BRICS

Basic Research in Computer Science

Control-Flow Analysis of Functional Programs

Jan Midtgaard

BRICS Report Series RS-07-18

ISSN 0909-0878 December 2007

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
IT-parken, Aabogade 34
DK–8200 Aarhus N
Denmark

Telephone: +45 8942 9300
Telefax: +45 8942 5601
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/07/18/`

Control-flow analysis of functional programs

Jan Midtgaard
BRICS
Department of Computer Science
University of Aarhus*

November 16, 2007

Abstract

We present a survey of control-flow analysis of functional programs, which has been the subject of extensive investigation throughout the past 25 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 paper we survey control-flow analysis of functional programs by structuring the multitude of formulations and approximations and comparing them.

*IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: `jmi@brics.dk`
Current affiliation: IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France.

Contents

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 run time. The problem has been named *data flow analysis* or just *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, e.g., 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 run time. A *control-flow analysis* approximates at compile time which functions may be applied at run time, i.e., 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. [112]. We also assume familiarity with functional programming and a basic acquaintance with continuation-passing style (CPS) [149] and with recursive equations [114]. We furthermore assume some knowledge about closures for representing functional values at run time [95], and with Reynolds's defunctionalization [130, 44].

1.1 History

Historically, Reynolds was the first to analyse LISP programs [129]. Later Jones and Muchnick independently analysed programs with Lisp-like structured data [84]. Jones was the first to analyse lambda expressions and to use the term *control-flow analysis* [80, 81] for the problem of approximating the control flow of higher-order programs. Shivers formulated control-flow analysis for Scheme programs including side-effects, and suggested a number of improvements and applications [144, 145]. Sestoft then developed a *closure analysis* for programs in direct style [139, 140]. The latter was reformulated first by Bondorf [23], and later by Palsberg [115], whose account is closest to how control-flow analysis is typically presented in textbooks today [112].

1.2 Terminology

1.2.1 Flow vs. closure analysis

Jones and Shivers named their analyses *control-flow analysis* [80, 81, 145] whereas Sestoft [139] named his analysis *closure analysis*. Even though they are presented with different terminology, all three analyses compute *flow information*, i.e., they approximate where a given first-class function is applied and which first-class functions are applied at a given call site. The term 'control flow analysis' was originally used by Allen [7] to refer to the extraction of properties of already given control-flow graphs.

A different line of analysis introduced by Steele in his MS thesis [149] is also referred to as closure analysis [31, 137]. 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 [93].

1.2.2 Approximating allocation

In control-flow analysis one typically approximates a dynamically allocated closure by its code component, representing its place of creation. The idea is well-known from analyses approximating other heap-allocated structures [84, 10, 30], where it is named the *birth-place* approach. Consel, for example, uses the birth-place approach in his work on binding-time analysis [32].

More generally dynamic allocated storage can be represented by the (approximate) state of the computation at allocation time [70, 46] — an idea which has been named the *birth-time*, *birth-date*, or *time-stamp* approach [58, 59, 162]. The state of the computation can be represented by the (approximate) paths or traces leading to it. One such example is *contours* [145], which are finite string encodings approximating the calling context, i.e., 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 [79].

1.2.3 Sensitivity

Established terminology from static analysis has been used to characterize and compare the precision of analyses [69]. Much of this terminology has its roots in data-flow analysis, where one distinguishes *intra-procedural* analyses, i.e., *local* analyses operating on procedures independently, from *inter-procedural* analyses, i.e., *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 paper we will use the established terminology where appropriate.

A *context-sensitive* analysis distinguishes different calling contexts when analysing 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 [112]. 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 MS thesis [139], Sestoft suggests 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 [125]. 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 [167]. 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 in, e.g., dead code. On the other hand the approximation is safe for languages, such as Prolog, where arity checks are performed at compile time.

2.3 Escape analysis

A lightweight approach to compiling higher-order functions is the so-called *escape analysis* [11, 12]. This approach is based on a rough flow approximation originally due to Steele [149]. In its simplest formulation, the analysis draws a distinction between so-called *escaping functions*, i.e., functions that (potentially) escape their lexical scope by being returned, passed as a parameter, stored in a pair, etc., and *known functions*, i.e., functions that do not escape [142]. The categorization is 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 [149], Steele used the analysis to decide whether to *close* lambda expressions, i.e., create a *closure*, over their free variables or not.

A slightly better approximation supplements the above categorization of source lambdas with a categorization of function calls. Function calls are separated into *known* and *unknown calls* [12]. This categorization is formulated as a simple mapping from call sites to a binary domain. As a consequence a function can both *escape* and also be the operator of a *known* call. In the Orbit compiler [92, 93], 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 [93] was to show that Scheme could be compiled as efficiently as, e.g., Pascal, which was designed to be stack-implementable.

Escape analysis is a modular flow approximation, i.e., separate modules can be analysed independently, as opposed to a whole-program flow analysis. The flow approximation is useful for both closure-conversion [149, 11] and defunctionalization [159]. 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 [149, ch.8]. Kranz refers to the distinction as *escape analysis* [93, ch.4]. Both Steele and Kranz refer to their decision procedure for choosing closure representation and layout as *closure analysis* [149, 93]. Clinger and Hansen [31] 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* [137]. Cejtin et al. [30] 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 [68], after having devised a similar binding-time analysis [67]. The analysis is heavily 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 [124]. Bondorf and Jørgensen [25] 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 [62]. Tolmach and Oliva [159, 160] as well as Mossin [104] have since formulated (typed) variants of the approach.

Henglein entitled the approach *simple closure analysis*. The analysis has later been named *equality-based flow analysis* [25, 117] as well as *control flow analysis via equality* [62]. The idea is also referred to as *unification-based* [30, 45, 69], or *bi-directional* [126, 69]. Aiken refers to the analysis as based on *term equations* [3]. We shall sometimes refer to it as unification-based analysis, when contrasting it with other analyses.

2.5 0-CFA

Shivers developed the context-insensitive *zeroth-order control-flow analysis* (0-CFA) for Scheme [145, 144], and suggested several context-sensitive flow analyses (see below). The 0-CFA originally suggested by Shivers had worst-case time-complexity $O(n^3)$. During his work on globalization, i.e., statically determining which function parameters can be turned into global variables, Sestoft had developed a similar flow analysis [139, 140], in order to handle higher-order programs.

Later Bondorf simplified the equations of Sestoft's analysis [23] in order to extend the Similix [24] self-applicable partial evaluator to higher-order functions. Palsberg then limited Bondorf's equations to the pure lambda calculus [115, 116]. He presented a simplified analysis as well as a constraint-based formulation, and proved the equivalence of the three analyses.

Though Shivers's and Sestoft's analyses were thought to be equivalent, Mossin [104] proved that Shivers's analysis is evaluation-order dependent, contrary to Sestoft's closure analysis. However one should note that Shivers's original analysis operated on a CPS-based intermediate language, i.e., an evaluation-order-independent language. Mossin's proof concerns a reformulation in a direct-style evaluation-order-dependent language. As a consequence his result does not directly concern Shivers's original analysis. The term 'control-flow analysis' by itself has since become synonymous with 0-CFA [112].

