Basic Research in Computer Science

On Static and Dynamic Control-Flow Information in Program Analysis and Transformation

Daniel Damian

BRICS Dissertation Series **DS-01-5**

ISSN 1396-7002 **August 2001**

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

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
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 `DS/01/5/`

On Static and Dynamic Control-Flow Information in Program Analysis and Transformation

Daniel Damian

Ph.D. Dissertation

BRICS

BRICS Ph.D. School
Department of Computer Science
University of Aarhus
Denmark

July 2001 **Supervisor:** Olivier Danvy

On Static and Dynamic Control-Flow Information in Program Analysis and Transformation

A dissertation
presented to the Faculty of Science
of the University of Aarhus
in partial fulfillment of the requirements for the
Ph.D. degree

by
Daniel Damian
31 July 2001

Abstract

This thesis addresses several aspects of static and dynamic control-flow information in programming languages, by investigating its interaction with program transformation and program analysis.

Control-flow information indicates for each point in a program the possible program points to be executed next. Control-flow information in a program may be static, as when the syntax of the program directly determines which parts of the program may be executed next. Control-flow information may be dynamic, as when run-time values and inputs of the program are required to determine which parts of the program may be executed next. A control-flow analysis approximates the dynamic control-flow information with conservative static control-flow information.

We explore the impact of a continuation-passing-style (CPS) transformation on the result of a constraint-based control-flow analysis over Moggi's computational metalanguage. A CPS transformation makes control-flow explicitly available to the program by abstracting the remainder of the computation into a continuation. Moggi's computational metalanguage supports reasoning about higher-order programs in the presence of computational effects. We show that a non-duplicating CPS transformation does not alter the result of a monovariant constraint-based control-flow analysis.

Building on control-flow analysis, we show that traditional constraint-based binding-time analysis and traditional partial evaluation benefit from the effects of a CPS transformation, while the same CPS transformation does not affect continuation-based partial evaluation and its corresponding binding-time analysis. As an intermediate result, we show that reducing a program in the computational metalanguage to monadic normal form also improves binding times for traditional partial evaluation while it does not affect continuation-based partial evaluation.

In addition, we show that linear β-reductions have no effect on control-flow analysis. As a corollary, we solve a problem left open in Palsberg and Wand's CPS transformation of flow information. Furthermore, using Danvy and Nielsen's first-order, one-pass CPS transformation, we present a simpler CPS transformation of flow information with a simpler correctness proof.

We continue by exploring Shivers's time-stamps-based technique for approximating program analyses over programs with dynamic control flow. We formalize a time-stamps-based algorithm for approximating the least fixed point

of a generic program analysis over higher-order programs, and we prove its correctness.

We conclude by investigating the translation of first-order structured programs into first-order unstructured programs. We present a one-pass translation that integrates static control-flow information and that produces programs containing no chains of jumps, no unused labels, and no redundant labels.

Acknowledgments

First of all, I wish to thank Torben Mogensen, Mogens Nielsen and Olin Shivers for accepting to serve in my PhD committee.

Then, I wish to thank my supervisor, Olivier Danvy, for his scientific advice and for his moral support. Olivier introduced me in the world of continuations and partial evaluation, while encouraging me to follow my own scientific ideas. Ever since I came to BRICS, Olivier has been an extraordinary resourceful teacher and a patient and thoughtful mentor. I am truly grateful to him for a remarkable fellowship and for a wonderful scientific collaboration.

I am also grateful to Norman Ramsey for hosting me at Harvard University for six months. I have enjoyed working with him and his group, as I have found in him a boundless source of knowledge and inspiration.

I wish to thank Andrzej Filinski for his courses at DAIMI, and for his insightful comments and advice. I also wish to thank David Toman and Torben Amtoft for collaboration over the last four years.

Zhe Yang and I have shared an office for a large part of our PhD studies. Zhe has been helpful and supportive, both as a colleague and as a friend. His vitality and sharpness have been an encouraging and stimulating example. I also have fond memories of the enjoyable evenings spent together with my colleagues Bernd and Marcin.

The programming-languages group at BRICS has been a robust circle for discussions and for sharing research ideas. Our regular meetings in the O building were a time of lucid review, passionate debate and delicious cookies.

This thesis would not have been possible without the financial support of BRICS. I have always felt privileged to be a member of the BRICS community. At the same time, DAIMI has provided an excellent environment. Not little of this is due to our kind and helpful secretaries, Janne Christensen and Karen Kjær Møller.

Last, and most, I wish to thank my wife, Mihaela, for her love and understanding, and all my family for its support and encouragement.

Contents

Chapter 1

Introduction and Overview

Understanding the flow of control in computer programs is an essential step in the design and implementation of program analysis and program transformation. Most of today's compilers and optimizers make use of control-flow information in an automated process, in the purpose of generating more efficient programs. Control-flow information in a program indicates the possible choices of instructions following the execution of an instruction in the program.

As a notation for expressing computation, programming languages have evolved from early languages with explicit control structures into modern languages facilitating high-level description of computations. In early programming languages, explicit control structures provide static control-flow information (e.g., flowcharts). In contrast, modern higher-order programming languages increasingly rely on an automated process for extracting dynamic control-flow information and for converting high-level specifications into implementations using explicit control structures.

This thesis investigates several aspects of static and dynamic control-flow information in programming languages, and the interaction between control-flow information and program analysis and transformation. We consider the control-flow analysis of high-level languages and its interaction with the continuation-passing-style (CPS) transformation. As a follow-up, we consider the interaction between the CPS transformation and binding-time analysis as used in offline partial evaluation. We also show several methods for constructing flow information of a program after the continuation-passing-style transformation.

We consider the analysis of programs with dynamic control flow, namely higher-order programs in which the possibilities of flow of control depend on run-time values in the program. We formalize and prove the correctness of a generic technique for approximating a range of program analyses. We also consider the problem of translating structured first-order programs into unstructured first-order programs with conditional and unconditional jumps.

In the rest of the introduction we describe the main objects of discourse of the dissertation, together with background and related work relevant to the later results. We continue by providing a short description of the technical

results which can be found in the following chapters of the dissertation, and we discuss their relevance with respect to recent related work. The remainder of the dissertation is composed from the text of research articles documenting our technical results.

1.1 Background

Throughout this dissertation we use the λ-calculus [8] as a metalanguage for reasoning about higher-order programming languages. Based on a mathematical notation for expressing function abstraction and function application, the λ-calculus has proved itself as a useful tool for reasoning about programming languages semantics and transformations; for a long time it has been considered to be the foundational concept behind programming languages [75]. In its typed version, the λ-calculus has further use as a tool for reasoning about safety properties of languages.

Various formal semantics of λ-terms have been defined, among them, denotational semantics [109] and operational semantics [99]. A semantics for λ-terms usually involves a choice of evaluation strategy, for instance call-by-name or call-by-value [98]. A semantics, incorporating an evaluation strategy, determines reasoning principles, as, for instance, relations among meanings of programs and their transformations. In this dissertation, we consider programs with call-by-value semantics, although we avoid being specific about the precise definition of their semantics.

We also consider languages with computational effects. Computational effects, such as nondeterminism, mutable state, continuations or exceptions, are powerful and useful programming tools. Introducing computational effects breaks however the reasoning principles available at the level of pure λ-calculus. For instance, programs which may be equivalent in a the setting of the pure λ-calculus are no longer equivalent in a setting which allows computational effects.

The notion of monads [81, 118] has proved to be a useful tool for reasoning about programming languages with computational effects. Moggi's computational metalanguage [82] extends the λ-calculus and provides a theoretical foundation which restores some of the reasoning principles familiar in the pure λ-calculus. Using Moggi's computational metalanguage, computational effects can be abstracted and sound reasoning principles can be established independently of such effects. Furthermore, the computational metalanguage has also been used as a support for reasoning about partial evaluation or program analysis.

1.1.1 Continuations and continuation-passing style

Functional programming languages are also based on the notion of function abstraction and function application. In this sense, the control of the execution of the program is implicit. Functional programs are declarative rather than imperative: the user declares the functions that are used, and declares the way

in which they are composed in order to compute the final result. Therefore, in functional programming languages, the user has little direct interaction with the control flow of the program, which is generally believed to lead to more readable programs [37, 59].

Nevertheless, the need for a handle on the control-flow of the program, allowing, for instance, jumps out of the current scope or current procedure, has lead to the introduction of continuations. After Strachey and Wadsworth [114], continuations are essentially an abstraction of the remainder of the computation. In functional programming languages, one can abstract the context of an expression into a continuation: a function that receives the value of the expression and returns it to the context. The evaluation of an expression completes by calling the continuation with the computed value.

The notion of continuations has emerged in several forms in the 1960's and early 1970's, see for instance Reynolds's survey [102]. Since then, continuations have been used in compilation [3, 6, 44, 113], logic [48, 83], and semantics [43, 101, 109]. Turning a term into continuation passing style amounts to making explicit the continuations inside the term. Such transformation enables a simple implementation of control constructs present in modern programming languages as Scheme [73] or ML [7]. On a semantic level, a continuation semantics [109, 114] enables reasoning about explicit control features as jumps in first-order languages or call-with-current-continuation in higher-order languages. Together with mutable locations, continuations can be used to model arbitrary monadic computational effects [42].

The transformation of λ-terms into continuation-passing style has been the subject of a long line of research. Plotkin [98] has introduced call-by-name and call-by-value CPS transformations, and has established the key property of evaluation-order independence. The CPS transformations yield large terms, and Plotkin identifies so-called "administrative redexes" which are redexes introduced by the translation which are not legitimate part of the computation in the original program. Danvy and Filinski [31] identify the flow of continuations in administrative reductions and produce a one-pass CPS transformation which yields CPS terms without administrative redexes. The one-pass CPS transformation is higher-order though: it involves a function which takes a term and returns another term. Reasoning on such a transformation becomes difficult [36].

CPS terms without administrative redexes may be obtained by embedding the original term in the computational metalanguage, reducing it into monadic normal form, and introducing continuations [49]. The result is equivalent to the result of a Plotkin-style CPS transformation followed by administrative reductions. Alternative CPS transformations have been developed, at the level of syntactic theories [105], or, non-compositionally, at the level of terms [107].

Recently, Danvy and Nielsen have discovered a one-pass CPS transformation [34, 86] which yields CPS terms without administrative redexes through a compositional first-order translation. This aspects make Danvy and Nielsen's CPS transformation a more suitable support for reasoning about CPS programs using structural-induction principles. In Chapter 4 we make use of this property to obtain a simpler proof of correctness of the CPS transformation of flow

information.

1.1.2 Static program analysis

Static program analysis is at the heart of today's modern compilers. The goal of program analysis is to extract program properties to enable program transformations, usually with the purpose of optimizing the run-time behavior of the program.

Typical question answered through program analysis concern run-time properties of programs. For instance, data-flow analysis in first-order languages [23, 71] determines properties of program points dependent on the set of values taken by variables during the run of a program. Such analyses are extensively used in compilers and code generators, but also in various other systems as profilers, security verifiers, etc.

Cousot and Cousot's seminal work in abstract interpretation [23] has opened the path to a wide area of research in formalizing and proving the correctness of program analyses. Kam and Ullman have introduced the notion of monotone frameworks [71]. In particular, they have characterized the importance of distributivity for data-flow analyses and have shown the undecidability of the meet-over-all-paths (MOP) solution for constant propagation. Nielson also has connected the maximal fixed-point (MFP) and the MOP solutions of a data-flow analysis with a direct-style and, respectively, continuation-passing style semantic formulation of a collecting interpretation [87].

A range of analyses are used in determining the static/dynamic aspect of program points in partial evaluation [66] (binding-time analyses) or determining the security level of program points and information flow [1, 55, 127] (security analyses). Other analyses determine properties relative to memory management, for instance region inference for higher-order languages [117]. But of crucial importance to analysis of higher-order languages is control-flow analysis, which is the topic of next section.

1.1.3 Control-flow analysis

In higher-order languages, extracting run-time properties of programs often involves a control-flow analysis. In functional languages, for instance, such an analysis determines a conservative approximation of what functions may be called at each application point in the program.

Early control-flow analyses by Jones and Mycroft [68] extracted minimal control-flow graphs. Shivers designed and proved the semantic correctness of a control-flow analysis for CPS Scheme [112], presenting at the same time several direct applications of control-flow analysis such as induction variable elimination, type recovery and super-β. Sestoft's closure analysis [111] uses abstract interpretation to extract the control-flow information as a least fixed point of a functional; Bondorf's thesis [12] makes use of control-flow information to perform defunctionalization [35, 103].

Recent developments have introduced the specification of a control-flow analysis through a set of constraints generated by the input program. Originally introduced by Palsberg and Schwartzbach in the context of object-oriented languages [95], constraint-based control-flow analysis has soon been extended to higher-order applicative languages [47, 61]. The analysis of a program proceeds by extracting a set of constraints and employing a standard worklist algorithm to solve the constraints. The algorithm works in cubic time on the size of the program.

Constraint-based analysis has been connected with other forms of analysis, in particular with set-based analysis [51, 52] and type inference [53, 94]. These connections are further used to extend flow analysis to typed intermediate languages [62].

Extensions of the analysis include adding data-flow information or by introducing polyvariance on various forms of context information [89, 90, 112, 125]. Such extensions yield more precise results, often at a price in the run-time of the analysis. On the other direction, observing that the cubic time upper-bound of the constraint-based analysis may still be prohibitive for large programs, other authors propose more tradeoffs between speed and precision [54, 56, 116].

1.1.4 Partial evaluation and binding-time analysis

Partial evaluation [22, 66] is a technique of automatic program specialization. The goal if partial evaluation is to produce more efficient specialized versions of generic programs, by exploiting the knowledge about static inputs. Most often, such knowledge includes the static parts of the control flow of the input programs [80].

A partial evaluator is a program to perform partial evaluation. Given an input program and a part of its inputs, a partial evaluator produces as output a residual program. The residual program, evaluated together with the remaining input of the original program, produces the same result as the evaluation of the original program with the complete input.

An online partial evaluator specializes programs by attempting to straightforwardly interpret the given program. The online partial evaluator performs computation where possible and generates code otherwise. Naturally, this interpretation involves a large amount of symbolic computation. Moreover, termination is difficult to ensure and there is little control over the specialization process and over the size and shape of the residual program.

The offline approach to partial evaluation involves two phases. First, a binding-time analysis is performed. The analysis requires that the user distinguishes the binding times of the inputs of the program into static (known at specialization time) and dynamic (not known at specialization time). The analysis then traverses the input program and determines the binding times of control structures and variables in the program. The result of binding-time analysis is usually in the form of static/dynamic annotations on individual statements of the source program. The second phase then amounts to performing the computation annotated as static and generating code for the statements annotated as

dynamic.

Offline partial evaluation based on annotations specified through a two-level λ-calculus [66, 88] has been thoroughly investigated [14, 65, 78, 79, 92, 121]. More recently, Hatcliff and Danvy have proven the correctness of a partial evaluator for Moggi's computational metalanguage [50]. The partial evaluator relies on a binding-time analysis through type inference. Such an analysis is closely related to a flow-based binding-time analysis [93].

Continuation-based partial evaluation [13, 50, 66, 76] has emerged from the need for binding-time improvements. A continuation-based partial evaluator implements a continuation-based reduction engine for the static language. Such reduction engine has the property of extracting potentially static code out of dynamic contexts, which is known to be a binding-time improvement [32, 33].

1.1.5 Continuation-passing style and program analysis

The interaction between the continuation-passing style and program analysis has been studied on several lines of research. Nielson's work on data-flow analysis [87] showed how a continuation semantics yields a more precise (and possibly uncomputable) analysis, by relating the direct-style abstract collecting semantics with the MFP solution and the continuation-passing abstract collecting semantics with the MOP solution. Consel and Danvy show that BTA yields more precise results on a CPS-transformed program [20]. Likewise, Muylaert-Filho and Burn's work [84] suggests a benefic effect of a syntactic CPS transformation on strictness analysis.

On a different note, Sabry and Felleisen's article "Is continuation-passing useful for data-flow analysis?" [106] shows that the results of a constant-propagation analysis on a CPS transformed program may be incomparable with the results of the same constant-propagation analysis over the original program. The result has a negative tone, since, for instance, a compiler writer might not be able to decide whether to CPS-transform a program before or after the analysis phase.

More recent work, by Palsberg and Wand [97] and Damian and Danvy [26], independently observed that a definite answer can be found if one restricts one self to a constraint-based control-flow analysis. Such result gives to a compiler writer more freedom of choice, since, while the constant-propagation analysis of Sabry and Felleisen gives more precise results due to its inherited flow-awareness, it also has a higher worst-case complexity [25]. In turn, the preservation of flow information obtained by a constraint-based analysis enables one to prove properties about the interaction between CPS and binding-time analysis used in partial evaluation.

1.1.6 Static transition compression

In partial-evaluation terminology [66, Section 4.4], static transition compression amounts to compressing chains of jumps in residual programs at specialization

time. The process of specializing first-order flow-chart programs leads to residual programs in form of unstructured labeled commands where control flow is expressed in terms of conditional and unconditional jumps.

Residual programs often contain chains of jumps, namely jumps to other unconditional jumps. Such chains of jumps impose a run-time overhead, which can be statically eliminated, since we are able to compute the ending point of the chain of jumps.

The issue of transition compression is traditional in compilers. Machine-level languages express control flow in terms of conditional and unconditional jumps. Therefore, a standard step in a compiler involves a translation from a structured language to an unstructured language where control flow is expressed only in terms of conditional and unconditional jumps [2, 4, 45, 123].

A translation of structured flow commands as conditional statements and while-loops into an unstructured language of conditional and unconditional jumps is a simple exercise. The resulting translation is simple and compositional and therefore simple to implement and to reason about. The drawback of this simplicity is the quality of the generated code. In particular, nested control-flow statements give rise to chains of jumps, potentially linear in the size of the program.

It is also simple to devise a post-processing phase removing such chains of jumps by replacing each jump in the chain to a jump to the final destination. The drawback of such a phase is that it may require multiple passes on the code to remove such jumps. One of the topics of this dissertation is a one-pass translation which generates no chains of jumps.

1.2 Overview of the dissertation

This thesis addresses several issues about static and dynamic control-flow information and its interaction with program analysis and program transformation. We consider the impact of the CPS transformation on control-flow analysis and on binding time-analysis, by exploring several different CPS transformations of flow information and binding times. We consider the issue of analyzing programs with dynamic control flow, i.e., programs in which the possibilities of flow of control depends on run-time values in the program. We formalize Shivers's time-stamps technique as a generic fixed-point algorithm and we prove its correctness. We also present a translation which achieves static transition compression.

A overview of the results included in the dissertation is as follows:

Chapter 2: Starting from the folklore observation that program analyses are fragile with respect to program transformation, we explore the impact of the CPS transformation over the results of control-flow analysis and binding-time analysis.

We use Moggi's computational metalanguage as a support for reasoning about program analysis and CPS transformation. Such choice is justified

by the reasoning principles available for the computational metalanguage, which allow us to reason about a more realistic setting of programs with computational effects. Moreover, we are able to use Hatcliff and Danvy's approach to the CPS transformation [49]: reduction to monadic normal form followed by the introduction of continuations. This separation in two steps enables us to distinguish between the effect of let-flattening and let-β-reductions on program analysis and the effect of introducing continuations. Therefore, the results are not particular to a full CPS transformation, they also account for monadic normal forms, which are a useful intermediate form for compilation and partial evaluation. We consider three target monovariant constraint-based analyses: control-flow analysis, binding-time analysis for traditional partial evaluation and binding-time analysis for continuation-based partial evaluation.

We show that reduction to monadic normal form and introduction of continuations does not affect the results of the control-flow analysis. Given the flow information extracted by the analysis of the source program, for each of these two transformations, we are able to construct, in linear time, the flow information extracted by the analysis of the transformed program. Such a construction is particularly useful in the case of control-flow analysis after the introduction of continuations. In this case, due to the larger size of the CPS terms and the cubic complexity of the analysis, it is cheaper to perform the analysis on the original program, also because we now know that no information is lost or gained by the CPS transformation.

Using this result, we also show that the CPS transformation improves the result of the constraint-based binding-time analysis for traditional partial evaluation. Our results formalize folklore knowledge about binding-time-improvements by CPS transformation in partial evaluation [20]. The improvements come from two main sources.

1. As part of the reduction to monadic normal forms, let-flattening is shown to improve the results of the binding-time analysis. This result essentially provides a static version of Hatcliff and Danvy's online let-flattening [50], where let-flattening occurs inside the specializer, after binding-time separation.
2. The introduction of continuations is proven to provide a binding-time improvement. The abstraction of contexts provides a support for maintaining static computations inside dynamic contexts. Binding-time-improvements originate in the let-rule which requires that the body of a let must be dynamic if its header is also dynamic. As explained in Chapter 2, such rule is needed in order to preserve dynamic computational effects. Binding-time improvements also originate in the relocation of contexts over conditional branches. Such improvements, however, might lead to code duplication, unless avoided by let-insertion.

Together, these two binding-time improvements suggest an avenue of improvements of results in similar program analyses. A CPS transformation might also lead to improvements in analyses which have similar constraints, as for instance, the ones emerging from strong non-interference requirements [1]. As we discuss in Chapter 2, security analysis might be such an analysis, but, on the other hand, region analysis might lead to mixed results.

Finally, we show that, naturally, the CPS transformation has no effect on the constraint-based binding-time analysis for continuation-based partial evaluation. Such analysis does not enforce the let rule and accounts for the CPS-based specializer's ability to relocate static computation outside dynamic contexts. This result thus leaves the user the freedom of choice between CPS-transforming a program before making use of a traditional partial evaluator, or leaving the program in direct style and using a continuation-based partial evaluator.

Chapter 3: We consider the traditional way of performing a CPS transformation, namely a Plotkin-style transformation followed by administrative reductions and we construct the corresponding CPS transformation of flow information.

Palsberg and Wand [97] have defined a CPS transformation of flow for the Plotkin-style CPS transformation. They transform the flow information obtained by the constraint-based analysis on a source program into the flow information for the CPS program. Plotkin's CPS transformation leads to prohibitively large terms including administrative redexes, and administrative reductions remain unaccounted.

Starting from the observation that administrative reductions are linear, we prove a general statement about linear β-reductions and constraint-based control-flow analysis. We show that the control-flow information for a program after a linear reduction is a simple restriction of the control-flow information of the original program. We are therefore able to say that linear β-reductions do not affect the result of the control-flow analysis. We also conclude that we are able to extract the flow information for the resulting program from the control-flow information of the original program by a simple projection.

Turning back to the CPS transformation of flow, by successively applying linear reductions, we are able to conclude that a Plotkin-style CPS transformation followed by administrative reductions does not affect the control-flow analysis. We are also able to construct the flow information for the CPS term without administrative reductions as a direct projection of the CPS flow information.

Chapter 4: Following Danvy and Nielsen's recent discovery of a compositional, first-order, one-pass CPS transformation, we adapt the CPS transformation of flow to this new syntactic support. As we also explain in Chapter 4,

there are several reasons for which the new CPS transformation of flow has a simpler definition and simpler proof of correctness.

The compositional formulation of the transformation is essential for performing a proof by induction on the CPS transformation. Relying also on the compositional formulation of the flow analysis, we are able to prove inductively that the constructed flow information is correct. Since the CPS transformation is first-order, we do not require reasoning on higher-order objects, and, since the transformation is one-pass, we directly generate flow information for programs without going through administrative reductions.

The simpler transformation of flow re-confirms the result of the preservation of flow information through a CPS transformation. It provides a direct, linear-time method for computing the CPS flow information. Compared with the results of the previous chapter, we no longer need to construct the intermediate flow information for the larger CPS program with administrative redexes.

Chapter 5: We reconsider Shivers's time-stamps technique in the context of program analysis for languages with dynamic control-flow graphs. We prove its correctness as a generic algorithm for approximating the least fixed point of such program analyses.

In his PhD thesis [112], Shivers extracts a formal specification of a control-flow analysis for a higher-order applicative language. At application points, the analysis explores the set of functions which may be called at run time at the application point. The number of functions is unknown, due to the dynamic aspect of the control-flow graph. Shivers proposes an algorithm based on time-stamps which approximates the result of the analysis by using a conservative approximation of the set of functions called at each application site. Due to the mutual dependency between the set computed by this analysis and the approximation obtained using time-stamps, the correctness of the method is non-trivial and only informally justified.

We present a formalization of the time-stamps technique and we prove its correctness. We define a generic formulation of a program analysis on programs with dynamic control-flow graphs, where we consider that the set of the possible future program points to be explored at a node in the graph is dependent on current analysis information. We define a time-stamps-based version of the analysis and we prove that it computes a conservative approximation of the original analysis.

In addition, we extend the initial formalization of the algorithm to non-tail-recursive formulations of program analyses. In particular, we show how to extend the time-stamps-based algorithm to apply to Sabry and Felleisen's constant propagation, and we are able to compare its result with a constraint-based analysis. We also obtain a polynomial estimate on the time and space complexity of the algorithm as a function of the

complexity of the basic operations involved. We conclude that further assessment on the practicality of the method is needed.

Chapter 6: We consider the issue of static transition compression. We consider a simple source language of structured commands using as control structures conditional statements and while loops. We also consider a simple target language of labeled unstructured commands, using as control structures conditional and unconditional jumps to labels of commands.

We show how a simple compositional translation generates chains of jumps, spurious empty instructions and redundant (multiple) labels. We show how such translation induces a run-time overhead, potentially linear on the size of the program. Taking an inspiration from the notion of continuations we extend the translation with inherited attributes including the label of the next generated command.

We obtain a compositional, one-pass and linear time translation which provably does not generate chains of jumps, spurious empty instructions, unused labels or redundant labels. The translation therefore achieves static transition compression. It thus removes the need for an expensive post-processing phase for compressing chains of jumps or for removing unused or redundant labels. It is therefore most suitable for run-time code generation environments like, for instance, just-in-time compilation or program specialization.

1.3 Conclusions and perspectives

At the term of this study, let us place our key results into perspective.

One of the main concepts presented in this dissertation is the CPS transformation of flow information. Several CPS transformations of flow are presented, based on several CPS transformations of terms. The existence of a reversible flow transformation indicates that, from the point of view of flow analysis, continuation-passing style and direct-style intermediate forms are equally attractive choices for an optimizing compiler, as long as a monovariant constraint-based analysis is being used.

More work is needed to assess the impact of the CPS transformation on the result of program analysis. Other methods of program analysis could also be investigated in this respect, possibly by relating them with constraint-based analyses which we have considered here. More precise control-flow analyses could also be explored, as, for instance, k-CFA or polymorphic-splitting-based CFA. As long as more precise information is extracted in an uniform way, my thesis (as illustrated in Chapter 2) provides a methodology for comparing results of program analyses across program transformations.

We also formalize CPS as a binding-time improvement. For constraint-based binding-time analysis, improvements originate mainly in the let-rule, both when reducing programs to monadic normal form and when introducing continuations. In essence, continuation-based partial evaluation includes all these binding-time

improvements. Analyses that share similar properties with binding-time analysis are likely to exhibit similar improvements after a CPS transformation. Given more time, we could investigate whether such improvements can be integrated in a continuation-based processor, similar to a continuation-based partial evaluator.

We also proved time-stamps to be a generic tool for approximating a whole range of analyses over programs with dynamic control flow. Further optimizations of the time-stamps-based algorithm would be worth investigating. More practical experiments and comparisons with alternative implementations could provide more new answers.

Finally, we present a simple one-pass translation from a language of structured commands to a language of unstructured commands. It should be straightforward to extend the translation to short-cut boolean operators. The formulation of the translation in terms of code blocks could also allow an integration with block-reshuffling strategies.

Control-flow information is pervasive in our thesis. We use it to construct the flow information after a CPS transformation. We also use it to compute the result of a binding-time analysis and to determine the binding times of a CPS-transformed program. A simpler definition of the CPS transformation makes control flow in the resulting program more explicit, leading to simpler proofs. Dynamic control-flow information can be approximated using time-stamps. Static control-flow information in the translation of first-order programs yields simpler and more efficient code. Our thesis therefore provides further evidence that control-flow information plays a key role in the understanding of program analysis and transformation.

Chapter 2

Syntactic Accidents in Program Analysis: On the Impact of the CPS Transformation

Abstract[1]

We show that a non-duplicating transformation into continuation-passing style (CPS) has no effect on control-flow analysis, a positive effect on binding-time analysis for traditional partial evaluation, and no effect on binding-time analysis for continuation-based partial evaluation: a monovariant control-flow analysis yields equivalent results on a direct-style program and on its CPS counterpart, a monovariant binding-time analysis yields less precise results on a direct-style program than on its CPS counterpart, and an enhanced monovariant binding-time analysis yields equivalent results on a direct-style program and on its CPS counterpart. Our proof technique amounts to constructing the CPS counterpart of flow information and of binding times.

Our results formalize and confirm a folklore theorem about traditional binding-time analysis, namely that CPS has a positive effect on binding times. What may be more surprising is that the benefit does not arise from a standard refinement of program analysis, as, for instance, duplicating continuations.

The present study is symptomatic of an unsettling property of program analyses: their quality is unpredictably vulnerable to syntactic accidents in source programs, i.e., to the way these programs

[1] This chapter is an extended version of [26]. The chapter is joint work with Olivier Danvy.

are written. More reliable program analyses require a better understanding of the effect of syntactic change.

2.1 Introduction

2.1.1 Motivation

Program analyses are vulnerable to syntactic accidents in source programs in that innocent-looking, meaning-preserving transformations may substantially alter the precision of an analysis.

For a simple example, binding-time analysis (BTA) is vulnerable to reassociation: given two static expressions s_1 and s_2 and one dynamic expression d, it makes a difference whether the source program is expressed as $(s_1 + s_2) + d$ or as $s_1 + (s_2 + d)$. In the former case, the inner addition is classified as static and the outer one is classified as dynamic. In the latter case, both additions are classified as dynamic.

With the exception of BTA (and of region inference, see Section 2.8.1.1), little is known about the effect of programming style on program analyses. BTA is an exception because its output critically determines the amount of specialization carried out by an offline partial evaluator [22, 66]. Therefore, the output of binding-time analyses has been intensively studied, especially in connection with syntactic changes in their input. As a result, "binding-time improvements" have been developed to milk out extra precision from binding-time analyses [66, Chapter 12], to the point that partial-evaluation users are encouraged to write programs in a particular style [64]. That said, binding-time-improvements are not specific to offline partial evaluation—they are also routine in staging transformations [70] and in the formal specification of programming languages for semantics-directed compiling [88, Section 8.2].

Since one of the most effective binding-time improvements is the transformation of source programs into continuation-passing style (CPS) [20, 113], people have wondered whether CPS may help program analysis in general. Nielson's early work on data-flow analysis [87] suggests so, since it shows that for a non-distributive analysis, a continuation semantics yields more precise results than a direct semantics. The CPS transformation is therefore a Good Thing, since for a direct semantics, it gives the effect of a continuation semantics. In the early 1990's, Muylaert-Filho and Burn's work [84] was providing further indication of the value of the CPS transformation for abstract interpretation when Sabry and Felleisen entered the scene.

In their stunning article "Is continuation-passing useful for data-flow analysis?" [106], Sabry and Felleisen showed that for constant propagation, analyzing a direct-style program and analyzing its CPS counterpart yields incomparable results. They showed that CPS might increase precision by duplicating continuations, and also that CPS might decrease precision by confusing return points. These results are essentially confirmed by Palsberg and Wand's recent CPS transformation of flow information [97]. At any rate, except for continuation-

based partial evaluation [50], there seems to have been no further work about the effect of CPS on the precision of program analysis in general.

The situation is therefore that the CPS transformation is known to have an unpredictable effect on constant propagation and is also believed to have a positive effect on binding-time analysis. Still, we do not know for sure whether this positive effect is truly positive, or whether it worsens binding times elsewhere in the source program. One may also wonder whether, besides distributive monotone frameworks, there exist other program analyses on which CPS has no effect.

In this article, we answer these two questions by studying the effect of a non-duplicating CPS transformation on two off-the-shelf constraint-based program analyses—control-flow analysis (CFA) and BTA. Using a uniform proof technique, we formally show that:

(1) CPS has no effect on CFA, i.e., analyzing a direct-style program and analyzing its CPS counterpart yields equivalent results.

(2) CPS does not make BTA yield less precise results, and for the class of examples for which continuation-based partial evaluation was developed, it makes BTA yield results that are strictly more precise.

(3) CPS has no effect on an enhanced BTA which takes into account continuation-based partial evaluation.

This increased precision entailed by CPS also concerns analyses that have been noticed to be structurally similar to BTA, such as security analysis, program slicing, and call tracking [1]. These analyses display a similar symptom: for example, we are told that, in practice, users tend to find security analyses too conservative, without quite knowing what to do to obtain more precise results. (Here, "more precise results" means that more parts of the source program can be classified as low security.)

In the next section, we point out how the dependency induced by let-expressions leads to a loss of precision.

2.1.2 A loophole: the let rule

Partial evaluation is a transformation technique for specializing programs. Offline partial evaluation [66] is a staged technique for specializing programs. In a first phase, the binding times of a source program (i.e., which parts are static and should be evaluated at partial-evaluation time and which parts are dynamic and should be part of the specialized program) are analyzed. In a second phase, specialization proper takes place (i.e., the static parts are evaluated and the dynamic parts are residualized). Binding-time analysis is thus a data-flow analysis and when source programs are higher-order, it is driven by control-flow information. Such information is in turn obtained by a control-flow analysis.

A partial evaluator is correct when the meaning of the residual program is the same as the meaning of the source program applied to the static input. In

particular, if the source language includes computational effects (for instance non-termination), the specializer must ensure that all the dynamic side effects of the source program are exhibited by the residual program.

To ensure this contextual coherence, a binding-time analysis classifies a let expression to be dynamic if its header is dynamic, because of possible side effects in the header and regardless of the binding time of the body. (Similarly, if a let header is classified to be of high security, the whole let expression is also classified to be of high security, regardless of the security level of its body.) Therefore, the body of the following λ-abstraction is classified as dynamic if e is dynamic:

$$\lambda x.\textbf{let } v = e \textbf{ in } b$$

The CPS counterpart of this λ-abstraction reads as follows:

$$\lambda x.\lambda k.e'\ (\lambda v.b'\ k)$$

where e' and b' are the CPS counterparts of e and b, respectively. Now, assume that b naturally yields a static result independently of x, but is coerced to be dynamic because of the let rule. In the CPS term, e' also yields a dynamic result, i.e., intuitively, v is classified to be dynamic.[2] Intuitively, b' also yields a static result and sends it to its continuation k. Therefore, in direct style, b yields a dynamic result whereas in CPS, it yields a static result.

Two observations need to be made at this point:

(1) The paragraph above is the standard motivation for improving binding times by CPS transformation [20] (see Section 2.8.2 for further detail). Nevertheless, what this paragraph leaves unsaid—and what actually has always been left unsaid—is whether this local binding-time improvement corresponds to a global improvement as well, or whether it may make things worse elsewhere in the source program. (In Section 2.7, we prove that this local improvement actually is a global improvement as well.)

(2) In their core calculus of dependency [1], Abadi et al. make a point that any function classified as $d \to s$ (resp. $h \to l$, etc.) is necessarily a constant function. Nevertheless, as argued above, given a direct-style function classified to be $d \to d$ because of the let rule, its CPS counterpart may very well be classified as $d \to (s \to o) \to o$ and *not* be a constant function in continuation-passing style (i.e., a function applying its continuation to a constant).

Together, these two observations tell us that the let rule is overly conservative in BTA, security analysis, etc. CPS makes it possible to exploit the untapped precision of this rule non-trivially by providing a local improvement which—and this is a point of this article—is also a global improvement.

This global improvement is distinct from the common method of improving precision of program analysis by duplicating the analysis over the same program

[2] This intuition is formalized in the rest of this article.

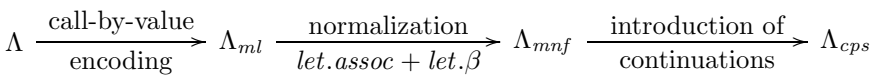


Figure 2.1: Staged CPS transformation

points. Sabry and Felleisen, for example, said that any improvement in precision provided by CPS is solely due to continuation duplication [106]. This assessment is true for their analysis, but it does not hold in general, as we have just shown for binding-time analysis.

Other approaches to improving analysis results amount to refining the definition of the analysis by including more information, such as, for instance, context information [61, 89, 90, 112]. In contrast, CPS-transforming the source program naturally provides a representation of the context as a syntactic support for refinement to the (unchanged) analysis.

