML (ML'06)*. 74–83.

REYNOLDS, J. C. 1969. Automatic computation of dataset definitions. In *Information Processing 68*, vol. 1. A. J. H. Morrell, Ed. 456–461.

REYNOLDS, J. C. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*. 717–740. Reprinted in *Higher-Order Symbol. Comput. 11*, 4, 363–397, with a foreword [Reynolds 1998].

REYNOLDS, J. C. 1998. Definitional interpreters revisited. *Higher-Order Symbol. Comput. 11,* 4, 355–361.

ROZAS, G. J. 1984. Liar, an Algol-like compiler for Scheme. M.S. thesis, Massachusetts Institute of Technology, Cambridge, MA.

ROZAS, G. J. 1992. Taming the Y operator. In *Proceedings of the ACM Conference on LISP and Functional Programming*, W. Clinger, Ed. *LISP Pointers 5*, 1, 226–234.

RYTTER, W. 1985. Fast recognition of pushdown automaton and context-free languages. *Inform. Control 67,* 1-3, 12–22.

SABRY, A. AND FELLEISEN, M. 1994. Is continuation-passing useful for data flow analysis? In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, V. Sarkar, Ed. 1–12.

SAHA, B., HEINTZE, N., AND OLIVA, D. 1998. Subtransitive CFA using types. Res. rep. 1166, Department of Computer Science, Yale University, New Haven, CT.

SCHMIDT, D. A. 1995. Natural-semantics-based abstract interpretation (preliminary version). In *Proceedings of the 2nd International Symposium on Static Analysis (SAS'95)*. Lecture Notes in Computer Science, vol. 983. Springer-Verlag, Berlin, 1–18.

SCHMIDT, D. A. 1997. Abstract interpretation of small-step semantics. In *Selected Papers from the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*. Lecture Notes in Computer Science, vol. 1192. Springer-Verlag, Berlin, 76–99.

SCHMIDT, D. A. 1998. Trace-based abstract interpretation of operational semantics. *LISP Symbol. Comput. 10,* 3, 237–271.

SCHWARTZ, J. T. 1975. Optimization of very high-level languages—I. value transmission and its corollaries. *Comput. Lang. 1,* 2, 161–194.

SCHWARTZ, J. T., DEWAR, R. B., SCHONBERG, E., AND DUBINSKY, E. 1986. *Programming with Sets: An Introduction to SETL*. Springer-Verlag New York, Inc., New York, NY.

SERENI, D. 2006. Termination analysis of higher-order functional programs. Ph.D. dissertation, Oxford University, Oxford, U.K.

SERENI, D. 2007. Termination analysis and call graph construction for higher-order functional programs. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*. 71–84.

SERENI, D. AND JONES, N. D. 2005. Termination analysis of higher-order functional programs. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*. K. Yi, Ed. Lecture Notes in Computer Science, vol. 3780. Springer-Verlag, Berlin, 281–297.

SERRANO, M. 1995. Control flow analysis: A functional languages compilation paradigm. In *Proceedings of the ACM Symposium on Applied Computing*. 118–122.

SERRANO, M. AND FEELEY, M. 1996. Storage use analysis and its applications. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*, R. K. Dybvig, Ed. 50–61.

SESTOFT, P. 1988. Replacing function parameters by global variables. M.S. thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark.

SESTOFT, P. 1989. Replacing function parameters by global variables. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, J. E. Stoy, Ed. 39–53.

SESTOFT, P. 1991. Analysis and efficient implementation of functional programs. Ph.D. dissertation, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark. DIKU Rapport 92/6.

SHAO, Z. AND APPEL, A. W. 1994. Space-efficient closure representations. In *Proceedings of the ACM Conference on LISP and Functional Programming. LISP Pointers, 7*, 3, 150–161.

SHARIR, M. AND PNUELI, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Upper Saddle River, NJ. 189–233.

SHIVERS, O. 1988. Control-flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, M. D. Schwartz, Ed. 164–174.

SHIVERS, O. 1991a. Control-flow analysis of higher-order languages or taming lambda. Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, Tech. rep. CMU-CS-91-145.

SHIVERS, O. 1991b. The semantics of Scheme control-flow analysis. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, P. Hudak and N. D. Jones, Eds. 190–198.

SISKIND, J. M. 1999. Flow-directed lightweight closure conversion. Tech. rep. 99-190R, NEC Research Institute, Inc. Tokyo, Japan.

SMITH, S. F. AND WANG, T. 2000. Polyvariant flow analysis with constrained types. In *Proceedings of the 9th European Symposium on Programming*. Lecture Notes in Computer Science, vol. 1782. Springer-Verlag, Berlin, 382–396.

STECKLER, P. A. AND WAND, M. 1997. Lightweight closure conversion. *ACM Trans. Program. Lang. Syst. 19,* 1, 48–86.