Serrano [137] 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 [128] later describes a refinement of Serrano's algorithm reformulated as a pure functional program. This analysis incorporates type information from ML's module system. The analyses of Serrano and Reppy do not infer control-flow information for all expressions, they infer control-flow information only for variables, i.e., they compute an approximation of the run-time environment.

As opposed to a unification-based control-flow analysis, 0-CFA is sometimes referred to as an *inclusion-based* [25], or *subset-based* [117] control-flow analysis. In the terminology of pointer analysis it is a (*uni-*) *directional* flow analysis [69]. Variants of 0-CFA are used within the Bigloo optimizing Scheme compiler [137] and within the MLton whole-program optimizing SML compiler [30].

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 [170]. The analysis is inspired by type systems — in particular Hindley-Milner (Damas-Milner) let-bound polymorphism [102], 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 [170].

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 [110].

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 [143] as a means for improving the precision of interprocedural data-flow analyses. Inspired by call strings, Shivers [145] formulated the context-sensitive *first-order control-flow analysis* (1-CFA) and suggested the extension to *kth-order control-flow analysis* (*k-CFA*) [145, p.55] as a refined choice of contours. Since then Jagannathan and Weeks [76] have suggested a *polynomial 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 [110, 112]. It uses a uniform "contour distinction", i.e., 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. Recently Van Horn and Mairson have proved that k-CFA is NP-hard for $k = 1$, and that k-CFA is complete for EXPTIME for the case $k = n$, where n is the size of the program [161]. Their proofs are developed for the uniform k-CFA variant.

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 [110].

3.3 Beyond the k-CFA hierarchy

An alternative context-sensitive flow analysis suggested by Agesen [2] takes argument types of the calling context into account. The original formulation of his *Cartesian-product algorithm* 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 [76] outline a control-flow analysis variant hereof, as do Nielson et al. [112, p.196]. 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* [58, 59] to capture both *procedure calls* and *returns* in a compact format. Recently Might and Shivers have suggested a new context-sensitive control-flow analysis [100] 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' [145], i.e., an analysis which statically detects when two run-time environments agree on a variable.

4 Type-based flow analysis

A parallel line of work has investigated control-flow analysis for 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 additional 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 [130], where the function space of the environment and the function space of expressible values in his definitional interpreter were defunctionalized separately. Indeed Reynolds recognizes that his defining language is typed in a later commentary [131].

Tolmach [159] 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'" [159, p.2]. Tolmach and Oliva [159, 160] 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 [65] present a linear-time algorithm for answering a number of context-insensitive CFA-related questions. Their algorithm has two stages. The first stage builds in linear time 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, e.g., list up to k functions for each call site, otherwise "many". However their algorithm only works on programs of bounded types — for untyped or recursively typed programs their algorithm may not terminate.

Later (unpublished) work by Heintze et al. [133] establishes that the above approach does not scale since real-world (functorized) programs do not always exhibit such bounded types. They therefore suggest a hybrid approach, where the above algorithm is combined with a complementary demand-driven algorithm.

Independently, Mossin arrived at a quadratic-time analysis for simply-typed programs with bounded types [104, 105]. His analysis is based on *flow graphs*. It is furthermore *modular*, i.e., different parts of the program may be analysed separately.

4.3 Context-sensitive type-based analysis (typed)

Mossin gave two context-sensitive analyses for a simply-typed programming language [104]: one inspired by let-polymorphism and one inspired by polymorphic recursion. Rehof and Fähndrich [126] later gave $O(n^3)$ algorithms for the two, improving their earlier complexity bounds on $O(n^7)$ and $O(n^8)$, respectively [104]. Rehof and Fähndrich achieve the speed-up by avoiding copying constraints when they are instantiated — instead they remember the substitution (instantiation constraint), leaving the original constraints unmodified.

5 Formulations

Control-flow analysis comes in many different formulations. As an example of the diversity, Malacaria and Hankin even present a cubic time flow-analysis for PCF based on game semantics [97]. The resulting analysis is similar to Shivers's 0-CFA, despite their different starting point and formulation. 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 [164] is an example of a constraint-based analysis that has inspired many later analyses [67, 155, 68, 151, 57, 104]. 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 [3].

Initially Reynolds conceived the idea of formulating analyses in terms of *recursive set definitions* [129] which resembled context-free grammars [129, p.456]. 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*, i.e., *regular tree grammars* extended with selectors, to analyse programs with LISP-like structures [84].

Heintze and Jaffar extended the idea of *grammar-based analyses* to handle projection (selectors) and intersection [64, 63] originally in the context of analysing logic programs, and introduced the term *set constraints*. Aiken and Murphy [5] formulated a type-inference algorithm with types implemented as *regular tree expressions*, and described their implementation [4]. In a later paper [6], Aiken and Wimmers gave an algorithm for solving constraint systems over regular tree expressions — now under the name *set constraints*.

Heintze coined the term *set-based analysis* [60] 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 [61]. Independently, Palsberg reformulated Bondorf's simplification of Sestoft's control-flow analysis in terms of *conditional set constraints* [115, 116]. Conditional constraints have later been shown to be equally expressive to a constraint system with selectors [3] such as Heintze's [60].

Cousot and Cousot [39] clarified how grammar-based, constraint-based, and set-based analyses are instances of the general theory of *abstract interpretation*. They suggest that a formulation in terms of abstract interpretation allows for the use of *widening* and for an easy integration with other non-grammar domains. Gallagher and Peralta [53] have investigated such a regular tree language domain in the context of partial evaluation.

As an extension to Heintze's set-based analysis [60], Flanagan and Felleisen [51, 52] 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 avoiding the re-extraction of constraints from an unmodified file upon later re-analysis. Flanagan used the analysis for a static debugger [51]. Meunier et al. [99] later identified that selectors complicated the analysis and suggested to use conditional constraints in the style of Palsberg [116] instead.

Henglein's simple closure analysis [68] and Bondorf and Jørgensen's efficient closure analysis for partial evaluation [25] 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 [3].

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 [106] distinguishes the two, by referring to them as Church-style analysis and Currystyle analysis, respectively. The field of type-based analysis is big enough to deserve a separate treatment. We refer to Palsberg [118] and Jensen [77] for surveys of the area.

Heintze and Palsberg et al. have investigated the relationship between flow analyses and type systems [62, 119, 117]. Their systems applies to untyped terms, and are strictly speaking not type-based analyses. See Section 5.3.1 for more details.

Mossin presented a number of type-based analyses for simply-typed programs in his PhD thesis [104]. He formulated two context-insensitive analyses: a *simple* analysis and a subtype-based analysis equivalent to Sestoft's analysis. He furthermore formulated two context-sensitive analyses: One based on let-polymorphism and one based on polymorphic recursion. Mossin also developed a context-sensitive control-flow analysis based on intersection types [104, 106], which he called *exact*. He showed the analysis to be decidable; it is however non-elementary recursive and therefore of limited practical value [104, 106].