In his work on data-flow analysis [87], Nielson shows that duplicating the analysis over conditional branches improves the analysis results. Let us point out that the CPS transformation also leads to binding-time improvements for conditional expressions. Indeed, to ensure contextual coherence for conditionals, the binding-time analysis makes conditional branches dynamic if the test is dynamic. This approximation can be circumvented with a CPS transformation. Therefore, the improvement is not produced by duplicating the analysis, but merely by the context relocation induced by the CPS transformation. This point is developed further in Section 2.7.4.

2.1.3 Overview

In this work we use a staged CPS transformation. Several equivalent methods exist for performing a global CPS transformation of a program. For example, one can use a Plotkin-style CPS transformation with administrative reductions [98], or one can stage the CPS transformation as normalization to a monadic normal form followed by introduction of continuations [49]. Other CPS transformations exist [105, 107], but we have not connected them with program analysis.

Therefore, we use a CPS transformation obtained as follows:

1. call-by-value embedding of the input program into Moggi's computational metalanguage [49, 82],
2. normalization under *let.assoc* and *let.β* (as defined in Hatcliff and Danvy's account of CPS [49]), and
3. introduction of continuations.

The staged transformation is visualized in the diagram of Figure 2.1.

The rest of this article is organized as follows: in Section 2.2 we define the input language, the transformation steps leading to CPS, and the program

analyses. More specifically, in Section 2.2.1 we present the labeled language of input programs. In Section 2.2.2 we review the computational metalanguage and the corresponding call-by-value encoding of the input language. In Section 2.2.3 we recall the monadic let-reductions.

We continue by introducing the constraint-based analyses for the computational metalanguage. In Section 2.2.5 we specify the control-flow analysis. In Section 2.2.6 we specify the binding-time analysis corresponding to traditional partial evaluation. In Section 2.2.7 we specify the binding-time analysis corresponding to continuation-based partial evaluation.

In Section 2.3 we outline how to compare the results of a constraint-based program analysis across a program transformation.

In Section 2.4 we evaluate the effect on constraint-based analyses incurred by the normalization of the source program with respect to let-reductions: we investigate the effect of $let.\beta$ (Section 2.4.1) and $let.assoc$ (Section 2.4.2) reductions over each of the analyses. We conclude (Section 2.4.3) that linear let-reductions and let flattening do not change the result of the control-flow analysis, while they do improve the results of the traditional binding-time analysis.

In the remainder of the article, we evaluate the effect of introducing continuations (Section 2.5) over the result of control-flow analysis (Section 2.6), binding-time analysis for traditional partial evaluation (Sections 2.7.1 to 2.7.3) and binding-time analysis for continuation-based partial evaluation (Section 2.7.4). In Section 2.8 we review related work. In Section 2.9 we conclude and discuss further issues.

2.2 Constraint-based analyses for a computational metalanguage

We introduce the language of input programs and the individual transformations performed by the CPS transformation. We then present the three program analyses: CFA, BTA and BTA*.

2.2.1 The language Λ

We consider that programs are given in an untyped λ-language Λ. The terms of the language are expressions given by the grammar of Figure 2.2. The language includes literals, λ-abstractions, recursive function definitions, conditionals and base-type operators (for simplicity, we only consider unary operators here). We focus on call by value. Since the evaluation of terms in the language may not terminate, programs in Λ may exhibit non-termination as a computational effect.

A program p is a closed expression.

$$
\begin{array}{rcll}
e & \in & \mathit{Exp} & ::= \quad x \mid n \mid \lambda x.e \mid \mathbf{rec}\ f(x).e \mid e_0\ e_1 \mid op(e) \mid \mathbf{if0}\ e\ e_0\ e_1 \\
x, f & \in & \mathit{Ide} & \text{(identifiers)} \\
n & \in & \mathit{Int} & \text{(integers)} \\
op & \in & \multicolumn{2}{l}{\text{(an unspecified set of base-type operators)}}
\end{array}
$$

Figure 2.2: The language Λ

$$
\begin{array}{rcl}
\mathcal{V}[\![x]\!] & = & \mathbf{unit}\ x \\
\mathcal{V}[\![n]\!] & = & \mathbf{unit}\ n \\
\mathcal{V}[\![\lambda x.e]\!] & = & \mathbf{unit}\ \lambda x.\mathcal{V}[\![e]\!] \\
\mathcal{V}[\![\mathbf{rec}\ f(x).e]\!] & = & \mathbf{unit}\ \mathbf{rec}\ f(x).e \\
\mathcal{V}[\![e_0\ e_1]\!] & = & \mathbf{let}\ x_0 = \mathcal{V}[\![e_0]\!] \\
 & & \mathbf{in\ let}\ x_1 = \mathcal{V}[\![e_1]\!] \\
 & & \quad \mathbf{in\ let}\ x_2 = x_0\ x_1 \\
 & & \quad\quad \mathbf{in\ unit}\ x_2 \\
\mathcal{V}[\![op(e)]\!] & = & \mathbf{let}\ x_0 = \mathcal{V}[\![e]\!]\ \mathbf{in\ let}\ x_1 = op(x_0)\ \mathbf{in\ unit}\ x_1 \\
\mathcal{V}[\![\mathbf{if0}\ e\ e_0\ e_1]\!] & = & \mathbf{let}\ x_0 = \mathcal{V}[\![e]\!] \\
 & & \mathbf{in\ let}\ x_1 = \mathbf{if0}\ x_0\ \mathcal{V}[\![e_0]\!]\ \mathcal{V}[\![e_1]\!] \\
 & & \quad \mathbf{in\ unit}\ x_1 \\
 & & \qquad\qquad \text{(where the } x_i \text{ are fresh)}
\end{array}
$$

Figure 2.3: Call-by-value encoding into the computational metalanguage

2.2.2 The computational metalanguage

The computational metalanguage Λ_{ml} [49] enforces the order of evaluation by introducing a **let** construct for naming intermediate computations and a **unit** construct for lifting a value into a computation.

$$e ::= \ldots \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{unit}\ e$$

The computational metalanguage comes with a set of sound reasoning principles about programs which may have computational effects, such as nontermination. Such principles can be used to validate program transformations performed, for instance, inside a compiler. They can also be used to validate, for instance, a partial evaluator [50].

In order to make use of such principles, an input program in the language Λ is encoded into the computational metalanguage, enforcing its order of evaluation. For call by value, the encoding into the monadic metalanguage is defined in Figure 2.3.

Notice that, in addition to other known call-by-value encodings [10, 49, 107], we name the result of the application of two values (x_1 and x_2 in the translation of an application). This cosmetic change (indeed, it is only a $let.\eta$ expansion in the computational metalanguage) is part of our development of the CPS transformation of flow information.

$$\begin{array}{rl}
\textbf{let } x = \textbf{unit } t \textbf{ in } e \rightarrow_{let.\beta} & e[t/x] \\
\textbf{let } x = e \textbf{ in unit } x \rightarrow_{let.\eta} & e \\
\textbf{let } x_2 = \textbf{let } x_1 = e_1 \textbf{ in } e_2 \textbf{ in } e \rightarrow_{let.assoc} & \textbf{let } x_1 = e_1 \textbf{ in let } x_2 = e_2 \textbf{ in } e
\end{array}$$

Figure 2.4: The monadic let reductions

2.2.3 The monadic let reductions

The call-by-value encoding leads to a separation of terms into two categories: trivial terms (noted with t) and serious terms (noted with s). Trivial terms represent values: constants, variables, λ-abstractions and recursive function definitions. Serious terms represent computations: applications, basic operations, conditionals, nesting of computations by naming intermediate results.

We recall the monadic let-reductions. Normalization under the let-reductions is the first step in a staged CPS transformation [49]. The let-β reduction, let-η reduction and the let flattening reduction are presented in Figure 2.4.

2.2.4 Λ_v: a call-by-value subset of the computational metalanguage

In this paper, we focus on the call-by-value embedding. Therefore, we restrict ourselves to the subset of Λ_{ml} that forms the image of the call-by-value embedding of Λ. The language Λ_v of labeled terms is defined in Figure 2.5. Indeed, the call-by-value embedding produces either trivial terms ($t \in \mathit{Triv}$) or let-expressions. All serious terms ($s \in \mathit{Step}$) are named. Since for the call-by-value embedding the occurrences of the **unit** construct can be deduced from the context, we omit them in Λ_v terms.

Note that the language is such that the final result of a computation is also named. We no longer perform *let*.η reductions in Λ_v before introducing continuations. This aspect is part of our development of the CPS transformation of flow information, and will be illustrated further in Section 2.6.

For the purpose of program analysis, terms are labeled with labels ℓ taken from a countable set *Lab*. In addition, λ-abstractions and recursive functions are identified by labels π from another set *Lam*, so that, for example, in $(\lambda^{\pi} x.e^{\ell_1})^{\ell_0}$, ℓ_0 and ℓ_1 belong to *Lab* and π belongs to *Lam*.

Definition 1. *A* properly labeled *expression is a labeled expression in which all labels are distinct and all variables are distinct.*

We should note that, for the purpose of control-flow analysis or binding-time analysis, it is not essential that the input program is properly labeled. But the precision of the analysis is increased if distinct program points have distinct labels, and distinct variables have distinct names. Since we want to compare the absolute precision of an analysis before and after program transformation, we consider the best results that the analysis can give over the program. For this

$$
\begin{array}{rcl}
p \in \mathit{Pgm} & ::= & e^{\ell} \\
e \in \mathit{Exp} & ::= & t \mid \textbf{let } x = s \textbf{ in } e^{\ell} \\
s \in \mathit{Step} & ::= & t^{\ell} \mid t_0^{\ell_0}\ t_1^{\ell_1} \mid \mathit{op}(t^{\ell}) \mid \textbf{if0 } t^{\ell}\ e_0^{\ell_0}\ e_1^{\ell_1} \mid (\textbf{let } x = s \textbf{ in } e^{\ell_1})^{\ell_2} \\
t \in \mathit{Triv} & ::= & n \mid x \mid \lambda^{\pi} x.e^{\ell} \mid \textbf{rec}^{\pi} f(x).e^{\ell} \\
x \in \mathit{Ide} & & \text{(identifiers)} \\
n \in \mathit{Int} & & \text{(integers)} \\
\ell \in \mathit{Lab} & & \text{(term labels)} \\
\pi \in \mathit{Lam} & & (\lambda\text{-abstraction labels}) \\
\mathit{op} \in & & \text{an unspecified set of base-type operators}
\end{array}
$$

Figure 2.5: Λ_v: The call-by-value subset of the computational metalanguage

reason, we consider only properly labeled programs and only transformations that lead to properly labeled programs.

2.2.5 Control-flow analysis for Λ_v

We consider a constraint-based, monovariant control-flow analysis (CFA) over programs in Λ_v. The constraint-based version [47, 61, 89, 93] is known to be equivalent to other versions, based on different methods such as set-based analysis [52] and type inference [94]; it is also known to be an instance of abstract interpretation [24]. For uniformity, we adopt the same definition and notation as in Nielson, Nielson and Hankin's recent textbook on program analysis [90].[3]

The flow information computed by the analysis is a pair consisting of an abstract cache $\widehat{C}_{\mathrm{cf}}$ mapping terms to abstract values and an abstract environment $\widehat{\rho}_{\mathrm{cf}}$ mapping variables to abstract values. Abstract values are sets of labels of λ-abstractions to which a term can be reduced and a variable can be bound. The constraint-based control-flow analysis is specified as a relation $\models_{\mathrm{cf}}$ on caches, environments and terms. Given a term e, $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) \models_{\mathrm{cf}} e$ means that $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$ is a result of the control-flow analysis of e.[4]

In this work we use the syntax-directed variant of the analysis [90, Chapter 3], and we restrict its analysis relation to a relation $\models^p_{\mathrm{cf}}$ associated to each program p being analyzed. Given a properly labeled program $p \in \Lambda_{ml}$, the functionality of the associated relation $\models^p_{\mathrm{cf}}$ is defined in Figure 2.6. The analysis relation is defined in Figure 2.7 by induction over the syntax of the program.

Any solution $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$ accepted by the relation $\models^p_{\mathrm{cf}}$ (i.e., such that the statement $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) \models^p_{\mathrm{cf}} p$ holds) is a conservative approximation of the exact flow information [90, Chapter 3]. Furthermore, the analysis relation $\models^p_{\mathrm{cf}}$ has a model-intersection property, i.e., the set of solutions accepted by $\models^p_{\mathrm{cf}}$ is closed under intersection. The model-intersection property ensures the existence of a least solution of the analysis, i.e., a most precise one. (Here, the order relation is given

[3] Nielson, Nielson and Hankin's CFA is developed for a call-by-value language with recursion and let-constructs. It is thus compatible with the language considered here.

[4] In the notation of Nielson, Nielson, and Hankin [90], $\models_{\mathrm{cf}}$ is simply $\models$.

$$
\begin{array}{rl}
Lam^p & \text{The set of } \lambda\text{-abstraction labels in } p \\
Var^p & \text{The set of identifiers in } p \\
Lab^p & \text{The set of term labels in } p
\end{array}
$$

$$
\begin{array}{rrll}
 & Val^p_{\mathrm{cf}} & = \mathcal{P}(Lam^p) & \text{Abstract values} \\
\widehat{C}_{\mathrm{cf}} \in & Cache^p_{\mathrm{cf}} & = Lab^p \to Val^p_{\mathrm{cf}} & \text{Abstract cache} \\
\widehat{\rho}_{\mathrm{cf}} \in & Env^p_{\mathrm{cf}} & = Var^p \to Val^p_{\mathrm{cf}} & \text{Abstract environment}
\end{array}
$$

$$
\models^p_{\mathrm{cf}} \subseteq (Cache^p_{\mathrm{cf}} \times Env^p_{\mathrm{cf}}) \times Lab^p
$$

Figure 2.6: CFA relation for a program p

by the pointwise ordering of functions induced by set inclusion.) In practice, a work-list based algorithm computes the least solution.

2.2.6 Binding-time analysis for Λ_v and traditional partial evaluation

We consider a constraint-based binding-time analysis (BTA) for the call-by-value subset Λ_v of the computational metalanguage. The analysis is an adaptation of Hatcliff and Danvy's BTA for the computational metalanguage [50], presented in constraint form [92, 93, 96]. The analysis determines binding times of program points and program variables. The binding-time information is used in offline partial evaluation [22, 66, 92]. The result of the analysis determines the static computations performed at specialization time.

The constraint-based BTA uses flow information to determine the binding times of the operators and operands of applications. Alternatively, we could have considered an analysis computing both flow and binding-time information at the same time, which is known to give equivalent results [93]. We have chosen to separate the flow analysis from the binding-time analysis in order to reuse results on preservation of flow.

The formal definition of the analysis is similar to the definition of the CFA of Section 2.2.5. The analysis is a relation defined on essentially the same domains (Figure 2.8); the difference is that the domain of abstract values is now the standard lattice $\{\mathbf{S} \sqsubseteq \mathbf{D}\}$ of static and dynamic annotations. The analysis relation is defined inductively over the syntax (Figure 2.9). At application points, the definition of the BTA refers to the flow information $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$, which is considered to be the least solution of the control-flow analysis of Section 2.2.5.

In contrast to the CFA of Section 2.2.5, the BTA accepts non-closed terms. Following the tradition, we consider the program to be dynamic and its free variables to be dynamic as well. The flow information for the free variables is considered to be empty, which is the result of applying the CFA to the program closed by abstraction over the free variables. Another difference with the CFA of Section 2.2.5 is that the constraints generated by the BTA are equality

$$
\begin{array}{lll}
(\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} n^{\ell} & \Longleftrightarrow & \mathit{true} \\
(\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} x^{\ell} & \Longleftrightarrow & \widehat{\rho}_{\mathrm{cf}}(x) \subseteq \widehat{C}_{\mathrm{cf}}(\ell) \\
(\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} (\lambda^{\pi} x.e^{\ell_1})^{\ell} & \Longleftrightarrow & \{\pi\} \subseteq \widehat{C}_{\mathrm{cf}}(\ell) \wedge (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} e^{\ell_1} \\
(\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} (\mathbf{rec}^{\pi} f(x).e^{\ell_1})^{\ell} & \Longleftrightarrow & \{\pi\} \subseteq \widehat{C}_{\mathrm{cf}}(\ell) \wedge \{\pi\} \subseteq \widehat{\rho}_{\mathrm{cf}}(f) \wedge \\
 & & (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} e^{\ell_1} \\
(\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} (\mathbf{let}\ x = t^{\ell}\ \mathbf{in}\ e^{\ell_1})^{\ell_2} & \Longleftrightarrow & (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} t^{\ell} \wedge (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} e^{\ell_1} \wedge \\
 & & \widehat{C}_{\mathrm{cf}}(\ell) \subseteq \widehat{\rho}_{\mathrm{cf}}(x) \wedge \widehat{C}_{\mathrm{cf}}(\ell_1) \subseteq \widehat{C}_{\mathrm{cf}}(\ell_2) \\
(\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} (\mathbf{let}\ x = t_0^{\ell_0}\ t_1^{\ell_1} & \Longleftrightarrow & (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} t_0^{\ell_0} \wedge (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} t_1^{\ell_1} \wedge \\
\quad \mathbf{in}\ e^{\ell_2})^{\ell_3} & & (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} e^{\ell_2} \wedge \widehat{C}_{\mathrm{cf}}(\ell_2) \subseteq \widehat{C}_{\mathrm{cf}}(\ell_3) \wedge \\
 & & \forall(\lambda^{\pi} y.e_1^{\ell}) \in \widehat{C}_{\mathrm{cf}}(\ell_0). \\
 & & \quad (\widehat{C}_{\mathrm{cf}}(\ell_1) \subseteq \widehat{\rho}_{\mathrm{cf}}(y) \wedge \widehat{C}_{\mathrm{cf}}(\ell) \subseteq \widehat{\rho}_{\mathrm{cf}}(x)) \\
(\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} (\mathbf{let}\ x = op(t^{\ell}) & \Longleftrightarrow & (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} t^{\ell} \wedge (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} e^{\ell_1} \wedge \\
\quad \mathbf{in}\ e^{\ell_1})^{\ell_2} & & \widehat{C}_{\mathrm{cf}}(\ell_1) \subseteq \widehat{C}_{\mathrm{cf}}(\ell_2) \\
(\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} (\mathbf{let}\ x = & \Longleftrightarrow & (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} t^{\ell} \wedge (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} e_0^{\ell_0} \wedge \\
\quad \mathbf{if0}\ t^{\ell}\ e_0^{\ell_0}\ e_1^{\ell_1} & & (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} e_1^{\ell_1} \wedge (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} e^{\ell_2} \wedge \\
\quad \mathbf{in}\ e^{\ell_2})^{\ell_3} & & \widehat{C}_{\mathrm{cf}}(\ell_0) \subseteq \widehat{\rho}_{\mathrm{cf}}(x) \wedge \widehat{C}_{\mathrm{cf}}(\ell_1) \subseteq \widehat{\rho}_{\mathrm{cf}}(x) \wedge \\
 & & \widehat{C}_{\mathrm{cf}}(\ell_2) \subseteq \widehat{C}_{\mathrm{cf}}(\ell_3) \\
(\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} (\mathbf{let}\ x = (\mathbf{let}\ x_1 = s & \Longleftrightarrow & (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} (\mathbf{let}\ x_1 = s\ \mathbf{in}\ e_1^{\ell_1})^{\ell_2} \wedge \\
\quad \mathbf{in}\ e_1^{\ell_1})^{\ell_2} & & (\widehat{C}_{\mathrm{cf}},\widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} e^{\ell_3} \wedge \widehat{C}_{\mathrm{cf}}(\ell_2) \subseteq \widehat{\rho}_{\mathrm{cf}}(x) \wedge \\
\quad \mathbf{in}\ e^{\ell_3})^{\ell_4} & & \widehat{C}_{\mathrm{cf}}(\ell_3) \subseteq \widehat{C}_{\mathrm{cf}}(\ell_4)
\end{array}
$$

Figure 2.7: Control-flow analysis (CFA)

constraints.

Finally, additional constraints are generated for λ-abstractions, conditionals and let-expressions. For example, the argument and body of an abstraction are dynamic if the abstraction itself is dynamic. As mentioned in Section 2.1.2, the following binding-time constraints ensure contextual coherence. In each let expression, the body is constrained to be dynamic if the header is dynamic. In each conditional expression, both branches are constrained to be dynamic if the test is dynamic. Note that we allow static operations in dynamic contexts so that static computations can take place at partial-evaluation time. A proof of correctness of a specializer using the annotations obtained by this traditional BTA can be found in Hatcliff and Danvy's work [50].

2.2.7 Binding-time analysis for Λ_v and continuation-based partial evaluation

As mentioned in Section 2.1.2, the traditional binding-time analysis from Section 2.2.6 is overly conservative because of the context coherence constraint imposed in the let rule. The constraint reflects the concern about which reductions can be safely performed by the specializer. Indeed, in the computational

$$
\begin{array}{rcll}
Val_{\mathrm{bt}} & = & \{\mathbf{S}, \mathbf{D}\} & \text{Abstract values} \\
\widehat{C}_{\mathrm{bt}} \in Cache^p_{\mathrm{bt}} & = & Lab^p \to Val_{\mathrm{bt}} & \text{Abstract cache} \\
\widehat{\rho}_{\mathrm{bt}} \in Env^p_{\mathrm{bt}} & = & Var^p \to Val_{\mathrm{bt}} & \text{Abstract environment}
\end{array}
$$

$$
\vDash^p_{\mathrm{bt}} \subseteq (Cache^p_{\mathrm{bt}} \times Env^p_{\mathrm{bt}}) \times Lab^p
$$

Figure 2.8: BTA relation for a program p

metalanguage [50], a named dynamic computation cannot be discarded due to possible computational effects. Similarly, the contextual coherence constraint over the conditional branches is introduced because one cannot decide statically which conditional branch should be selected. We will show in Sections 2.4 and 2.7 that these context coherence constraints are the source of binding-time improvements by CPS transformation.

The context coherence constraint on the body of a let-expression can be relaxed if one uses a *continuation-based program specializer* [13, 50, 76]. The context coherence constraint connecting the conditional branches with the test can be relaxed as well if one allows the same continuation-based specializer to lift the test above the context, either by duplicating the context or by naming the continuation with a let-expression.

We consider a binding-time analysis which takes into account a continuation-based specializer. More formally, we consider the BTA of Figure 2.9, without the context coherence constraints mentioned above. The functionality of the new relation $\vDash^p_{\mathrm{bt}^\star}$ is defined in Figure 2.10, and it is identical to the functionality of the traditional BTA relation $\vDash^p_{\mathrm{bt}}$ (Figure 2.8). To define the new BTA relation, we replace the rules for let-expressions and conditional expressions as specified in Figure 2.11. The result is BTA*.

2.3 Comparing analysis results across program transformations

How do we compare the results of a program analysis before and after a program transformation? The result of an analysis is a function mapping labels and program variables to analysis information. For simplicity, we expect that the transformation preserves some of the labels and variables of the initial program. Under this assumption, we relate the results of the analysis by comparing the analysis information associated with the labels and variables preserved by the transformation.

Let us say that the program p is transformed into the program p'. Let us assume that the points (labels and variables) common to p and p' are identified as a set L. Let S be an arbitrary solution of the analysis of p and S' be an arbitrary solution of the analysis of p'. We consider that the solutions S and S' are equivalent if $S'|_L = S|_L$, where $S|_L$ is the restriction of the mapping S to

$$
\begin{array}{lcl}
(\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} n^\ell & \Longleftrightarrow & \mathit{true} \\
(\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} x^\ell & \Longleftrightarrow & \widehat{\rho}_{\mathrm{bt}}(x) = \widehat{C}_{\mathrm{bt}}(\ell) \\
(\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} (\lambda^\pi x.e^{\ell_1})^\ell & \Longleftrightarrow & (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} e^{\ell_1} \wedge \\
 & & (\widehat{C}_{\mathrm{bt}}(\ell) = \mathbf{D} \Rightarrow \widehat{C}_{\mathrm{bt}}(\ell_1) = \widehat{\rho}_{\mathrm{bt}}(x) = \mathbf{D}) \\
(\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} (\mathbf{rec}^\pi f(x).e^{\ell_1})^\ell & \Longleftrightarrow & (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} e^{\ell_1} \wedge \widehat{C}_{\mathrm{bt}}(\ell) = \widehat{\rho}_{\mathrm{bt}}(f) \wedge \\
 & & (\widehat{C}_{\mathrm{bt}}(\ell) = \mathbf{D} \Rightarrow \widehat{C}_{\mathrm{bt}}(\ell_1) = \widehat{\rho}_{\mathrm{bt}}(x) = \mathbf{D}) \\
(\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} (\mathbf{let}\ x = t^\ell & \Longleftrightarrow & (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} t^\ell \wedge (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} e^{\ell_1} \wedge \\
\quad \mathbf{in}\ e^{\ell_1})^{\ell_2} & & \widehat{C}_{\mathrm{bt}}(\ell) = \widehat{\rho}_{\mathrm{bt}}(x) \wedge \widehat{C}_{\mathrm{bt}}(\ell_1) = \widehat{C}_{\mathrm{bt}}(\ell_2) \wedge \\
 & & \widehat{\rho}_{\mathrm{bt}}(x) = \mathbf{D} \Rightarrow \widehat{C}_{\mathrm{bt}}(\ell_1) = \mathbf{D} \\
(\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} (\mathbf{let}\ x = t_0^{\ell_0}\ t_1^{\ell_1} & \Longleftrightarrow & (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} t_0^{\ell_0} \wedge (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} t_1^{\ell_1} \wedge \\
\quad \mathbf{in}\ e^{\ell_2})^{\ell_3} & & (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} e^{\ell_2} \wedge \widehat{C}_{\mathrm{bt}}(\ell_2) = \widehat{C}_{\mathrm{bt}}(\ell_3) \wedge \\
 & & (\widehat{C}_{\mathrm{bt}}(\ell_0) = \mathbf{D} \Rightarrow \widehat{C}_{\mathrm{bt}}(\ell_1) = \widehat{\rho}_{\mathrm{bt}}(x) = \mathbf{D}) \wedge \\
 & & (\widehat{\rho}_{\mathrm{bt}}(x) = \mathbf{D} \Rightarrow \widehat{C}_{\mathrm{bt}}(\ell_2) = \mathbf{D}) \wedge \\
 & & \forall(\lambda^\pi y.e_1^\ell) \in \widehat{C}_{\mathrm{cf}}(\ell_0).(\widehat{C}_{\mathrm{bt}}(\ell_1) = \widehat{\rho}_{\mathrm{bt}}(y) \wedge \\
 & & \qquad \widehat{C}_{\mathrm{bt}}(\ell) = \widehat{\rho}_{\mathrm{bt}}(x)) \\
(\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} (\mathbf{let}\ x = op(t^\ell) & \Longleftrightarrow & (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} t^\ell \wedge (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} e^{\ell_1} \wedge \\
\quad \mathbf{in}\ e^{\ell_1})^{\ell_2} & & \widehat{C}_{\mathrm{bt}}(\ell) \sqsubseteq \widehat{\rho}_{\mathrm{bt}}(x) \wedge \widehat{C}_{\mathrm{bt}}(\ell_1) = \widehat{C}_{\mathrm{bt}}(\ell_2) \wedge \\
 & & (\widehat{\rho}_{\mathrm{bt}}(x) = \mathbf{D} \Rightarrow \widehat{C}_{\mathrm{bt}}(\ell_1) = \mathbf{D}) \\
(\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} (\mathbf{let}\ x = & \Longleftrightarrow & (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} t^\ell \wedge (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} e_0^{\ell_0} \wedge \\
\quad \mathbf{if0}\ t^\ell\ e_0^{\ell_0}\ e_1^{\ell_1} & & (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} e_1^{\ell_1} \wedge (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} e^{\ell_2} \wedge \\
\quad \mathbf{in}\ e^{\ell_2})^{\ell_3} & & \widehat{C}_{\mathrm{bt}}(\ell_0) = \widehat{C}_{\mathrm{bt}}(\ell_1) = \widehat{\rho}_{\mathrm{bt}}(x) \wedge \\
 & & (\widehat{C}_{\mathrm{bt}}(\ell) = \mathbf{D} \Rightarrow \widehat{C}_{\mathrm{bt}}(\ell_0) = \widehat{C}_{\mathrm{bt}}(\ell_1) = \mathbf{D}) \wedge \\
 & & (\widehat{\rho}_{\mathrm{bt}}(x) = \mathbf{D} \Rightarrow \widehat{C}_{\mathrm{bt}}(\ell_2) = \mathbf{D}) \wedge \\
 & & \widehat{C}_{\mathrm{bt}}(\ell_2) = \widehat{C}_{\mathrm{bt}}(\ell_3) \\
(\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} (\mathbf{let}\ x = & \Longleftrightarrow & (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} (\mathbf{let}\ x_1 = s\ \mathbf{in}\ e_1^{\ell_1})^{\ell_2} \wedge \\
\quad (\mathbf{let}\ x_1 = s & & (\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} e^{\ell_3} \wedge \widehat{C}_{\mathrm{bt}}(\ell_2) = \widehat{\rho}_{\mathrm{bt}}(x) \wedge \\
\quad \mathbf{in}\ e_1^{\ell_1})^{\ell_2} & & (\widehat{\rho}_{\mathrm{bt}}(x) = \mathbf{D} \Rightarrow \widehat{C}_{\mathrm{bt}}(\ell_1) = \mathbf{D}) \wedge \\
\quad \mathbf{in}\ e^{\ell_3})^{\ell_4} & & \widehat{C}_{\mathrm{bt}}(\ell_3) = \widehat{C}_{\mathrm{bt}}(\ell_4) \\
(\widehat{C}_{\mathrm{bt}},\widehat{\rho}_{\mathrm{bt}}) \models^p_{\mathrm{bt}} p & \Longleftrightarrow & (\forall x.x \text{ free in } p \Rightarrow \widehat{\rho}_{\mathrm{bt}}(x) = \mathbf{D}) \wedge \\
 & & (p = e^\ell \Rightarrow \widehat{C}_{\mathrm{bt}}(\ell) = \mathbf{D})
\end{array}
$$

Figure 2.9: Binding-time analysis for traditional partial evaluation (BTA)

the set L of common program points.

To establish a relationship between the two best analysis results we use a constructive technique. Given an arbitrary solution S of a constraint-based analysis of a program p, we show how to construct an equivalent solution S' of the analysis of the transformed program p'. We then show that the construction is valid, i.e., that S' is a valid solution of the analysis. Our construction induces a monotone mapping Φ between the two spaces of solutions. From the model-

$$
\begin{array}{rrll}
 & Val_{\mathrm{bt}} & = \{\mathbf{S}, \mathbf{D}\} & \text{Abstract values} \\
\widehat{C}_{\mathrm{bt}} \in & Cache^p_{\mathrm{bt}} & = Lab^p \to Val_{\mathrm{bt}} & \text{Abstract cache} \\
\widehat{\rho}_{\mathrm{bt}} \in & Env^p_{\mathrm{bt}} & = Var^p \to Val_{\mathrm{bt}} & \text{Abstract environment}
\end{array}
$$

$$
\vDash^p_{\mathrm{bt}^\star} \subseteq (Cache^p_{\mathrm{bt}} \times Env^p_{\mathrm{bt}}) \times Lab^p
$$

Figure 2.10: BTA* relation for a program p

$$
\begin{array}{lcl}
(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} n^\ell & \Longleftrightarrow & \mathit{true} \\
(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} x^\ell & \Longleftrightarrow & \widehat{\rho}_{\mathrm{bt}}(x) = \widehat{C}_{\mathrm{bt}}(\ell) \\
(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} (\lambda^\pi x.e^{\ell_1})^\ell & \Longleftrightarrow & (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} e^{\ell_1} \wedge \\
 & & (\widehat{C}_{\mathrm{bt}}(\ell) = \mathbf{D} \Rightarrow \widehat{C}_{\mathrm{bt}}(\ell_1) = \widehat{\rho}_{\mathrm{bt}}(x) = \mathbf{D}) \\
(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} (\mathbf{rec}^\pi f(x).e^{\ell_1})^\ell & \Longleftrightarrow & (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} e^{\ell_1} \wedge \widehat{C}_{\mathrm{bt}}(\ell) = \widehat{\rho}_{\mathrm{bt}}(f) \wedge \\
 & & (\widehat{C}_{\mathrm{bt}}(\ell) = \mathbf{D} \Rightarrow \widehat{C}_{\mathrm{bt}}(\ell_1) = \widehat{\rho}_{\mathrm{bt}}(x) = \mathbf{D}) \\
(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} (\mathbf{let}\ x = t^\ell & \Longleftrightarrow & (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} t^\ell \wedge (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} e^{\ell_1} \wedge \\
\quad \mathbf{in}\ e^{\ell_1})^{\ell_2} & & \widehat{C}_{\mathrm{bt}}(\ell) = \widehat{\rho}_{\mathrm{bt}}(x) \wedge \widehat{C}_{\mathrm{bt}}(\ell_1) = \widehat{C}_{\mathrm{bt}}(\ell_2) \\
(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} (\mathbf{let}\ x = t_0^{\ell_0}\ t_1^{\ell_1} & \Longleftrightarrow & (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} t_0^{\ell_0} \wedge (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} t_1^{\ell_1} \wedge \\
\quad \mathbf{in}\ e^{\ell_2})^{\ell_3} & & (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} e^{\ell_2} \wedge \widehat{C}_{\mathrm{bt}}(\ell_2) = \widehat{C}_{\mathrm{bt}}(\ell_3) \wedge \\
 & & (\widehat{C}_{\mathrm{bt}}(\ell_0) = \mathbf{D} \Rightarrow \widehat{C}_{\mathrm{bt}}(\ell_1) = \widehat{\rho}_{\mathrm{bt}}(x) = \mathbf{D}) \wedge \\
 & & \forall(\lambda^\pi y.e_1^\ell) \in \widehat{C}_{\mathrm{cf}}(\ell_0).(\widehat{C}_{\mathrm{bt}}(\ell_1) = \widehat{\rho}_{\mathrm{bt}}(y) \wedge \\
 & & \qquad \widehat{C}_{\mathrm{bt}}(\ell) = \widehat{\rho}_{\mathrm{bt}}(x)) \\
(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} (\mathbf{let}\ x = \mathit{op}(t^\ell) & \Longleftrightarrow & (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} t^\ell \wedge (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} e^{\ell_1} \wedge \\
\quad \mathbf{in}\ e^{\ell_1})^{\ell_2} & & \widehat{C}_{\mathrm{bt}}(\ell) \sqsubseteq \widehat{\rho}_{\mathrm{bt}}(x) \wedge \widehat{C}_{\mathrm{bt}}(\ell_1) = \widehat{C}_{\mathrm{bt}}(\ell_2) \\
(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} (\mathbf{let}\ x = & \Longleftrightarrow & (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} t^\ell \wedge (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} e_0^{\ell_0} \wedge \\
\quad \mathbf{if0}\ t^\ell\ e_0^{\ell_0}\ e_1^{\ell_1} & & (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} e_1^{\ell_1} \wedge (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} e^{\ell_2} \wedge \\
\quad \mathbf{in}\ e^{\ell_2})^{\ell_3} & & \widehat{C}_{\mathrm{bt}}(\ell_0) = \widehat{C}_{\mathrm{bt}}(\ell_1) = \widehat{\rho}_{\mathrm{bt}}(x) \wedge \\
 & & \widehat{C}_{\mathrm{bt}}(\ell_2) = \widehat{C}_{\mathrm{bt}}(\ell_3) \\
(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} (\mathbf{let}\ x = & \Longleftrightarrow & (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} (\mathbf{let}\ x_1 = s\ \mathbf{in}\ e_1^{\ell_1})^{\ell_2} \wedge \\
\quad (\mathbf{let}\ x_1 = s & & (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} e^{\ell_3} \wedge \widehat{C}_{\mathrm{bt}}(\ell_2) = \widehat{\rho}_{\mathrm{bt}}(x) \wedge \\
\quad\ \mathbf{in}\ e_1^{\ell_1})^{\ell_2} & & \widehat{C}_{\mathrm{bt}}(\ell_3) = \widehat{C}_{\mathrm{bt}}(\ell_4) \\
\quad \mathbf{in}\ e^{\ell_3})^{\ell_4} & & \\
(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^p_{\mathrm{bt}^\star} p & \Longleftrightarrow & (\forall x.x \text{ free in } p \Rightarrow \widehat{\rho}_{\mathrm{bt}}(x) = \mathbf{D}) \wedge \\
 & & (p = e^\ell \Rightarrow \widehat{C}_{\mathrm{bt}}(\ell) = \mathbf{D})
\end{array}
$$

Figure 2.11: Binding-time analysis for continuation-based partial evaluation (BTA*)

(Compared to Figure 2.9, we disabled the context coherence constraints in the 5th, 6th, 7th, 8th and 9th case.)