STEELE JR., G. L. 1978. Rabbit: A compiler for Scheme. M.S. thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA. Tech. rep. AI-TR-474.

STEENSGAARD, B. 1996a. Points-to analysis by type inference of programs with structures and unions. In *Proceedings of the 6th International Conference on Compiler Construction*. Lecture Notes in Computer Science, vol. 1060. 136–150.

STEENSGAARD, B. 1996b. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*. 32–41.

STEENSGAARD, B. AND MARQUARD, M. 1994. A polyvariant closure analysis with dynamic widening. Unpublished note. Available at ftp://ftp.research.microsoft.com/pub/analysts/closure.ps.Z.

STEFANESCU, D. AND ZHOU, Y. 1994. An equational framework for the flow analysis of higher-order functional programs. In *Proceedings of the ACM Conference on LISP and Functional Programming. LISP Pointers, 7*, 3, 318–327.

TANG, Y. M. AND JOUVELOT, P. 1992. Control-flow effects for escape analysis. In *Actes WSA'92 Workshop on Static Analysis*. 313–321.

TANG, Y. M. AND JOUVELOT, P. 1994. Separate abstract interpretation for control-flow analysis. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Software*, M. Hagiya and J. C. Mitchell, Eds. Lecture Notes in Computer Science, vol. 789. Springer-Verlag, Berlin, 224–243.

THIEMANN, P. 1993. A safety analysis for functional programs. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, D. A. Schmidt, Ed. 133–144.

TOLMACH, A. 1997. Combining closure conversion with closure analysis using algebraic types. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*. Available as tech. rep. BCCS-97-03, Computer Science Department, Boston College, MA.

TOLMACH, A. AND OLIVA, D. P. 1998. From ML to Ada: Strongly-typed language interoperability via source translation. *J. Funct. Program. 8,* 4, 367–412.

VAN HORN, D. AND MAIRSON, H. G. 2007. Relating complexity and precision in control flow analysis. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*. 85–96.

VAN HORN, D. AND MAIRSON, H. G. 2008a. Deciding kCFA is complete for EXPTIME. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, P. Thiemann, Ed. 275–282.

VAN HORN, D. AND MAIRSON, H. G. 2008b. Flow analysis, linearity, and PTIME. In *Proceedings of the 15th International Symposium on Static Analysis (SAS'08)*. Lecture Notes in Computer Science, vol. 5079. 255–269.

VARDOULAKIS, D. AND SHIVERS, O. 2010. CFA2: A context-free approach to control-flow analysis. In *Proceedings of the 19th European Symposium on Programming Languages and Systems (ESOP'10)*, A. D. Gordon, Ed. Lecture Notes in Computer Science, vol. 6012. Springer-Verlag, Berlin, 570–589.

VENET, A. 2002. Nonuniform alias analysis of recursive data structures and arrays. In *Proceedings of the 9th International Symposium on Static Analysis (SAS'02)*, M. V. Hermenegildo and G. Puebla, Eds. Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, Berlin, 36–51.

WADDELL, O. AND DYBVIG, R. K. 1997. Fast and effective procedure inlining. In *Proceedings of the 4th International Symposium on Static Analysis (SAS'97)*. Lecture Notes in Computer Science, vol. 1302. 35–52.

WAND, M. 1987. A simple algorithm and proof for type inference. *Fundamenta Infomaticae 10*, 115–122.

WAND, M. 2002. Analyses that distinguish different evaluation orders, or, unsoundness results in control-flow analysis. Unpublished manuscript. Available at ftp://ftp.ccs.neu.edu/pub/people/wand/papers/order-sensitive-cfa.ps.

WAND, M. AND WILLIAMSON, G. B. 2002. A modular, extensible proof method for small-step flow analyses. In *Proceedings of the 11th European Symposium on Programming Languages and Systems (ESOP'02)*, D. Le Métayer, Ed. Lecture Notes in Computer Science, vol. 2305. Springer-Verlag, Berlin, 213–227.

WARREN, D. H. D. 1982. Higher-order extensions to PROLOG: Are they needed? In *Machine Intelligence*, vol. 10. J. E. Hayes, D. Michie, and Y.-H. Pao, Eds. Ellis Horwood, 441–454.

WEEKS, S. 2006. Whole-program compilation in MLton. In *Proceedings of the ACM SIGPLAN Workshop on ML (ML'06)*.

WELLS, J. B., DIMOCK, A., MULLER, R., AND TURBAK, F. 2002. A calculus with polymorphic and polyvariant flow types. *J. Funct. Program. 12,* 3, 183–227.

WRIGHT, A. K. AND JAGANNATHAN, S. 1998. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Trans. Program. Lang. Syst. 20,* 1, 166–207.

YOUNG, J. AND HUDAK, P. 1986. Finding fixpoints on function spaces. Tech. rep. res. rep. YALEEU/DCS/RR-505, Yale University, New Haven, CT.

Received November 2007; revised November 2008, June 2010; accepted August 2010