Banerjee et al. [17] prove 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 sub-type relation on control-flow types. The analysis resembles one of Heintze's [62] systems modulo Heintze's super type for handling otherwise untypable programs.

Wells et al. [169] have investigated a type-based intermediate language with intersection and union flow types. Their focus has been type-based compilation, rather than flow analysis [169]. As such, they have inferred control-flow information using known flow analyses, and afterwards decorated the flow types with the inferred flow information [48].

The flow analysis of the MLton Standard ML compiler operates on simply typed programs [30], i.e., 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 inclusions on datatype elimination and tuple introduction and elimination substituted with equalities, which are then solved by unification [168]. Apparently the resulting analysis does not exhibit cubic time behavior [168], which seems consistent with linear-time CFA on bounded-type programs [65, 105].

Recently, Reppy presented an analysis [128] 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 [159, 160], or for termination or time complexity [65, 104], 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, and data-flow and context-free grammar reachability.

5.3.1 Equivalences between type systems and analyses

Palsberg and O'Keefe [119] show that a 0-CFA-based safety analysis (cf. Section 7.1) is equivalent to a type system due to Amadio and Cardelli [8] with subtyping and recursive types. Independently, Heintze showed a number of similar equivalences [62] between equality-based and subset-based control-flow analyses and their counterpart type systems with simple types and sub-typing, including the above.

Palsberg later refuted Heintze's claim that equality-based flow analysis is equivalent to a type system with recursive types [117], by giving counter examples. 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 [120] 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 [169].

5.3.2 Equivalences between CFL reachability and analyses

Melski and Reps [98] have shown how to convert in linear time a class of set constraints into a corresponding context-free-language reachability problem, and vice versa. They also show how to extend the result to Heintze's *ML set constraints* [61] for closure analysis. Recently Kodumal and Aiken [90] have shown a particularly simple reduction from a context-free-language reachability problem to set constraints in the special case of Dyck context-free languages, i.e., languages of matching parentheses.

Heintze and McAllester [66] prove a number of problems to be 2NPDA-complete: *data-flow reachability* (in a formulation equivalent to the set constraints of Melski and Reps), *control-flow reachability*, and the complement of *Amadio-Cardelli typability* [8].

5.4 Specification-based

Nielson and Nielson have championed the specification approach to program analysis [110, 54, 113, 111]. 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 analyzed iteratively.

Nielson and Nielson coined the term *flow logic* for such a tight declarative format describing analyses [113]. They show how such a specification can be gradually transformed into a more verbose constraint-based formulation [113]. Their gradual transformation towards constraints involves formulations in terms of *(extended) attribute grammars*, which should be compared to the above mentioned grammar/constraint correspondence.

Indeed a specification-based analysis offers a constructive way of calculating a solution. Cachera et al. [28] have illustrated this point by formalizing a specification-based analysis in constructive logic using the Coq proof assistant.

5.5 Abstract interpretation-based

As pointed out by Aiken [3] 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 [35, 37], which suggests that analyses should be derived systematically from a formal semantics, e.g., through Galois connections. We refer here to abstract interpretation in the latter meaning.

In his PhD thesis [58], Harrison used abstract interpretation of Scheme programs to automatically parallelize them. Harrison treats a statement-based 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 [145] 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, i.e., abstraction functions mapping concrete objects to abstract counterparts.

Ayers also treated higher-order flow analysis based on abstract interpretation in his PhD thesis [15]. 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 [134, 135, 136], Schmidt 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.

5.6 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 [36]. The idea of considering only necessary arguments in the context of function graphs was suggested and named *minimal function graphs* by Jones and Mycroft [86].

Jones and Rosendahl [87] 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 to how many arguments the function has been partially applied.

Control-flow analyses defined as functions on an expression-based language do not attempt to give non-trivial approximate characterizations for all possible expressions. Instead such analyses are often specified as finite partial functions or as total functions on a program-dependent domain [116, 112], which is finite for any given (finite) program.

6 Formulation issues

6.1 Evaluation-order dependence

Flow-sensitivity of program analyses in functional languages can potentially model evaluation order and strategy, e.g., 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 prior linearization of the program. A flow-insensitive analysis approximates all evaluation orders and strategies.

Reynolds's seminal paper [129] inspired Jones to develop control-flow analyses for lambda expressions under both call-by-value [80] and call-by-name [81] 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 wrt. a strict call-by-value semantics [139] and a lazy call-by-name semantics [141]. Palsberg [116] then claimed the soundness of closure analysis wrt. general β-reduction. Unfortunately his proof was flawed, which was later pointed out and corrected by Wand and Williamson [166]. In an unpublished report [165], Wand then compared prior soundness results wrt. different semantics.

6.2 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 simplified his earlier analysis approach by limiting his input to recursive equations [82] as obtained, e.g., by lambda lifting [78]. Sestoft's analysis was also specified for recursive equations [139, 140]. Shivers argued that in CPS lambda expressions capture all control flow in one unifying construct. As a consequence he formulated his original analyses for linearized source programs in CPS [144, 145] and continues to do so today [100]. Ayers's analysis was also formulated for a core language in CPS [15]. The flow analysis of Ashley and Dybvig operates on linearized source programs in a variant of CPS [13].

Consel and Danvy [33] pointed out that CPS transforming a program could improve the outcome of a binding-time analysis, and Muylaert-Filho and Burn [108] showed a similar result for strictness analysis. Sabry and Felleisen [132] 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 [43, 122], however, that Sabry and Felleisen were comparing a flow-sensitive analysis to a flow-insensitive analysis.

Damian and Danvy [43] 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 [122] proved that a non-duplicating Plotkin-style CPS transformation does not change the precision of a standard constraint-based 0-CFA, a result that Damian and Danvy [42] extended to a 'one-pass' CPS transformation that performs administrative reductions. 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.

6.3 Cache-based analysis and iteration order

Hudak and Young [72] 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 sub-expressions 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 [145]. His implementation [145], 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 [23, 116, 110].

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"* [134]. 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 [39] 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 [116]:

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 amount of widening operators to still ensure convergence [27]. Deutsch [47] and Blanchet [22] have used this approach in the context of escape analyses. Bourdoncle [27] has suggested different iteration strategies, some of which are applicable to analysing higher-order programs. He concluded that more work is needed in the higher-order case.

6.4 Compositionality

Keeping an analysis compositional prevents it from diverging by recursively analysing 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 analysing the same terms: in an unpublished technical report [171], Young and Hudak invented *pending analysis*, of which Shivers's *time-stamps* are a variant [146, 145]; and Ashley and Dybvig [13] use a similar concept which they name *pending sets*. A related technique is the worklist algorithm from data flow analysis [89, 112].