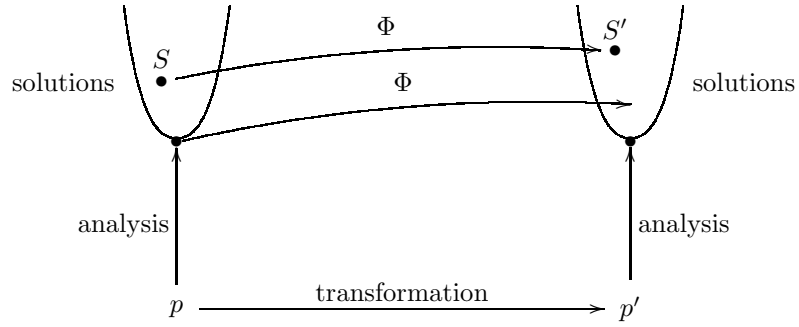


Figure 2.12: Comparing results of constraint-based analyses

intersection property of the constraint-based analyses we conclude that the best result of the analysis of p' is at least as good as the results of the analysis of p. This situation is pictured in Figure 2.12.

In some cases, given a solution of the analysis of p', we are also able to construct an equivalent solution of the analysis of p, inducing an inverse mapping Ψ. When Φ and Ψ are both monotone and their composition in both ways leads to contractions (similarly to a Galois connection), we are able to show that the best result of the analysis of p is equivalent to the best result of the analysis of p'. In such cases we conclude that the specific program transformation has no impact on the result of the analysis.

2.4 Control-flow analysis, binding-time analysis and monadic let reductions

In order to avoid generating administrative redexes when introducing continuations, Λ_{ml}-programs need to be normalized with respect to the monadic let-reductions [49]. The Λ_v language is closed under the $let.\beta$ and $let.assoc$ reductions. In this section, we investigate the effect of each of the two reductions over the constraint-based analyses defined in Section 2.2.

Concerning preservation of analysis results, the subject-reduction property of the control-flow or binding-time analyses shows that a result of an analysis will be a result of the analysis after a let-reduction. What it is not clear, however, is whether such reductions may strictly improve the results of the analysis. We rely on the linearity of the transformations to show that flow information is not improved. We also show that strict binding-time improvements may be the result of let reduction; we also show that the context coherence constraints are the cause of such improvements: disabling them leads to no improvements after a let reduction.

The let-expressions introduced by the call-by-value embedding of Figure 2.3

are linear: they do not duplicate or throw away code. Moreover, their linearity is preserved by the *let.β* and *let.assoc* reductions. In the following sections, we formalize the notion of linearity (Section 2.4.1), and use it to characterize the effect of the *let.β* and *let.assoc* reductions over CFA, BTA and BTA* (Sections 2.4.1.1 through 2.4.2.3).

2.4.1 Linear let-reduction

We formalize the notion of linear let-reduction as a *let.β* reduction such that the let body contains a unique occurrence of the variable named in the let header. The key observation, which we will prove in Section 2.4.1.1, is that linear reductions have no effect on the flow analysis. Linearity is essential: it is simple to show that non-linear (code-duplicating) reductions may improve the result of the flow analysis.

Definition 2. *A* linear context *is an expression with a unique hole* $[\cdot]$*. Linear contexts are defined by the following grammar:*

$$
\begin{array}{rl}
E ::= & T \mid (\textbf{let } x = S \textbf{ in } e^{\ell_1})^{\ell} \mid (\textbf{let } x = s \textbf{ in } E)^{\ell} \\
S ::= & T \mid T\ t_1^{\ell_1} \mid t_0^{\ell_0}\ T \mid op(T) \mid \textbf{if0}\ T\ e_0^{\ell_0}\ e_1^{\ell_1} \mid \textbf{if0}\ t^{\ell}\ E\ e_1^{\ell_1} \mid \textbf{if0}\ t^{\ell}\ e_0^{\ell_0}\ E \mid \\
& (\textbf{let } x = S \textbf{ in } e^{\ell_1})^{\ell} \mid (\textbf{let } x = s \textbf{ in } E)^{\ell} \\
T ::= & [\cdot] \mid (\lambda^{\pi} x.E)^{\ell} \mid (\textbf{rec}^{\pi} f(x).E)^{\ell}
\end{array}
$$

We use linear contexts to identify contexts which are filled as the result of a *let.β* reduction. Note that linear contexts as defined in Definition 2 are more expressive than contexts that may result from the call-by-value embedding: the CPS transformation does not extract terms from inside lambda-expressions and conditional branches. Nevertheless, the results we are presenting hold in this enlarged setting.

We also formalize the notion of a let-context as a context where a let-reduction might take place.

Definition 3. *A* let context *is an expression which contains a unique hole* $[\cdot]$ *in the place of a let-expression. Let-contexts are defined by the following grammar:*

$$
\begin{array}{rl}
E ::= & [\cdot] \mid T \mid (\textbf{let } x = S \textbf{ in } e^{\ell_1})^{\ell} \mid (\textbf{let } x = s \textbf{ in } E)^{\ell} \\
S ::= & [\cdot] \mid T \mid T\ t_1^{\ell_1} \mid t_0^{\ell_0}\ T \mid op(T) \mid \textbf{if0}\ T\ e_0^{\ell_0}\ e_1^{\ell_1} \mid \textbf{if0}\ t^{\ell}\ E\ e_1^{\ell_1} \mid \\
& \textbf{if0}\ t^{\ell}\ e_0^{\ell_0}\ E \mid (\textbf{let } x = S \textbf{ in } e^{\ell_1})^{\ell} \mid (\textbf{let } x = s \textbf{ in } E)^{\ell} \\
T ::= & (\lambda^{\pi} x.E)^{\ell} \mid (\textbf{rec}^{\pi} f(x).E)^{\ell}
\end{array}
$$

Given a linear context E and a trivial term t^{ℓ}, we use $E[t^{\ell}]$ to denote the context E with the hole $[\cdot]$ replaced with t^{ℓ}. It is trivial to see that $E[t^{\ell}]$ is a well-formed expression. We use the same notation for plugging a labeled let-expression into a let context. Again, the operation is well defined.

We use $FV(e)$ to denote the set of free variables of the expression e. This notation naturally extends to contexts, by considering the hole $[\cdot]$ to contain no free variables. We also use L as the function extracting the label of an expression. By definition, for any labeled expression e^{ℓ}, $L(e^{\ell}) = \ell$.

Definition 4. *A* linear let *is an expression of the form* **let** $x = s$ **in** e^ℓ *such that* e^ℓ *contains a unique free occurrence of* x*.*

It is immediate to see that if a let-expression **let** $x = s$ **in** e^ℓ is linear, then there exists a linear context E and a label ℓ_1 such that $e^\ell = E[x^{\ell_1}]$.

Definition 5. *A* linear *let.*β reduction *is a let.*β *reduction of a linear* **let***.*

It is relevant to notice that all the *let.*β redexes introduced by the call-by-value embedding are linear and that reducing any of these redexes does not change this property.

2.4.1.1 Linear *let.*β reduction and CFA

Let us show that a linear *let.*β reduction does not alter the results of the CFA. Let p be a properly labeled program such that there exist a let context E and a linear context E_1 such that

$$p = E[(\textbf{let } x = t^\ell \textbf{ in } E_1[x^{\ell_1}])^{\ell_2}]$$

Let p' be the program p after performing the linear *let.*β reduction:

$$p' = E[E_1[t^\ell]]$$

It is immediate to see that p' is a properly labeled program.

We show that the least solution of the flow analysis of p is equivalent to the least solution of the analysis of p'. In fact, the least solution for p' is obtained from the least solution for p by projection on the labels and variables preserved by the transformation.

We define the following functions:

- $\Phi^{let.\beta}_{\mathrm{cf}} : (Cache^p_{\mathrm{cf}} \times Env^p_{\mathrm{cf}}) \to (Cache^{p'}_{\mathrm{cf}} \times Env^{p'}_{\mathrm{cf}})$ such that
$$\Phi^{let.\beta}_{\mathrm{cf}}(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) = (\widehat{C}_{\mathrm{cf}}|_{Lab^{p'}}, \widehat{\rho}'_{\mathrm{cf}}|_{Var^{p'}})$$
- $\Psi^{let.\beta}_{\mathrm{cf}} : (Cache^{p'}_{\mathrm{cf}} \times Env^{p'}_{\mathrm{cf}}) \to (Cache^p_{\mathrm{cf}} \times Env^p_{\mathrm{cf}})$ such that, if $\Psi^{let.\beta}_{\mathrm{cf}}(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) = (\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$, then
  - $\widehat{C}_{\mathrm{cf}} = \widehat{C}'_{\mathrm{cf}} \sqcup [\ell_1 \mapsto \widehat{C}'_{\mathrm{cf}}(\ell), \ell_2 \mapsto \widehat{C}'_{\mathrm{cf}}(L(E_1[t^\ell]))]$
  - $\widehat{\rho}_{\mathrm{cf}} = \widehat{\rho}'_{\mathrm{cf}} \sqcup [x \mapsto \widehat{C}'_{\mathrm{cf}}(\ell)]$.

The two functions mediate between solutions for p and p'.

Lemma 2.4.1. *If* $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) \vDash^p_{\mathrm{cf}} p$ *then* $\Phi^{let.\beta}_{\mathrm{cf}}(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) \vDash^{p'}_{\mathrm{cf}} p'$*, and if* $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) \vDash^{p'}_{\mathrm{cf}} p'$ *then* $\Psi^{let.\beta}_{\mathrm{cf}}(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) \vDash^p_{\mathrm{cf}} p$*.*

It is immediate to show that $\Phi^{let.\beta}_{\mathrm{cf}}$ and $\Psi^{let.\beta}_{\mathrm{cf}}$ form an embedding/projection pair. The following lemma is a direct consequence.

Lemma 2.4.2. *If* $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$ *is the least solution of the CFA of* p *and* $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}})$ *is the least solution of the CFA of* p'*, then* $\Phi^{let.\beta}_{\mathrm{cf}}(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) = (\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}})$ *and* $\Psi^{let.\beta}_{\mathrm{cf}}(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) = (\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$*.*

Lemma 2.4.2 says that the result of the CFA is preserved by a linear *let.*β reduction.

2.4.1.2 Linear *let.β* reduction and BTA

We show that a linear *let.β* reduction may improve the results of the BTA. Let p and p' be as defined in the previous section. We show that the least binding times of p are as good and possibly better than the binding times of p'.

We define the function $\Phi_{\mathrm{bt}}^{let.\beta} : (Cache_{\mathrm{bt}}^{p} \times Env_{\mathrm{bt}}^{p}) \to (Cache_{\mathrm{bt}}^{p'} \times Env_{\mathrm{bt}}^{p'})$ as

$$\Phi_{\mathrm{bt}}^{let.\beta}(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) = (\widehat{C}_{\mathrm{bt}}|_{Lab^{p'}}, \widehat{\rho}'_{\mathrm{bt}}|_{Var^{p'}})$$

Lemma 2.4.3. *If* $(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash_{\mathrm{cf}}^{p} p$ *then* $\Phi_{\mathrm{bt}}^{let.\beta}(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash_{\mathrm{cf}}^{p'} p'$.

Lemma 2.4.3 says that the binding times are not worsened by a linear *let.β* reduction. Yet the analysis can yield strictly better results after a linear *let.β* reduction. In some cases, the binding times of the reduced program are strictly better than the binding times of the initial program.

For example, the call-by-value embedding of the term $(\lambda x.2)\ z$ followed by one linear *let.β* reduction yields the term:

$$\begin{array}{l}\textbf{let } x_1 = z \textbf{ in}\\ \textbf{let } x_2 = (\lambda x.2)\ x_1 \textbf{ in } x_2\end{array}$$

Considering z to be dynamic, the let-rule forces the variable x_2 to be dynamic. Therefore, the constant 2 has to be dynamic as well, and, consequently, it will be residualized at specialization time. In contrast, after one more (linear) *let.β* reduction we obtain the term

$$\textbf{let } x_2 = (\lambda x.2)\ z \textbf{ in } x_2$$

and we can see that, in a global static context, the value 2 is no longer coerced to be dynamic.

The context coherence constraint seems unjustified in the above case since evaluating the variable z has no side effects. But it is the call-by-value embedding which forces the variable z into a computation. The BTA has to impose the constraint in such cases as well [50]. At any rate, this initial loss of precision is avoided by performing the second *let.β* reduction.

In the next section we show that disabling the context coherence constraints leads to no loss or gain in the precision of the binding times.

2.4.1.3 Linear *let.β* reduction and BTA*

Let us show that the context coherence constraints from the standard BTA are the source of the benefit obtained by a linear *let.β* reduction. To do so, we show that a linear *let.β* reduction does not alter the results of the binding-time analysis for continuation-based partial evaluation, BTA*. We use the constructive technique outlined in Section 2.3. The function $\Phi_{\mathrm{bt}^\star}^{let.\beta} : (Cache_{\mathrm{bt}}^{p} \times Env_{\mathrm{bt}}^{p}) \to (Cache_{\mathrm{bt}}^{p'} \times Env_{\mathrm{bt}}^{p'})$ is identical to the one from Section 2.4.1.2. The function $\Psi_{\mathrm{bt}^\star}^{let.\beta} : (Cache_{\mathrm{bt}}^{p'} \times Env_{\mathrm{bt}}^{p'}) \to (Cache_{\mathrm{bt}}^{p} \times Env_{\mathrm{bt}}^{p})$ is defined similarly to $\Psi_{\mathrm{cf}}^{let.\beta}$ in Section 2.4.1.1. It is immediate to show that $\Psi_{\mathrm{bt}^\star}^{let.\beta} \circ \Phi_{\mathrm{bt}^\star}^{let.\beta} = id$ and $\Phi_{\mathrm{bt}^\star}^{let.\beta} \circ \Psi_{\mathrm{bt}^\star}^{let.\beta} = id$. The following lemma is a direct consequence:

Lemma 2.4.4. *If* $(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}})$ *is the least solution of the BTA** *of* p *and* $(\widehat{C}'_{\mathrm{bt}}, \widehat{\rho}'_{\mathrm{bt}})$ *is the least solution of the BTA** *of* p'*, then* $\Phi^{let.\beta}_{\mathrm{bt}^\star}(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) = (\widehat{C}'_{\mathrm{bt}}, \widehat{\rho}'_{\mathrm{bt}})$ *and* $\Psi^{let.\beta}_{\mathrm{bt}^\star}(\widehat{C}'_{\mathrm{bt}}, \widehat{\rho}'_{\mathrm{bt}}) = (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}})$.

Lemma 2.4.4 says that the binding times obtained with BTA* are preserved by a linear *let.*β reduction.

2.4.2 Let flattening

We show that a *let.assoc* reduction has no effect on the CFA and on BTA*, and that it can improve and will not degrade the results of the standard BTA.

2.4.2.1 Let flattening and CFA

Let us show that a *let.assoc* reduction does not alter the results of the CFA. Let p be a properly labeled program as a let context E such that

$$p = E[(\mathbf{let}\ x_1 = (\mathbf{let}\ x = s\ \mathbf{in}\ e_1^{\ell_1})^{\ell}\ \mathbf{in}\ e_2^{\ell_2})^{\ell_3}]$$

Let p' be the program p after reassociating the let constructs:

$$p' = E[(\mathbf{let}\ x = s\ \mathbf{in}\ (\mathbf{let}\ x_1 = e_1^{\ell_1}\ \mathbf{in}\ e_2^{\ell_2})^{\ell_4})^{\ell_3}]$$

It is immediate to see that p' is a properly labeled program.

Again, we show that the least solution of the flow analysis of p is equivalent to the least solution of the analysis of p'. The least solution for p' is obtained from the least solution for p by projection on the labels and variables preserved by the transformation.

As in Section 2.4.1.1, we define the following functions:

- $\Phi^{let.assoc}_{\mathrm{cf}} : (Cache^{p}_{\mathrm{cf}} \times Env^{p}_{\mathrm{cf}}) \to (Cache^{p'}_{\mathrm{cf}} \times Env^{p'}_{\mathrm{cf}})$ such that
$$\Phi^{let.assoc}_{\mathrm{cf}}(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) = (\widehat{C}_{\mathrm{cf}}|_{Lab^{p}\setminus\{\ell\}} \sqcup [\ell_4 \mapsto \widehat{C}_{\mathrm{cf}}(\ell_2)], \widehat{\rho}_{\mathrm{cf}}).$$
- $\Psi^{let.assoc}_{\mathrm{cf}} : (Cache^{p'}_{\mathrm{cf}} \times Env^{p'}_{\mathrm{cf}}) \to (Cache^{p}_{\mathrm{cf}} \times Env^{p'}_{\mathrm{cf}})$ such that
$$\Psi^{let.assoc}_{\mathrm{cf}}(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) = (\widehat{C}'_{\mathrm{cf}}|_{Lab^{p'}\setminus\{\ell_4\}} \sqcup [\ell \mapsto \widehat{C}'_{\mathrm{cf}}(\ell_1)], \widehat{\rho}'_{\mathrm{cf}}).$$

The two functions mediate between solutions of the analysis of p and p'.

Lemma 2.4.5. *If* $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} p$ *then* $\Phi^{let.assoc}_{\mathrm{cf}}(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) \models^{p'}_{\mathrm{cf}} p'$*, and if* $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) \models^{p'}_{\mathrm{cf}} p'$ *then* $\Psi^{let.assoc}_{\mathrm{cf}}(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) \models^{p}_{\mathrm{cf}} p$.

Following the constructive technique from Section 2.3, we can easily prove the following lemma.

Lemma 2.4.6. *If* $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$ *is the least solution of the CFA of* p *and* $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}})$ *is the least solution of the CFA of* p'*, then* $\Phi^{let.assoc}_{\mathrm{cf}}(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) = (\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}})$ *and* $\Psi^{let.assoc}_{\mathrm{cf}}(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) = (\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$.

Lemma 2.4.6 says that the result of the CFA is preserved by let flattening.

2.4.2.2 Let flattening and BTA

Let us show that an isolated let flattening may improve the results of the BTA. Let p and p' be as defined in Section 2.4.2.1.

Again, we show that any binding times of p have their equivalent in p'. We define the function $\Phi_{\mathrm{bt}}^{let.assoc} : (Cache_{\mathrm{bt}}^{p} \times Env_{\mathrm{bt}}^{p}) \to (Cache_{\mathrm{bt}}^{p} \times Env_{\mathrm{bt}}^{p})$ such that

$$\Phi_{\mathrm{bt}}^{let.assoc}(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) = (\widehat{C}_{\mathrm{bt}}|_{Lab^{p} \setminus \{\ell\}} \sqcup [\ell_4 \mapsto \widehat{C}_{\mathrm{bt}}(\ell_2)], \widehat{\rho}_{\mathrm{bt}}).$$

Obviously $\Phi_{\mathrm{bt}}^{let.assoc}(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}})$ is the equivalent of $(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}})$. The following lemma shows that $\Phi_{\mathrm{bt}}^{let.assoc}$ constructs valid solutions.

Lemma 2.4.7. *If* $(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash_{\mathrm{bt}}^{p} p$ *then* $\Phi_{\mathrm{bt}}^{let.assoc}(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash_{\mathrm{bt}}^{p'} p'$.

Since $\Phi_{\mathrm{bt}}^{let.assoc}$ constructs valid equivalent solutions, by the considerations of Section 2.3, it follows that the binding times are not worsened by a let-flattening. The analysis, however, can yield strictly better results. In some cases, the binding times after a let-flattening are strictly better than the binding times of the initial program.

For example, the call-by-value embedding of the program $succ((\lambda x.2)\ (pred(z)))$, after a few $let.\beta$ and one $let.assoc$ reductions, leads to:

$$\begin{array}{l}\textbf{let } x_1 = \textbf{let } x_2 = pred(z)\\ \qquad\qquad \textbf{in let } x_3 = (\lambda x.2)\ x_2 \textbf{ in } x_3\\ \textbf{in let } x_4 = succ(x_1) \textbf{ in } x_4\end{array}$$

The program above reassociates to:

$$\begin{array}{l}\textbf{let } x_2 = pred(z)\\ \textbf{in let } x_1 = \textbf{let } x_3 = (\lambda x.2)\ x_2 \textbf{ in } x_3\\ \quad \textbf{in let } x_4 = succ(x_1) \textbf{ in } x_4\end{array}$$

In the first program, the let rule forces x_1 to be dynamic and the $succ(x_1)$ computation is dynamic. In the second program x_1 can be static, and the $succ(x_1)$ computation may be performed statically, and only its result (3) will be residualized.

2.4.2.3 Let flattening and BTA*

Let us show that for the $let.assoc$ reduction (similarly to the $let.\beta$ reduction in Section 2.4.1.3), all binding-time improvements come from the context coherence constraints. To do so, we show that a $let.assoc$ reduction has no effect on the binding-time analysis for continuation-based partial evaluation BTA*.

Taking p and p' as defined in Section 2.4.2.1, we define two functions $\Phi_{\mathrm{bt}^\star}^{let.assoc} : (Cache_{\mathrm{bt}}^{p} \times Env_{\mathrm{bt}}^{p}) \to (Cache_{\mathrm{bt}}^{p'} \times Env_{\mathrm{bt}}^{p'})$ and $\Psi_{\mathrm{bt}^\star}^{let.assoc} : (Cache_{\mathrm{bt}}^{p'} \times Env_{\mathrm{bt}}^{p'}) \to (Cache_{\mathrm{bt}}^{p} \times Env_{\mathrm{bt}}^{p})$ which map solutions of BTA* for p into solutions of BTA* for p' and vice-versa. The functions are essentially defined as in Section 2.4.2.1. One can show that $\Phi_{\mathrm{bt}^\star}^{let.assoc} \circ \Psi_{\mathrm{bt}^\star}^{let.assoc} = id$ and $\Phi_{\mathrm{bt}^\star}^{let.assoc} \circ \Psi_{\mathrm{bt}^\star}^{let.assoc} = id$. The following lemma is an immediate consequence:

$$
\begin{array}{rcl}
p \in \mathit{Pgm} & ::= & e^{\ell} \\
e \in \mathit{Exp} & ::= & t \mid \textbf{let } x = s \textbf{ in } e^{\ell} \\
s \in \mathit{Step} & ::= & t_0^{\ell_0}\ t_1^{\ell_1} \mid \mathit{op}(t^{\ell}) \mid \textbf{if0 } t^{\ell}\ e_0^{\ell_0}\ e_1^{\ell_1} \\
t \in \mathit{Triv} & ::= & n \mid x \mid \lambda^{\pi} x.e^{\ell} \mid \textbf{rec}^{\pi} f(x).e^{\ell}
\end{array}
$$

Figure 2.13: Λ_{mnf}: The subset of Λ_v normalized with respect to *let*.β and *let*.*assoc*

Lemma 2.4.8. *If* $(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}})$ *is the least solution of the BTA** *of* p *and* $(\widehat{C}'_{\mathrm{bt}}, \widehat{\rho}'_{\mathrm{bt}})$ *is the least solution of the BTA** *of* p'*, then* $\Phi_{\mathrm{bt}^{\star}}^{let.assoc}(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) = (\widehat{C}'_{\mathrm{bt}}, \widehat{\rho}'_{\mathrm{bt}})$ *and* $\Psi_{\mathrm{bt}^{\star}}^{let.assoc}(\widehat{C}'_{\mathrm{bt}}, \widehat{\rho}'_{\mathrm{bt}}) = (\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}})$.

2.4.3 Summary and conclusions

We have shown that, once the input program is embedded into the computational metalanguage, *let*.β and *let*.*assoc*-normalization can yield binding-time improvements. At the same time linear *let*.β and *let*.*assoc* preserve the quality of flow information. This property confirms that monadic normal forms are a valuable intermediate representation in a program transformer and in an optimizing compiler.

2.5 Introducing continuations

The language resulting from normalizing terms in Λ_v under the *let*.β and *let*.*assoc* reductions is the language Λ_{mnf} defined in Figure 2.13. The effect of the normalization is to eliminate naming of trivial values and to flatten all nested computations. Therefore, in Λ_{mnf} a computational step can no longer be a trivial value or a nested computation.

The language Λ_{mnf} is the support for introducing continuations by the transformation shown in Figure 2.14. Introducing continuations leads to terms in a CPS language.[5] CPS is a restriction of direct style. In order to use the same program analysis, we therefore embed the CPS language into the Λ_v language. For example, applications are transformed into let-expressions that name partially applied CPS λ-abstractions and intermediate computations. Figure 2.15 displays the corresponding CPS transformation and embedding.[6] (We have omitted the labels, because they only matter in the following sections. Suffice it to say that we label each CPS trivial term with the same label as its direct-style counterpart.)

We can apply now the constraint-based analyses of Section 2.2 on both the (*let*.β + *let*.*assoc*)-normalized program and on its CPS counterpart given by the

[5] In Figure 2.14, $\widetilde{op}$ is the CPS counterpart of *op*, to ensure evaluation-order independence [98].

[6] In Figure 2.15, we use *op* instead of $\widetilde{op}$ since the direct-style language is call-by-value.

$$
\begin{array}{rll}
[\![e]\!]^{Pgm} & = \lambda k.[\![e]\!]^{Exp}\,k & \text{where } k \text{ is fresh} \\
\\
[\![n]\!]^{Triv} & = n & \\
[\![x]\!]^{Triv} & = x & \\
[\![\lambda x.e]\!]^{Triv} & = \lambda x.\lambda k.[\![e]\!]^{Exp}\,k & \text{where } k \text{ is fresh} \\
[\![\textbf{rec } f(x).e]\!]^{Triv} & = \textbf{rec } f(x).\lambda k.[\![e]\!]^{Exp}\,k & \text{where } k \text{ is fresh} \\
\\
[\![t]\!]^{Exp}\,k & = k\ [\![t]\!]^{Triv} & \\
[\![\textbf{let } x = t_0\ t_1 \textbf{ in } e]\!]^{Exp}\,k & = [\![t_0]\!]^{Triv}\ [\![t_1]\!]^{Triv}\ \lambda x.[\![e]\!]^{Exp}\,k & \\
[\![\textbf{let } x = op(t) \textbf{ in } e]\!]^{Exp}\,k & = \widetilde{op}\ [\![t]\!]^{Triv}\ \lambda x.[\![e]\!]^{Exp}\,k & \\
[\![\textbf{let } x = \textbf{if0 } t\ e_0\ e_1 \textbf{ in } e]\!]^{Exp}\,k & = \textbf{let } k_1 = \lambda x.[\![e]\!]^{Exp}\,k & \\
 & \quad \textbf{in if0 } [\![t]\!]^{Triv}\ ([\![e_0]\!]^{Exp}\,k_1)\ ([\![e_1]\!]^{Exp}\,k_1) & \\
 & \quad \text{where } k_1 \text{ is fresh} &
\end{array}
$$

Figure 2.14: Introducing continuations

transformation of Figure 2.15.

2.6 Control-flow analysis and the introduction of continuations

In order to compare the results of the CFA before and after introducing continuations, we follow the constructive technique outlined in Section 2.3. Therefore, the rest of this section is organized as follows. First, we show how to CPS-transform control-flow information (Section 2.6.1). Given a direct-style program p and an arbitrary solution of its associated analysis $(\widehat{C}_{\text{cf}}, \widehat{\rho}_{\text{cf}})$, we construct a solution $(\widehat{C}'_{\text{cf}}, \widehat{\rho}'_{\text{cf}})$ of the analysis associated to p', the CPS counterpart of p. We ensure that the construction $\Phi^{\text{CPS}}_{\text{cf}}$ builds a valid solution (Section 2.6.2). We present a converse transformation, $\Psi^{\text{CPS}}_{\text{cf}}$ (Section 2.6.3), which we also prove to be correct (Section 2.6.4). We then show that the two constructions preserve leastness (Section 2.6.5).

2.6.1 CPS transformation of control flow

Given a solution $(\widehat{C}_{\text{cf}}, \widehat{\rho}_{\text{cf}})$ of the analysis of a program p (i.e., a cache-environment pair such that $(\widehat{C}_{\text{cf}}, \widehat{\rho}_{\text{cf}}) \vDash^{p}_{\text{cf}} p$ holds), we now construct in linear time a solution $(\widehat{C}'_{\text{cf}}, \widehat{\rho}'_{\text{cf}})$ of the analysis of $p' = [\![p]\!]^{Pgm}$, the CPS counterpart of p (i.e., such that $(\widehat{C}'_{\text{cf}}, \widehat{\rho}'_{\text{cf}}) \vDash^{p'}_{\text{cf}} p'$ holds). By analogy, we refer to the construction of $(\widehat{C}'_{\text{cf}}, \widehat{\rho}'_{\text{cf}})$ out of $(\widehat{C}_{\text{cf}}, \widehat{\rho}_{\text{cf}})$ as the CPS transformation of $(\widehat{C}_{\text{cf}}, \widehat{\rho}_{\text{cf}})$ into $(\widehat{C}'_{\text{cf}}, \widehat{\rho}'_{\text{cf}})$.

As mentioned in Section 2.2.1, we have designed the CPS transformation on labeled terms so that it preserves the labels of each trivial term. In addition, each direct-style λ-abstraction is annotated with the same label as its CPS

$$
\begin{aligned}
[\![e]\!]^{Pgm} &= \lambda k.[\![e]\!]^{Exp}k && \text{where } k \text{ is fresh}\\
\\
[\![n]\!]^{Triv} &= n\\
[\![x]\!]^{Triv} &= x\\
[\![\lambda x.e]\!]^{Triv} &= \lambda x.\lambda k.[\![e]\!]^{Exp}k && \text{where } k \text{ is fresh}\\
[\![\mathbf{rec}\ f(x).e]\!]^{Triv} &= \mathbf{rec}\ f(x).\lambda k.[\![e]\!]^{Exp}k && \text{where } k \text{ is fresh}\\
\\
[\![t]\!]^{Exp}k &= \mathbf{let}\ x = k\ [\![t]\!]^{Triv}\ \mathbf{in}\ x && \text{where } x \text{ is fresh}\\
\\
[\![\mathbf{let}\ x = t_0\ t_1\ \mathbf{in}\ e]\!]^{Exp}k &= \mathbf{let}\ x_0 = [\![t_0]\!]^{Triv}\ [\![t_1]\!]^{Triv}\\
&\quad\ \mathbf{in\ let}\ x_1 = x_0\ \lambda x.[\![e]\!]^{Exp}k\ \mathbf{in}\ x_1\\
&\quad\ \text{where } x_0 \text{ and } x_1 \text{ are fresh}\\
\\
[\![\mathbf{let}\ x = op(t)\ \mathbf{in}\ e]\!]^{Exp}k &= \mathbf{let}\ x = op([\![t]\!]^{Triv})\ \mathbf{in}\ [\![e]\!]^{Exp}k\\
\\
[\![\mathbf{let}\ x = \mathbf{if0}\ t\ e_0\ e_1\ \mathbf{in}\ e]\!]^{Exp}k &= \mathbf{let}\ k_1 = \lambda x.[\![e]\!]^{Exp}k\ \mathbf{in}\\
&\quad\ \mathbf{let}\ x_1 = \mathbf{if0}\ [\![t]\!]^{Triv}\ ([\![e_0]\!]^{Exp}k_1)\ ([\![e_1]\!]^{Exp}k_1)\\
&\quad\ \mathbf{in}\ x_1\\
&\quad\ \text{where } k_1 \text{ and } x_1 \text{ are fresh}
\end{aligned}
$$

Figure 2.15: Introducing continuations and embedding into the Λ_v language

counterpart. As a consequence, the abstract values in direct style are included into the abstract values in CPS, i.e., $Lam^p \subseteq Lam^{p'}$ and $Val^p_{\mathrm{cf}} \subseteq Val^{p'}_{\mathrm{cf}}$. When introducing continuations, all the variables defined in the original direct-style program are preserved. Therefore $Var^p \subseteq Var^{p'}$. In essence, we construct a solution for the CPS program such that the flow information assigned to the variables and to the trivial terms preserved by the transformation is identical to the information found in the direct-style solution.

We also assign flow information to the newly introduced terms and variables, in particular to continuation abstractions and continuation identifiers. To this end, we use two auxiliary functions γ and ξ.

- γ extracts the labels of partially applied CPS λ-abstractions. Formally, given A a set of λ-abstractions from the program p', $\gamma(A)$ is defined as the set of λ-abstractions $\lambda^{\pi'}k.e^{\ell}$ such that $\lambda^{\pi}x.\lambda^{\pi'}k.e^{\ell} \in A$ or such that $\mathbf{rec}^{\pi}f(x).\lambda^{\pi'}k.e^{\ell} \in A$.
- ξ assigns flow information to each continuation identifier k introduced by the CPS transformation of p (at λ-abstractions and recursive function definitions). This information can be obtained from the direct-style flow information, since we can *syntactically* identify the continuation of the CPS counterpart of any direct-style application.

  Given p, $\widehat{C}_{\mathrm{cf}}$, $\widehat{\rho}_{\mathrm{cf}}$, and a continuation identifier k introduced by the trans-

$$
\begin{aligned}
[\![e^\ell]\!]^{Pgm} &= (\lambda^\pi k.[\![e^\ell]\!]^{Exp}\,k)^{\ell_0} & \widehat{C}'_{\mathrm{cf}}(\ell_0) &= \{\pi\} \quad \widehat{\rho}'_{\mathrm{cf}}(k) = \emptyset \\
[\![n^\ell]\!]^{Triv} &= n^\ell & \widehat{C}'_{\mathrm{cf}}(\ell) &= \widehat{C}_{\mathrm{cf}}(\ell) \\
[\![x^\ell]\!]^{Triv} &= x^\ell & \widehat{C}'_{\mathrm{cf}}(\ell) &= \widehat{C}_{\mathrm{cf}}(\ell)
\end{aligned}
$$

$$
\begin{aligned}
[\![(\lambda^\pi x.e^{\ell_0})^\ell]\!]^{Triv} &= (\lambda^\pi x.(\lambda^{\pi_1} k.[\![e^{\ell_0}]\!]^{Exp}\,k)^{\ell_2})^\ell \\
&\widehat{C}'_{\mathrm{cf}}(\ell) = \widehat{C}_{\mathrm{cf}}(\ell) \quad \widehat{C}'_{\mathrm{cf}}(\ell_2) = \{\pi_1\} \\
&\widehat{\rho}'_{\mathrm{cf}}(x) = \widehat{\rho}_{\mathrm{cf}}(x) \quad \widehat{\rho}'_{\mathrm{cf}}(k) = \xi(k)
\end{aligned}
$$

$$
\begin{aligned}
[\![(\mathbf{rec}^\pi f(x).e^{\ell_0})^\ell]\!]^{Triv} &= (\mathbf{rec}^\pi f(x).(\lambda^{\pi_1} k.[\![e^{\ell_0}]\!]^{Exp}\,k)^{\ell_2})^\ell \\
&\widehat{C}'_{\mathrm{cf}}(\ell) = \widehat{C}_{\mathrm{cf}}(\ell) \quad \widehat{C}'_{\mathrm{cf}}(\ell_2) = \{\pi_1\} \\
&\widehat{\rho}'_{\mathrm{cf}}(x) = \widehat{\rho}_{\mathrm{cf}}(x) \quad \widehat{\rho}'_{\mathrm{cf}}(f) = \widehat{\rho}_{\mathrm{cf}}(f) \quad \widehat{\rho}'_{\mathrm{cf}}(k) = \xi(k)
\end{aligned}
$$

$$
\begin{aligned}
[\![t^\ell]\!]^{Exp}\,k &= (\mathbf{let}\ x = k^{\ell_0}\ [\![t^\ell]\!]^{Triv}\ \mathbf{in}\ x^{\ell_1})^{\ell_2} \\
&\widehat{C}'_{\mathrm{cf}}(\ell_0) = \widehat{\rho}'_{\mathrm{cf}}(k) \\
&\widehat{C}'_{\mathrm{cf}}(\ell_2) = \widehat{C}'_{\mathrm{cf}}(\ell_1) = \widehat{\rho}'_{\mathrm{cf}}(x) = \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\left[\!\!\left[\begin{array}{l} (\mathbf{let}\ x = t_0^{\ell_0}\ t_1^{\ell_1} \\ \mathbf{in}\ e^\ell)^{\ell_2} \end{array} \right]\!\!\right]^{Exp} k &= (\mathbf{let}\ x_0 = [\![t_0^{\ell_0}]\!]^{Triv}\ [\![t_1^{\ell_1}]\!]^{Triv}\ \mathbf{in} \\
&\quad (\mathbf{let}\ x_1 = x_0^{\ell_3}\ (\lambda^\pi x.[\![e^\ell]\!]^{Exp}\,k)^{\ell_4}\ \mathbf{in}\ x_1^{\ell_5})^{\ell_6})^{\ell_7} \\
&\widehat{C}'_{\mathrm{cf}}(\ell_3) = \widehat{\rho}'_{\mathrm{cf}}(x_0) = \gamma(\widehat{C}_{\mathrm{cf}}(\ell_0)) \\
&\widehat{C}'_{\mathrm{cf}}(\ell_4) = \{\pi\} \quad \widehat{\rho}'_{\mathrm{cf}}(x) = \widehat{\rho}_{\mathrm{cf}}(x) \\
&\widehat{C}'_{\mathrm{cf}}(\ell_7) = \widehat{C}'_{\mathrm{cf}}(\ell_6) = \widehat{C}'_{\mathrm{cf}}(\ell_5) = \widehat{\rho}'_{\mathrm{cf}}(x_1) = \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\left[\!\!\left[\begin{array}{l} (\mathbf{let}\ x = op(t^\ell) \\ \mathbf{in}\ e^{\ell_0})^{\ell_1} \end{array} \right]\!\!\right]^{Exp} k &= (\mathbf{let}\ x = op([\![t^\ell]\!]^{Triv})\ \mathbf{in}\ [\![e^{\ell_0}]\!]^{Exp}\,k)^{\ell_2} \\
&\widehat{\rho}'_{\mathrm{cf}}(x) = \widehat{\rho}_{\mathrm{cf}}(x) \quad \widehat{C}'_{\mathrm{cf}}(\ell_2) = \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\left[\!\!\left[\begin{array}{l} (\mathbf{let}\ x = \\ \quad \mathbf{if0}\ t^\ell\ e_0^{\ell_0}\ e_1^{\ell_1} \\ \mathbf{in}\ e^{\ell_2})^{\ell_3} \end{array} \right]\!\!\right]^{Exp} k &= (\mathbf{let}\ k_1 = (\lambda^\pi x.[\![e^{\ell_2}]\!]^{Exp}\,k)^{\ell_4}\ \mathbf{in} \\
&\quad (\mathbf{let}\ x_1 = \mathbf{if0}\ [\![t^\ell]\!]^{Triv}\ ([\![e_0^{\ell_0}]\!]^{Exp}\,k_1)\ ([\![e_1^{\ell_1}]\!]^{Exp}\,k_1) \\
&\quad \mathbf{in}\ x_1^{\ell_5})^{\ell_6})^{\ell_7} \\
&\widehat{\rho}'_{\mathrm{cf}}(k_1) = \widehat{C}'_{\mathrm{cf}}(\ell_4) = \{\pi\} \quad \widehat{\rho}'_{\mathrm{cf}}(x) = \widehat{\rho}_{\mathrm{cf}}(x) \\
&\widehat{C}'_{\mathrm{cf}}(\ell_7) = \widehat{C}'_{\mathrm{cf}}(\ell_6) = \widehat{C}'_{\mathrm{cf}}(\ell_5) = \widehat{\rho}'_{\mathrm{cf}}(x_1) = \emptyset
\end{aligned}
$$

Figure 2.16: Transformation of control flow from direct style to CPS

formation of a λ-abstraction from p:

$$[\![\lambda^{\pi_1} x.e]\!]^{Triv} = \lambda^{\pi_1} x.\lambda k.[\![e]\!]^{Exp}\,k$$

we gather in $\xi(k)$ all the continuations that are passed at the program points where $\lambda^{\pi_1} x.e$ can be applied. Formally, $\xi(k)$ is defined as the set of all labels π such that in the CPS transformation of p into p' there exists a

transformation step

$$[\![\textbf{let } x = t_0^{\ell_0}\ t_1 \textbf{ in } e]\!]^{Exp} k_1 = \textbf{let } x_0 = [\![t_0^{\ell_0}]\!]^{Triv}\ [\![t_1]\!]^{Triv} \textbf{ in let } x_1 = x_0\ \lambda^{\pi} x.[\![e]\!]^{Exp} k_1 \textbf{ in } x_1$$

such that $\pi_1 \in \widehat{C}_{\mathrm{cf}}(\ell_0)$. We make a similar definition for the continuation identifiers introduced at recursive function definitions.

Using γ and ξ, we define $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}})$ inductively, following Figure 2.16. In the right part, for each CPS-transformation step, we assign flow values into $\widehat{C}'_{\mathrm{cf}}$ and $\widehat{\rho}'_{\mathrm{cf}}$ using previously defined values.

The construction of flow information defines a function

$$\Phi_{\mathrm{cf}}^{\mathrm{CPS}} : (Cache_{\mathrm{cf}}^{p} \times Env_{\mathrm{cf}}^{p}) \to (Cache_{\mathrm{cf}}^{p} \times Env_{\mathrm{cf}}^{p}).$$

It is easy to show that $\Phi_{\mathrm{cf}}^{\mathrm{CPS}}$ is monotone.

2.6.2 Correctness of the transformation

Let us show that the cache-environment pair constructed by $\Phi_{\mathrm{cf}}^{\mathrm{CPS}}$ is indeed a valid solution of the analysis of the CPS counterpart of p.

Theorem 2.6.1. *Given a direct-style program p and its CPS counterpart $p' = [\![p]\!]^{Pgm}$, let $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$ be a solution of the CFA of p (i.e., such that $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) \vDash_{\mathrm{cf}}^{p} p$ holds) and let $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) = \Phi_{\mathrm{cf}}^{\mathrm{CPS}}(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$. Then $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) \vDash_{\mathrm{cf}}^{p'} p'$ holds.*

Under the assumptions of the theorem, we start by observing three immediate properties of the flow transformation.

Lemma 2.6.2. *For all variables x in p, $\widehat{\rho}'_{\mathrm{cf}}(x) = \widehat{\rho}_{\mathrm{cf}}(x)$; for all trivial terms t^{ℓ} in p, $\widehat{C}'_{\mathrm{cf}}(\ell) = \widehat{C}_{\mathrm{cf}}(\ell)$; and for all expressions e^{ℓ} in p', $\widehat{C}'_{\mathrm{cf}}(\ell) = \emptyset$.*

For an arbitrary expression, we define the notion of return label to capture the return point from which CFA collects flow information, as shown just below in Lemma 2.6.3.

Definition 6. *Given a labeled expression $e^{\ell} \in Exp$, we define the return label $\mathcal{R}[\![e^{\ell}]\!]$ of e^{ℓ} by structural induction as follows:*

$$\begin{aligned} \mathcal{R}[\![t^{\ell}]\!] &= \ell \\ \mathcal{R}[\![(\textbf{let } x = s \textbf{ in } e^{\ell_1})^{\ell}]\!] &= \mathcal{R}[\![e^{\ell_1}]\!] \end{aligned}$$

Lemma 2.6.3. *Let e^{ℓ} be an arbitrary subexpression of p. Then $\widehat{C}_{\mathrm{cf}}(\mathcal{R}[\![e^{\ell}]\!]) \subseteq \widehat{C}_{\mathrm{cf}}(\ell)$.*

A return label identifies the point where a continuation is called in the CPS-transformed program. Return labels thus provide a syntactic connection between the points where flow information is collected in direct style and the points where flow information is sent to continuations in CPS.

$$
\begin{array}{rll}
[\![e^\ell]\!]^{Pgm} = & (\lambda^\pi k.[\![e^\ell]\!]^{Exp} k)^{\ell_0} & \\
[\![n^\ell]\!]^{Triv} = & n^\ell & \widehat{C}_{\mathrm{cf}}(\ell) = \widehat{C}'_{\mathrm{cf}}(\ell) \cap Lam^p \\
[\![x^\ell]\!]^{Triv} = & x^\ell & \widehat{C}_{\mathrm{cf}}(\ell) = \widehat{C}'_{\mathrm{cf}}(\ell) \cap Lam^p \\
[\![(\lambda^\pi x.e^{\ell_0})^\ell]\!]^{Triv} = & (\lambda^\pi x.(\lambda^{\pi_1} k.[\![e^{\ell_0}]\!]^{Exp} k)^{\ell_2})^\ell & \\
 & \widehat{C}_{\mathrm{cf}}(\ell) = \widehat{C}'_{\mathrm{cf}}(\ell) \cap Lam^p & \widehat{\rho}_{\mathrm{cf}}(x) = \widehat{\rho}'_{\mathrm{cf}}(x) \cap Lam^p \\
[\![(\mathbf{rec}^\pi f(x).e^{\ell_0})^\ell]\!]^{Triv} = & (\mathbf{rec}^\pi f(x).(\lambda^{\pi_1} k.[\![e^{\ell_0}]\!]^{Exp} k)^{\ell_2})^\ell & \\
 & \widehat{C}_{\mathrm{cf}}(\ell) = \widehat{C}'_{\mathrm{cf}}(\ell) \cap Lam^p & \widehat{\rho}_{\mathrm{cf}}(x) = \widehat{\rho}'_{\mathrm{cf}}(x) \cap Lam^p \\
 & & \widehat{\rho}_{\mathrm{cf}}(f) = \widehat{\rho}'_{\mathrm{cf}}(f) \cap Lam^p
\end{array}
$$

$$
[\![t^\ell]\!]^{Exp} k = (\mathbf{let}\ x = k^{\ell_0}\ [\![t^\ell]\!]^{Triv}\ \mathbf{in}\ x^{\ell_1})^{\ell_2}
$$

$$
\left[\!\!\left[\begin{array}{l} (\mathbf{let}\ x = t_0^{\ell_0}\ t_1^{\ell_1} \\ \mathbf{in}\ e^\ell)^{\ell_2} \end{array} \right]\!\!\right]^{Exp} k = \begin{array}{l} (\mathbf{let}\ x_0 = [\![t_0^{\ell_0}]\!]^{Triv}\ [\![t_1^{\ell_1}]\!]^{Triv}\ \mathbf{in} \\ \quad (\mathbf{let}\ x_1 = x_0^{\ell_3}\ (\lambda^\pi x.[\![e^\ell]\!]^{Exp} k)^{\ell_4}\ \mathbf{in}\ x_1^{\ell_5})^{\ell_6})^{\ell_7} \end{array}
$$
$$
\widehat{C}_{\mathrm{cf}}(\ell_2) = \widehat{C}_{\mathrm{cf}}(\ell) \quad \widehat{\rho}_{\mathrm{cf}}(x) = \widehat{\rho}'_{\mathrm{cf}}(x) \cap Lam^p
$$

$$
\left[\!\!\left[\begin{array}{l} (\mathbf{let}\ x = op(t^\ell) \\ \mathbf{in}\ e^{\ell_0})^{\ell_1} \end{array} \right]\!\!\right]^{Exp} k = (\mathbf{let}\ x = op([\![t^\ell]\!]^{Triv})\ \mathbf{in}\ [\![e^{\ell_0}]\!]^{Exp} k)^{\ell_2}
$$
$$
\widehat{C}_{\mathrm{cf}}(\ell_1) = \widehat{C}_{\mathrm{cf}}(\ell_0) \quad \widehat{\rho}_{\mathrm{cf}}(x) = \widehat{\rho}'_{\mathrm{cf}}(x) \cap Lam^p
$$

$$
\left[\!\!\left[\begin{array}{l} (\mathbf{let}\ x = \\ \quad \mathbf{if0}\ t^\ell\ e_0^{\ell_0}\ e_1^{\ell_1} \\ \mathbf{in}\ e^{\ell_2})^{\ell_3} \end{array} \right]\!\!\right]^{Exp} k = \begin{array}{l} (\mathbf{let}\ k_1 = (\lambda^\pi x.[\![e^{\ell_2}]\!]^{Exp} k)^{\ell_4}\ \mathbf{in} \\ \quad (\mathbf{let}\ x_1 = \mathbf{if0}\ [\![t^\ell]\!]^{Triv}\ ([\![e_0^{\ell_0}]\!]^{Exp} k_1)\ ([\![e_1^{\ell_1}]\!]^{Exp} k_1) \\ \qquad \mathbf{in}\ x_1^{\ell_5})^{\ell_6})^{\ell_7} \end{array}
$$
$$
\widehat{C}_{\mathrm{cf}}(\ell_3) = \widehat{C}_{\mathrm{cf}}(\ell_2) \quad \widehat{\rho}_{\mathrm{cf}}(x) = \widehat{\rho}'_{\mathrm{cf}}(x) \cap Lam^p
$$

Figure 2.17: Transformation of control flow from CPS to direct style

Lemma 2.6.4. *Let k be a continuation identifier introduced by the CPS transformation of a λ-abstraction from p:*

$$
[\![\lambda^{\pi_1} x_1.e^{\ell_0}]\!]^{Triv} = \lambda^{\pi_1} x_1.\lambda k.[\![e^{\ell_0}]\!]^{Exp} k
$$

Then, for each $\lambda^\pi x.e^{\ell_1} \in \widehat{\rho}'_{\mathrm{cf}}(k)$, $\widehat{C}_{\mathrm{cf}}(\mathcal{R}[\![e^{\ell_0}]\!]) \subseteq \widehat{\rho}'_{\mathrm{cf}}(x)$. Let k be a continuation identifier introduced by the CPS transformation of a recursive function definition from p:

$$
[\![\mathbf{rec}^{\pi_1} f(x_1).e^{\ell_0}]\!]^{Triv} = \mathbf{rec}^{\pi_1} f(x_1).\lambda k.[\![e^{\ell_0}]\!]^{Exp} k
$$

Then, for each $\lambda^\pi x.e^{\ell_1} \in \widehat{\rho}'_{\mathrm{cf}}(k)$, $\widehat{C}_{\mathrm{cf}}(\mathcal{R}[\![e^{\ell_0}]\!]) \subseteq \widehat{\rho}'_{\mathrm{cf}}(x)$.

Let us consider the first case. By the definition of ξ, the only possibility such that $\lambda^\pi x.e^{\ell_1} \in \widehat{\rho}'_{\mathrm{cf}}(k)$ is that the function is the continuation of an application point where $\lambda^{\pi_1} x_1.e^{\ell_0}$ is applied. Focusing on the application point, we show that $\widehat{C}_{\mathrm{cf}}(\ell_0) \subseteq \widehat{\rho}_{\mathrm{cf}}(x) = \widehat{\rho}'_{\mathrm{cf}}(x)$. From Lemma 2.6.3, $\widehat{C}_{\mathrm{cf}}(\mathcal{R}[\![e^{\ell_0}]\!]) \subseteq \widehat{C}_{\mathrm{cf}}(\ell_0)$.

The proof of Theorem 2.6.1 is sketched in Appendix 2.A.

2.6.3 Reversing the transformation

In the previous section we have shown that direct-style flow information can be transformed into CPS flow information. We can also show that any result of the analysis of a CPS-transformed program can be matched by a result of the analysis of its direct-style counterpart. Using again the structure given by the CPS transformation, we exhibit a direct-style flow transformation. Given a direct-style program p and its CPS counterpart p', and given $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}})$ a valid solution of the analysis on p', we recover in linear time a valid solution $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$ of the analysis of p.

Recovering a direct-style solution is straightforward. For variables and trivial terms in p, we are only "filtering out" the labels of continuations from the results of the analysis of p'. We define the direct-style solution by induction on the CPS transformation, following Figure 2.17. In the right part, for each CPS-transformation step, we assign flow values into $\widehat{C}_{\mathrm{cf}}$ and $\widehat{\rho}_{\mathrm{cf}}$. The left parts of Figures 2.16 and 2.17 are identical.

We can show that Figure 2.17 defines another function

$$\Psi^{\mathrm{CPS}}_{\mathrm{cf}} : (\mathit{Cache}^p_{\mathrm{cf}} \times \mathit{Env}^p_{\mathrm{cf}}) \to (\mathit{Cache}^p_{\mathrm{cf}} \times \mathit{Env}^p_{\mathrm{cf}}).$$

It is also easy to show that, like $\Phi^{\mathrm{CPS}}_{\mathrm{cf}}$ in Section 2.6.2, $\Psi^{\mathrm{CPS}}_{\mathrm{cf}}$ is monotone.

2.6.4 Correctness of the reverse transformation

Let us show that the reverse transformation indeed yields a valid solution of the analysis of the original program.

Theorem 2.6.5. *Given a direct-style program p and its CPS counterpart $p' = [\![p]\!]^{Pgm}$, let $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}})$ be a solution of the CFA of p' (i.e., such that $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) \vDash^{p'}_{\mathrm{cf}} p'$ holds) and let $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) = \Psi^{\mathrm{CPS}}_{\mathrm{cf}}(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}})$. Then $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) \vDash^{p}_{\mathrm{cf}} p$ holds.*

As in Section 2.6.2, we use intermediate results to prove Theorem 2.6.5. Working under the assumptions of the theorem, we observe two immediate properties of the reverse transformation:

Lemma 2.6.6. *For all $x \in \mathit{Var}^p$, $\widehat{\rho}_{\mathrm{cf}}(x) = \widehat{\rho}'_{\mathrm{cf}}(x) \cap \mathit{Lam}^p$; and for all trivial terms t^ℓ in p, $\widehat{C}_{\mathrm{cf}}(\ell) = \widehat{C}'_{\mathrm{cf}}(\ell) \cap \mathit{Lam}^p$.*

For an arbitrary expression, the new solution collects all the flow information from the return point of the expression.

Lemma 2.6.7. *Let e^ℓ be an expression in p. Then $\widehat{C}_{\mathrm{cf}}(\ell) = \widehat{C}_{\mathrm{cf}}(\mathcal{R}[\![e^\ell]\!])$.*

As a parallel of Lemma 2.6.4, the following lemma connects the flow at the return points of functions with the flow collected for the variables declared by continuations.

Lemma 2.6.8. *Let k be a continuation identifier introduced by the transformation of a λ-abstraction from p:*

$$[\![\lambda^{\pi_1} x_1.e^{\ell_0}]\!]^{Triv} = \lambda^{\pi_1} x_1.\lambda k.[\![e^{\ell_0}]\!]^{Exp} k$$

Then, for each $\lambda^{\pi} x.e^{\ell_1} \in \widehat{\rho}'_{\mathrm{cf}}(k)$, $\widehat{C}_{\mathrm{cf}}(\mathcal{R}[\![e^{\ell_0}]\!]) \subseteq \widehat{\rho}'_{\mathrm{cf}}(x)$. Let k be a continuation identifier introduced by the transformation of a recursive function definition from p:

$$[\![\mathbf{rec}^{\pi_1} f(x_1).e^{\ell_0}]\!]^{Triv} = \mathbf{rec}^{\pi_1} f(x_1).\lambda k.[\![e^{\ell_0}]\!]^{Exp} k$$

Then, for each $\lambda^{\pi} x.e^{\ell_1} \in \widehat{\rho}'_{\mathrm{cf}}(k)$, $\widehat{C}_{\mathrm{cf}}(\mathcal{R}[\![e^{\ell_0}]\!]) \subseteq \widehat{\rho}'_{\mathrm{cf}}(x)$.

The proof of Theorem 2.6.5 is sketched in Appendix 2.A.

2.6.5 Equivalence of flow

Let p be an arbitrary direct-style program and $p' = [\![p]\!]^{Pgm}$ its CPS counterpart. It is a matter of tedious calculations to prove the following lemma.

Lemma 2.6.9. *Given $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$ a solution of the CFA of p (i.e., such that $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) \vDash^{p}_{\mathrm{cf}} p$ holds), $\Psi^{\mathrm{CPS}}_{\mathrm{cf}}(\Phi^{\mathrm{CPS}}_{\mathrm{cf}}(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})) \subseteq (\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$. Given $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}})$ a solution of the CFA of p', (i.e., such that $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) \vDash^{p'}_{\mathrm{cf}} p'$ holds), then it holds that $\Phi^{\mathrm{CPS}}_{\mathrm{cf}}(\Psi^{\mathrm{CPS}}_{\mathrm{cf}}(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}})) \subseteq (\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}})$.*

From these two properties the following main theorem follows directly.

Theorem 2.6.10 (Equivalence of flow). *Given a direct-style program p and its CPS counterpart $p' = [\![p]\!]^{Pgm}$, let $(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$ be the least solution of the CFA of p and let $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}})$ be the least solution of the CFA of p'. Then $\Phi^{\mathrm{CPS}}_{\mathrm{cf}}(\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}}) = (\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}})$ and $\Psi^{\mathrm{CPS}}_{\mathrm{cf}}(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) = (\widehat{C}_{\mathrm{cf}}, \widehat{\rho}_{\mathrm{cf}})$.*

2.6.6 Summary and conclusions

Theorem 2.6.10 shows that the best flow information obtained by a constraint-based analysis on a direct-style program is transformed into the best flow information obtainable by the same analysis on the CPS counterpart of this program and vice versa. Lemma 2.6.2 and Lemma 2.6.6 show that the two solutions are equal on the variables and program points common to the two programs. We conclude that, for CFA as defined in Figure 2.7, no information is lost or gained by the CPS transformation.

2.7 Binding-time analysis and the introduction of continuations

We describe the effect of the introduction of continuations on the result of the BTA of a program in Λ_{mnf}. First, we define a CPS transformation of binding

times (Section 2.7.1), which we show to be correct and to preserve the quality of the binding times (Section 2.7.2). Unlike for CFA, however, we show examples where BTA on CPS terms gives more precise results than on the corresponding direct-style terms, thus showing that introducing continuations may lead to more specialization opportunities (Section 2.7.3). Finally (Section 2.7.4) we show that if we relax the constraints of the BTA to take into account continuation-based partial evaluation, then, just like CFA, no loss and no gain of information can be observed after the introduction of continuations.

2.7.1 CPS transformation of binding times

We show that the binding times obtained by analyzing the CPS counterpart of a program are at least as good as the ones obtained by analyzing the original program. We construct in linear time a solution of the BTA over the CPS-transformed program from a solution of the BTA over the original program, such that the quality of the binding times is preserved.

Given the program p and $(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}})$ a solution of the BTA over p, we define $(\widehat{C}'_{\mathrm{bt}}, \widehat{\rho}'_{\mathrm{bt}})$ as a solution of the BTA over p', the CPS counterpart of p. The definition is by induction on the introduction of continuations and is given in Figure 2.18, where the left parts are identical to the left parts of Figures 2.16 and 2.17. In the right part, we assign binding times into $\widehat{C}'_{\mathrm{bt}}$ and $\widehat{\rho}'_{\mathrm{bt}}$. As in Section 2.6, we use $\Phi^{\mathrm{CPS}}_{\mathrm{bt}}$ to denote the function induced by the transformation.

$$\Phi^{\mathrm{CPS}}_{\mathrm{bt}} : (\mathit{Cache}^{p}_{\mathrm{bt}} \times \mathit{Env}^{p}_{\mathrm{bt}}) \to (\mathit{Cache}^{p'}_{\mathrm{bt}} \times \mathit{Env}^{p'}_{\mathrm{bt}}).$$

2.7.2 Correctness of the transformation

Let us show that the solution defined in Figure 2.18 is indeed a valid solution of the BTA. We follow the same technique as in Section 2.6.2. The correctness of the transformation is established by the following theorem.

Theorem 2.7.1. *Given a direct-style program p and its CPS counterpart $p' = [\![p]\!]^{Pgm}$, let $(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}})$ be an arbitrary solution of the BTA of p (i.e., such that $(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}}) \vDash^{p}_{\mathrm{bt}} p$ holds). If $(\widehat{C}'_{\mathrm{bt}}, \widehat{\rho}'_{\mathrm{bt}}) = \Phi^{\mathrm{CPS}}_{\mathrm{bt}}(\widehat{C}_{\mathrm{bt}}, \widehat{\rho}_{\mathrm{bt}})$ then $(\widehat{C}'_{\mathrm{bt}}, \widehat{\rho}'_{\mathrm{bt}}) \vDash^{p'}_{\mathrm{bt}} p'$ holds.*

Under the assumption of the theorem, we first observe immediate properties of the CPS transformation of binding times, similar to the ones stated in Lemma 2.6.2. For instance, the binding time for expressions in CPS is equal to the binding time of the result of the program, which, as mentioned in Section 2.2.6, is dynamic.

Lemma 2.7.2. *For all variables x in p, $\widehat{\rho}'_{\mathrm{bt}}(x) = \widehat{\rho}_{\mathrm{bt}}(x)$; for all trivial terms t^{ℓ} in p, $\widehat{C}'_{\mathrm{bt}}(\ell) = \widehat{C}_{\mathrm{bt}}(\ell)$; and for all expressions e in p', $\widehat{C}'_{\mathrm{bt}}(e) = \mathbf{D}$.*

The binding time of an expression in p is equal to the binding time of its return point.

$$[\![e^\ell]\!]^{Pgm} = (\lambda^\pi k.[\![e^\ell]\!]^{Exp} k)^{\ell_0} \qquad \widehat{C}'_{\mathrm{bt}}(\ell_0) = \widehat{\rho}'_{\mathrm{bt}}(k) = \mathbf{D}$$
$$[\![n^\ell]\!]^{Triv} = n^\ell \qquad \widehat{C}'_{\mathrm{bt}}(\ell) = \widehat{C}_{\mathrm{bt}}(\ell)$$
$$[\![x^\ell]\!]^{Triv} = x^\ell \qquad \widehat{C}'_{\mathrm{bt}}(\ell) = \widehat{C}_{\mathrm{bt}}(\ell)$$
$$[\![(\lambda^\pi x.e^{\ell_0})^\ell]\!]^{Triv} = (\lambda^\pi x.(\lambda^{\pi_1} k.[\![e^{\ell_0}]\!]^{Exp} k)^{\ell_2})^\ell$$
$$\widehat{C}'_{\mathrm{bt}}(\ell_2) = \widehat{C}_{\mathrm{bt}}(\ell) \quad \widehat{C}'_{\mathrm{bt}}(\ell) = \widehat{C}_{\mathrm{bt}}(\ell)$$
$$\widehat{\rho}'_{\mathrm{bt}}(x) = \widehat{\rho}_{\mathrm{bt}}(x) \quad \widehat{\rho}'_{\mathrm{bt}}(k) = \widehat{C}_{\mathrm{bt}}(\ell)$$

$$[\![(\mathbf{rec}^\pi f(x).e^{\ell_0})^\ell]\!]^{Triv} = (\lambda^\pi x.(\mathbf{rec}^{\pi_1} f(k).[\![e^{\ell_0}]\!]^{Exp} k)^{\ell_2})^\ell$$
$$\widehat{C}'_{\mathrm{bt}}(\ell_2) = \widehat{C}_{\mathrm{bt}}(\ell) \quad \widehat{C}'_{\mathrm{bt}}(\ell) = \widehat{C}_{\mathrm{bt}}(\ell)$$
$$\widehat{\rho}'_{\mathrm{bt}}(f) = \widehat{\rho}_{\mathrm{bt}}(f) \quad \widehat{\rho}'_{\mathrm{bt}}(x) = \widehat{\rho}_{\mathrm{bt}}(x) \quad \widehat{\rho}'_{\mathrm{bt}}(k) = \widehat{C}_{\mathrm{bt}}(\ell)$$

$$[\![t^\ell]\!]^{Exp} k = (\mathbf{let}\ x = k^{\ell_0}\ [\![t^\ell]\!]^{Triv}\ \mathbf{in}\ x^{\ell_1})^{\ell_2}$$
$$\widehat{C}'_{\mathrm{bt}}(\ell_0) = \widehat{\rho}'_{\mathrm{bt}}(k)$$
$$\widehat{C}'_{\mathrm{bt}}(\ell_2) = \widehat{C}'_{\mathrm{bt}}(\ell_1) = \widehat{\rho}_{\mathrm{bt}}(x) = \mathbf{D}$$

$$\left[\!\!\left[\begin{array}{l}(\mathbf{let}\ x = t_0^{\ell_0}\ t_1^{\ell_1} \\ \mathbf{in}\ e^\ell)^{\ell_2}\end{array}\right]\!\!\right]^{Exp} k = \begin{array}{l}(\mathbf{let}\ x_0 = [\![t_0^{\ell_0}]\!]^{Triv}\ [\![t_1^{\ell_1}]\!]^{Triv}\ \mathbf{in} \\ \quad (\mathbf{let}\ x_1 = x_0^{\ell_3}\ (\lambda^\pi x.[\![e^\ell]\!]^{Exp} k)^{\ell_4}\ \mathbf{in}\ x_1^{\ell_5})^{\ell_6})^{\ell_7}\end{array}$$
$$\widehat{C}'_{\mathrm{bt}}(\ell_4) = \widehat{C}'_{\mathrm{bt}}(\ell_3) = \widehat{\rho}'_{\mathrm{bt}}(x_0) = \widehat{C}_{\mathrm{bt}}(\ell_0)$$
$$\widehat{\rho}'_{\mathrm{bt}}(x) = \widehat{\rho}_{\mathrm{bt}}(x) \quad \widehat{\rho}'_{\mathrm{bt}}(x_1) = \mathbf{D}$$
$$\widehat{C}'_{\mathrm{bt}}(\ell_7) = \widehat{C}'_{\mathrm{bt}}(\ell_6) = \widehat{C}'_{\mathrm{bt}}(\ell_5) = \mathbf{D}$$

$$\left[\!\!\left[\begin{array}{l}(\mathbf{let}\ x = op(t^\ell) \\ \mathbf{in}\ e^{\ell_0})^{\ell_1}\end{array}\right]\!\!\right]^{Exp} k = (\mathbf{let}\ x = op([\![t^\ell]\!]^{Triv})\ \mathbf{in}\ [\![e^{\ell_0}]\!]^{Exp} k)^{\ell_2}$$
$$\widehat{\rho}'_{\mathrm{bt}}(x) = \widehat{\rho}_{\mathrm{bt}}(x) \quad \widehat{C}'_{\mathrm{bt}}(\ell_2) = \mathbf{D}$$

$$\left[\!\!\left[\begin{array}{l}(\mathbf{let}\ x = \\ \quad \mathbf{if0}\ t^\ell\ e_0^{\ell_0}\ e_1^{\ell_1} \\ \mathbf{in}\ e^{\ell_2})^{\ell_3}\end{array}\right]\!\!\right]^{Exp} k = \begin{array}{l}(\mathbf{let}\ k_1 = (\lambda^\pi x.[\![e^{\ell_2}]\!]^{Exp} k)^{\ell_4}\ \mathbf{in} \\ \quad (\mathbf{let}\ x_1 = \mathbf{if0}\ [\![t^\ell]\!]^{Triv}\ ([\![e_0^{\ell_0}]\!]^{Exp} k_1)\ ([\![e_1^{\ell_1}]\!]^{Exp} k_1) \\ \quad\quad \mathbf{in}\ x_1^{\ell_5})^{\ell_6})^{\ell_7}\end{array}$$
$$\widehat{\rho}'_{\mathrm{bt}}(k_1) = \widehat{C}'_{\mathrm{bt}}(\ell_4) = \widehat{\rho}'_{\mathrm{bt}}(x) = \widehat{\rho}_{\mathrm{bt}}(x)$$
$$\widehat{C}'_{\mathrm{bt}}(\ell_7) = \widehat{C}'_{\mathrm{bt}}(\ell_6) = \widehat{C}'_{\mathrm{bt}}(\ell_5) = \widehat{\rho}'_{\mathrm{bt}}(x_1) = \mathbf{D}$$

Figure 2.18: Transformation of binding times from direct style to CPS

Lemma 2.7.3. *Let e^ℓ be an arbitrary subexpression of p. Then $\widehat{C}_{\mathrm{bt}}(\mathcal{R}[\![e^\ell]\!]) = \widehat{C}_{\mathrm{bt}}(\ell)$.*

The flow of the continuation abstractions connects the binding times of the return point of expressions and continuation variables. The binding time of the value abstracted by a continuation is equal to the binding time of any expression that the continuation can be passed to.

Lemma 2.7.4. *Let k be a continuation identifier introduced by the transforma-*

tion of a λ-abstraction from p:

$$[\![\lambda^{\pi_1} x_1.e^{\ell_0}]\!]^{Triv} = \lambda^{\pi_1} x_1.\lambda k.[\![e^{\ell_0}]\!]^{Exp} k$$

Then, for each $\lambda^{\pi} x.e^{\ell_1} \in \widehat{\rho}'_{\mathrm{cf}}(k)$, $\widehat{C}_{\mathrm{bt}}(\mathcal{R}[\![e^{\ell_0}]\!]) = \widehat{\rho}'_{\mathrm{bt}}(x)$. *Let k be a continuation identifier introduced by the transformation of a recursive function definition from p:*

$$[\![\mathbf{rec}^{\pi_1} f(x_1).e^{\ell_0}]\!]^{Triv} = \mathbf{rec}^{\pi_1} f(x_1).\lambda k.[\![e^{\ell_0}]\!]^{Exp} k$$

Then, for each $\lambda^{\pi} x.e^{\ell_1} \in \widehat{\rho}'_{\mathrm{cf}}(k)$, $\widehat{C}_{\mathrm{bt}}(\mathcal{R}[\![e^{\ell_0}]\!]) = \widehat{\rho}'_{\mathrm{bt}}(x)$.

The proof of Theorem 2.7.1 is sketched in Appendix 2.A.

Theorem 2.7.1 and Lemma 2.7.2 show that we can transform any binding-time solution of a direct-style program into a solution of its CPS counterpart in such a way that the binding times of variables and trivial terms are preserved. This preservation implies that no values are forced to be dynamic just by introducing continuations. It also implies that the static computations (applications, tests or base-type operations) in a direct-style program remain static as well in its CPS counterpart. We thus conclude that the same amount of specialization of the input program can be achieved after introducing continuations.

2.7.3 Reversing the transformation

We show that it is not always possible to reverse the CPS transformation of binding times. There are cases when the least analysis of a CPS-transformed program produces strictly more static annotations than the least analysis of its direct-style counterpart. Here is a canonical example [50], where *succ* is the successor function, and the free variable f and z are considered to be dynamic (f might denote a potentially diverging function):

$$\mathbf{let}\ r = (\lambda^{\pi} y.\mathbf{let}\ v = f\ z\ \mathbf{in}\ 2)\ 1\ \mathbf{in}\ \mathbf{let}\ r_1 = succ(r)\ \mathbf{in}\ r_1$$

In the least solution of the BTA on this term, even if the application of λ^{π} to 1 is classified as static, its result is classified as dynamic because of the dynamic application in the header of its inner let-expression. Thus r is dynamic. Since the second increment operation depends on r, it is dynamic as well. Simply discarding the dynamic computation $f\ z$ is not meaning-preserving since the computation may diverge.

The CPS counterpart of the canonical example above reads as follows (without embedding it into direct style, for readability):

$$\lambda k.(\lambda^{\pi} y.\lambda k_1.f\ z\ (\lambda v.k_1\ 2))\ 1\ (\lambda r.\mathbf{let}\ r_1 = succ(r)\ \mathbf{in}\ k\ r_1)$$

The continuation denoted by k_1 is static, and thus the application $k_1\ 2$ is performed statically (even if its result is dynamic). Thus, r is static as well, and further computation based on r can be performed at specialization time.

Other binding-time improvements can be obtained when a dynamic test disables further computations based on its result. The canonical example is as follows:

$$\textbf{let } v = \textbf{if0 } z\ 0\ 1 \textbf{ in let } v_1 = \mathit{succ}(v) \textbf{ in } v_1$$

It is true that one benefits from such an improvement only by allowing code duplication. But the code duplication takes place at specialization time, not at BTA time. Thus in contrast to Sabry and Felleisen's analysis [106], the improvement in precision is not due to duplicating the analysis on the two branches.

2.7.4 Continuation-based partial evaluation

In the two examples above the binding-time improvements come from the context coherence constraints in the specification of the BTA (Figure 2.9): the body of a let-expression has to be dynamic if the header is dynamic, and both branches of a conditional have to be dynamic if the test is dynamic.

In this section, we show that these contextual coherence constraints are the only ones leading to binding-time improvements. Using the same proof technique as in Section 2.6, we can formally show that introducing continuations has no effect on BTA*, i.e., it entails no local increase and also no loss of precision elsewhere in the program: the best binding times in direct style are the best binding times in CPS as well.

More precisely, we can define $\Phi^{\mathrm{CPS}}_{\mathrm{bt}^\star}$, the CPS transformation of the binding times obtained by BTA*. The definition is only a slight modification of the definition of $\Phi^{\mathrm{CPS}}_{\mathrm{bt}}$ in Section 2.7.1. Given the program p and a solution $(\widehat{C}_{\mathrm{bt}^\star}, \widehat{\rho}_{\mathrm{bt}^\star})$ of BTA* (i.e., such that $(\widehat{C}_{\mathrm{bt}^\star}, \widehat{\rho}_{\mathrm{bt}^\star}) \vDash^{p}_{\mathrm{bt}^\star} p$ holds), we can show that $\Phi^{\mathrm{CPS}}_{\mathrm{bt}^\star}(\widehat{C}_{\mathrm{bt}^\star}, \widehat{\rho}_{\mathrm{bt}^\star}) \vDash^{p'}_{\mathrm{bt}^\star} p'$ holds. We can also define the reverse binding-time transformation $\Psi^{\mathrm{CPS}}_{\mathrm{bt}^\star}$, which is essentially the same as the reverse flow transformation of Section 2.6.3 and also operates in linear time: for each term we just extract the binding time of its CPS counterpart. We can show that given a solution $(\widehat{C}'_{\mathrm{bt}^\star}, \widehat{\rho}'_{\mathrm{bt}^\star})$ of BTA* for p' (i.e., such that $(\widehat{C}'_{\mathrm{bt}^\star}, \widehat{\rho}'_{\mathrm{bt}^\star}) \vDash^{p'}_{\mathrm{bt}^\star} p'$ holds), $\Psi^{\mathrm{CPS}}_{\mathrm{bt}^\star}(\widehat{C}'_{\mathrm{bt}^\star}, \widehat{\rho}'_{\mathrm{bt}^\star}) \vDash^{p}_{\mathrm{bt}^\star} p$ holds too.