The original formulation of 0-CFA in Shivers's PhD thesis [145, p.32] is not compositional. The formulation in the later paper proving the soundness of the approximation is however compositional [146, p.196]. Shivers's implementation [145] used a time-stamping approach to ensure convergence on recursive programs. The formal correctness of time-stamping was later established by Damian [41]. Neither Serrano's nor Reppy's 0-CFA formulations are compositional [137, 128]. In order to avoid re-analysing 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 [110, 54], but later they were reformulated compositionally [113, 111] (in which case induction and co-induction coincide [112]).

The context-sensitive analyses — the k-CFA formulation of Shivers [145], the polymorphic splitting formulation of Wright and Jagannathan [170], and the uniform k-CFA formulation of Nielson, Nielson and Hankin [110, 112] are non-compositional. The analysis framework of Nielson and Nielson's later paper on higher-order flow analysis supporting side-effects [111] is however compositional, as is Rehof and Fähndrich's [126] context-sensitive flow analysis of typed higher-order programs.

6.5 Frameworks

A line of papers have formulated control-flow analysis frameworks following Shivers's initial presentation [144, 145]. Stefanescu and Zhou [153] 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 [145].

Jagannathan and Weeks [76] 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 prove their 0-CFA instantiation equivalent to Heintze's set-based analysis [61].

Ashley and Dybvig [13] developed a flow analysis framework for the *Chez Scheme* compiler. The framework is parameterized by an abstract allocation function and a 'projection operator'. By different instantiations they obtain a 0-CFA, a 1-CFA 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 show that the sub-0-CFA instantiation enables effectively the same optimizations in the underlying compiler as the 0-CFA.

Nielson and Nielson [110] 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 — a k-CFA variant with better worst-case time complexity.

In a later paper [111], Nielson and Nielson 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 [145], with call strings in the style of Sharir and Pnueli [143], and with *assumption sets* [112].

In an unpublished report [147], Siskind developed a framework with a precise flow analysis 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 compile times or time complexity of the approach.

6.6 Abstract compilation and partial evaluation of CFA

Boucher and Feeley [26] illustrate two approaches to eliminate the interpretive overhead of an analysis. They name these approaches *abstract compilation.* Their first approach serializes the program-specific analysis textually in a file, that is later interpreted, e.g., using the Scheme *eval*-function. Their second approach avoids the interpretive overhead and the I/O of the above serialization by utilizing the closures of the host language. Ashley and Dybvig note [13] that the prototype implementation of their analysis is staged. In their own words [13, p.857] "code is compiled to closures", i.e., they are effectively performing abstract compilation.

Boucher and Feeley [26] suggest two optimizations, namely η-reduction and static look-up of constants and lambda-expressions. They note that 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 [40] 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 results [26].

In a related technical report [9], Amtoft partially evaluates two constraint interpreters with respect to a set of (program specific) CFA constraints (on the same set of benchmarks [26]). He compares the two to their un-specialized counterparts and reports of 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 handling 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.

6.7 Demand-driven analysis

A standard control-flow analysis analyses all terms of a source program regardless of whether they will be used during execution or not. A line of work has therefore investigated *reachability* or *demand-driven analysis*, in order to limit to a minimum the execution time of a full control-flow analysis.

Ayers [14, 15] illustrate how limiting the analysis to the *live* parts of the program can yield a speed-up in analysis time. The abstract semantics of Jagannathan and Weeks's framework [76] contains a 'reachability predicate' to minimize the size of the generated flow graphs. Biswas [21] augmented a set-based analysis in the style of Heintze [60] with boolean constraints to formulate a demand-driven flow analysis for detecting dead code in higher-order functional programs. Gasser et al. [54] formulated a control-flow analysis for Concurrent ML [127]. Starting with an abstract specification, they incorporate tracking of reachable sub-expressions and arrive at a constraint-based formulation.

6.8 Modular and separate analysis

Tang and Jouvelot [156] combine a type and effect system with a control-flow analysis in the style of Shivers's 1-CFA [145] 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 used as starting points for *re-analysis* using the more precise 1-CFA. The approach extends earlier work that formulated a control-flow effect system [155].

Banerjee [16] developed a modular and polyvariant control-flow and type-inference system for untyped programs. In a follow-up paper, Banerjee and Jensen [18] formulated a modular and polyvariant control-flow analysis for simply-typed programs. Both analyses are based on intersection types, in particular they rely on the principal typing property of rank 2 intersection types. Their analyses are *compositional* and *modular* in that the analysis of an expression can be calculated by combining the analyses of its sub-expressions using intersection types without re-analysis of any sub-expressions.

Lee, Yi, and Paek [96] 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 analysed separately in topological order of their (acyclic) dependencies. The resulting analysis is more precise than a 0-CFA, because of the module-variance.

7 Related analyses

7.1 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, i.e., to statically guarantee the absence of run-time 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 [121] 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 wrt. both call-by-value and call-by-name evaluation. Thiemann [157] had earlier used the term *safety analysis* for an unrelated analysis for functional programs that detects when in-place updating is safe, i.e., when it does not affect the outcome of programs.

7.2 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 [69] 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 [49]. 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 [10] and Steensgaard's equality-based pointer analysis [151, 150]. Andersen's pointer analysis was formulated in terms of subset-inclusion constraints [10], whereas Steensgaard's pointer analysis was formulated as a type system with a non-standard set of types and unification [151].

Andersen's pointer analysis [10] was conceived simultaneously with Palsberg's control-flow analysis in constraint form [115] and Heintze's set-based analysis [61]. On the other hand Steensgaard's pointer analysis [151] postdates Henglein's technical report on closure analysis by type inference [68] by four years, and indeed Steensgaard [151] cites Henglein [67] as a source of inspiration for his unification-based pointer analysis.

More recently Das [45] 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. Das's analysis seems in line with Henglein's original analysis relying on a limited form of (flow-)subtyping [68] and with Palsberg's *funny* type system equivalent to equality-based CFA [117].

7.3 Escape analysis and stackability

Control-flow analysis is concerned with *flows-from* information, i.e., 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, i.e., 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, e.g., 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 [123] devised an *escape analysis* for higher-order programs. Their initial analysis handled constants and procedural values [55]. It was later extended to handle lists [123]. The analysis was formulated as a forward analysis requiring exponential time even in the first-order case. Deutsch [47] 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 DEXPTIME-hard, suggesting that an extension to higher-order functions would demand further approximation. Blanchet [22] extended Deutsch's backwards escape analysis [47] to a higher-order ML-like core language incorporating further approximation to ensure rapid termination.

Banerjee and Schmidt [19] developed a static *stackability* criterion for simply-typed call-by-value λ-calculus terms, i.e., 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 [155] formulated a control-flow effect system that infers control-flow information. The system infers both which function a given expression may evaluate to, *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 [57] 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 [138] presented a *storage use analysis*. Their analysis is an extension of Shivers 0-CFA with modules and general data storage. They present two applications of the analysis: *stack allocation* and *unboxing*.