We are now in position to connect the binding times in direct style and in CPS as obtained by BTA*:

Theorem 2.7.5. *Given a direct-style program p and its CPS counterpart $p' = [\![p]\!]^{Pgm}$, let $(\widehat{C}_{\mathrm{bt}^\star}, \widehat{\rho}_{\mathrm{bt}^\star})$ be the least solution of BTA* for p and let $(\widehat{C}'_{\mathrm{bt}^\star}, \widehat{\rho}'_{\mathrm{bt}^\star})$ be the least solution of BTA* for p'. Then for all variables x in p, $\widehat{\rho}_{\mathrm{bt}^\star}(x) = \widehat{\rho}'_{\mathrm{bt}^\star}(x)$ and for all trivial terms t^ℓ in p, $\widehat{C}_{\mathrm{bt}^\star}(\ell) = \widehat{C}'_{\mathrm{bt}^\star}(\ell)$.*

We thus conclude that introducing continuations has no effect on the amount of specialization that can be performed when using continuation-based partial evaluation.

2.7.5 Summary and conclusions

We have shown that, given an input program as a call-by-value encoding of a Λ-program, introducing continuations does not degrade and may improve the results of the BTA for traditional partial-evaluation. We have also shown that introducing continuations does not affect the results of the BTA for continuation-based partial evaluation.

We therefore conclude that, unless one is willing to use continuation-based partial evaluation, a complete CPS transformation of the program is beneficial to the quality of the results of the BTA.

2.8 Related work

2.8.1 Program analysis in general

Even though the issue of syntactic accidents is not treated in textbooks and tutorials on program analysis, it appears to be folklore in the program-analysis community. An outstanding recent example is region inference (Section 2.8.1.1). To some extent, a similar situation occurs in programming practice: who has never modified a program with the sole purpose of improving its performance?

We are only aware of three other studies of the effect of continuations on program analysis: an early work by Nielson [87], Sabry and Felleisen's PLDI'94 paper [106], and a recent unpublished work by Palsberg and Wand [97].

Nielson's work compares the precision of two data-flow analyses: one based on a direct-style semantics and the other on a continuation semantics. In contrast, we compare the precision of the (same) analysis of a program and of its CPS counterpart. Sabry and Felleisen's work shows that a CPS transformation leads to incomparable results for a constant propagation analysis (Section 2.8.1.2). Palsberg and Wand's work is similar to ours since it involves a CPS transformation of flow information (Section 2.8.1.3).

2.8.1.1 Region inference and the CPS transformation

Region inference [117] aims at detecting program points where run-time storage can be deallocated—typically at exit points for blocks and at return points for functions. To overcome syntactic accidents, a programming discipline has therefore been developed to make region inference yield better results.

We note that region improvements and binding-time improvements may come at cross purpose. For example, consider let reassociation:

$$\begin{array}{l}\textbf{let } x_2 = \textbf{let } x_1 = e^{\ell_1} \\ \qquad\quad \textbf{in } e^{\ell_2} \\ \textbf{in } e^{\ell_3}\end{array} \quad \underset{\text{let ``deepening''}}{\overset{\text{let flattening}}{\rightleftarrows}} \quad \begin{array}{l}\textbf{let } x_1 = e^{\ell_1} \\ \textbf{in let } x_2 = e^{\ell_2} \\ \qquad \textbf{in } e^{\ell_3}\end{array}$$

Let flattening allows the region for x_1 to be released after the region for x_2. Let deepening allows the region for x_1 to be released earlier and requires the region for x_2 to be allocated earlier. Therefore, let deepening provides a region improvement, especially if e^{ℓ_3} contains a recursive call. But on the other hand, and as pointed out by an anonymous reviewer, if e^{ℓ_1} contains a recursive call, it is let flattening that provides a region improvement. Similarly, for functions, the CPS transformation yields a binding-time improvement whereas the direct-style transformation yields a region improvement (since in CPS, functions "never return").

2.8.1.2 Data-flow analysis and the CPS transformation

In their PLDI'94 paper [106], Sabry and Felleisen have shown that after a CPS transformation, a data-flow analysis may confuse the continuations used at return points. An example of confusion of return points is given by the term

$$\begin{array}{l}\textbf{let } x_1 = f\ 1\\ \textbf{in let } x_2 = f\ 2\\ \quad \textbf{in } x_1\end{array}$$

and its CPS counterpart

$$\lambda k.f\ 1\ (\lambda^{\pi_1} x_1.f\ 2\ (\lambda^{\pi_2} x_2.k\ x_1))$$

analyzed in contexts where f is bound to $\lambda x.x$ and to its CPS counterpart $\lambda x.\lambda k_1.k_1\ x$, respectively. The analysis of the direct-style term starts by examining the first application and detects that x and afterwards x_1 evaluate to the constant 1. Then, by analyzing the second application, the analysis approximates that the value of x is not constant (it can evaluate to both 1 and 2). The value of x_2 is also considered unknown. Nevertheless, x_1 is still considered constant, and the analysis is able to deduce that the whole expression evaluates to the constant 1.

In the CPS program, the analysis of the first application determines that the continuation k_1 evaluates to π_1, and, afterwards, that x_1 evaluates to 1. After the analysis of the second application, the continuation k_1 evaluates to both π_1 and π_2. The variable x evaluates to both 1 and 2 and is approximated as unknown. The approximation is passed by the application $k_1\ x$, into both x_1 and x_2. Therefore, a loss of precision occurs: the result of the whole expression is no longer detected as being a constant.

One can observe, however, that in a constant-propagation analysis the chronological order of the two applications may affect the result. In direct style, the first application of the function f is analyzed in a different context than the second application. Switching the order of the two applications leads to a different result of the analysis, for an essentially equivalent program. Therefore a limited form of context dependency is built in the constant-propagation analysis that Sabry and Felleisen considered. In contrast, the constraint-based analyses (in the monovariant case) propagate the result of a function at once to all the

application sites of this function. These analyses do not exhibit the sequentiality dependency of the constant propagation, and therefore, no precision is lost after a source CPS transformation.

Sabry and Felleisen also present examples where the analysis of a program is improved after the CPS transformation, reflecting that the constant-propagation analysis is not distributive [72, 87]. The improvements are attributed to the fact that the constant-propagation analysis is duplicated over conditional branches (and their corresponding continuations). In contrast, the constraint-based analyses propagate results from one branch of a conditional to another, and therefore, no precision is gained by the CPS transformation.

To summarize, Sabry and Felleisen's analysis depends on the order in which the source program is traversed and it is duplicated over conditional branches. These two properties led Sabry and Felleisen to conclude that the CPS transformation does not preserve the result of constant propagation. In contrast, our monovariant constraint-based analyses do not depend on the order in which constraints are solved and the analyses are not duplicated over conditional branches. These two properties led us to conclude that the CPS transformation does preserve the results of CFA and of BTA*.

2.8.1.3 CPS transformation of flow information

Recently, Palsberg and Wand have conducted a study of CFA [97], supporting Sabry and Felleisen's conclusion that the extra precision enabled by the CPS transformation is due to the duplication of the analysis. They developed a CPS transformation of flow information comparable to the one of Figure 2.16, but independently and prior to us. Palsberg and Wand also mention that least solutions may or may not be preserved by administrative reductions of CPS-transformed programs. In that, they implicitly share our concern about syntactic accidents, even though their primary goal was to transfer Wand's pioneer results on the CPS transformation of types [77, 119] to the CPS transformation of flow types.

2.8.2 Binding-time analysis and the CPS transformation

Binding-time improvements have always been customary for users of binding-time analysis [66, 88]. One of them amounts to considering source programs in CPS [21, 29], which suggests that source programs should be systematically CPS-transformed [20]. (Muylaert-Filho and Burn take the same stand for strictness analysis and the call-by-name CPS transformation [84].)

Essentially, the CPS transformation relocates potentially static contexts inside definitely dynamic contexts (let expressions and conditionals), thereby providing a binding-time improvement. To this end, the CPS transformation itself is continuation-based [30], which paved the way to continuation-based partial evaluation [13, 76].

Hatcliff and Danvy have characterized the full effect of continuation-based partial evaluation as online let flattening in Moggi's computational metalan-

guage [50]. This characterization justifies why offline let flattening is also, partially, a binding-time improvement [58]. In any case, offline let flattening is known to be part of the CPS transformation [49].

What had not been shown before, however, and what we have addressed here, is whether such "improvements" worsen binding times elsewhere in a source program.

2.9 Conclusion and issues

Observing that program analyses are vulnerable to syntactic accidents, we have considered a radical syntactic change: a transformation into CPS. We have studied the interaction between a non-duplicating CPS transformation and two program analyses: control-flow analysis (CFA) and binding-time analysis. Through a systematic construction of the CPS counterpart of flow information, we have found that constraint-based CFA is insensitive to continuation-passing, and that the CPS transformation does improve binding times for traditional partial evaluation. Using the same technique, we have also found that the binding-time analysis for continuation-based partial evaluation is insensitive to the CPS transformation.

These results suggest two further avenues of study:

- In BTA, the beneficial effect of the CPS transformation can be accounted for by disabling the context coherence constraints for let expressions (and for conditionals as well, if one is willing to duplicate static contexts at specialization time). The price of this change, however, is that the corresponding program specializer has to be made continuation-based [50]. We conjecture that the situation is similar, e.g., for security analysis, which has similar let and case rules. Just like BTA, a security analysis thus ought to yield more precise results over CPS-transformed programs. We therefore also conjecture that the beneficial effect of the CPS transformation can be accounted for by disabling the context coherence constraints in the let and case rules, if one is willing to develop a corresponding continuation-based processor of security information.
- More generally, as a step towards more robust program analyses that are less vulnerable to syntactic accidents, we need to understand better the program-analysis perspective over syntactic landscapes. Two key questions arise which may be general to program analysis or specific to individual program analyses: which program transformations affect precision? And among those that do, which ones affect precision *monotonically*? Answering these questions would enable one to develop more reliable program analyses, possibly with some kind of subject-reduction property or with some kind of intermediate language for program analysis. Henglein's invariance properties of polymorphic typing judgments with respect to let unfolding and folding and η-reduction [57] is a step in this direction.

2.A Proofs

Proof of Theorem 2.6.1. The proof proceeds by induction on the transformation of p into p'. We sketch the induction steps.

We show that $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) \vDash^{p'}_{\mathrm{cf}} (\mathbf{let}\ x = k^{\ell_0}\ [\![t^{\ell}]\!]^{Triv}\ \mathbf{in}\ x^{\ell_1})^{\ell_2}$ holds. For an arbitrary continuation $\lambda^{\pi} y.e^{\ell_3}$ in the set $\widehat{C}'_{\mathrm{cf}}(\ell_0) = \widehat{\rho}'_{\mathrm{cf}}(k)$, we show that two flow constraints are satisfied.

The first constraint is $\widehat{C}'_{\mathrm{cf}}(\ell) \subseteq \widehat{\rho}'_{\mathrm{cf}}(y)$. By Lemma 2.6.2, $\widehat{C}'_{\mathrm{cf}}(\ell) = \widehat{C}_{\mathrm{cf}}(\ell)$. We make a case analysis on the introduction of k by the CPS transformation.

If k is the top-level continuation, then the constraints are vacuously satisfied. If k is introduced by the transformation of a named conditional, then ℓ is the return point of one of the two branches of the test. Obviously $\widehat{C}_{\mathrm{cf}}(\ell) \subseteq \widehat{\rho}'_{\mathrm{cf}}(y)$. Otherwise, k comes from the transformation of a λ-abstraction $\lambda^{\pi_1} x_1.e^{\ell_4}$ from p, such that $\ell = \mathcal{R}[\![e^{\ell_4}]\!]$. We apply Lemma 2.6.4.

The second constraint is $\widehat{C}'_{\mathrm{cf}}(\ell_3) \subseteq \widehat{\rho}'_{\mathrm{cf}}(x)$. Following Lemma 2.6.2, it amounts to $\emptyset \subseteq \emptyset$.

For the rest of the induction steps, the induction hypotheses and the definition of γ suffice to show that the constraints are satisfied. □

Proof of Theorem 2.6.5. The proof is by induction on the transformation of p into p'. We sketch the induction steps.

For the transformation step $[\![t^{\ell}]\!]^{Triv}$, the constraints follow from the induction hypothesis. The same applies for the transformation step $[\![t^{\ell}]\!]^{Exp} k$.

For the transformation of a named application:

$$[\![\mathbf{let}\ x = t_0^{\ell_3}\ t_1\ \mathbf{in}\ e_2]\!]^{Exp} k = \begin{array}{l} \mathbf{let}\ x_0 = [\![t_0^{\ell_3}]\!]^{Triv}\ [\![t_1]\!]^{Triv} \\ \mathbf{in\ let}\ x_1 = x_0\ \lambda^{\pi} x.e^{\ell_2}\ \mathbf{in}\ x_1 \end{array}$$

let $\lambda^{\pi_1} y.e_1^{\ell_4}$ be an arbitrary λ-abstraction from p such that $\pi_1 \in \widehat{C}_{\mathrm{cf}}(\ell_3)$. Let the CPS transformation of the λ-abstraction be $\lambda^{\pi_1} y.\lambda k_1.e_2$. Then $\pi \in \widehat{\rho}'_{\mathrm{cf}}(k_1)$. From Lemma 2.6.7 and Lemma 2.6.8 we obtain that $\widehat{C}_{\mathrm{cf}}(\ell_4) \subseteq \widehat{\rho}_{\mathrm{cf}}(x)$. □

Proof of Theorem 2.7.1. The proof is an adaptation of the proof of Theorem 2.6.1 to equality constraints. In addition, we need to prove the satisfaction of the additional constraints introduced by BTA. We sketch the induction steps.

We show that $(\widehat{C}'_{\mathrm{cf}}, \widehat{\rho}'_{\mathrm{cf}}) \vDash^{p'}_{\mathrm{cf}} (\mathbf{let}\ x = k^{\ell_0}\ [\![t^{\ell}]\!]^{Triv}\ \mathbf{in}\ x^{\ell_1})^{\ell_2}$ holds. For this purpose, given an arbitrary $\lambda^{\pi} x.e^{\ell_3} \in \widehat{C}'_{\mathrm{cf}}(\ell_0) = \widehat{\rho}'_{\mathrm{cf}}(k)$ we must show that two equality constraints are satisfied. Similarly to the proof of Theorem 2.6.10, we make a case analysis on the introduction of k, using Lemma 2.7.3 and Lemma 2.7.4 to prove the satisfaction of the constraints.

We also need to show that $\widehat{C}'_{\mathrm{bt}}(\ell_0) = \mathbf{D} \Rightarrow \widehat{C}'_{\mathrm{bt}}(\ell) = \mathbf{D}$. Again, we make a case analysis on the introduction of k. The top-level case is trivial. The case where k is introduced by the transformation of a function $(\lambda y.e_1^{\ell_5})^{\ell_4}$ implies that $\widehat{C}_{\mathrm{bt}}(\ell_4) = \mathbf{D}$. Thus $\widehat{C}_{\mathrm{bt}}(\ell_5) = \mathbf{D}$ and then $\widehat{C}'_{\mathrm{bt}}(\ell) = \mathbf{D}$, since $\ell = \mathcal{R}[\![e_1^{\ell_5}]\!]$. The same reasoning follows for the case where k comes from the transformation of a named conditional.

The remaining cases follow directly from the induction hypotheses and the definition of $\widehat{C}'_{\mathrm{bt}}$, $\widehat{\rho}'_{\mathrm{bt}}$, $\widehat{C}_{\mathrm{cf}}$ and γ. □

Chapter 3

CPS Transformation of Flow Information, Part II: Administrative Reductions

Abstract[1]

We characterize the impact of a linear β-reduction on the result of a control-flow analysis. (By "a linear β-reduction" we mean the β-reduction of a linear λ-abstraction, i.e., of a λ-abstraction whose parameter occurs exactly once in its body.)

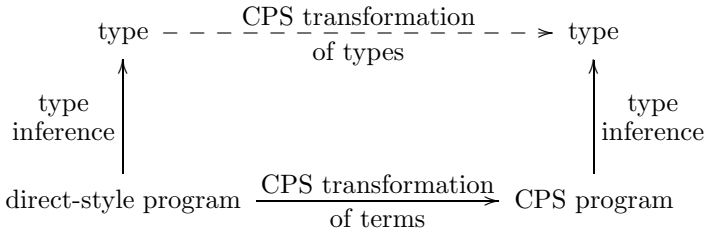
As a corollary, we consider the administrative reductions of a Plotkin-style transformation into continuation-passing style (CPS), and how they affect the result of a constraint-based control-flow analysis and in particular the least element in the space of solutions. We show that administrative reductions preserve the least solution. Since we know how to construct least solutions, preservation of least solutions solves a problem that was left open in Palsberg and Wand's paper "CPS Transformation of Flow Information."

Therefore, together, Palsberg and Wand's article "CPS Transformation of Flow Information" and the present article show how to map, in linear time, the least solution of the flow constraints of a program into the least solution of the flow constraints of the CPS counterpart of this program, after administrative reductions. Furthermore, we show how to CPS transform control-flow information in one pass.

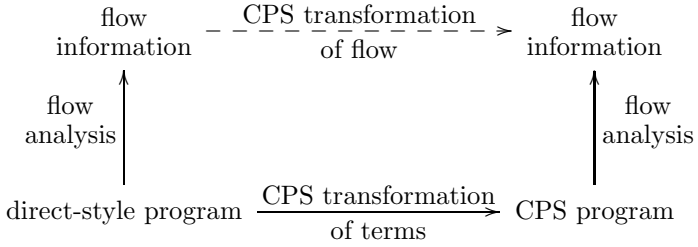
[1] This chapter is joint work with Olivier Danvy.

3.1 Background and introduction

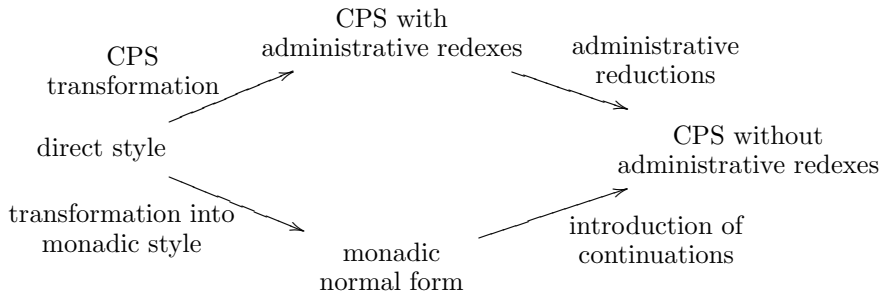
Since their inception, over thirty years ago [102], continuations and the transformation into continuation-passing style (CPS) have been the topic of much study, ranging from semantics and logic to implementations of sequential, concurrent, and distributed programming languages and systems. Fifteen years ago [77, 119], Meyer and Wand noticed that the CPS transformation preserves types and constructed a CPS transformation of types.



Over the last couple of years, Palsberg and Wand have extended this observation to flow types and the flow information gathered by a control-flow analysis [97], designing a CPS transformation of flow information.



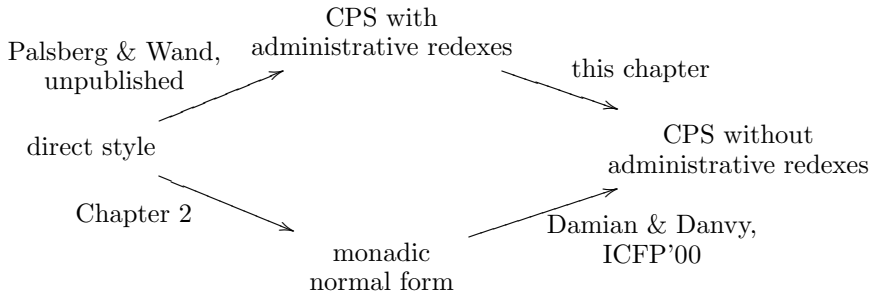
Independently, and with a different motivation, we have also designed a CPS transformation of flow information for control flow and binding times [26]. The two CPS transformations of flow information correspond to two different takes on the CPS transformation of λ-terms:



The CPS transformation is Plotkin's [98]. It is a first-order, compositional rewriting system generating numerous administrative redexes that need to be

post-reduced in practice [113]. Alternatively [49, 107], the CPS transformation can be staged into a transformation into monadic normal form followed by an introduction of continuations.

The two CPS transformations of flow information can be depicted as follows.



Palsberg and Wand show how to construct, in linear time, the flow information corresponding to a CPS program obtained through a Plotkin-style CPS transformation [97, 98]. The resulting programs contain all administrative redexes induced by Plotkin's transformation. Therefore, the corresponding CPS information of flow also contains spurious information which accounts for the extraneous λ-abstractions and their flow. The problem of eliminating this spurious information is open.

Damian and Danvy show how to construct, in linear time, the flow information corresponding to the introduction of continuations, starting from monadic normal forms [26, 49]. Damian shows how to construct, in linear time, the flow information corresponding to the transformation into monadic normal forms (as done in Chapter 2).

In this work, we complete the picture above by showing how to perform, in linear time, administrative reductions on CPS transformed programs (Section 3.4). Our result hinges on linear reductions (Section 3.3). But first, we present the source language and a constraint-based control-flow analysis (Section 3.2).

3.2 Preliminaries

3.2.1 The source language

We consider that input terms are given by the grammar in Figure 3.1. Terms are annotated with distinct labels taken from a countable set *Lab*. Each λ-abstraction is annotated with a distinct label π from a set *Lam*, and we consider that there exists a bijection between λ-abstractions and their labels.

We consider that the language has a standard call-by-value semantics, which is left unspecified. A program p is a closed labeled expression e^{ℓ}.

Definition 7. *A* properly labeled *expression is a labeled expression in which all labels are distinct and all variables are distinct.*

$$
\begin{array}{rcll}
e & \in & \mathit{Exp} & ::= \quad x \mid n \mid e_1^{\ell_1}\ e_2^{\ell_2} \mid \mathbf{if0}\ e^{\ell}\ e_0^{\ell_0}\ e_1^{\ell_1} \mid \lambda^{\pi} x.e^{\ell} \\
\pi & \in & \mathit{Lam} & \quad (\lambda\text{-abstraction labels}) \\
\ell & \in & \mathit{Lab} & \quad (\text{term labels}) \\
n & \in & \mathit{Lit} & \quad (\text{integer literals})
\end{array}
$$

Figure 3.1: The language of labeled λ-terms

$$
\begin{array}{rl}
\mathit{Lam}^p & \text{The set of } \lambda\text{-abstraction labels in } p \\
\mathit{Var}^p & \text{The set of identifiers in } p \\
\mathit{Lab}^p & \text{The set of term labels in } p
\end{array}
$$

$$
\begin{array}{rrll}
 & \mathit{Val}^p & = \mathcal{P}(\mathit{Lam}^p) & \text{Abstract values} \\
\widehat{C} \in & \mathit{Cache}_p & = \mathit{Lab}^p \to \mathit{Val}^p & \text{Abstract cache} \\
\widehat{\rho} \in & \mathit{Env}_p & = \mathit{Var}^p \to \mathit{Val}^p & \text{Abstract environment}
\end{array}
$$

$$\vDash_p \subseteq (\mathit{Cache}_p \times \mathit{Env}_p) \times \mathit{Lab}^p$$

Figure 3.2: Control-flow analysis relation for a program p: functionality

3.2.2 Control-flow analysis

We consider a constraint-based control-flow analysis. We use the same notations and definitions as in Nielson, Nielson and Hankin's recent textbook on program analysis [90].

Given a program p, the control-flow analysis is defined as a relation $\vDash_p$ whose functionality is displayed in Figure 3.2.

A solution of the analysis of p is a pair $(\widehat{C}, \widehat{\rho})$ such that $(\widehat{C}, \widehat{\rho}) \vDash p$. The set of solutions of the analysis is ordered by the natural pointwise ordering of functions, and has a least element. This property ensures the existence of a least solution of the analysis of p. The analysis relation is defined inductively on the syntax as defined in Figure 3.3.

3.3 Linear reductions

We observed that linear reductions preserve flow information. A linear reduction is a β-reduction in which the λ-abstraction in the function position is linear, i.e., such that its argument occurs free once and only once inside the body. Let us formalize the notion of linear reduction using linear contexts.

Definition 8. *A* linear context *is a labeled expression with a unique hole* $[\cdot]$.

$$
\begin{array}{lcl}
(\widehat{C},\widehat{\rho}) \vDash_p n^\ell & \iff & \mathit{true} \\
(\widehat{C},\widehat{\rho}) \vDash_p x^\ell & \iff & \widehat{\rho}(x) \sqsubseteq \widehat{C}(\ell) \\
(\widehat{C},\widehat{\rho}) \vDash_p (\lambda^\pi x.e^\ell)^{\ell_1} & \iff & \pi \in \widehat{C}(\ell_1) \wedge (\widehat{C},\widehat{\rho}) \vDash_p e^\ell \\
(\widehat{C},\widehat{\rho}) \vDash_p (e_1^{\ell_1}\ e_2^{\ell_2})^\ell & \iff & (\widehat{C},\widehat{\rho}) \vDash_p e_1^{\ell_1} \wedge (\widehat{C},\widehat{\rho}) \vDash_p e_2^{\ell_2} \wedge \\
 & & \forall \lambda^\pi x.e_0^{\ell_0} \in \widehat{C}(\ell_1).\widehat{C}(\ell_2) \subseteq \widehat{\rho}(x) \wedge \\
 & & \qquad \widehat{C}(\ell_0) \subseteq \widehat{C}(\ell) \\
(\widehat{C},\widehat{\rho}) \vDash_p (\mathbf{if0}\ e^\ell\ e_0^{\ell_0}\ e_1^{\ell_1})^{\ell_2} & \iff & (\widehat{C},\widehat{\rho}) \vDash_p e^\ell \wedge (\widehat{C},\widehat{\rho}) \vDash_p e_0^{\ell_0} \wedge (\widehat{C},\widehat{\rho}) \vDash_p e_1^{\ell_1} \wedge \\
 & & \widehat{C}(\ell_0) \subseteq \widehat{C}(\ell_2) \wedge \widehat{C}(\ell_1) \subseteq \widehat{C}(\ell_2)
\end{array}
$$

Figure 3.3: Control-flow analysis relation for a program p: definition

Linear contexts are defined by the grammar:

$$
\begin{array}{rcl}
E & ::= & [\cdot] \mid x^\ell \mid n^\ell \mid (E\ e_2^{\ell_2})^\ell \mid (e_1^{\ell_1}\ E)^\ell \mid \\
 & & (\mathbf{if0}\ E\ e_0^{\ell_0}\ e_1^{\ell_1})^\ell \mid (\mathbf{if0}\ e^\ell\ E\ e_1^{\ell_1})^{\ell_0} \mid (\mathbf{if0}\ e^\ell\ e_0^{\ell_0}\ E)^{\ell_1} \mid \\
 & & (\lambda^\pi x.E)^\ell
\end{array}
$$

Given a linear context E and a labeled expression e^ℓ, we use $E[e^\ell]$ to denote the context E with the hole $[\cdot]$ replaced with e^ℓ. It is trivial to see that $E[e^\ell]$ is a well-formed expression.

We also use $FV(e)$ to denote the set of free variables of the expression e. This notation naturally extends to contexts: given the context E, by definition $FV(E) = FV(E[n])$, where n is an arbitrary literal. We use L as the function extracting the label of an expression. By definition, for any labeled expression e^ℓ, $L(e^\ell) = \ell$.

Definition 9. *A λ-abstraction $\lambda^\pi x.e^\ell$ is* linear *if and only if it is properly labeled and e^ℓ contains a unique free occurrence of x, i.e., if there exists a linear context E such that $x \notin FV(E)$ and $e = E[x^{\ell_1}]$ for some label ℓ_1.*

Definition 10. *A* linear redex *is a β-redex $(\lambda x.e_1)\ e_2$ such that $\lambda x.e_1$ is a linear λ-abstraction.*

Definition 11. *A* linear reduction *is the β-reduction of a linear redex.*

Example:

$$((\lambda^\pi x.E[x^\ell])^{\ell_1}\ e^{\ell_2})^{\ell_3} \to E[e^{\ell_2}]$$

where E is a linear context where x does not occur free. Note that such a reduction might not necessarily be sound with respect to call-by-value semantics. Nevertheless, we show that it does not affect the result of the monovariant control-flow analysis. In any case, we treat linear reductions in CPS, which is evaluation-order independent [98].

3.4 Control-flow analysis and linear reduction

We show that performing a linear reduction does not alter the results of the analysis of a properly labeled program. More precisely, we show that, given a properly labeled program which contains a linear β-redex, control-flow analysis yields strictly equivalent results before and after performing a linear β-reduction.

We are given a program that contains a linear redex and the least solution of its analysis. The goal of this section is to construct the least solution of the analysis of this program after a linear β-reduction.

Let p be a properly labeled program containing a linear β-redex. Therefore, there exists two linear contexts E and E_1, an expression e, a fresh variable x, and labels $\pi, \ell_0, \ell_1, \ell_2$ and ℓ_3 such that

$$p = E[((\lambda^{\pi} x.E_1[x^{\ell_0}])^{\ell_1}\ e^{\ell_2})^{\ell_3}]$$

and $x \notin FV(E)$. Let then

$$p' = E[E_1[e^{\ell_2}]]$$

be the program p with the linear redex above reduced. It is immediate to see that p' is also a properly labeled program.

In the rest of this section, we define a monotone function $\mathcal{F}_p$ which, given a solution of the analysis of p, constructs a solution of the analysis of p'. We then define a reverse function $\mathcal{G}_p$, monotone as well, which, given a solution of the analysis of p', constructs a solution of the analysis of p. Using the two functions and their monotonicity, we show that the best solution for p is transformed into the best solution for p'. We then show how to construct, in linear time, the least solution of the analysis of p' from the least solution of the analysis of p.

3.4.1 Flow constructions

For the programs p and p' defined as above, by construction,

- $Lab^p = Lab^{p'} \cup \{\ell_0, \ell_1, \ell_3\}$,
- $Lam^p = Lam^{p'} \cup \{\pi\}$, and
- $Var^p = Var^{p'} \cup \{x\}$.

We define a function $\mathcal{F}_p : (Cache_p \times Env_p) \to (Cache_{p'} \times Env_{p'})$ as $\mathcal{F}_p(\widehat{C}, \widehat{\rho}) = (\widehat{C}|_{Lam^{p'}}, \widehat{\rho}|_{Lam^{p'}})$. Obviously, $\mathcal{F}_p$ is a projection function and it is monotone with respect to the ordering of solutions.

We define a reverse function $\mathcal{G}_p : (Cache_{p'} \times Env_{p'}) \to (Cache_p \times Env_p)$ as follows. If $\mathcal{G}_p(\widehat{C}', \widehat{\rho}') = (\widehat{C}, \widehat{\rho})$ then:

- for all $\ell \in Lab^{p'}$, $\widehat{C}(\ell) = \widehat{C}'(\ell)$; $\widehat{C}(\ell_3) = \widehat{C}'(L(E_1[e^{\ell_2}]))$; $\widehat{C}(\ell_0) = \widehat{\rho}(x) = \widehat{C}'(\ell_2)$; $\widehat{C}(\ell_1) = \{\pi\}$; and
- for all $y \in Var^{p'}$, $\widehat{\rho}(y) = \widehat{\rho}'(y)$.

Obviously, $\mathcal{G}_p$ is an embedding function and it is monotone as well.

Lemma 3.4.1. *Let* $(\widehat{C},\widehat{\rho}) \in (Cache_p \times Env_p)$ *such that* $(\widehat{C},\widehat{\rho}) \vDash_p p$. *Then* $\mathcal{F}_p(\widehat{C},\widehat{\rho}) \vDash_{p'} p'$.

Proof. Let $(\widehat{C}',\widehat{\rho}') = \mathcal{F}_p(\widehat{C},\widehat{\rho})$. We show that $(\widehat{C}',\widehat{\rho}') \vDash_{p'} p'$. The proof has two steps:

i) A proof of the fact that $(\widehat{C}',\widehat{\rho}') \vDash_{p'} E_1[e^{\ell_2}]$. The proof is by structural induction on the context E_1, using the assumption that $(\widehat{C},\widehat{\rho}) \vDash_p E_1[x^{\ell_0}]$.

ii) A proof of the fact that $(\widehat{C}',\widehat{\rho}') \vDash_{p'} E[E_1[e^{\ell_2}]]$. The proof is by structural induction on the context E.

□

Lemma 3.4.2. *Let* $(\widehat{C}',\widehat{\rho}') \in (Cache_{p'} \times Env_{p'})$ *such that* $(\widehat{C}',\widehat{\rho}') \vDash_{p'} p'$. *Then* $\mathcal{G}_p(\widehat{C}',\widehat{\rho}') \vDash_p p$.

Proof. Let $(\widehat{C},\widehat{\rho}) = \mathcal{G}_p(\widehat{C}',\widehat{\rho}')$. We show that $(\widehat{C},\widehat{\rho}) \vDash_p p$. The proof has two steps:

i) A proof of the fact that $(\widehat{C},\widehat{\rho}) \vDash_p E_1[x^{\ell_0}]$. The proof is by structural induction on the context E_1, using the assumption that $(\widehat{C}',\widehat{\rho}') \vDash_{p'} E_1[e^{\ell_2}]$.

ii) A proof of the fact that $(\widehat{C},\widehat{\rho}) \vDash_p ((\lambda^\pi x.E_1[x^{\ell_0}])^{\ell_1}\ e^{\ell_2})^{\ell_3}$. Using i), the proof amounts to showing that a small set of constraints are satisfied.

iii) A proof of the fact that $(\widehat{C},\widehat{\rho}) \vDash_p E[((\lambda^\pi x.E_1[x^{\ell_0}])^{\ell_1}\ e^{\ell_2})^{\ell_3}]$. The proof is by structural induction on the context E.

□

Lemma 3.4.3. *Let* $(\widehat{C},\widehat{\rho})$ *be the least solution of the analysis of* p. *Let* $(\widehat{C}',\widehat{\rho}')$ *be the least solution of the analysis of* p'. *Then* $\mathcal{F}_p(\widehat{C},\widehat{\rho}) = (\widehat{C}',\widehat{\rho}')$ *and* $\mathcal{G}_p(\widehat{C}',\widehat{\rho}') = (\widehat{C},\widehat{\rho})$.

Proof. We can immediately see that $\mathcal{F}_p(\mathcal{G}_p(\widehat{C}',\widehat{\rho}')) = (\widehat{C}',\widehat{\rho}')$ and that $\mathcal{G}_p(\mathcal{F}_p(\widehat{C},\widehat{\rho})) \sqsubseteq (\widehat{C},\widehat{\rho})$. Therefore, $\mathcal{G}_p$ and $\mathcal{F}_p$ form an embedding/projection pair. Using the monotonicity of the two functions, we obtain that $\mathcal{F}_p(\widehat{C},\widehat{\rho}) = (\widehat{C}',\widehat{\rho}')$ and $\mathcal{G}_p(\widehat{C}',\widehat{\rho}') = (\widehat{C},\widehat{\rho})$. □

3.4.2 CPS transformation of flow information and administrative reductions

Lemma 3.4.3 says that the least analysis after a linear β-reduction is a restriction of the least analysis of the initial term. From this, we can infer that any linear β-reduction does not alter the results of the CFA. We use this result to show that administrative reductions after Plotkin's CPS transformation do not change the result of the flow analysis.

Theorem 3.4.4. *Let p be a program, p_1 be its CPS counterpart without administrative reductions, and p_2 be its CPS counterpart after administrative reduction. Let $(\widehat{C}_1, \widehat{\rho}_1)$ be the least solution of the analysis of p_1. The least solution $(\widehat{C}_2, \widehat{\rho}_2)$ of the analysis of p_2 can be obtained in linear time from $(\widehat{C}_1, \widehat{\rho}_1)$, by restricting $(\widehat{C}_1, \widehat{\rho}_1)$ to the program points preserved by the administrative reductions.*

Proof. All administrative reductions are linear, and furthermore, administrative reduction is known to terminate [31]. We apply Lemma 3.4.3. □

Corollary 3.4.5. *Let p be a program, p_1 be its CPS counterpart without administrative reductions, and p_2 be its CPS counterpart after administrative reduction. Let $(\widehat{C}, \widehat{\rho})$ be the least solution of the analysis of p. The least solution $(\widehat{C}_2, \widehat{\rho}_2)$ of the analysis of p_2 can be obtained in linear time from $(\widehat{C}, \widehat{\rho})$.*

Proof. We compose the construction given by Theorem 3.4.4 with Palsberg and Wand's CPS transformation of flow information [97], which also works in linear time. □

3.5 Conclusion and issues

We have shown how to complement Palsberg and Wand's CPS transformation of flow information with administrative reductions, while preserving its linear-time complexity. Our extension hinges on the linearity of administrative redexes.

Let us now show how to integrate administrative reductions in Palsberg and Wand's CPS transformation, therefore making it operate in one pass, still in linear time. As shown in "Representing Control" [31], at CPS-transformation time, one can segregate the administrative lambdas and applications and the residual ones. (The residual lambdas and applications are the ones preserved by the administrative reductions.) Therefore, in Palsberg and Wand's CPS transformation of flow information, we can segregate the labels of the administrative lambdas and applications and the labels of the residual ones as well. In practice, the solution after administrative reduction is thus obtained simply by restricting Palsberg and Wand's solution to the residual labels. In the overall process of (1) CPS transformation and (2) administrative reduction, the administrative labels are used transitorily, just as in the one-pass CPS transformation, which is conceptually fitting.

Chapter 4

A Simple CPS Transformation of Control-Flow Information

Abstract[1]

We show how to compute control-flow information for CPS-transformed programs from control-flow information for direct-style programs and vice-versa. As a corollary, we obtain a known result that CPS transformation has no effect on the control-flow information obtained by constraint-based control-flow analysis.

Compared to previous work by Palsberg and Wand, we compute control-flow information for CPS programs after administrative reductions. Compared to previous work by Damian and Danvy, our input programs need not be in monadic normal form. We use Danvy and Nielsen's recent compositional CPS transformation. The CPS transformation of control flow is therefore simpler. The transformation has immediate applications in assessing the effect of the CPS transformation over other analyses as, for instance, binding-time analysis.

4.1 Introduction

The continuation-passing-style (CPS) transformation is a source-to-source program transformation of λ-terms which makes explicit the continuation [114] of each λ-expression. The call-by-value and call-by-name CPS transformations due to Plotkin [98] yield λ-terms which are independent on the order of evaluation. The CPS transformation has been extended to types [77, 119], which led to dis-

[1] This chapter is joint work with Olivier Danvy.

covering its logical content [48, 83]. Recently, Danvy and Nielsen have presented a new and simpler CPS transformation [34].

Recently, Damian and Danvy [26] have discovered a CPS transformation of control flow and used it to shown that a CPS transformation does not affect the control-flow information collected by a monovariant constraint-based control-flow analysis; a similar work has been done by Palsberg and Wand [97]. In this article, we build on Danvy and Nielsen's new and simpler CPS transformation and we present a new and simpler CPS transformation of control-flow information.

4.1.1 Formulating the CPS transformation

The CPS transformation has motivated a long line of research. Plotkin [98] observed that introducing continuations over λ-terms gives rise to large terms. A set of so-called administrative redexes is identified: a practically useful CPS transformation need not contain these redexes. Plotkin interleaves administrative and essential reductions: Steele [113] performs all administrative reductions immediately after the CPS transformation.

Administrative redexes may be avoided altogether by using a one-pass CPS transformation [3, 31, 120]. The one-pass CPS transformation is defined through a higher-order accumulator. This accumulator expects a term and yields a CPS transformed term.

Alternatively, a first-order CPS transformation based on evaluation contexts such as the one by Sabry and Felleisen [105] and by Sabry and Wadler [107] also yields administratively reduced terms. Compact CPS programs also can be obtained by bringing the source program into monadic normal form [49] and then introducing continuations.

Recently, Danvy and Nielsen [34] have discovered a first-order, compositional one-pass CPS transformation. The transformation directly yields terms without administrative redexes, without the need of an intermediate form.

4.1.2 Reasoning on a CPS transformation

A CPS transformation introducing administrative redexes complicates a simulation proof: Plotkin had to devise a so-called colon translation [98]. Administrative redexes are not a problem for proving a CPS transformation of types due to the subject-reduction property.

Palsberg and Wand's original CPS transformation of flow does not address administrative reductions. In turn, Damian and Danvy [26] explore a similar transformation by considering the introduction of continuations over terms in monadic normal form [49]. The transformation of λ-terms into monadic normal form is an additional step on which further reasoning needs to be projected, as in Chapter 2.

Reasoning on a higher-order, one-pass CPS transformation becomes even more difficult since a higher-order argument is necessary. At the same time, adapting the control-flow transformation to either Sabry and Felleisen's or Sabry

and Wadler's CPS transformations appears to be difficult due their non-compositional formulation. In the first case, evaluation contexts need to be characterized and control-flow preservation needs to be established. In the later case, the non-compositional definition of the transformation makes the flow of procedures unclear, even if a compositional reformulation is actually possible [34].

Indeed, proving predicates defined by structural induction on a CPS transformed term appears to be most natural to be done using a compositionally-defined CPS transformation. In this respect, Danvy and Nielsen's recent compositional, first-order, and one-pass CPS transformation [34] appears to be most suitable to prove results the preservation of the results obtained by control-flow analysis.

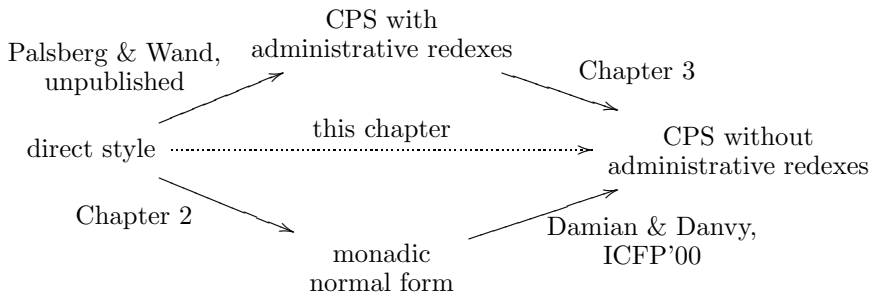
4.1.3 This work

In a previous work [26], the authors have considered only a step of the CPS transformation, namely the introduction of continuations on terms in monadic normal form [49]. Showing that reducing source terms to monadic normal form in turn does not affect the flow analysis requires additional reasoning, and the results may not be immediately predictable, as shown in Chapter 2.

In a related work [97], Palsberg and Wand have shown how to compute flow information for a CPS-transformed program. The work addresses CPS programs obtained through a Plotkin-style CPS transformation and does not addresses the issue of administrative reductions. For practical applications, performing administrative reductions requires further reasoning at the level of flow information.

In this work we show how to directly construct control-flow information for a CPS program after administrative reductions, without the need of an intermediate form. As a syntactic support, we use a recent compositional CPS transformation by Danvy and Nielsen [34]. Our construction confirms that the CPS transformation does not affect the result of a monovariant constraint-based control-flow analysis [26]. In the same time, it opens the way to investigating the effect of the CPS transformation on other analyses, as for instance, binding-time analysis.

Graphically:



Our CPS transformation of control flow is simpler than previous versions

and addresses the λ-calculus without the need of an intermediate form or administrative reductions. The proofs of correctness are similar to the ones in our earlier work [26], but here source terms need not be in monadic normal form. The proofs are slightly simpler than Palsberg and Wand's [97], due to the lack of administrative redexes.

4.2 Control-flow analysis for λ-terms

4.2.1 The language of λ-terms

We consider a language of labeled λ-terms, defined in Figure 4.1. Following Reynolds [103] and Moggi [82], we distinguish among trivial terms t which denote values and serious terms s which may denote computations. Expressions are annotated with distinct labels ℓ from a countable set Lab. Each λ-abstraction has a unique associated label π. A program p is a closed labeled expression e^{ℓ}.

4.2.2 Control-flow analysis

We consider a standard constraint-based control-flow analysis (CFA) on λ-terms [24, 47, 52, 56, 61, 89, 90, 95], specifically, we consider the CFA specified in Nielson, Nielson and Hankin's textbook [90]. Given an input program p, the functionality of the syntax-directed control-flow analysis relation $\models^p$ is defined in Figure 4.2. The analysis relation is defined inductively in Figure 4.3.

The relation is defined on a pair of a tuple $(\widehat{C}, \widehat{\rho})$ and a labeled expression e^{ℓ}. In the relation, $(\widehat{C}, \widehat{\rho})$ is a cache mapping each expression label to a set of λ-abstractions that the expression might evaluate to, while $\widehat{\rho}$ is an environment mapping each program variable to a set of λ-abstractions that the variable might denote. It is known [90, Chapter 3] that a pair $(\widehat{C}, \widehat{\rho})$ satisfying the relation $(\widehat{C}, \widehat{\rho}) \models^p p$ is a safe analysis of the program p.

Given a source program p, solutions of the analysis of p always exist. The set of solutions of the analysis of p is closed under intersection: the pointwise intersection of two solutions always exists. Therefore, there exists a least solution of the analysis of p. The least solution can be computed with a standard work-list based algorithm [90, Chapter 3]. Through the rest of this article we use "the result of the analysis of p" to refer to the least analysis result.

4.3 CPS transformation and control-flow analysis

We show that the CPS transformation preserves the result of the control-flow analysis defined in Section 4.2.2. To this end, we define a transformation from control-flow information for a direct-style program into control-flow information for the CPS counterpart of this program. We also define a transformation of

$e \in \mathit{Expr}$	(terms)	$e ::= s \mid t$
$s \in \mathit{Comp}$	(serious terms, i.e., computations)	$s ::= e_0^{\ell_0}\ e_1^{\ell_1}$
$t, K \in \mathit{Triv}$	(trivial terms, i.e., values)	$t ::= x \mid \lambda^{\pi} x.e^{\ell}$
$x \in \mathit{Ide}$	(identifiers)	
$\ell \in \mathit{Lab}$	(term labels)	
$\pi \in \mathit{Lam}$	(λ-abstraction labels)	

Figure 4.1: The language of labeled λ-terms

Lam^p	The set of λ-abstraction labels in p
Var^p	The set of identifiers in p
Lab^p	The set of term labels in p

$$\begin{array}{rrll} & \mathit{Triv}^p & = \mathcal{P}(\mathit{Lam}^p) & \text{Abstract values} \\ \widehat{C} \in & \mathit{Cache}^p & = \mathit{Lab}^p \to \mathit{Triv}^p & \text{Abstract cache} \\ \widehat{\rho} \in & \mathit{Env}^p & = \mathit{Var}^p \to \mathit{Triv}^p & \text{Abstract environment} \end{array}$$

$$\vDash^p \subseteq (\mathit{Cache}^p \times \mathit{Env}^p) \times \mathit{Lab}^p$$

Figure 4.2: Control-flow analysis relation for a program p

$$\begin{array}{lcl} (\widehat{C}, \widehat{\rho}) \vDash^p x^{\ell} & \iff & \widehat{\rho}(x) \subseteq \widehat{C}(\ell) \\ (\widehat{C}, \widehat{\rho}) \vDash^p (\lambda^{\pi} x.e^{\ell})^{\ell_1} & \iff & (\widehat{C}, \widehat{\rho}) \vDash^p e^{\ell} \wedge \pi \in \widehat{C}(\ell_1) \\ (\widehat{C}, \widehat{\rho}) \vDash^p (e_0^{\ell_0}\ e_1^{\ell_1})^{\ell_2} & \iff & (\widehat{C}, \widehat{\rho}) \vDash^p e_0^{\ell_0} \wedge (\widehat{C}, \widehat{\rho}) \vDash^p e_1^{\ell_1} \wedge \\ & & \forall \lambda^{\pi} x.e^{\ell} \in \widehat{C}(\ell_0).\widehat{C}(\ell_1) \subseteq \widehat{\rho}(x) \wedge \\ & & \qquad \widehat{C}(\ell) \subseteq \widehat{C}(\ell_2) \end{array}$$

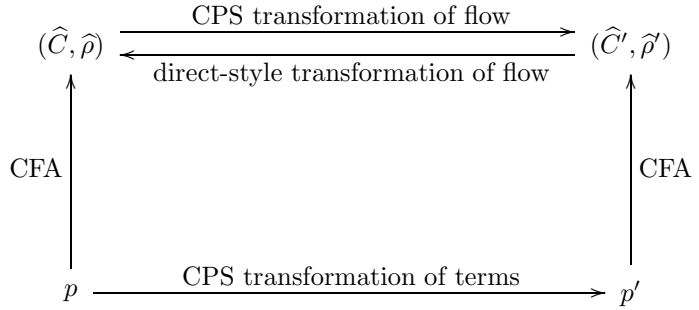
Figure 4.3: Control-flow analysis

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathcal{E} : \mathit{Expr} \times \mathit{Ide} \to \mathit{Comp} \times \mathit{Lab}} \\
\mathcal{E}[\![t]\!]k & = & k\ \mathcal{T}[\![t]\!] \\
\mathcal{E}[\![s]\!]k & = & \mathcal{S}[\![s]\!](\lambda x.k\ x) \\
 & & \\
\multicolumn{3}{l}{\mathcal{S} : \mathit{Comp} \times \mathit{Triv} \to \mathit{Comp}} \\
\mathcal{S}[\![t_0\ t_1]\!]K & = & \mathcal{T}[\![t_0]\!]\ \mathcal{T}[\![t_1]\!]\ K \\
\mathcal{S}[\![t_0\ s_1]\!]K & = & \mathcal{S}[\![s_1]\!](\lambda x_1.\mathcal{T}[\![t_0]\!]\ x_1\ K) \\
\mathcal{S}[\![s_0\ t_1]\!]K & = & \mathcal{S}[\![s_0]\!](\lambda x_0.x_0\ \mathcal{T}[\![t_1]\!]\ K) \\
\mathcal{S}[\![s_0\ s_1]\!]K & = & \mathcal{S}[\![s_0]\!](\lambda x_0.\mathcal{S}[\![s_1]\!](\lambda x_1.x_0\ x_1\ K)) \\
 & & \\
\multicolumn{3}{l}{\mathcal{T} : \mathit{Triv} \to \mathit{Triv}} \\
\mathcal{T}[\![x]\!] & = & x \\
\mathcal{T}[\![\lambda x.e]\!] & = & \lambda x.\lambda k.\mathcal{E}[\![e]\!]k
\end{array}
$$

Figure 4.4: First-order one-pass CPS transformation (labels omitted)

control-flow information for a CPS-transformed program into control-flow information for the direct-style counterpart of the program. Using the monotonicity of the two transformations, we show that the least analysis of a direct-style program is equivalent with the least analysis of its CPS counterpart and vice-versa.

Graphically:



4.3.1 CPS transformation

In this article CPS programs are obtained using Danvy and Nielsen's first-order CPS transformation [34]. The CPS transformation for (unlabeled) λ-terms is defined in Figure 4.4. As in our earlier work [26, 27], we consider a transformation with η-expanded tail calls: the continuation passed at a function call is always a syntactic λ-abstraction.

Danvy and Nielsen's one-pass CPS transformation yields CPS terms without administrative redexes. In Section 4.3.2, using this CPS transformation as a syntactic support, we are able to define the CPS transformation of control-flow information for CPS programs without administrative redexes.

$\mathcal{E} : \mathit{Expr} \times \mathit{Lab} \times \mathit{Ide} \to \mathit{Comp} \times \mathit{Lab}$

$$\begin{array}{rcl}
\mathcal{E}[\![t^\ell]\!]k & = & (k^{\ell_0}\ (\mathcal{T}[\![t]\!])^\ell)^{\ell_1} \qquad \widehat{C}'(\ell_0) = \widehat{\rho}'(k) \\
& & \qquad \widehat{C}'(\ell) = \widehat{C}'(\ell) \quad \widehat{C}'(\ell_1) = \emptyset \\
\mathcal{E}[\![s^\ell]\!]k & = & (\mathcal{S}[\![s]\!](\lambda^\pi x.(k^{\ell_0}\ x^{\ell_1})^{\ell_2})^{\ell_3})^{\ell_4} \\
& & \qquad \widehat{C}'(\ell_0) = \widehat{\rho}'(k) \quad \widehat{C}'(\ell_1) = \widehat{\rho}'(x) = \widehat{C}(\ell) \\
& & \qquad \widehat{C}'(\ell_3) = \{\pi\} \quad \widehat{C}'(\ell_4) = \widehat{C}'(\ell_2) = \emptyset
\end{array}$$

$\mathcal{S} : \mathit{Comp} \times \mathit{Triv} \times \mathit{Lab} \to \mathit{Comp}$

$$\begin{array}{rcl}
\mathcal{S}[\![t_0^{\ell_0}\ t_1^{\ell_1}]\!]K^\ell & = & ((\mathcal{T}[\![t_0]\!])^{\ell_0}\ (\mathcal{T}[\![t_1]\!])^{\ell_1})^{\ell_2}\ K^\ell \\
& & \qquad \widehat{C}'(\ell_0) = \widehat{C}(\ell_0) \quad \widehat{C}'(\ell_1) = \widehat{C}(\ell_1) \\
& & \qquad \widehat{C}'(\ell_2) = \gamma(\widehat{C}(\ell_0)) \\
\mathcal{S}[\![t_0^{\ell_0}\ s_1^{\ell_1}]\!]K^\ell & = & \mathcal{S}[\![s_1]\!](\lambda^\pi x_1.(((\mathcal{T}[\![t_0]\!])^{\ell_0}\ x_1^{\ell_1})^{\ell_2}\ K^\ell)^{\ell_3})^{\ell_4} \\
& & \qquad \widehat{C}'(\ell_0) = \widehat{C}(\ell_0) \quad \widehat{C}'(\ell_1) = \widehat{\rho}'(x_1) = \widehat{C}(\ell_1) \\
& & \qquad \widehat{C}'(\ell_2) = \gamma(\widehat{C}(\ell_0)) \\
& & \qquad \widehat{C}'(\ell_3) = \emptyset \quad \widehat{C}'(\ell_4) = \{\pi\} \\
\mathcal{S}[\![s_0^{\ell_0}\ t_1^{\ell_1}]\!]K^\ell & = & \mathcal{S}[\![s_0]\!](\lambda^\pi x_0.((x_0^{\ell_0}\ (\mathcal{T}[\![t_1]\!])^{\ell_1})^{\ell_2}\ K^\ell)^{\ell_3})^{\ell_4} \\
& & \qquad \widehat{C}'(\ell_0) = \widehat{\rho}'(x_0) = \widehat{C}(\ell_0) \quad \widehat{C}'(\ell_1) = \widehat{C}(\ell_1) \\
& & \qquad \widehat{C}'(\ell_2) = \gamma(\widehat{C}(\ell_0)) \\
& & \qquad \widehat{C}'(\ell_3) = \emptyset \quad \widehat{C}'(\ell_4) = \{\pi\} \\
\mathcal{S}[\![s_0^{\ell_0}\ s_1^{\ell_1}]\!]K^\ell & = & \mathcal{S}[\![s_0]\!](\lambda^\pi x_0.(\mathcal{S}[\![s_1]\!](\lambda^{\pi_1} x_1.((x_0^{\ell_0}\ x_1^{\ell_1})^{\ell_2}\ K^\ell)^{\ell_3})^{\ell_4})^{\ell_5})^{\ell_6} \\
& & \qquad \widehat{C}'(\ell_0) = \widehat{\rho}'(x_0) = \widehat{C}(\ell_0) \\
& & \qquad \widehat{C}'(\ell_1) = \widehat{\rho}'(x_1) = \widehat{C}(\ell_1) \quad \widehat{C}'(\ell_2) = \gamma(\widehat{C}(\ell_0)) \\
& & \qquad \widehat{C}'(\ell_3) = \emptyset \quad \widehat{C}'(\ell_4) = \{\pi_1\} \\
& & \qquad \widehat{C}'(\ell_5) = \emptyset \quad \widehat{C}'(\ell_6) = \{\pi\}
\end{array}$$

$\mathcal{T} : \mathit{Triv} \to \mathit{Triv}$

$$\begin{array}{rcl}
\mathcal{T}[\![x]\!] & = & x \\
\mathcal{T}[\![\lambda^\pi x.e^\ell]\!] & = & \lambda^\pi x.(\lambda^{\pi_1} k.\mathcal{E}[\![e^\ell]\!]k)^{\ell_0} \quad \widehat{C}'(\ell_0) = \{\pi_1\} \quad \widehat{\rho}'(k) = \xi(k)
\end{array}$$

Figure 4.5: Transformation of control flow from direct style to CPS

4.3.2 CPS transformation of control flow

We define a CPS transformation of control-flow information following the CPS transformation of Figure 4.4. We show how control-flow information for a direct-style term can be used to compute control-flow information for the CPS transformed program.

To transformation relies on two auxiliary functions:

- γ extracts the labels of partially applied CPS λ-abstractions. Formally, considering A to be a set of CPS λ-abstractions $\{\lambda^{\pi^i} x_i.\lambda^{\pi_1^i} k_i.e_i | 1 \leq i \leq n\}$, for some n, then $\gamma(A) = \{\pi_1^i | 1 \leq i \leq n\}$.
- ξ assigns flow information to each continuation identifier k introduced by

the CPS transformation of a λ-abstraction from p. This information can be obtained from the direct-style flow information, since we can syntactically identify the continuation of the CPS counterpart of any direct-style application.
Given p, $\widehat{C}$, $\widehat{\rho}$, and a continuation identifier k introduced by the transformation of a λ-abstraction from p:

$$\mathcal{T}[\![\lambda^{\pi} x.e^{\ell}]\!] = \lambda^{\pi} x.\lambda k.\mathcal{E}[\![e]\!]k$$

we define $\xi(k)$ as the union of all sets $\widehat{C}'(\ell)$ such that in the CPS transformation of p into p' there exists a transformation step

$$\mathcal{S}[\![e_0^{\ell_0}\ e_1^{\ell_1}]\!]K^{\ell} = \ldots$$

such that $\pi \in \widehat{C}(\ell_0)$.

We construct the CPS control-flow information in two steps. First, in a a recursive descent on the tree of the transformation, we compute $\widehat{C}'(\ell)$ for each label ℓ attached on the newly introduced λ-abstractions (continuations) and we construct the function ξ.

The second step consists of another recursive descent on the tree of the transformation. We assign control-flow information recursively, as defined for each step in Figure 4.5. At each transformation step, on the right-hand side, we construct the labeled CPS term corresponding to the left-hand side. We then assign flow information for each fresh label or variable. Trivial terms preserve their label and their flow information. Flow information for serious terms is transferred through calls to continuations. Fresh continuation identifiers are assigned flow information as computed by the ξ function.

Note that in contrast to the CPS transformation of unlabeled terms of Figure 4.4, the transformation of labeled serious terms takes an extra argument, namely the label of the syntactic continuation being passed as an argument. At each case in Figure 4.5, we do not make the label explicit: we rather place it directly over the constructed continuation. Similarly, the CPS transformation of a labeled expression returns a serious term and its enclosing label.

The CPS transformation of control flow is therefore defined as a monotone function:

$$\Phi_{\mathrm{cf}}^{\mathrm{CPS}} : (Cache^{p} \times Env^{p}) \rightarrow (Cache^{p'} \times Env^{p'}).$$

Theorem 4.3.1. *Let $p = e^{\ell}$ be a uniquely labeled program. If $(\widehat{C}, \widehat{\rho}) \vDash^{p} e^{\ell}$ then $(\Phi_{\mathrm{cf}}^{\mathrm{CPS}}(\widehat{C}, \widehat{\rho})) \vDash^{p'} \lambda^{\pi} k.\mathcal{E}[\![e^{\ell}]\!]k$.*

Proof. By structural induction on the resulting CPS program. We prove a predicate stating that a CPS-transformed serious term satisfies the relation when the term passed as a continuation is also satisfying the relation. □

The proof of the theorem is similar to the proof of Theorem 1 of our earlier work [26] and Theorem 2.6.1 of Chapter 2, which addressed terms in monadic normal forms. The proof is slightly less complex than Palsberg and Wand's proof [97], due to the lack of administrative redexes.

$$
\begin{array}{l}
\mathcal{E} : \mathit{Expr} \times \mathit{Ide} \to \mathit{Comp} \times \mathit{Lab} \\
\quad \mathcal{E}[\![t^{\ell}]\!]k = (k^{\ell_0}\ (\mathcal{T}[\![t]\!])^{\ell})^{\ell_1} \qquad \widehat{C}(\ell) = \widehat{C}'(\ell) \quad \widehat{C}(\ell_1) = \emptyset \\
\quad \mathcal{E}[\![s^{\ell}]\!]k = (\mathcal{S}[\![s]\!](\lambda^{\pi} x.(k^{\ell_0}\ x^{\ell_1})^{\ell_2})^{\ell_3})^{\ell_4} \qquad \widehat{C}(\ell) = \widehat{\rho}'(x) \\
\\
\mathcal{S} : \mathit{Comp} \times \mathit{Triv} \times \mathit{Lab} \to \mathit{Comp} \\
\quad \mathcal{S}[\![t_0^{\ell_0}\ t_1^{\ell_1}]\!]K^{\ell} = ((\mathcal{T}[\![t_0]\!])^{\ell_0}\ (\mathcal{T}[\![t_1]\!])^{\ell_1})^{\ell_2}\ K^{\ell} \\
\qquad\qquad \widehat{C}(\ell_0) = \widehat{C}'(\ell_0) \quad \widehat{C}(\ell_1) = \widehat{C}'(\ell_1) \\
\quad \mathcal{S}[\![t_0^{\ell_0}\ s_1^{\ell_1}]\!]K^{\ell} = \mathcal{S}[\![s_1]\!](\lambda^{\pi} x_1.(((\mathcal{T}[\![t_0]\!])^{\ell_0}\ x_1^{\ell_1})^{\ell_2}\ K^{\ell})^{\ell_3})^{\ell_4} \\
\qquad\qquad \widehat{C}(\ell_0) = \widehat{C}'(\ell_0) \quad \widehat{C}(\ell_1) = \widehat{\rho}'(x_1) \\
\quad \mathcal{S}[\![s_0^{\ell_0}\ t_1^{\ell_1}]\!]K^{\ell} = \mathcal{S}[\![s_0]\!](\lambda^{\pi} x_0.((x_0^{\ell_0}\ (\mathcal{T}[\![t_1]\!])^{\ell_1})^{\ell_2}\ K^{\ell})^{\ell_3})^{\ell_4} \\
\qquad\qquad \widehat{C}(\ell_0) = \widehat{\rho}'(x_0) \quad \widehat{C}(\ell_1) = \widehat{C}'(\ell_1) \\
\quad \mathcal{S}[\![s_0^{\ell_0}\ s_1^{\ell_1}]\!]K^{\ell} = \mathcal{S}[\![s_0]\!](\lambda^{\pi} x_0.(\mathcal{S}[\![s_1]\!](\lambda^{\pi_1} x_1.((x_0^{\ell_0}\ x_1^{\ell_1})^{\ell_2}\ K^{\ell})^{\ell_3})^{\ell_4})^{\ell_5})^{\ell_6} \\
\qquad\qquad \widehat{C}(\ell_0) = \widehat{\rho}'(x_0) \quad \widehat{C}(\ell_1) = \widehat{\rho}'(x_1) \\
\\
\mathcal{T} : \mathit{Triv} \to \mathit{Triv} \\
\quad \mathcal{T}[\![x]\!] = x \\
\quad \mathcal{T}[\![\lambda^{\pi} x.e^{\ell}]\!] = \lambda^{\pi} x.(\lambda^{\pi_1} k.\mathcal{E}[\![e^{\ell}]\!]k)^{\ell_0}
\end{array}
$$

Figure 4.6: Transformation of control flow from CPS into direct style

4.3.3 Direct-style transformation of control flow

The CPS transformation of flow from Figure 4.5 shows that the analysis of a CPS-transformed term can be at least as good as the analysis of the direct-style original term. The resulting CPS solution is the equivalent of the direct-style one, but may not be the best. We show that the direct-style and CPS analysis results are equivalent by exhibiting a direct-style transformation of flow.

We define a direct-style transformation of control-flow information. In other words, we transform control-flow information for the CPS-transformed term into control-flow information for the original direct-style term. The transformation is defined recursively in Figure 4.6. At each transformation step, on the right-hand side we construct flow information $(\widehat{C}, \widehat{\rho})$ for the direct-style program from the flow information $(\widehat{C}', \widehat{\rho}')$ for the CPS program.

Since at each function call the continuation is an explicit syntactic continuation, we are able to determine the control-flow information returned by each expression. In particular, at a transformation step

$$\mathcal{E}[\![s^{\ell}]\!]k = (\mathcal{S}[\![s]\!](\lambda^{\pi} x.(k^{\ell_0}\ x^{\ell_1})^{\ell_2})^{\ell_3})^{\ell_4}$$

we are able to assign control-flow information for the return label ℓ from the control-flow information collected by the continuation $\lambda^{\pi} x.(k^{\ell_0}\ x^{\ell_1})^{\ell_2}$.

Control-flow information can therefore be constructed bottom-up. The direct-style transformation of control flow is therefore defined as a monotone function:

$$\Psi_{\mathrm{cf}}^{\mathrm{CPS}} : (\mathit{Cache}^{p'} \times \mathit{Env}^{p'}) \to (\mathit{Cache}^{p} \times \mathit{Env}^{p})$$

Theorem 4.3.2. *Let $p = e^\ell$ be a uniquely labeled program. If $(\widehat{C}', \widehat{\rho}') \models^p \lambda^\pi k.\mathcal{E}[\![e^\ell]\!]k$ and $\widehat{\rho}'(k) = \emptyset$, then $\Psi_{\mathrm{cf}}^{\mathrm{CPS}}(\widehat{C}', \widehat{\rho}') \models^{p'} e^\ell$.*

Proof. By structural induction on the direct-style source program. We prove a predicate stating that the constructed solution satisfies the flow constraints for any serious sub-term together with its enclosing label. □

Again, the proof is similar to the proof of Theorem 2 of our earlier work [26] and with the proof of Theorem 2.6.5 of Chapter 2.

4.3.4 Preservation of flow

Following the construction of the CPS control-flow information in Figure 4.5, it is immediate to see that the flow information assigned to the program's original variables in CPS is identical to the one extracted from the direct-style original. The same is valid for the reverse transformation of Figure 4.6: the control-flow information assigned to direct-style variables is identical to the one extracted from the CPS program.

This following theorem follows from the monotonicity of the two transformations of control flow.

Theorem 4.3.3. *Let p be a direct-style program and p' its CPS counterpart.*

i) Let $(\widehat{C}, \widehat{\rho})$ be the solution of the control-flow analysis of p. Then $\Psi_{\mathrm{cf}}^{\mathrm{CPS}}(\Phi_{\mathrm{cf}}^{\mathrm{CPS}}(\widehat{C}, \widehat{\rho})) = (\widehat{C}, \widehat{\rho})$.

ii) Let $(\widehat{C}', \widehat{\rho}')$ be the solution of the control-flow analysis of p'. Then $\Phi_{\mathrm{cf}}^{\mathrm{CPS}}(\Psi_{\mathrm{cf}}^{\mathrm{CPS}}(\widehat{C}', \widehat{\rho}')) = (\widehat{C}', \widehat{\rho}')$.

4.4 Conclusions and future work

The complete CPS transformation of control flow can be used to assess the impact of the CPS transformation on the result of binding-times analysis. In a previous work [26], the authors have shown that introducing continuations may improve the binding-times obtained by the standard binding-time analysis for traditional partial evaluation. Translating programs into monadic normal form may lead to further binding-time improvements [50], also in Chapter 2. Our inital investigations show that the current transformation of control-flow can be used to fully characterize the binding-time improvements obtained by the CPS transformation.

An interesting further aspect is the issue of tail-call optimization and control-flow analysis. In the CPS transformation of Figure 4.4, the η-expansion of tail calls has the benefit of providing an explicit continuation for each function call for which we can extract control-flow information. More precisely, the CPS transformation of an expression introduces an explicit continuation:

$$\mathcal{E}[\![s]\!]k = \mathcal{S}[\![s]\!](\lambda x.k\ x)$$

The presence of an explicit continuation facilitates the definition of the CPS transformation of control flow. We are currently investigating the observation that reducing the tail-call η-redexes also preserves the control-flow information assigned to the original program points in the program.

Chapter 5

Time Stamps for Fixed-Point Approximation

Abstract[1]

Time stamps were introduced in Shivers's PhD thesis for approximating the result of a control-flow analysis. We show them to be suitable for computing program analyses where the space of results (e.g., control-flow graphs) is large. We formalize time-stamping as a top-down, fixed-point approximation algorithm which maintains a single copy of intermediate results. We then prove the correctness of this algorithm.

5.1 Introduction

5.1.1 Abstract interpretation and fixed-point computation

Abstract interpretation [23, 67] is a framework for systematic derivation of program analyses. In this framework, the standard semantics of a program is approximated by an abstract semantics. The abstract semantics simulates the standard semantics and is used to extract properties of the actual run-time behavior of the program.

Abstract interpretation often yields program analyses specified by a set of recursive equations. Formally, the analysis is defined as the least fixed point of a functional over a specific lattice. Analyzing a program then amounts to computing such a least fixed point. The design and analysis of algorithms for computing least fixed points has thus become a classic research topic.

This article presents a top-down algorithm that computes an approximate solution for a specific class of program analyses. This class includes analyses

[1] This chapter is an extended version of [25].

of programs with dynamic control-flow, namely programs whose control-flow is determined by the run-time values of program variables. Such programs are common, for instance, in higher-order and object-oriented languages.

The common problem of analyzing programs with dynamic control flow is to compute a static approximation of the dynamic control-flow graph. The flow information is usually represented as a table mapping each program point to the set of points that form possible outgoing edges from that point. The analysis may compute flow information either as a separate phase, or as an integral component of the abstract interpretation. In any case, flow information is itself computed as a least fixed point of a functional.

An algorithm for computing a solution of such an analysis is met with a difficult practical constraint: due to the potential size of the control-flow graph embedded in the result of the analysis, one cannot afford to maintain multiple intermediate results. The time-stamps-based algorithm considered here only needs to maintain a single intermediate analysis result throughout the computation.

5.1.2 The time-stamping technique

The time-stamping technique has been previously introduced in Shivers's PhD thesis [112] on control-flow analysis for Scheme, based on ideas from Hudak and Young's "memoized pending analysis" [126]. Using time stamps Shivers implements a top-down algorithm which computes an approximation of the semantic specification of the analysis and which does not maintain multiple intermediate results. The termination of the algorithm relies on the required monotonicity of the abstract semantics and on the use of time stamps on abstract environments. The algorithm yields an approximation by using increasingly approximate environments on the sequential analysis of program paths.

To our knowledge, Shivers's thesis contains the only description of the time-stamping technique. The thesis provides a formal account of some of the transformations performed on the abstract control-flow semantics in order to obtain an efficient implementation (as, for instance, the "aggressive cutoff" approach). The introduction of time stamps, however, remains only informally described. In particular, his account of the time-stamps algorithm [112, Chapter 6] relies on the property that the recursion sets computed by the modified algorithm are included in the recursion sets computed by the basic algorithm. Such property relies on the monotonicity of the original semantics, and the relationship with the algorithm modified to use a single-threaded environment remains unclear.

Our work: We formalize the time-stamps-based approximation algorithm as a generic fixed-point approximation algorithm, and we prove its correctness.

5.1.3 Overview

The rest of the article is organized as follows: In Section 5.2 we describe the time-stamps-based approximation algorithm. In Section 5.2.1 we define the class

of recursive equations on which the algorithm is applicable. In Section 5.2.2 we describe the intuition behind the time stamps. We proceed in Section 5.3 to formalize the time-stamps-based algorithm (Section 5.3.1) and prove its correctness (Section 5.3.2). In Section 5.3.3 we estimate the complexity of the algorithm. In Section 5.4 we show how to extend the algorithm to a wider class of analyses. In Section 5.5 we review related work and in Section 5.6 we conclude.

5.2 The time-stamps-based approximation algorithm

5.2.1 A class of recursive equations

We consider a class of recursive equations that model a program analysis by abstract interpretation. The analysis gathers information about a program by simulating its execution. Abstracting the details, we consider that a given program p induces a finite set of program points Lab. Transitions from a program point to another are modeled as directed edges in the graph. The analysis collects information as an element $\widehat{\rho}$ of a complete lattice A (we assume that A has finite height). Typically, such analysis information is in the form of a cache collecting information about program points and variables.