Mohnen [103] gives an (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, i.e., 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 gives a measure for determining whether a closure will only have one active activation at a time during execution, in which case he suggests static allocation. Mohnen's work extends earlier work by Hughes [74], who formulated an inheritance analysis for lists in higher-order programs. Hughes's main application was compile-time garbage collection.

7.4 Must analysis and abstract cardinality

Whereas much work in control-flow analysis has focused on inferring *may alias* information, Jagannathan et al. [75] formulated a constraint-based *must alias* 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. The resulting information enables *lightweight closure conversion* [148]. Their analysis determines a related property for reference cells that enables other optimizing transformations.

Might and Shivers [101] recently 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 and, e.g., 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.

8 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 [145]. Wright and Jagannathan refined 0-CFA into *polymorphic splitting* [170], and Nielson and Nielson reformulated k-CFA into a *uniform k-CFA* [110]. In contrast, the traditional abstract interpretation approach is top-down [37]. The starting point is here 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 [116, 110]. In contrast, abstract interpretation offers best approximations [37] in the form of abstraction functions. Together with a companion concretization function, the two can form a Galois connection [37]. Few papers investigating control-flow analysis relate them by Galois connections [15, 153, 111]. Ayers's work on Galois connections is available only in his PhD thesis. 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"* [110] control-flow analyses.

8.1 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 [60, 39, 3]. The data representation of constraints (or the equivalent *regular-tree grammar* [39]) is a finite representative on a potentially infinite domain. An abstract interpretation can always "inherit" that finite representation and their corresponding convergence guarantee [3, p.106] to yield a terminating analysis. To emphasize the point, Cousot and Cousot develop a finitary grammar domain [39], 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 [38].

8.2 CFA with widening

Few control-flow analyses have been formulated with an explicit widening operator. Steensgaard and Marquard [152] include a dynamic widening operator in their (unpublished) analysis to ensure convergence in an infinite domain. Correspondingly Ashley and Dybvig [13] include in their framework a projection operator similar to a widening operator to ensure rapid termination.

Schmidt [136] outlines an alternative closure analysis that approximates environments less crudely. To still ensure termination of his analysis he suggests to index 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 [37] in abstract interpretation.

There is a clear line of research headed towards more precise modeling of contexts [145, 110, 170, 100]. However one will not get full benefit of a very precise context representation if code and environment components of closures are analysed separately as *independent attributes* [85]. The key to precise control-flow analysis is to keep the code and its environment together in *abstract closures*, thereby obtaining a *relational* analysis [85] as in the above 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 Relevance

Serrano questions [137, p.122] 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, i.e., 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, of Blanchet, and of Venet [47, 22, 162]).

Research by Waddell and Dybvig [163] 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), e.g., ASTRÉE [34], one will still need precise control-flow analyses in order to bring the advances to verification of higher-order programs.

10 Conclusion

Over 25 years after Jones's initial flow analysis of lambda expressions [81], 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 [110], a simple and systematic development of control-flow analysis utilizing the tools of abstract interpretation still remains to be found. Such a development may provide the insight to extend recent developments in the verification of first-order programs to verifying higher-order programs.

Acknowledgments: This paper benefited from Olivier Danvy and Kevin Millikin's numerous comments and encouragement. Thanks are also due to Thomas Jensen and Janus Dam Nielsen for comments on an earlier version of this survey.

References

[1] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[2] Ole Agesen. The Cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In Walter G. Olthoff, editor, *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Science*, pages 2–26, Århus, Denmark, August 1995. Springer-Verlag.

[3] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2-3):79–111, 1999.

[4] Alexander Aiken and Brian R. Murphy. Implementing regular tree expressions. In Hughes [73], pages 427–447.

[5] Alexander Aiken and Brian R. Murphy. Static type inference in a dynamically typed language. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, Orlando, Florida, January 1991. ACM Press.

[6] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints (extended abstract). In Andre Scedrov, editor, *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science (LICS*

'92), pages 329–340, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[7] Frances E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, 1970.

[8] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[9] Torben Amtoft. Partial evaluation for constraint-based program analyses. Technical Report BRICS-RS-99-45, BRICS, Dept. of Computer Science, University of Aarhus, 1999.

[10] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, May 1994. DIKU Rapport 94/19.

[11] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[12] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In Michael J. O'Donnell and Stuart Feldman, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, Texas, January 1989. ACM Press.

[13] J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, 1998.

[14] Andrew Edward Ayers. Efficient closure analysis with reachability. In Billaud et al. [20], pages 126–134.

[15] Andrew Edward Ayers. *Abstract Analysis and Optimization of Scheme*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1993.

[16] Anindya Banerjee. A modular, polyvariant, and type-based closure analysis. In Tofte [158], pages 1–10.

[17] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, volume 2215 of *Lecture Notes in Computer Science*, pages 420–447, Sendai, Japan, October 2001. Springer-Verlag.

[18] Anindya Banerjee and Thomas Jensen. Modular control-flow analysis with rank 2 intersection types. *Mathematical Structures in Computer Science*, 13(1):87–124, 2003.

[19] Anindya Banerjee and David A. Schmidt. Stackability in the typed call-by-value lambda calculus. *Science of Computer Programming*, 31(1):47–73, 1998.

[20] Michel Billaud, Pierre Castéran, Marc-Michel Corsini, Kaninda Musumbu, and Antoine Rauzy, editors. *Actes WSA'92, Proceedings of the Second International Workshop on Static Analysis*, Bigre, Laboratoire Bordelais de Recherche en Informatique (LaBRI), September 1992. Atelier Irisa, IRISA, Campus de Beaulieu.

[21] Sandip K. Biswas. A demand-driven set-based analysis. In Jones [83], pages 372–385.

[22] Bruno Blanchet. Escape analysis: correctness proof, implementation and experimental results. In Cardelli [29], pages 25–37.

[23] Anders Bondorf. Automatic autoprojection of higher-order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, 1991. A preliminary version was presented at the Third European Symposium on Programming (ESOP 1990).

[24] Anders Bondorf. Similix 5.1 manual. Technical report, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, May 1993. Included in the Similix 5.1 distribution.

[25] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.

[26] Dominique Boucher and Marc Feeley. Abstract compilation: A new implementation paradigm for static analysis. In Gyimóthy [56], pages 192–207.

[27] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141, Academgorodok, Novosibirsk, Russia, June 1993. Springer-Verlag.

[28] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, 2005.

[29] Luca Cardelli, editor. *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1998. ACM Press.

[30] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In Gert Smolka, editor, *Proceedings of the Ninth European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 56–71, Berlin, Germany, March 2000. Springer-Verlag.

[31] William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label or a simple optimizing compiler for Scheme. In Talcott [154], pages 128–139.

[32] Charles Consel. Binding time analysis for higher order untyped functional languages. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 264–272, Nice, France, June 1990. ACM Press.

[33] Charles Consel and Olivier Danvy. For a better support of static data flow. In Hughes [73], pages 496–519.