In our setting, at a program point $\ell \in Lab$, with intermediate analysis information $\widehat{\rho}$, the result of the analysis is computed from local analysis information together with the union of the results obtained by analyzing all possible outgoing paths. For instance, the analysis of a branching statement, when the result of the boolean condition is unknown, may be obtained as the union of the analysis of both branches. In higher-order languages, the analysis of a function call $(e_0\ e_1)$ may be obtained as the union of the analysis of calls to all functions that the expression e_0 can evaluate to.

The choice of a specific outgoing edge may determine a specific update of the analysis information. For instance, after choosing one of the functions that may be called at an application point, one updates the information associated to the formal parameter with the information associated to the actual parameter.

We consider therefore that local analysis information is defined by a monotone function $B : (Lab \times A) \to A$. We also consider that the analysis information associated with the transition from a program point to another is defined by a monotone function $V : (Lab \times Lab \times A) \to A$. Such functions can model, for instance, Sagiv, Reps and Horowitz's environment transformers [108], but they can also model monotone frameworks [71, 90]. Transition information is added into the already computed analysis information, in a collecting analysis [23, 110] fashion.

To model dynamic control flow, we consider that, at a specific node ℓ and in the presence of already computed analysis information $\widehat{\rho}$, the set of possible future nodes is described by a monotone function $R : (Lab \times A) \to \mathcal{P}(Lab)$: transitions are obtained from the current node ℓ and the elements of $R(\ell, \widehat{\rho})$. A generic analysis function $F : (Lab \times A) \to A$ may therefore be defined by the

following recursive equation:

$$F(\ell, \widehat{\rho}) = B(\ell, \widehat{\rho}) \sqcup \bigsqcup_{\ell' \in R(\ell, \widehat{\rho})} F(\ell', \widehat{\rho} \sqcup V(\ell, \ell', \widehat{\rho})), \qquad (*)$$

If the functions B, R and V are monotone on $\widehat{\rho}$ (Lab is essentially a flat domain), it can be easily shown that Equation $(*)$ has solutions. Given the starting point of the program ℓ_0 and some initial (possibly empty) analysis information $\widehat{\rho}_0$, we are interested in computing a value $F(\ell_0, \widehat{\rho}_0)$, where F is the least solution of Equation $(*)$.

It is usually more expensive to compute the entire function F as the least solution of Equation $(*)$. Naturally, we only want to implement a program that computes the value of $F(\ell_0, \widehat{\rho}_0)$. Naturally, in order to compute a value $F(\ell, \widehat{\rho})$, one needs to control termination (repeating sequences of pairs $(\ell, \widehat{\rho})$ might appear) and one also needs to save intermediate copies of the current analysis information $\widehat{\rho}$ when the current node ℓ has multiple outgoing edges.

Memoization is a solution for controlling termination. When the space of analysis results is large, however, the cost of maintaining the memoization table, coupled with the cost of saving intermediate results, leads to a prohibitively expensive implementation. We can use Shivers's time-stamping technique [112] to solve these two problems, as long as we are satisfied with an approximation of $F(\ell_0, \widehat{\rho}_0)$.

5.2.2 The intuition behind time stamps

We present a pseudo-code formulation of the algorithm which informally describes the time-stamping technique. We will properly formalize the algorithm and prove its correctness in Section 5.3.

We assume that we are given an instance of the analysis specified by the functions B, R and V (which we assume are computable). The pseudo-code of the time-stamps-based approximation algorithm is given in Figure 5.1. The time-stamps-based algorithm uses a time counter t (initialized with 0) and a table τ which associates to each program point ℓ a time stamp $\tau[\ell]$, initialized with 0. We compute the result of the analysis into a global variable $\widehat{\rho}$, initialized with $\widehat{\rho}_0$. In essence, the function F' is obtained by lifting the $\widehat{\rho}$ parameter out of the F function.

The time counter t and the time-stamps table τ (modeled as an array of integers) are also global variables. The function U updates the global analysis with fresh information: if new results are obtained, the time counter is incremented before they are added in the global analysis. The function F' implements the time-stamps-based approximation. To approximate the value of $F(\ell)$, we first compute the local information $B(\ell, \widehat{\rho})$ and add the result into the global analysis. We then compute the set of future nodes $R(\ell, \widehat{\rho})$. For each future node $\ell' \in R(\ell, \widehat{\rho})$, *sequentially*, we compute the execution information $V(\ell, \ell', \widehat{\rho})$ along the edge (ℓ, ℓ'), we add its result to $\widehat{\rho}$ and we then call $F'(\ell')$. Because all the calls to F' on the second or later branches are made with a possibly larger $\widehat{\rho}$, an approximation may occur.

```
global ρ̂:A, t:N, τ:N array
proc U(ρ̂₁) = if ρ̂₁ ⋢ ρ̂ then t := t + 1; ρ̂ := ρ̂ ⊔ ρ̂₁
proc F′(ℓ) = if τ[ℓ] ≠ t then
                 τ[ℓ] := t;
                 U(B(ℓ, ρ̂));
                 foreach ℓ′ in R(ℓ, ρ̂)
                     U(V(ℓ, ℓ′, ρ̂)); F′(ℓ′)
```

Figure 5.1: Time-stamps-based approximation algorithm

Each time $\widehat{\rho}$ is increased by addition of new information, we increment the time counter. Each time we call F' on a program point ℓ, we record the current value of the time counter in the time-stamps table at ℓ's slot, i.e., $\tau[\ell] := t$. We use the time-stamps table to control the termination. If the function F' is called on a point ℓ such that $\tau[\ell] = t$, then there has already been a call to F' on ℓ, and the environment has not been updated since. Therefore, no fresh information is going to be added to the environment by this call, and we can simply return without performing any computation.

Such correctness argument is only informal, though. In his thesis, Shivers [112] makes a detailed description of the time-stamps technique in the context of a control-flow analysis for Scheme. He proves that memoization (the so-called "aggressive cutoff" method) preserves the results of the analysis. The introduction of time-stamps and the approximation obtained by collecting results in a global variable remain only informally justified. In the next section we provide a formal description of the time-stamps-based approximation algorithm and we prove its correctness.

5.3 A formalization of the time-stamps-based algorithm

5.3.1 State-passing recursive equations

We formalize the algorithm and the time-stamping technique as a new set of recursive equations. The equations describe precisely the computational steps of the algorithm. They are designed such that their solution can be immediately related with the semantics of an implementation of the algorithm from Figure 5.1 in a standard programming language. At the same time, they define an approximate solution of Equation $(*)$ on the preceding page. We prove that the solution of the new equations is indeed an approximation of the original form.

The equations are modeling a state-passing computation. The global state of the computation contains the analysis information $\widehat{\rho}$, the time-stamps table τ and the time counter t. The time-stamps table is modeled by a function

$$
\begin{array}{l}
F'(\ell,(\widehat{\rho},\tau,t)) = \textbf{if } \tau(\ell) = t \textbf{ then } (\widehat{\rho},\tau,t) \\
\qquad\qquad \textbf{else let} \\
\qquad\qquad\qquad \{\ell_1,...,\ell_n\} = R(\ell,\widehat{\rho}) \\
\qquad\qquad\qquad (\widehat{\rho}_0,\tau_0,t_0) = U(B(\ell,\widehat{\rho}),(\widehat{\rho},\tau[\ell \mapsto t],t)) \\
\qquad\qquad\qquad (\widehat{\rho}_1,\tau_1,t_1) = F'(\ell_1,\, U(V(\ell,\ell_1,\widehat{\rho}_0),(\widehat{\rho}_0,\tau_0,t_0))) \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad (\widehat{\rho}_n,\tau_n,t_n) = F'(\ell_n,\, U(V(\ell,\ell_n,\widehat{\rho}_{n-1}),(\widehat{\rho}_{n-1},\tau_{n-1},t_{n-1}))) \\
\qquad\qquad\qquad \textbf{in } (\widehat{\rho}_n,\tau_n,t_n) \\
U(\widehat{\rho}_1,(\widehat{\rho},\tau,t)) = \textbf{if } \widehat{\rho}_1 \not\sqsubseteq \widehat{\rho} \textbf{ then } (\widehat{\rho} \sqcup \widehat{\rho}_1,\tau,t+1) \textbf{ else } (\widehat{\rho},\tau,t)
\end{array}
$$

Figure 5.2: Time-stamps-based approximation equation

$\tau \in Lab \to \mathbf{N}$:

$$(\widehat{\rho},\tau,t) \in States = (A \times (Lab \to \mathbf{N}) \times \mathbf{N})$$

In contrast to the standard denotational semantics, we consider $\mathbf{N}$ with the usual ordering on natural numbers. Therefore *States* is an infinite domain containing infinite ascending chains. To limit the height of ascending chains, we restrict the space to reflect more precisely the set of possible states in the computation:

$$States = \{(\widehat{\rho},\tau,t) \in (A \times (Lab \to \mathbf{N}) \times \mathbf{N}) \mid t \leq h(\widehat{\rho}) \wedge \forall \ell \in Lab.\tau(\ell) \leq t\}$$

Here the function $h(\widehat{\rho})$ defines "the length of the longest chain of elements of A below $\widehat{\rho}$".

Informally, the restriction accounts for the fact that we increment t each time we add information into $\widehat{\rho}$. Starting from $\widehat{\rho} = \bot$ and $t = 0$, t is always smaller than the longest ascending path from bottom to $\widehat{\rho}$ in A. The second condition accounts for the fact that the time-stamps table records time stamps smaller than or equal to the value of the time counter.

The recursive equations that define the time-stamps approximation are stated in Figure 5.2. They define a function $F' : (Lab \times States) \to States$ that models a state-passing computation. It is easy to show that $U : (A \times States) \to States$ is well-defined (on the restricted space of states). The existence of solutions for the equations from Figure 5.2 can then be easily established, due to the monotonicity of B, V and R.

Note that the order in which the elements of the set of future nodes $R(\ell,\widehat{\rho})$ are processed remains unspecified. This aspect does not affect our further development, while leaving room for improving the evaluation strategy.

The main reason for the restriction on the states and for the non-standard semantics is that we restrict the definition of the function to the strictly terminating instances. It is easy to show that F' terminates on any initial program point and initial state. In fact, such initial configuration determines a trace of states which we use to show that the function F' computes a safe approximation of the analysis.

5.3.2 Correctness

The correctness of the time-stamps-based algorithm, i.e., the fact that it computes an approximation of the function defined by Equation $(*)$ on page 74, is established by the following theorem.

Theorem 5.3.1. *For any $\ell \in Lab$ and $\widehat{\rho} \in A$:*

$$F(\ell, \widehat{\rho}) \sqsubseteq \pi_1(F'(\ell, (\widehat{\rho}, \lambda\ell.0, 1)))$$

where π_1 is the projection of the first element of the tuple.

The theorem is proven in two steps. First, we show that using time stamps to control recursion does not change the result of the analysis. In this sense, we consider an intermediate equation defining a function $F'' : (Lab \times States) \to A$.

$$\begin{aligned} F''(\ell, (\widehat{\rho}, \tau, t)) \quad = \quad & \textbf{if } \tau(\ell) = t \textbf{ then } \bot \\ & \textbf{else } B(\ell, \widehat{\rho}) \sqcup \bigsqcup_{\ell' \in R(\ell, \widehat{\rho})} F''(\ell', U(V(\ell, \ell', \widehat{\rho}), (\widehat{\rho}, \tau[\ell \mapsto t], t))) \end{aligned}$$

We show that the function F'' computes the same analysis as the function defined by Equation $(*)$.

Lemma 5.3.2. *Let $\ell \in Lab$ be a program point and $(\widehat{\rho}, \tau, t) \in States$. Let $S = \{\ell' \in Lab | \tau(\ell') = t\}$. Then we have:*

$$F''(\ell, (\widehat{\rho}, \tau, t)) \sqcup \bigsqcup_{\ell' \in S} F(\ell', \widehat{\rho}) = F(\ell, \widehat{\rho}) \sqcup \bigsqcup_{\ell' \in S} F(\ell', \widehat{\rho})$$

Lemma 5.3.2 is proved using a well-founded induction on states, based on the observation that F'' calls itself on arguments strictly above the original (in the space of states). As an instance of the Lemma 5.3.2 we obtain:

Corollary 5.3.3.

$$\forall \ell \in Lab, \widehat{\rho} \in A \,.\, F(\ell, \widehat{\rho}) = F''(\ell, (\widehat{\rho}, \lambda\ell.0, 1))$$

We show that the time-stamps algorithm computes an approximation of the function F''.

Lemma 5.3.4.

$$\forall\, (\widehat{\rho}, \tau, t) \in States, \ell \in Lab \,.\, F''(\ell, (\widehat{\rho}, \tau, t)) \sqsubseteq \pi_1(F'(\ell, (\widehat{\rho}, \tau, t)))$$

The proof of Lemma 5.3.4 relates the recursion tree from the definition of function F'' and the trace of states in the computation of F'. In essence, the value of $F''(\ell, (\widehat{\rho}, \tau, t))$ is defined as the union of a tree of values of the form $B(\ell_i, \widehat{\rho}_i)$. We show by induction on the depth of the tree that each of these values is accumulated in the final result at some point on the trace of states

in the computation of F'. The complete proof of Lemma 5.3.4 is presented in Appendix 5.A.

Combining the two lemmas we obtain the statement of Theorem 5.3.1.

Despite the non-standard ordering and domains used when defining the solutions of the equations in Figure 5.2, showing that the function F' agrees on the starting configuration with a standard semantic definition of the algorithm in Figure 5.1 is trivial and is not part of the current presentation.

5.3.3 Complexity estimates

Let us assume that computing the function U takes m time units, and that B, R and V can be computed in constant time (one time unit). The time-counter can be incremented at most $h(A)$ times, where the function h defines the height (given by the longest ascending chain) of the lattice A. In the worst case, between two increments, each edge in the graph may be explored, and for each edge we might spend m time units in computing the U function. Thus, computing $F'(\ell, (\widehat{\rho}, \lambda\ell.0, 1))$ has a worst-case complexity of $O(|Lab|^2 \cdot m \cdot h(A))$.

Space-wise, it is immediate to see that at most two elements of A are in memory at any given time: the global value $\widehat{\rho}$ and one temporary value created at each call of B or V. The temporary value is not of a concern though: in most usual cases, the size of the results of B or V is at least one order of magnitude smaller than the size of $\widehat{\rho}$.

The worst-case space complexity is also driven by the exploration of edges. It is immediate to see that each edge might be put aside between two updates of the global environment. Denoting with $S(A)$ the size of an element in A, the worst-case space complexity might be $O(S(A) + |Lab|^2 \cdot h(A))$. It seems apparent, however, that many of the edges put aside are redundant. We are currently exploring possibilities of avoiding some of these redundancies.

5.4 An extension

The time-stamps method has originally been presented in the setting of flow analysis of Scheme programs in continuation-passing style (CPS) [112]. Indeed, the formulation of Equation $(*)$ on page 74 facilitates the analysis of a computation that "never returns" or, otherwise said, of an analysis function that has a tail-recursive formulation.

There is no particular reason for which the time-stamps technique may not be extended to non-tail-recursive analyses. We show how the time-stamps technique can be extended to a flow analysis of higher-order languages which has a non-tail-recursive formulation.

In their paper on CPS versus direct style in program analysis [106], Sabry and Felleisen also suggest using a memoization technique for computing the result of their constant propagation for a higher-order language in direct style. The constant-propagation analysis has a non-compositional, non-tail recursive definition. Indeed, in order to model the analysis of a term like **let** $x = V_1\ V_2$ **in** M,

Sabry and Felleisen's analysis explores all possible functions that can be called in the header of the let, joins the results and, afterwards, analyzes the term M.

We can apply the time-stamps technique to Sabry and Felleisen's analysis in order to compute an approximate solution more efficiently. We consider equations of the following form:

$$\begin{aligned} F(\ell,\widehat{\rho}) = \textbf{let } \widehat{\rho}_1 &= B(\ell,\widehat{\rho}) \sqcup \bigsqcup_{\ell' \in R(\ell,\widehat{\rho})} F(\ell', \widehat{\rho} \sqcup V(\ell,\ell',\widehat{\rho})) \\ \textbf{in } B'(\ell,\widehat{\rho}_1) &\sqcup \bigsqcup_{\ell' \in R'(\ell,\widehat{\rho}_1)} F(\ell', \widehat{\rho}_1 \sqcup V'(\ell,\ell',\widehat{\rho}_1)) \end{aligned}$$

The algorithm is straightforwardly extended to account for the second call with another iteration over $R'(\ell,\widehat{\rho}_1)$. The proof of correctness extends as well. It is remarkable that despite the more complicated formulation of the equations, the complexity of the algorithm remains the same, due to the bounds imposed by the time-stamps.

Applying the time-stamps-based algorithm to Sabry and Felleisen's analysis yields a more efficient algorithm than their proposed memoization-based implementation (for the reasons outlined in Section 5.2.1). The approximation obtained using time stamps is still precise enough. In particular, the time-stamps-based analysis is able to distinguish returns. Consider for instance the following example (also due to Sabry and Felleisen):

$$\begin{array}{lrl} \textbf{let} & f & = \lambda x.x \\ & x_1 & = f\ 1 \\ & x_2 & = f\ 2 \\ \textbf{in} & x_1 & \end{array}$$

The time-stamps-based analysis computes a solution in which x_1 (and, therefore, the result of the entire expression) is bound to 1, and x_2 is bound to $\top$. In contrast, a monovariant constraint-based data-flow analysis [90] computes a solution in which both x_1 and x_2 are bound to $\top$.

Formally, it is relatively easy to show that the time-stamps-based constant propagation always computes a result at least as good as a standard monovariant constraint-based data-flow analysis. The details are omitted from this article. Note that the improvement in the quality over the constraint-based analysis comes at a price in the worst-case time and space complexity.

5.5 Related work

A number of authors describe algorithms for computing least fixed points as solutions to program analyses using chaotic iteration, which are also adapted to compute approximation using widenings or narrowings [23]. O'Keefe's bottom-up algorithm [91] has inspired a significant number of articles, where the convergence speed is improved using refined strategies on choosing the next iteration, or exploiting locality properties of the specifications [15, 69, 104].

Such algorithms have also been applied to languages with dynamic control flow. Chen, Harrison and Yi [18] developed advanced techniques such as "waiting for all successors", "leading edge first", "suspend evaluation", which improve

the behavior of the bottom-up algorithm when applied to such languages. In a subsequent work [17], the authors use reachability information to implement a technique called "context projection" which reduces the amount of abstract information associated to each program point. In contrast, time stamps approximate the solution, by maintaining only one global context common to all program points.

Other algorithms that address languages with dynamic flow have been developed in the context of strictness analysis. Clack and Peyton-Jones [19] have introduced the frontier-based algorithm. The algorithm reduces space usage by representing the solution only with a subset of relevant values. The technique has been developed for binary lattices. Hunt's PhD thesis [60] contains a generalization to distributive lattices.

The top-down vs. bottom-up aspects of fixed-point algorithms for abstract interpretation of logic programs have been investigated by Le Charlier and Van Hentenryck. The two authors have developed a generic top-down fixed-point algorithm [16], and have compared it with the alternative bottom-up strategy. The evaluation strategy of their algorithm is similar to the time-stamps-based one in this article. In contrast, however, since their algorithm precisely computes the least fixed point, it also maintains multiple values from the lattice of results.

Fecht and Seidl [40] design the time-stamps solver "WRT" which combines the benefits of both the top-down and bottom-up approaches. The algorithm also uses time stamps, in a different manner though: The time stamps are used to interpret the algorithm's worklist as a priority queue. Our technique uses time stamps simply to control the termination of the computation. In a sequel paper [41], the authors derive a fixed-point algorithm for distributive constraint systems and use it, for instance, to compute a flow graph expressed as a set of constraints.

5.6 Conclusion

We have presented a polynomial-time algorithm for approximating the least fixed point of a certain class of recursive equations. The algorithm uses time stamps to control recursion and avoids duplication of analysis information over program branches by reusing intermediate results. The time-stamping technique has originally been introduced by Shivers in his PhD thesis [112]. To the best of our knowledge, the idea has not been pursued. We have presented a formalization of the technique and we have proven its correctness.

Several issues regarding the time-stamps-based algorithm might be worth further investigation. For instance, it is noticeable that the order in which the outgoing edges are processed at a certain node might affect the result of the analysis. Designing an improved strategy for selecting the next node to be processed is worth investigating. Also, as we observed in Section 5.3.3, an edge might be processed several times independently, each time with a larger analysis information. This suggests that some of the processing might be redundant. We are currently investigating such a possible improvement of the algorithm, and

its correctness proof.

5.A Operational specification

We give an an operational specification of the time-stamp algorithm. Configurations are triples formed by a global state and two stacks. The global state is defined as:

$$\mathit{States} = \{(\widehat{\rho}, \tau, t) \in (A \times (\mathit{Lab} \to \mathbf{N}) \times \mathbf{N}) \mid t \leq h(\widehat{\rho}) \wedge \forall \ell \in \mathit{Lab}.\tau(\ell) \leq t\}$$

Configurations are then defined as:

$$\sigma \in \mathit{Conf} = \mathit{States} \times (\mathit{Lab} \times (\mathit{Lab}^*))^*$$

where we use Lab^* to denote the domain of finite lists of program points from of Lab. Final configurations are configurations of the form

$$\langle (\widehat{\rho}, \tau, t), [] \rangle^A$$

We distinguish two types of configurations. A-configurations are reached in the middle of the for-loop of Figure 5.1. B-configurations are reached at the entry point of the main function.

The transition judgments are defined in Figure 5.3. Updates in the analysis result are modeled by $\vdash_U$ judgments. Time-stamps for nodes are verified in B-states and analysis transitions occur at A-states. It is easy to prove that from any initial configuration $\langle (\widehat{\rho}, \tau, t), ls, L \rangle$, the transition reaches a final configuration $\langle (\widehat{\rho}, \tau, t), [], [] \rangle$ in a finite number of steps. We use $\Rightarrow^*$ to denote the transitive closure of the relation $\Rightarrow$.

By definition, the time-stamp based analysis starting from the initial program point ℓ_0 and analysis information $\widehat{\rho}_0$ is the element $\widehat{\rho}$ such that

$$\vdash \langle (\widehat{\rho}_0, \lambda\ell.0, 1), (\ell_0, \ell_0 :: []) :: [] \rangle^B \Rightarrow^* \langle (\widehat{\rho}, \tau, t), [] \rangle^A$$

We also define the trace of an analysis to be the set of states σ such that $\sigma_0 \Rightarrow^* \sigma$.

Given a configuration $\langle (\widehat{\rho}, \tau, t), L \rangle^A$, we say that a node ℓ is on the stack if there exist a pair (ℓ', ls) in the list L such that ℓ belongs to the list ls.

Lemma 5.A.1 (Termination). *Given ℓ_0 and $\widehat{\rho}_0$ there exists a unique final configuration $\langle (\widehat{\rho}, \tau, t), [] \rangle^A$ such that*

$$\vdash \langle (\widehat{\rho}_0, \lambda\ell.0, 1), (\ell_0, \ell_0 :: []) :: [] \rangle^B \Rightarrow^* \langle (\widehat{\rho}, \tau, t), [] \rangle^A$$

Proof. First we show that in successive configurations, the environment always increases. We then show that there can not be an infinite number of B-configurations, due to the finite height of the lattice of results and the connection between time stamps and environment established by the $\vdash_U$ judgments. It follows that there can not be an infinite nuber of configurations and thus termination. Since the rules are deterministic, uniqueness follows. □

Tracking updates :

$$\frac{}{\vdash_U \langle(\widehat{\rho},\tau,t)\,,\widehat{\rho}_1\rangle \to (\widehat{\rho}\sqcup\widehat{\rho}_1,\tau,t+1)}\ \widehat{\rho}_1 \not\sqsubseteq \widehat{\rho}$$

$$\frac{}{\vdash_U \langle(\widehat{\rho},\tau,t)\,,\widehat{\rho}_1\rangle \to (\widehat{\rho},\tau,t)}\ \widehat{\rho}_1 \sqsubseteq \widehat{\rho}$$

Node handling:

$$\frac{\vdash_U \langle(\widehat{\rho},\tau[\ell\mapsto t],t)\,,B(\ell_1,\widehat{\rho})\rangle \to (\widehat{\rho}_1,\tau_1,t_1)}{\vdash \langle(\widehat{\rho},\tau,t)\,,(\ell,\ell_1{::}ls_1){::}L\rangle^B \Rightarrow \langle(\widehat{\rho}_1,\tau_1,t_1)\,,(\ell_1,R(\ell_1,\widehat{\rho})){::}(\ell,ls_1){::}L\rangle^A}\ \tau(\ell)\neq t$$

$$\frac{}{\vdash \langle(\widehat{\rho},\tau,t)\,,(\ell,\ell_1{::}ls_1){::}L\rangle^B \Rightarrow \langle(\widehat{\rho},\tau,t)\,,(\ell,ls_1){::}L\rangle^A}\ \tau(\ell)=t$$

Edge handling:

$$\frac{\vdash_U \langle(\widehat{\rho},\tau,t)\,,V(\ell_1,\ell_2,\widehat{\rho})\rangle \to (\widehat{\rho}_1,\tau_1,t_1)}{\vdash \langle(\widehat{\rho},\tau,t)\,,(\ell_1,\ell_2{::}ls_2){::}L\rangle^A \Rightarrow \langle(\widehat{\rho}_1,\tau_1,t_1)\,,(\ell_1,\ell_2{::}ls_2){::}L\rangle^B}$$

$$\frac{}{\vdash \langle(\widehat{\rho},\tau,t)\,,(\ell_1,[]){::}L\rangle^A \Rightarrow \langle(\widehat{\rho},\tau,t)\,,L\rangle^A}$$

Figure 5.3: Operational specification of the time-stamps-based algorithm

In the following, we assume that we are given a finite trace of the analysis starting from ℓ_0 and $\widehat{\rho}_0$.

Lemma 5.A.2 (Progress). *Let $\langle(\widehat{\rho},\tau,t)\,,L\rangle^A$ be a A-configuration in the trace of the time-stamp-based algorithm applied to ℓ_0 and $\widehat{\rho}_0$ and ℓ a node such that ℓ is in the stack L. Then ℓ is eventually processed, namely there exists a later B-configuration $\langle(\widehat{\rho}_1,\tau_1,t_1)\,,(\ell',\ell{::}ls_1){::}L\rangle^B$ in the trace.*

Proof. By induction on the depth of the position of ℓ in the list L. We use the fact that the trace of states is finite. □

Lemma 5.A.3 (Time-stamp control). *Let $\langle(\widehat{\rho}_1,\tau_1,t)\,,(\ell_1,\ell{::}ls_1){::}L_1\rangle^B$ be a B-configuration in the trace such that $\tau_1(\ell)=t$. Then there exists a previous B-configuration $\langle(\widehat{\rho}_2,\tau_2,t)\,,(\ell_2,\ell{::}ls_2){::}L_2\rangle^B$ such that $\widehat{\rho}_1=\widehat{\rho}_2$ and $\tau_2(\ell)<t$.*

Proof. By reduction to absurd. The only possiblity that $\tau_1(\ell)=t$ is that there has been a previous B-configuration at which $\tau_1(\ell)$ has been assigned to t, and since then, the environment has not changed. □

The above lemmas are used to show that the analysis information collected by the F'' function modeling the time-stamp based control of recursion is also

collected by the time-stamp based algorithm. Let us recall the equations defining the F'' function:

$$\begin{aligned} F''(\ell,(\widehat{\rho},\tau,t)) \quad = \quad & \textbf{if } \tau(\ell)=t \textbf{ then } \bot \\ & \textbf{else } B(\ell,\widehat{\rho}) \sqcup \bigsqcup_{\ell' \in R(\ell,\widehat{\rho})} F''(\ell', U(V(\ell,\ell',\widehat{\rho}),(\widehat{\rho},\tau[\ell \mapsto t],t))) \end{aligned}$$

Given an initial point ℓ_0 and initial analysis information $\widehat{\rho}_0$, computing the value of $F''(\ell_0,\widehat{\rho}_0)$ gives rise to a tree of recursive calls $F''(\ell,(\widehat{\rho},\tau,t))$. The value of $F''(\ell_0,\widehat{\rho}_0)$ is thus computed as the union of $B(\ell,\widehat{\rho})$ values for the $(\ell,(\widehat{\rho},\tau,t))$ pairs in the tree such that $\tau(\ell) < t$.

Lemma 5.A.4 (Approximative simulation). *Let $(\ell,(\widehat{\rho},\tau,t))$ be a tuple in the tree of calls to $F''(\ell_0,\widehat{\rho}_0)$ such that $\tau(\ell) < t$. Then, in the trace of configurations of the time-stamp-based algorithm applied to ℓ_0 and $\widehat{\rho}_0$, there exists a configuration $\langle(\widehat{\rho}_1,\tau_1,t_1),(\ell',\ell::ls_1)::L\rangle^B$ such that $\widehat{\rho} \sqsubseteq \widehat{\rho}_1$ and $\tau_1(\ell) < t_1$.*

Proof. By induction on the tree of calls to $F''(\ell_0,\widehat{\rho}_0)$. The starting node is trivial. Let us take a call $(\ell,(\widehat{\rho},\tau,t))$ such that $\tau(\ell) < t$. Since this is not the top-level call, there must exist a previous call $(\ell_2,(\widehat{\rho}_2,\tau_2,t_2))$ such that $\ell \in R(\ell_2,\widehat{\rho}_2)$. By I.H, there exists also a previous state $\langle(\widehat{\rho}_3,\tau_3,t_3),(\ell_3,\ell_2::ls_3)::L_3\rangle^B$ such that $\widehat{\rho}_2 \sqsubseteq \widehat{\rho}_3$. We make a case distinction on whether $\tau_2(\ell_2) = t_2$ or not. If not, then using Lemma 5.A.3 we obtain a similar state in which the above is true. If yes, then the outgoing nodes $R(\ell_2,\widehat{\rho}_3)$ have been pushed on the stack, and, among them, ℓ. From Lemma 5.A.2 we obtain that there is a later configuration $\langle(\widehat{\rho}_4,\tau_4,t_4),(\ell_4,\ell::ls_4)::L_4\rangle^B$. More importanttly, $\widehat{\rho}_3 \sqsubseteq \widehat{\rho}_4$. If $\tau_4(\ell) < t_4$, then we are done. Otherwise, we use Lemma 5.A.3 to obtain the desired configuration. □

From Lemma 5.A.4 it is easy to show that each value $B(\ell,\widehat{\rho})$ collected by the function F'' is also collected by the time-stamp-based algorithm. We obtain the following theorem:

Theorem 5.A.5 (Correctness of time-stamp approximation). *Let $(\widehat{\rho}_1,\tau_1,t_1) = F''(\ell_0,(\widehat{\rho}_0,\lambda\ell.0,1))$ and $(\widehat{\rho}_2,\tau_2,t_2)$ such that*

$$\vdash \langle(\widehat{\rho}_0,\lambda\ell.0,1),(\ell_0,\ell_0::[])::[]\rangle^B \Rightarrow^* \langle(\widehat{\rho}_2,\tau_2,t_2),[]\rangle^A$$

Then $\widehat{\rho}_1 \sqsubseteq \widehat{\rho}_2$.

Proof. By induction on the unfolding of $F''(\ell_0,(\widehat{\rho}_0,\lambda\ell.0,1))$. □

All that remains to be shown is the following theorem:

Theorem 5.A.6 (Accuracy). *Let $(\widehat{\rho}_1,\tau_1,t_1) = F'(\ell_0,(\widehat{\rho}_0,\lambda\ell.0,1))$ and $(\widehat{\rho}_2,\tau_2,t_2)$ such that*

$$\vdash \langle(\widehat{\rho}_0,\lambda\ell.0,1),(\ell_0,\ell_0::[])::[]\rangle^B \Rightarrow^* \langle(\widehat{\rho}_2,\tau_2,t_2),[]\rangle^A$$

Then $\widehat{\rho}_1 = \widehat{\rho}_2$.

Proof. By induction on the unfolding of $F'(\ell_0,(\widehat{\rho}_0,\lambda\ell.0,1))$. □

Chapter 6

Static Transition Compression

Abstract[1]

Starting from an operational specification of a translation from a structured to an unstructured imperative language, we point out how a compositional and context-insensitive translation gives rise to static chains of jumps. Taking an inspiration from the notion of continuation, we state a new compositional and context-sensitive specification that provably gives rise to no static chains of jumps, no redundant labels, and no unused labels. It is defined with one inference rule per syntactic construct and operates in linear time and space on the size of the source program (indeed it operates in one pass).

6.1 Introduction

The art of writing a compiler partly amounts to resolving the tensions between its various phases. For example, should each phase be simple, but generate redundancies that equally simple later phases would eliminate? Or should each phase avoid redundancies in order to avoid a later phase and save the resources spent processing the redundancies in between? Picturesquely, Richard Gabriel nicknamed this kind of choices "worse is better" and "the right thing" [46]. Worse is better yields results sooner, at the risk of never ending with the right thing. The right thing is the right thing, but takes longer to achieve, at the risk of funding running out. Resolving the tensions between these choices in a compiler is anything but easy due to its many circular dependencies.

In this article, we consider transition compression, i.e., collapsing chains of jumps into unique jumps. To this end, we formalize the translation of structured

[1] This chapter has been published as [28]. The chapter is joint work with Olivier Danvy.

programs into unstructured programs with jumps (Section 6.2). We show how a compositional and context-insensitive translation naturally generates chains of jumps (Section 6.3). We then specify a compositional and context-sensitive translation that is parameterized by the labels of the current command and of the next command, and we prove that it generates no chains of jumps, no redundant labels and no unused labels (Section 6.4 and 6.5). It therefore compresses transitions at translation time—hence the title of this article.

The issue of transition compression is standard [2] [66, Section 4.4], but we are not aware of any other formalized characterization of generating chains of jumps and avoiding them. Since we wrote this article, however, we became aware of Dybvig, Hieb, and Butler's work on destination-driven code generation [38], which shares the same goal as static transition compression and has been implemented as part of the back-end of an optimizing Scheme compiler.

6.2 Source and target languages

We consider the translation of structured programs that use conditional commands and while loops into unstructured programs that use labels and jumps.

6.2.1 An unstructured target language

Our target language is defined in Figure 6.1. Unstructured commands can be (unspecified) atomic commands, skip commands, sequences of commands, conditional jumps, and unconditional jumps. Commands can be labeled, and labels are used as targets of jumps. Conditional jumps are triggered by either the truth (if) or the falsity (ifn) of a boolean expression. A program is an unstructured command ending with a stop instruction, optionally labeled.

$u \in \mathrm{UCom} ::= a \mid \mathsf{skip} \mid u_1; u_2 \mid \ell : u \mid \mathsf{if}\ b\ \mathsf{goto}\ \ell \mid \mathsf{ifn}\ b\ \mathsf{goto}\ \ell \mid \mathsf{goto}\ \ell$
$a \in \mathrm{ACom}$ (atomic commands, omitted)
$b \in \mathrm{BExp}$ (boolean expressions, omitted)
$\ell \in \mathit{Lab}$ (code labels, unspecified)

Figure 6.1: An unstructured target language

Valid programs are commands where labels are unique and targets of jumps always exist. Their semantics is state-based and straightforward. (We define it as a trace semantics since we omit the specification of atomic commands. A state is thus the series of atomic commands executed in the course of a program.)

6.2.2 A structured source language

Our source language is defined in Figure 6.2. Structured commands can be atomic commands, sequences of commands, conditional commands, and while loops. Again, their (trace) semantics is state-based and straightforward [124].

$s \in \text{SCom} ::= a \mid s_1; s_2 \mid \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{end} \mid \mathsf{while}\ b\ \mathsf{do}\ s\ \mathsf{end}$
$a \in \text{ACom}$ (atomic commands, same as in Figure 6.1)
$b \in \text{BExp}$ (boolean expressions, same as in Figure 6.1)

Figure 6.2: A structured source language

Implementing a compiler for such a language requires one to translate structured commands into unstructured commands in a language such as the one in Figure 6.1. This translation is the topic of the next section.

6.3 A context-insensitive translation

We start with a compositional and context-insensitive translation. For each compound structured command, the translation generates code for all subcomponents, and combines them together with explicit conditional/unconditional jumps implementing the required flow of control. The translation is canonical [123, Section 2.8].

6.3.1 The translation

The judgment

$$\vdash s \longrightarrow u$$

holds whenever the structured command s translates into the unstructured command u. Fresh labels may be generated during the translation. Label generation can be modeled, e.g., by threading a counter through the derivation tree.

At the top level, a structured program s is translated into an unstructured program u; stop if $\vdash s \longrightarrow u$ holds.

The inference rules are displayed in Figure 6.3. Let us describe each of them in words.

Atomic commands: An atomic command is translated into the same atomic command.

Sequences of commands: A sequence of commands is translated into the sequence of the translated commands.