[34] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In Mooly Sagiv, editor, *Proceedings of the 14th European Symposium on Programming*, volume 2618 of *Lecture Notes in Computer Science*, pages 21–30, Edinburgh, Scotland, April 2005. Springer-Verlag.

[35] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Ravi Sethi, editor, *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977. ACM Press.

[36] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In Erich J. Neuhold, editor, *IFIP Conference on Formal Description of Programming Concepts*, pages 237–277, St-Andrews, Canada, 1977. North-Holland.

[37] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[38] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In Maurice Bruynooghe and Martin Wirsing, editors, *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming (PLILP '92)*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295, Leuven, Belgium, August 1992. Springer-Verlag.

[39] Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In Simon Peyton Jones, editor, *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 170–181, La Jolla, California, June 1995. ACM Press.

[40] Daniel Damian. Partial evaluation for program analysis. Progress report, BRICS PhD School, University of Aarhus. Available at `http://www.brics.dk/~damian/`, June 1999.

[41] Daniel Damian. Time stamps for fixed-point approximation. In Stephen Brookes and Michael Mislove, editors, *Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics*, volume 45 of *Electronic Notes in Theoretical Computer Science*, pages 43–54, Aarhus, Denmark, May 2001. Elsevier Science Publishers.

[42] Daniel Damian and Olivier Danvy. CPS transformation of flow information, part II: Administrative reductions. *Journal of Functional Programming*, 13(5):925–934, 2003.

[43] Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis: On the impact of the CPS transformation. *Journal of Functional Programming*, 13(5):867–904, 2003. A preliminary version was presented at the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000).

[44] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[45] Manuvir Das. Unification-based pointer analysis with directional assignments. In Lam [94], pages 35–46.

[46] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In Hudak [71], pages 157–168.

[47] Alain Deutsch. On the complexity of escape analysis. In Jones [83], pages 358–371.

[48] Allyn Dimock, Ian Westmacott, Robert Muller, Franklyn Turbak, and J. B. Wells. Functioning without closure: type-safe customized function representations for Standard ML. In Xavier Leroy, editor, *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 14–25, Firenze, Italy, September 2001. ACM Press.

[49] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In Lam [94], pages 253–263.

[50] John Field and Gregor Snelting, editors. *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001.

[51] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, Houston, Texas, May 1997.

[52] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, 1999.

[53] John P. Gallagher and Julio C. Peralta. Regular tree languages as an abstract domain in program specialisation. *Higher-Order and Symbolic Computation*, 14(2-3):143–172, 2001.

[54] Kirsten L. Solberg Gasser, Flemming Nielson, and Hanne Riis Nielson. Systematic realisation of control flow analyses for CML. In Tofte [158], pages 38–51.

[55] Benjamin Goldberg and Young Gil Park. Higher order escape analysis: optimizing stack allocation in functional program implementations. In Neil D. Jones, editor, *Proceedings of the Third European Symposium on Programming*, number 432 in Lecture Notes in Computer Science, pages 152–160, Copenhagen, Denmark, May 1990. Springer-Verlag.

[56] Tibor Gyimóthy, editor. *CC'96: Proceedings of the 6th International Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, Linköping, Sweden, April 1996. Springer-Verlag.

[57] John Hannan. A type-based escape analysis for functional languages. *Journal of Functional Programming*, 8(3):239–273, 1998.

[58] William Ludwell Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, October 1989.

[59] Williams Ludwell Harrison III and Zahira Ammarguellat. A program's eye view of Miprac. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 512–537, London, UK, 1993. Springer-Verlag.

[60] Nevin Heintze. *Set-Based Program Analysis.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, October 1992.

[61] Nevin Heintze. Set-based program analysis of ML programs. In Talcott [154], pages 306–317.

[62] Nevin Heintze. Control-flow analysis and type systems. In Mycroft [109], pages 189–206.

[63] Nevin Heintze and Joxan Jaffar. A decision procedure for a class of set constraints. In John Mitchell, editor, *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51, Philadelphia, Pennsylvania, June 1990. IEEE Computer Society Press.

[64] Nevin Heintze and Joxan Jaffar. A finite presentation theorem for approximating logic programs. In Hudak [71], pages 197–209.

[65] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In Ron K. Cytron, editor, *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 32, No 5, pages 261–272, Las Vegas, Nevada, June 1997. ACM Press.

[66] Nevin Heintze and David McAllester. On the cubic bottleneck in subtyping and flow analysis. In Glynn Winskel, editor, *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS '97)*, pages 342–351, Warsaw, Poland, June 1997. IEEE Computer Society.

[67] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In Hughes [73], pages 448–472.

[68] Fritz Henglein. Simple closure analysis. Technical Report Semantics Report D-193, DIKU, Computer Science Department, University of Copenhagen, 1992.

[69] Michael Hind. Pointer analysis: haven't we solved this problem yet? In Field and Snelting [50], pages 54–61.

[70] Paul Hudak. A semantic model of reference counting and its abstraction. In Abramsky and Hankin [1], pages 45–62.

[71] Paul Hudak, editor. *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, San Francisco, California, January 1990. ACM Press.

[72] Paul Hudak and Jonathan Young. Collecting interpretations of expressions. *ACM Transactions on Programming Languages and Systems*, 13(2):269–290, 1991.

[73] John Hughes, editor. *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, Cambridge, Massachusetts, August 1991. Springer-Verlag.

[74] Simon Hughes. Compile-time garbage collection for higher-order functional languages. *Journal of Logic and Computation*, 2(4):483–509, August 1992.

[75] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: must-alias analysis for higher-order languages. In Cardelli [29], pages 329–341.

[76] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In Peter Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 393–407, San Francisco, California, January 1995. ACM Press.

[77] Thomas Jensen. Types in program analysis. In Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 204–222. Springer-Verlag, 2002.

[78] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203, Nancy, France, September 1985. Springer-Verlag.

[79] John B. Johnston. The contour model of block structured processes. In *Proceedings of the ACM symposium on data structures in programming languages*, pages 55–82. ACM Press, 1971. SIGPLAN Notices, Vol. 6, No. 2, 1971.

[80] Neil D. Jones. Flow analysis of lambda expressions. Technical report PB-128, Aarhus University, Aarhus, Denmark, 1981.

[81] Neil D. Jones. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 114–128, London, UK, 1981. Springer-Verlag.

[82] Neil D. Jones. Flow analysis of lazy higher-order functional programs. In Abramsky and Hankin [1], pages 103–122.

[83] Neil D. Jones, editor. *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997. ACM Press.

[84] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of LISP-like structures. In Barry K. Rosen, editor, *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 244–256, San Antonio, Texas, January 1979. ACM Press.

[85] Neil D. Jones and Steven S. Muchnick. Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra. In Muchnick and Jones [107], pages 380–393.

[86] Neil D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs. In Mark Scott Johnson and Ravi Sethi, editors, *Proceedings of the Thirteenth Annual ACM Symposium*

on Principles of Programming Languages, pages 296–306, St. Petersburg, Florida, January 1986. ACM Press.

[87] Neil D. Jones and Mads Rosendahl. Higher-order minimal function graphs. *Journal of Functional and Logic Programming*, 1997(2), February 1997.

[88] Andrew Kennedy and François Pottier, editors. *ACM SIGPLAN Workshop on ML*, September 2006.

[89] G. Kildall. A unified approach to global program optimization. In Jeffrey D. Ullman, editor, *Proceedings of the First Annual ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, January 1973. ACM Press.

[90] John Kodumal and Alex Aiken. The set constraint/CFL reachability connection in practice. In Craig Chambers, editor, *Proceedings of the ACM SIGPLAN'04 Conference on Programming Languages Design and Implementation*, pages 207–218, Washington DC, USA, June 2004. ACM Press.

[91] Kai Koskimies, editor. *CC'98: Proceedings of the 7th International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, London, UK, April 1998. Springer-Verlag.

[92] David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. In Stuart I. Feldman, editor, *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 21, No 7, pages 219–233, Palo Alto, California, June 1986. ACM Press.

[93] David A. Kranz. *ORBIT: An Optimizing Compiler for Scheme.* PhD thesis, Computer Science Department, Yale University, New Haven, Connecticut, February 1988. Research Report 632.

[94] Monica Lam, editor. *Proceedings of the ACM SIGPLAN'00 Conference on Programming Languages Design and Implementation*, Vancouver, British Columbia, Canada, June 2000. ACM Press.

[95] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[96] Oukseh Lee, Kwangkeun Yi, and Yunheung Paek. A proof method for the correctness of modularized 0CFA. *Information Processing Letters*, 81(4):179–185, 2002.

[97] Pasquale Malacaria and Chris Hankin. A new approach to control flow analysis. In Koskimies [91], pages 95–108.

[98] David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1-2):29–98, 2000.

[99] Philippe Meunier, Robert Bruce Findler, Paul Steckler, and Mitchell Wand. Selectors make set-based analysis too hard. *Higher-Order and Symbolic Computation*, 18(3-4):245–269, 2005.

[100] Matthew Might and Olin Shivers. Environmental analysis via ΔCFA. In Simon Peyton Jones, editor, *Proceedings of the Thirty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 127–140, Charleston, South Carolina, January 2006. ACM Press.

[101] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In Julia Lawall, editor, *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, pages 13–25, Portland, Oregon, September 2006. ACM Press.

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

[103] Markus Mohnen. Efficient closure utilisation by higher-order inheritance analysis. In Mycroft [109], pages 261–278.

[104] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1997. Technical Report DIKU-TR-97/1.

[105] Christian Mossin. Higher-order value flow graphs. *Nordic Journal of Computing*, 5(3):214–234, 1998.

[106] Christian Mossin. Exact flow analysis. *Mathematical Structures in Computer Science*, 13(1):125–156, 2003.

[107] Steven S. Muchnick and Neil D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[108] Juarez A. Muylaert-Filho and Geoffrey L. Burn. Continuation passing transformation and abstract interpretation. In G. L. Burn, S. J. Gay, and M. D. Ryan, editors, *Theory and Formal Methods 1993: Proceedings of the First Imperial College Department of Computing Workshop on Theory and Formal Methods*, Workshops in Computing Series, pages 247–259, Isle of Thorns, Sussex, 1993. Springer-Verlag.

[109] Alan Mycroft, editor. *Proceedings of the Second International Symposium on Static Analysis (SAS'95)*, volume 983 of *Lecture Notes in Computer Science*, Glasgow, Scotland, September 1995. Springer-Verlag.

[110] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In Jones [83], pages 332–345.

[111] Flemming Nielson and Hanne Riis Nielson. Interprocedural control flow analysis. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 20–39, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

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

[113] Hanne Riis Nielson and Flemming Nielson. Flow logics for constraint based analysis. In Koskimies [91], pages 109–127.

[114] Michael O'Donnell. *Computing in Systems Described by Equations*. Number 58 in Lecture Notes in Computer Science. Springer-Verlag, 1977.

[115] Jens Palsberg. Global program analysis in constraint form. In Sophie Tison, editor, *19th Colloquium on Trees in Algebra and Programming (CAAP'94)*, volume 787 of *Lecture Notes in Computer Science*, pages 276–290, Edinburgh, Scotland, April 1994. Springer-Verlag.

[116] Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, 1995.

[117] Jens Palsberg. Equality-based flow analysis versus recursive types. *ACM Transactions on Programming Languages and Systems*, 20(6):1251–1264, 1998.

[118] Jens Palsberg. Type-based analysis and applications. In Field and Snelting [50], pages 20–27.

[119] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, 1995.

[120] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(3):263–317, 2001.

[121] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

[122] Jens Palsberg and Mitchell Wand. CPS transformation of flow information. *Journal of Functional Programming*, 13(5):905–923, 2003.

[123] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In Christopher W. Fraser, editor, *Proceedings of the ACM SIGPLAN'92 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 27, No 7, pages 116–127, San Francisco, California, July 1992. ACM Press.

[124] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[125] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19(1):125–162, 2006. A preliminary version was presented at the Thirty-First Annual ACM Symposium on Principles of Programming Languages (POPL 2004).

[126] Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: from polymorphic subtyping to CFL-reachability. In Hanne Riis Nielson, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 54–66, London, United Kingdom, January 2001. ACM Press.

[127] John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[128] John Reppy. Type-sensitive control-flow analysis. In Kennedy and Pottier [88], pages 74–83.

[129] John C. Reynolds. Automatic computation of data set definitions. In A. J. H. Morrell, editor, *Information Processing 68*, volume 1, pages 456–461, Amsterdam, 1969. North-Holland.

[130] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. ACM Press. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [131].

[131] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[132] Amr Sabry and Matthias Felleisen. Is continuation-passing useful for data flow analysis? In Vivek Sarkar, editor, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 29, No 6, pages 1–12, Orlando, Florida, June 1994. ACM Press.

[133] Bratin Saha, Nevin Heintze, and Dino Oliva. Subtransitive CFA using types. Research Report 1166, Department of Computer Science, Yale University, 1998.

[134] David A. Schmidt. Natural-semantics-based abstract interpretation (preliminary version). In Mycroft [109], pages 1–18.

[135] David A. Schmidt. Abstract interpretation of small-step semantics. In Mads Dam, editor, *Selected papers from the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 76–99. Springer-Verlag, 1997.

[136] David A. Schmidt. Trace-based abstract interpretation of operational semantics. *Lisp and Symbolic Computation*, 10(3):237–271, 1998.

[137] Manuel Serrano. Control flow analysis: a functional languages compilation paradigm. In *Proceedings of the 1995 ACM Symposium on Applied Computing*, pages 118–122, Nashville, TN, USA, February 1995. ACM.

[138] Manuel Serrano and Marc Feeley. Storage use analysis and its applications. In R. Kent Dybvig, editor, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, Philadelphia, Pennsylvania, May 1996. ACM Press.

[139] Peter Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, October 1988.

[140] Peter Sestoft. Replacing function parameters by global variables. In Joseph E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 39–53, London, England, September 1989. ACM Press.

[141] Peter Sestoft. *Analysis and efficient implementation of functional programs*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1991. DIKU Rapport 92/6.

[142] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In Talcott [154], pages 150–161.

[143] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Muchnick and Jones [107], pages 189–233.

[144] Olin Shivers. Control-flow analysis in Scheme. In Mayer D. Schwartz, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 23, No 7, pages 164–174, Atlanta, Georgia, June 1988. ACM Press. Reprinted in *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979–1999): A Selection*.

[145] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[146] Olin Shivers. The semantics of Scheme control-flow analysis. In Paul Hudak and Neil D. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, pages 190–198, New Haven, Connecticut, June 1991. ACM Press.

[147] Jeffrey Mark Siskind. Flow-directed lightweight closure conversion. Technical Report 99-190R, NEC Research Institute, Inc., Dec 1999.

[148] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, 1997.

[149] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.

[150] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In Gyimóthy [56], pages 136–150.

[151] Bjarne Steensgaard. Points-to analysis in almost linear time. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, Florida, January 1996. ACM Press.

[152] Bjarne Steensgaard and Morten Marquard. A polyvariant closure analysis with dynamic widening. Unpublished note. Available at `ftp://ftp.research.microsoft.com/pub/analysts/closure.ps.Z`, May 1994.

[153] Dan Stefanescu and Yuli Zhou. An equational framework for the flow analysis of higher order functional programs. In Talcott [154], pages 318–327.

[154] Carolyn L. Talcott, editor. *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.

[155] Yan Mei Tang and Pierre Jouvelot. Control-flow effects for escape analysis. In Billaud et al. [20], pages 313–321.

[156] Yan Mei Tang and Pierre Jouvelot. Separate abstract interpretation for control-flow analysis. In Masami Hagiya and John C. Mitchell, editors, *Proceedings of the 1994 International Symposium on Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 224–243, Sendai, Japan, April 1994. Springer-Verlag.

[157] Peter Thiemann. A safety analysis for functional programs. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 133–144, Copenhagen, Denmark, June 1993. ACM Press.

[158] Mads Tofte, editor. *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, Amsterdam, The Netherlands, June 1997. ACM Press.

[159] Andrew Tolmach. Combining closure conversion with closure analysis using algebraic types. In *Proceedings of the 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997. Available as Boston College Computer Science Technical Report BCCS-97-03.

[160] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

[161] David Van Horn and Harry G Mairson. Relating complexity and precision in control flow analysis. In Norman Ramsey, editor, *Proceedings of the Twelfth ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, Freiburg, Germany, October 2007. ACM Press.

[162] Arnaud Venet. Nonuniform alias analysis of recursive data structures and arrays. In Manuel V. Hermenegildo and Germán Puebla, editors, *Static Analysis, 9th International Symposium, SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 36–51, Madrid, Spain, September 2002. Springer-Verlag.

[163] Oscar Waddell and R. Kent Dybvig. Fast and effective procedure inlining. In Pascal Van Hentenryck, editor, *Static Analysis*, volume 1302 of *Lecture Notes in Computer Science*, pages 35–52, Paris, France, September 1997. Springer-Verlag.

[164] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Infomaticae*, 10:115–122, 1987.

[165] Mitchell Wand. Analyses that distinguish different evaluation orders, or, unsoundness results in control-flow analysis. unpublished, July 2002.

[166] Mitchell Wand and Galen B. Williamson. A modular, extensible proof method for small-step flow analyses. In Daniel Le Métayer, editor, *Programming Languages and Systems, 11thEuropean Symposium on Programming, ESOP 2002*, volume 2305 of *Lecture Notes in Computer Science*, pages 213–227, Grenoble, France, April 2002. Springer-Verlag.

[167] David H. D. Warren. Higher-order extensions to PROLOG: are they needed? In J. E. Hayes, Donald Michie, and Y.-H. Pao, editors, *Machine Intelligence*, volume 10, pages 441–454. Ellis Horwood, 1982.

[168] Stephen Weeks. Whole-program compilation in MLton. In Kennedy and Pottier [88], page 1. Invited talk.

[169] J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *Journal of Functional Programming*, 12(3):183–227, 2002.

[170] Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *TOPLAS*, 20(1):166–207, 1998.

[171] Jonathan Young and Paul Hudak. Finding fixpoints on function spaces. Technical Report Research Report YALEEU/DCS/RR-505, Yale University, December 1986.

Recent BRICS Report Series Publications

RS-07-18 Jan Midtgaard. *Control-Flow Analysis of Functional Programs*. December 2007. iii+38 pp.

RS-07-17 Luca Aceto, Willem Jan Fokkink, and Anna Ingólfsdóttir. *A Cancellation Theorem for 7BCCSP*. December 2007. 30 pp.

RS-07-16 Olivier Danvy and Kevin Millikin. *On the Equivalence between Small-Step and Big-Step Abstract Machines: A Simple Application of Lightweight Fusion*. November 2007. ii+11 pp. To appear in Information Processing Letters (extended version). Supersedes BRICS RS-07-8.

RS-07-15 Jooyong Lee. *A Case for Dynamic Reverse-code Generation*. August 2007. ii+10 pp.

RS-07-14 Olivier Danvy and Michael Spivey. *On Barron and Strachey's Cartesian Product Function*. July 2007. ii+14 pp.

RS-07-13 Martin Lange. *Temporal Logics Beyond Regularity*. July 2007. 82 pp.

RS-07-12 Gerth Stølting Brodal, Rolf Fagerberg, Allan Grønlund Jørgensen, Gabriel Moruz, and Thomas Mølhave. *Optimal Resilient Dynamic Dictionaries*. July 2007.

RS-07-11 Luca Aceto and Anna Ingólfsdóttir. *The Saga of the Axiomatization of Parallel Composition*. June 2007. 15 pp. To appear in the Proceedings of CONCUR 2007, the 18th International Conference on Concurrency Theory (Lisbon, Portugal, September 4–7, 2007), Lecture Notes in Computer Science, Springer-Verlag, 2007.

RS-07-10 Claus Brabrand, Robert Giegerich, and Anders Møller. *Analyzing Ambiguity of Context-Free Grammars*. May 2007. 17 pp. Full version of paper presented at CIAA '07.

RS-07-9 Janus Dam Nielsen and Michael I. Schwartzbach. *The SMCL Language Specification*. March 2007.

RS-07-8 Olivier Danvy and Kevin Millikin. *A Simple Application of Lightweight Fusion to Proving the Equivalence of Abstract Machines*. March 2007. ii+6 pp.

RS-07-7 Olivier Danvy and Kevin Millikin. *Refunctionalization at Work*. March 2007. ii+16 pp. Invited talk at the 8th International Conference on Mathematics of Program Construction, MPC '06.