Conditional commands: A conditional command is translated into a conditional jump determining whether to execute the translated then branch or the translated else branch. The translated branches merge into a labeled skip instruction. One branch jumps to it and the other one flows into it.

While loops: A while loop is translated into a labeled conditional jump determining whether to execute the translated body of the loop or to exit the loop. The translated body is followed by an unconditional jump to close the loop. A labeled skip instruction follows as the exit point of the loop.

$$\frac{}{\vdash a \longrightarrow a}$$

$$\frac{\vdash s_1 \longrightarrow u_1 \qquad \vdash s_2 \longrightarrow u_2}{\vdash s_1;\, s_2 \longrightarrow u_1;\, u_2}$$

$$\frac{\vdash s_1 \longrightarrow u_1 \qquad \vdash s_2 \longrightarrow u_2}{\begin{array}{ll}\vdash \mathsf{if}\ b & \longrightarrow \quad \mathsf{if}\ b\ \mathsf{goto}\ \ell_1;\\ \quad \mathsf{then}\ s_1 & \quad\ \ u_2;\\ \quad \mathsf{else}\ s_2 & \quad\ \ \mathsf{goto}\ \ell_2;\\ \quad \mathsf{end} & \ell_1\colon u_1;\\ & \ell_2\colon \mathsf{skip}\end{array}}$$ where ℓ_1 and ℓ_2 are fresh

$$\frac{\vdash s \longrightarrow u}{\begin{array}{ll}\vdash \mathsf{while}\ b & \longrightarrow \quad \ell_1\colon \mathsf{ifn}\ b\ \mathsf{goto}\ \ell_2;\\ \quad \mathsf{do}\ s & \quad\ \ u;\\ \quad \mathsf{end} & \quad\ \ \mathsf{goto}\ \ell_1;\\ & \ell_2\colon \mathsf{skip}\end{array}}$$ where ℓ_1 and ℓ_2 are fresh

Figure 6.3: A compositional and context-insensitive translation

In Figure 6.3, the translation judgments are deterministic and therefore the translation of a program always exists and is unique. Using rule induction [124] one can show that the translated program is valid and also that its semantics agrees with the semantics of the source program.

6.3.2 Variations

The translation defined in Figure 6.3 is only one among many in the range of choices for translating conditional commands and while loops. Let us take three examples.

- For conditional commands, the conditional branches could be swapped.
- For while loops, the translation starts with a conditional jump and finishes with an unconditional jump. More compact programs can be obtained by starting with an unconditional jump and finishing with a conditional jump:

  $$\frac{\vdash s \longrightarrow u}{\begin{array}{ll}\vdash \mathsf{while}\ b & \longrightarrow \quad \mathsf{goto}\ \ell_2;\\ \quad \mathsf{do}\ s & \quad\ \ \ell_1\colon u;\\ \quad \mathsf{end} & \quad\ \ \ell_2\colon \mathsf{if}\ b\ \mathsf{goto}\ \ell_1\end{array}}$$ where ℓ_1 and ℓ_2 are fresh

  The two translations differ in the number of steps executed at run-time by the resulting programs.

- For while loops, one could also choose Baskett's technique [9], which duplicates b and mixes the two previous translations:

$$\frac{\vdash s \longrightarrow u}{\begin{array}{ll}\vdash \mathsf{while}\ b \longrightarrow & \mathsf{ifn}\ b\ \mathsf{goto}\ \ell_2;\\ \mathsf{do}\ s & \ell_1\colon u;\\ \mathsf{end} & \mathsf{if}\ b\ \mathsf{goto}\ \ell_1;\\ & \ell_2\colon \mathsf{skip}\end{array}} \quad \text{where } \ell_1 \text{ and } \ell_2 \text{ are fresh}$$

These alternatives embody tradeoffs between the size and the speed of the resulting code. A compiler could decide among them based on compile-time information or on profiling information.

6.3.3 Analysis

A compositional and context-insensitive translation is simple to write and to reason about. By its very nature, however, it does not take into account the contexts of the translated terms and therefore, it gives rise to redundancies, e.g., when sequencing conditional commands and while loops, as illustrated next.

Example 6.3.1 (Sequencing).

$$\begin{array}{lll}\vdash \mathsf{if}\ b_1 & \longrightarrow & \mathsf{if}\ b_1\ \mathsf{goto}\ 1;\\ \mathsf{then}\ a_1 & & a_2;\\ \mathsf{else}\ a_2 & & \mathsf{goto}\ 2;\\ \mathsf{end}; & & 1\colon a_1;\\ \mathsf{while}\ b_2 & & 2\colon \mathsf{skip};\\ \mathsf{do}\ a_3 & & 3\colon \mathsf{ifn}\ b_2\ \mathsf{goto}\ 4;\\ \mathsf{end} & & a_3;\\ & & \mathsf{goto}\ 3;\\ & & 4\colon \mathsf{skip}\end{array}$$

Sequencing a conditional command and a while loop gives rise to a skip *instruction (the one labeled by* 2*) that could be eliminated and to labels (*2 *and* 3*) that could be merged.* □

The translation also gives rise to further spurious skip instructions, e.g., for nested conditional commands, as illustrated next.

Example 6.3.2 (Then-nested conditional commands).

$$\begin{array}{lll}\vdash \mathsf{if}\ b_1 & \longrightarrow & \mathsf{if}\ b_1\ \mathsf{goto}\ 1;\\ \mathsf{then\ if}\ b_2 & & a_3;\\ \quad \mathsf{then}\ a_1 & & \mathsf{goto}\ 2;\\ \quad \mathsf{else}\ a_2 & & 1\colon \mathsf{if}\ b_2\ \mathsf{goto}\ 3;\\ \quad \mathsf{end} & & a_2;\\ \mathsf{else}\ a_3 & & \mathsf{goto}\ 4;\\ \mathsf{end} & & 3\colon a_1;\\ & & 4\colon \mathsf{skip};\\ & & 2\colon \mathsf{skip}\end{array}$$

Nesting a conditional command in the then branch of a conditional command gives rise to a spurious skip *instruction (here, the one labeled with* 4*).* □

Given n then-nested conditional commands, the direct translation generates a chain of n consecutive skip instructions. It can also give rise to chains of jumps, as illustrated next.

Example 6.3.3 (Else-nested conditional commands).

```
⊢ if b1              ⟶      if b1 goto 1;
  then a1                    if b2 goto 3;
  else if b2                 a3;
       then a2               goto 4;
       else a3          3: a2;
       end              4: skip;
  end                        goto 2;
                       1: a1;
                       2: skip
```

Nesting a conditional command in the else branch of a conditional command gives rise to a chain of skip *instructions and unconditional jumps (label* 4 *is part of the chain here).* □

Given n else-nested conditional commands, the direct translation generates a chain of n successive jumps.

Chains of jumps also arise when translating nested while loops, as illustrated next.

Example 6.3.4 (Nested while loops).

```
⊢ while b1           ⟶      1: ifn b1 goto 2;
  do while b2                3: ifn b2 goto 4;
     do while b3             5: ifn b3 goto 6;
        do a                    a;
        end                     goto 5;
     end                   6: skip;
  end                          goto 3;
                        4: skip;
                           goto 1;
                        2: skip
```

Nesting a while loop in a while loop gives rise to a chain of skip *instructions and unconditional jumps (labels* 6 *and* 4 *are part of the chain here).* □

Given n nested while loops, the direct translation generates a chain of n successive jumps. The alternative translations for while loops mentioned in Section 6.3.1 share the same problem.

The redundancies add up when translating more complex code structures, as illustrated next.

Example 6.3.5 (Nested heterogeneous commands).

```
⊢ while b1                ⟶  1: ifn b1 goto 2;
  do a1;                        a1;
     if b2                      if b2 goto 3;
     then while b3              a3;
          do a2                 goto 4;
          end                3:
     else a3                 5: ifn b3 goto 6;
     end                        a2;
  end                           goto 5;
                             6: skip;
                             4: skip;
                                goto 1;
                             2: skip
```

This example illustrates overlapping chains of jumps (the skip *instruction labeled with* 4 *can both be flowed into and jumped to) and redundant aliased labels (*3 *and* 5*).* □

Finding a proper strategy to eliminate overlapping chains of jumps increases the requirements on the post-processing phase. An extra phase is also necessary to eliminate redundant aliased labels.

6.3.4 Chains of jumps

Chains of jumps are a classical issue. Compiler textbooks list two choices:

1. use backpatching [2, Chapter 8]; and
2. have a simple code-generation scheme and eliminate chains of jumps in a post-processing phase, e.g., peephole optimization [2, Chapter 9] or block reordering [100].

Post-unfolding chains of jumps is also known as *transition compression* in partial evaluation [66, Section 4.4].

On one hand, a simple code-generation scheme is easy to maintain and to extend. On the other hand, a post-processing phase is likely to be an expensive part of the compiler, since it requires one to identify the chains of jumps. As for backpatching, it requires one to maintain an additional list structure in the memory, potentially linear in the size of the source program. Ensuring correctness also becomes more difficult.

Other authors [4] propose using code blocks, exploring code traces and reshuffling blocks in order to minimize the amount of redundant jumps. On one hand, such an approach often leads to better code by optimizing the redundancies in the original code. On the other hand, it requires one to design and implement a code-reshuffling strategy.

In the following sections we present and prove a simple translation scheme that avoids generating redundant labels and chains of jumps, does not introduce

additional computational steps, does not require allocation of additional space, and operates in one pass. Furthermore, the generated code is still amenable to code reshuffling and dynamic transition compression, as in Erosa and Hendren's work [39]. Our solution is not unique in the sense that it can be adapted to other translation rules.

6.4 Context awareness

Examining Figure 6.3 makes it clear where chains of jumps can arise. For example, a conditional command is always punctuated with a labeled skip instruction, in order to establish a program point where the conditional branches can join. Therefore, as illustrated by Example 6.3.3, else-nested conditional commands give rise to chains of jumps—one for each joining program point.

6.4.1 Continuations and duplication

To prevent these chains of jumps, and in the spirit of continuations [114], one can parameterize the translation with the label of the next instruction and generate explicit jumps to this label. The resulting translation, however, is not optimal because sequenced atomic commands yield redundant jumps to the next instruction. The problem is reminiscent of administrative redexes introduced by a Plotkin-style CPS transformation [31, 98].

One could then duplicate the translation judgment. A first judgment would hold when the translated command ends with an explicit jump to the label of its next command, and a second would hold when the translated command flows into the next instruction. Duplication, however, is a slippery road because it leads one to uncomfortably large specifications. Alternatively, the translation judgment could be parameterized with an inherited attribute, and this is the topic of the next section.

6.4.2 Towards the right thing

We parameterize the translation judgment with attributes. The judgment relates a structured command s and an unstructured command u.

- Both conditional commands and while loops need a label for the next instruction. We therefore parameterize the judgment with an inherited label for the instruction following u (the 'next' label). On the other hand, atomic commands do not need a next label. We therefore parameterize the judgment with an inherited flag indicating whether u can flow into the next label or whether it must jump to it, and with a synthesized flag indicating whether or not the next label is used in u.
- While loops need a label for the current instruction, but other commands do not. We therefore parameterize the judgment with an inherited label for u (the 'current' label) and with a synthesized flag indicating whether or not the current label is used in u.

Thus equipped, we can specify a translation by combinatorially enumerating all the possible values of the flags. The next labels make it possible to short-cut static chains of jumps and the inherited flags to only declare the labels that are actually used. The translation avoids both chains of jumps and unused labels in translated programs, works in one pass, and is simple to prove correct.

On the other hand, the combinatorial enumeration gives rise to a large number of inference rules. We would rather have a solution with one inference rule per syntactic construct. Such a solution is the topic of the next section.

6.5 A context-sensitive translation

We present a compositional and context-sensitive translation that has one inference rule per syntactic construct and that avoids generating chains of jumps as well as redundant or unused labels.

6.5.1 The translation

Compared to Section 6.4.2, our insight is to combine inherited flags and inherited labels into *qualified labels*, and to reduce the two synthesized flags to one, for the next label.

The qualified current label, qcl, for a translated command u: qcl is inherited. It is either $MAY\ \ell$ or $MUST\ \ell$, for some label ℓ. The label ℓ is associated to the entry point of u. If ℓ is qualified as $MUST$, the translation must declare it at the entry point of u. Otherwise, ℓ is qualified as MAY and, unless necessary (e.g., for a while loop), the translation can ignore ℓ.

Qualified current labels are attached (or not) to translated commands according to the judgment

$$\vdash_{\mathrm{def}} \langle qcl, u\rangle \longrightarrow u'$$

where qcl is a qualified current label and u and u' are unstructured commands. If the qualifier is $MUST$, u is labeled in the result u'. Otherwise, the qualifier is MAY and u is not labeled in u'. This auxiliary judgment thus concerns the definition of labels for commands. It is defined in Figure 6.4.

The qualified next label, qnl, for a translated command u: qnl is inherited. It is either $FLOW\ \ell$ or $JUMP\ \ell$, for some label ℓ. The label ℓ is associated to the instruction following u. If ℓ is qualified as $JUMP$, the translation must generate an explicit jump to it. Otherwise, ℓ is qualified as $FLOW$ and, unless it needs it (e.g., for a conditional command or for a while loop), the translation can let u flow into the next instruction.

The result flag, r, for a translated command u: r is synthesized. It is either $USED$ or $UNUSED$. If it is $USED$, then the command following u must be labeled with the next label.

Qualified next labels are jumped to (or not) in translated commands according to the judgment

$$\vdash_{\text{use}} \langle qnl, u\rangle \longrightarrow \langle u', r\rangle$$

where qnl is a qualified next label, u and u' are unstructured commands, and r is a result flag. If the qualification is $FLOW$, then $u' = u$ and r is $UNUSED$. Otherwise, the qualification is $JUMP$, a jump to the next label is generated, and r is $USED$. This second auxiliary judgment thus concerns the use of a label after a command. It is defined in Figure 6.4.

Our translation is compositional, uses one inference rule per syntactic construct, and relates a structured command s into an unstructured command u. It is parameterized with two inherited attributes, qcl and qnl, and one synthesized attribute, r, and reads as follows:

$$\vdash \langle qcl, s, qnl\rangle \longrightarrow \langle u, r\rangle$$

The judgment is satisfied whenever s translates into u. It uses the two auxiliary judgments defined above.

At the top level, a structured program s is translated into an unstructured program u; u' as follows:

$$\frac{\begin{array}{l}\vdash \langle MAY\ \ell_1, s, FLOW\ \ell_0\rangle \longrightarrow \langle u, r\rangle \\ \vdash_{\text{def}} \langle qcl, \mathsf{stop}\rangle \longrightarrow u'\end{array}}{\vdash s \longrightarrow u;\ u'} \quad \begin{array}{l}\text{where } \ell_0 \text{ and } \ell_1 \text{ are fresh} \\ \text{and } qcl = transfer\ r\ \ell_0\end{array}$$

where *transfer* is defined just below.
N.B.: u' is either stop or ℓ_0 : stop, depending on whether r is $UNUSED$ or $USED$.

We use the following auxiliary functions (*project* is overloaded):

$$\begin{array}{lcl}
transfer : result\text{-}flag \times Lab & \rightarrow & qualified\text{-}current\text{-}label \\
transfer\ UNUSED\ \ell & = & MAY\ \ell \\
transfer\ USED\ \ell & = & MUST\ \ell \\
\\
project : qualified\text{-}current\text{-}label & \rightarrow & Lab \\
project\ (MAY\ \ell) & = & \ell \\
project\ (MUST\ \ell) & = & \ell \\
\\
project : qualified\text{-}next\text{-}label & \rightarrow & Lab \\
project\ (FLOW\ \ell) & = & \ell \\
project\ (JUMP\ \ell) & = & \ell
\end{array}$$

N.B.: In an implementation, since there is no ambiguity, one would probably represent qualified labels as pairs of tags and labels, and overload MAY, $FLOW$ and $UNUSED$ into one tag and $MUST$, $JUMP$ and $USED$ into another tag, at

$$\frac{}{\vdash_{\mathrm{def}} \langle \mathit{MAY}\ \ell, u\rangle \longrightarrow u} \qquad \frac{}{\vdash_{\mathrm{def}} \langle \mathit{MUST}\ \ell, u\rangle \longrightarrow \ell :\ u}$$

$$\frac{}{\vdash_{\mathrm{use}} \langle \mathit{FLOW}\ \ell, u\rangle \longrightarrow \langle u, \mathit{UNUSED}\rangle}$$

$$\frac{}{\vdash_{\mathrm{use}} \langle \mathit{JUMP}\ \ell, u\rangle \longrightarrow \langle (u;\ \mathsf{goto}\ \ell), \mathit{USED}\rangle}$$

$$\frac{\vdash_{\mathrm{def}} \langle qcl, a\rangle \longrightarrow u \qquad \vdash_{\mathrm{use}} \langle qnl, u\rangle \longrightarrow \langle u', r\rangle}{\vdash \langle qcl, a, qnl\rangle \longrightarrow \langle u', r\rangle}$$

$$\frac{\begin{array}{l}\vdash \langle qcl, s_1, \mathit{FLOW}\ \ell_1\rangle \longrightarrow \langle u_1, r_1\rangle \\ \vdash \langle qcl_2, s_2, qnl\rangle \longrightarrow \langle u_2, r_2\rangle\end{array}}{\vdash \langle qcl, (s_1;\ s_2), qnl\rangle \longrightarrow \langle (u_1;\ u_2), r_2\rangle} \quad \begin{array}{l}\text{where } \ell_1 \text{ is fresh} \\ \text{and } qcl_2 = \mathit{transfer}\ r_1\ \ell_1\end{array}$$

$$\frac{\begin{array}{l}\vdash \langle \mathit{MUST}\ \ell_1, s_1, qnl\rangle \longrightarrow \langle u_1, r_1\rangle \\ \vdash \langle \mathit{MAY}\ \ell_2, s_2, \mathit{JUMP}\ \ell\rangle \longrightarrow \langle u_2, r_2\rangle \\ \vdash_{\mathrm{def}} \langle qcl, (\mathsf{if}\ b\ \mathsf{goto}\ \ell_1;\ u_2;\ u_1)\rangle \longrightarrow u\end{array}}{\vdash \langle qcl, \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{end}, qnl\rangle \longrightarrow \langle u, \mathit{USED}\rangle} \quad \begin{array}{l}\text{where } \ell = \mathit{project}\ qnl \\ \text{and } \ell_1 \text{ and } \ell_2 \\ \text{are fresh}\end{array}$$

$$\frac{\begin{array}{l}\vdash \langle \mathit{MAY}\ \ell_3, s, \mathit{JUMP}\ \ell_1\rangle \longrightarrow \langle u, r\rangle \\ \vdash_{\mathrm{def}} \langle \mathit{MUST}\ \ell_1, (\mathsf{ifn}\ b\ \mathsf{goto}\ \ell_2;\ u)\rangle \longrightarrow u'\end{array}}{\vdash \langle qcl, \mathsf{while}\ b\ \mathsf{do}\ s\ \mathsf{end}, qnl\rangle \longrightarrow \langle u', \mathit{USED}\rangle} \quad \begin{array}{l}\text{where } \ell_1 = \mathit{project}\ qcl \\ \ell_2 = \mathit{project}\ qnl \\ \text{and } \ell_3 \text{ is fresh}\end{array}$$

Figure 6.4: A compositional and context-sensitive translation

the price of clarity. Then *transfer* would be the identity function and *project* would be the second projection function.

The inference rules are displayed in Figure 6.4. We describe each of them in words next.

Atomic commands: An atomic command is translated into the same atomic command. Depending on the qualified current label, the resulting atomic command might be labeled. Depending on the qualified next label, it might be followed by a jump to the next label and gives rise to the corresponding synthesized result flag.

Sequences of commands: A sequence of commands is translated into the sequence of the translated commands. The qualified current label is inherited by the translation of the first command, and the qualified next label by the translation of the second command. In the translation of the first command, to reflect sequencing, the qualification of the next label is FLOW.

The result flag synthesized from the translation of the first command is transferred to the translation of the second command as an inherited attribute. The final result flag is inherited from the translation of the second command.

Conditional commands: A conditional command is translated into a conditional jump determining whether to execute the translated then branch or the translated else branch. The translated branches merge into the next label (the else branch must jump to it), and thus their result flags are not used.

While loops: A while loop is translated into a conditional jump labeled by the current label of the translation and determining whether to execute the translated body of the loop or to exit the loop. The current label serves as a next label to translate the body, to close the loop. The next label of the translation serves as the exit point of the loop.

In contrast to the context-insensitive translation of Figure 6.3, the context-sensitive translation of Figure 6.4 does not introduce any skip commands. Moreover, none of the generated unconditional jumps are labeled. The following theorem is straightforward to prove by rule induction [124].

Theorem 6.5.1 (No chains). *Let s be a program in* SCom *such that $\vdash s \longrightarrow u$ is satisfied. Then,*

1. *there exist no commands* goto ℓ *and* $\ell :$ goto ℓ' *in u;*
2. *there exist no commands* if b goto ℓ *and* $\ell :$ goto ℓ' *in u; and*
3. *there exist no commands* ifn b goto ℓ *and* $\ell :$ goto ℓ' *in u.*

Furthermore, the translation of Figure 6.4 generates no unused labels, as stated by the following theorem, which is also straightforward to prove.

Theorem 6.5.2 (All labels are used). *Let s be a program in* SCom *such that $\vdash s \longrightarrow u$ is satisfied. Then, for any label ℓ in u, at least one of the following conditions is true:*

1. *There exists a command* goto ℓ *in u.*
2. *There exists a command* if b goto ℓ *in u.*
3. *There exists a command* ifn b goto ℓ *in u.*

Finally, the translation of Figure 6.4 does not generate redundant aliased labels, as stated by the following theorem, which is also straightforward to prove.

Theorem 6.5.3 (No redundant labels). *Let s be a program in* SCom *such that $\vdash s \longrightarrow u$ is satisfied. Then there exist no labels ℓ_1 and ℓ_2 such that there exists a command $\ell_1 : \ell_2 : u'$ in u.*

6.5.2 Variations

The translation defined in Figure 6.4 is only one among many in the range of choices for translating conditional commands and while loops. For instance, Baskett's translation of while loops [9] can be adapted as follows.

$$\frac{\begin{array}{l}\vdash_{\mathrm{def}} \langle qcl, \mathsf{ifn}\ b\ \mathsf{goto}\ \ell_2\rangle \longrightarrow u_0 \\ \vdash \langle \mathit{MUST}\ \ell_1, s, \mathit{FLOW}\ \ell_3\rangle \longrightarrow \langle u_1, r_3\rangle \\ \vdash_{\mathrm{def}} \langle qcl_3, \mathsf{if}\ b\ \mathsf{goto}\ \ell_1\rangle \longrightarrow u_2\end{array}}{\vdash \langle qcl, \mathsf{while}\ b\ \mathsf{do}\ s\ \mathsf{end}, qnl\rangle \longrightarrow \langle (u_0; u_1; u_2), \mathit{USED}\rangle}$$

where $\ell_2 = \mathit{project}\ qnl$
ℓ_1, ℓ_3 are fresh
and $qcl_3 = \mathit{transfer}\ r_3\ \ell_3$

The alternative translation of while loops from Section 6.3.2, however, does not blend as directly in the present context-sensitive translation. The guilty part is its opening unconditional jump.

6.5.3 Analysis

Let us revisit the examples of Section 6.3.3 and show that no redundancies occur.

Example 6.5.1 (Sequencing, revisited).

$\vdash$ if b_1	$\longrightarrow$	if b_1 goto 3;
then a_1		a_2;
else a_2		goto 2;
end;		3: a_1;
while b_2		2: ifn b_2 goto 0;
do a_3		a_3;
end		goto 2;
		0: stop

In contrast to Example 6.3.1, the translated code contains no additional labels or spurious skip *instructions.* □

Example 6.5.2 (Then-nested conditional commands, revisited).

$\vdash$ if b_1	$\longrightarrow$	if b_1 goto 2;
then if b_2		a_3;
then a_1		goto 0;
else a_2		2: if b_2 goto 4;
end		a_2;
else a_3		goto 0;
end		4: a_1;
		0: stop

In contrast to Example 6.3.2, no spurious skip *instructions have been generated.* □

Example 6.5.3 (Else-nested conditional commands, revisited).

```
⊢ if b1                 ⟶      if b1 goto 2;
  then a1                       if b2 goto 4;
  else if b2                    a3;
       then a2                  goto 0;
       else a3             4: a2;
       end                      goto 0;
  end                      2: a1;
                           0: stop
```

In contrast to Example 6.3.3, all jumps to 0 *are now direct, and the last statement flows directly to* 0. □

Example 6.5.4 (Nested while loops, revisited).

```
⊢ while b1              ⟶      1: ifn b1 goto 0;
  do while b2                   2: ifn b2 goto 1;
     do while b3                3: ifn b3 goto 2;
        do a                       a;
        end                        goto 3;
     end                     0: stop
  end
```

In contrast to Example 6.3.4, no additional labels or spurious skip *instructions have been generated.* □

Example 6.5.5 (Nested heterogeneous commands, revisited).

```
⊢ while b1              ⟶      1: ifn b1 goto 0;
  do a1;                         a1;
     if b2                       if b2 goto 4;
     then while b3               a3;
          do a2                  goto 1;
          end                4: ifn b3 goto 1;
     else a3                     a2;
     end                         goto 4;
  end                       0: stop
```

In contrast to Example 6.3.5, no chains of jumps, no useless labels, and no redundant aliased labels have been generated. □

6.6 Conclusions

We have presented a context-sensitive translation from a language with structured control-flow constructs into a language with unstructured control-flow constructs. The translation is compositional and operates in linear time and

space on the size of the input program. It is defined with one inference rule per syntactic construct and operates in one pass. The resulting program is semantically equivalent to the original program, contains the same atomic commands, and contains no chains of jumps, no unused labels, and no redundant labels. The translation should thus be useful in a JIT compiler for a structured language.

Compiling nested conditional commands naturally gives rise to chains of jumps to join their control flow if the translation is context-insensitive. We have pointed out how to avoid generating these chains of jumps. In SSA terms [5, 74, 122], our translation naturally yields fewer merge points without duplicating contexts. It also generates fewer basic blocks and thus makes it faster to compute an SSA form.

Turning to the CPS transformation [31, 113], we observe that the issue of chains of jumps arises there in the form of spurious η-redexes such as $\lambda v.k\ v$, where k denotes a continuation. These η-redexes appear in the translation of tail calls and for nested conditional expressions, just like here for while loops and nested conditional commands. This coincidence should not come as a surprise since CPS and, more generally, functional programming are known to be connected to SSA [5, 74].

Our closest related work is Dybvig, Hieb, and Butler's destination-driven code generation [38], where commands are also translated based on their context. While destination-driven code generation is not formalized and yields both redundant labels and unreferenced labels, it is defined for a richer source language and has been implemented in the back-end of a Scheme optimizing compiler, where it has been found very effective.

Bibliography

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, January 1999. ACM Press.

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

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

[4] Andrew W. Appel. *Modern Compiler Implementation in {C, Java, ML}*. Cambridge University Press, New York, 1998.

[5] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, April 1998.

[6] 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.

[7] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Maluszyński and Martin Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991. Springer-Verlag.

[8] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, 1984. Revised edition.

[9] Forest Baskett. The best simple code generation technique for WHILE, FOR, and DO loops. *ACM SIGPLAN Notices*, 13(4):31–32, April 1978.

[10] Nick Benton and Philip Wadler. Linear logic, monads and the lambda calculus. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 420–431, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[11] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[12] Anders Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1990. DIKU Rapport 90/17.

[13] Anders Bondorf. Improving binding times without explicit cps-conversion. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 1–10, San Francisco, California, June 1992. ACM Press.

[14] Anders Bondorf, Neil D. Jones, Torben Mogensen, and Peter Sestoft. Binding time analysis and the taming of self-application. Diku rapport, University of Copenhagen, Copenhagen, Denmark, 1988.

[15] 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. Springer-Verlag, 1993.

[16] Baudouin Le Charlier and Pascal Van Hentenryck. A universal top-down fixpoint algorithm. Technical Report CS-92-25, Brown University, Providence, Rhode Island, 1992.

[17] Li-Ling Chen and Williams Ludwell Harrison. An efficient approach to computing fixpoints for complex program analysis. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 98–106. ACM Press, July 1994.

[18] Li-Ling Chen, Williams Ludwell Harrison, and Kwangkeun Yi. Efficient computation of fixpoints that arise in complex program analysis. *Journal of Programming Languages*, 3(1):31–68, 1995.

[19] Chris Clack and Simon L. Peyton Jones. Strictness analysis—a practical approach. In Jean-Pierre Jouannaud, editor, *Proceedings of the Second International Conference on Functional Programming and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 35–49, Nancy, France, September 1985. Springer-Verlag.

[20] Charles Consel and Olivier Danvy. For a better support of static data flow. In John Hughes, editor, *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 496–519, Cambridge, Massachusetts, August 1991. ACM, Springer-Verlag.

[21] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, January 1991. ACM Press.

[22] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[23] 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. ACM Press, January 1977.

[24] 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.

[25] Daniel Damian. Time stamps for fixed-point approximation. In Michael Mislove, editor, *Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics*, volume 45 of *Electronic Notes in Theoretical Computer Science*, Aarhus, Denmark, May 2001. Elsevier Science.

[26] Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis. In Philip Wadler, editor, *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*, pages 209–220, Montréal, Canada, September 2000. ACM Press.

[27] Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis: On the impact of the CPS transformation. Technical report, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, July 2001. Extended version of [26].

[28] Damian Daniel and Olivier Danvy. Static transition compression. In Walid Taha, editor, *Proceedings of the Second Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2001)*, number 2196 in Lecture Notes in Computer Science, Firenze, Italy, September 2001. Springer-Verlag.

[29] Olivier Danvy. Semantics-directed compilation of non-linear patterns. *Information Processing Letters*, 37:315–322, March 1991.

[30] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

[31] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[32] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, 1995. An earlier version appeared in the proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.

[33] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 8(6):730–751, 1996.

[34] Olivier Danvy and Lasse R. Nielsen. Compositionality in the CPS transformation. Technical report, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, April 2001.

[35] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International Conference on Principles and Practice of Declarative Programming*, pages 378–395, Firenze, Italy, September 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[36] Olivier Danvy and Lasse R. Nielsen. A higher-order colon translation. In Herbert Kuchen and Kazunori Ueda, editors, *Fifth International Symposium on Functional and Logic Programming*, number 2024 in Lecture Notes in Computer Science, pages 78–91, Tokyo, Japan, March 2001. Springer-Verlag. Extended version available as the technical report BRICS RS-00-33.

[37] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[38] R. Kent Dybvig, Robert Hieb, and Tom Butler. Destination-driven code generation. Technical Report 302, Computer Science Department, Indiana University, Bloomington, Indiana, February 1990.

[39] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In Henri Bal, editor, *Proceedings of the Fifth IEEE International Conference on Computer Languages*, pages 229–240, Toulouse, France, May 1994. IEEE Computer Society Press.

[40] Christian Fecht and Helmut Seidl. An even faster solver for general systems of equations. In Radhia Cousot and David A. Schmidt, editors, *Proceedings of 3rd Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 189–204, Aachen, Germany, September 1996. Springer-Verlag.

[41] Christian Fecht and Helmut Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In Chris Hankin, editor, *Proceedings of the 7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 90–104, Lisbon, Portugal, March 1998. Springer-Verlag.

[42] Andrzej Filinski. Representing monads. In Boehm [11], pages 446–457.

[43] Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996. Technical Report CMU-CS-96-119.

[44] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

[45] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.

[46] Richard P. Gabriel. LISP: Good news, bad news, how to win big. *AI Expert*, 6(6):30–39, June 1991.

[47] Kirsten L. Solberg Gasser, Flemming Nielson, and Hanne Riis Nielson. Systematic realisation of control flow analyses for CML. In Mads Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 38–51, Amsterdam, The Netherlands, June 1997. ACM Press.

[48] Timothy G. Griffin. A formulae-as-types notion of control. In Paul Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.

[49] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [11], pages 458–471.

[50] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, pages 507–541, 1997. Extended version available as the technical report BRICS RS-96-34.

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

[52] Nevin Heintze. Set-based program analysis of ML programs. In Talcott [115], pages 306–317.

[53] Nevin Heintze. Control-flow analysis and type systems. In Mycroft [85], pages 189–206.

[54] 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.

[55] Nevin Heintze and Jon G. Riecke. The slam calculus: Programming with secrecy and integrity. In Luca Cardelli, editor, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 365–377, San Diego, California, January 1998. ACM Press.

[56] Fritz Henglein. Simple closure analysis. Technical Report Semantics Report D-193, DIKU, Computer Science Department, University of Copenhagen, 1992. Accessible from `ftp://ftp.diku.dk/diku/users/henglein`.

[57] Fritz Henglein. Syntactic properties of polymorphic subtyping. Technical Report Semantics Report D-293, DIKU, Computer Science Department, University of Copenhagen, May 1996.

[58] Carsten K. Holst and Carsten K. Gomard. Partial evaluation is fuller laziness. 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 223–233, New Haven, Connecticut, June 1991. ACM Press.

[59] John Hughes. Why functional programming matters. In David A. Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley, 1990.

[60] Sebastian Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice*. PhD thesis, Department of Computing, Imperial College of Science Technology and Medicine, London, UK, 1991.

[61] 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.

[62] Suresh Jagannathan, Andrew Wright, and Stephen Weeks. Type-directed flow analysis for typed intermediate languages. In Pascal Van Hentenryck, editor, *Static Analysis*, number 1302 in Lecture Notes in Computer Science, pages 232–249, Paris, France, September 1997. Springer-Verlag.

[63] Mark Scott Johnson and Ravi Sethi, editors. *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 1986. ACM Press.

[64] Neil D. Jones. What *not* to do when writing an interpreter for specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 216–237, Dagstuhl, Germany, February 1996. Springer-Verlag.

[65] Neil D. Jones, Carsten K. Gomard, Anders Bondorf, Olivier Danvy, and Torben Æ. Mogensen. A self-applicable partial evaluator for the lambda calculus. In K. C. Tai and Alexander L. Wolf, editors, *Proceedings of the 1990 IEEE International Conference on Computer Languages*, pages 49–58, New Orleans, Louisiana, March 1990.

[66] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/pebook.html`.

[67] Neil D. Jones and Flemming Nielson. Abstract interpretation: A semantics-based tool for program analysis. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 527–636. Oxford University Press, 1995.

[68] Niel D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs. In Johnson and Sethi [63], pages 296–306.

[69] Niels Jørgensen. Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration. In Baudouin Le Charlier, editor, *Static Analysis*, number 864 in Lecture Notes in Computer Science, pages 329–345, Namur, Belgium, September 1994. Springer-Verlag.

[70] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In Johnson and Sethi [63], pages 86–96.

[71] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, January 1977.

[72] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[73] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[74] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In Michael Ernst, editor, *ACM SIGPLAN Workshop on Intermediate Representations*, SIGPLAN Notices, Vol. 30, No 3, pages 13–22, San Francisco, California, January 1995. ACM Press.

[75] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[76] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Talcott [115], pages 227–238.

[77] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, June 1985. Springer-Verlag.

[78] Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–363, 1992.

[79] Torben Æ. Mogensen. Self-applicable partial evaluation for the pure lambda calculus. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Research Report 909, Department of Computer Science, Yale University, pages 116–121, San Francisco, California, June 1992.

[80] Torben Æ. Mogensen. Partial evaluation: Concepts and applications. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 1–19, Copenhagen, Denmark, July 1998. Springer-Verlag.

[81] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[82] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[83] Chetan R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1990.

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

[85] Alan Mycroft, editor. *Second International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, Glasgow, Scotland, September 1995. Springer-Verlag.

[86] Lasse R. Nielsen. *A study of defunctionalization and continuation-passing style*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001.

[87] Flemming Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[88] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[89] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In Neil D. Jones, editor, *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 332–345, Paris, France, January 1997. ACM Press.

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

[91] Richard A. O'Keefe. Finite fixed-point problems. In Jean-Louis Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, pages 729–743. The MIT Press, 1987.

[92] Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–363, 1993.

[93] Jens Palsberg. Comparing flow-based binding-time analyses. In Peter Mosses, Mogens Nielsen, and Michael Schwartzbach, editors, *Proceedings of TAPSOFT '95*, number 915 in Lecture Notes in Computer Science, pages 561–574, Aarhus, Denmark, May 1995. Springer-Verlag.

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

[95] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings of the OOPSLA '91 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[96] Jens Palsberg and Michael I. Schwartzbach. Binding-time analysis: Abstract interpretation versus type inference. In Henri Bal, editor, *Proceedings of the Fifth IEEE International Conference on Computer Languages*, pages 289–298, Toulouse, France, May 1994. IEEE Computer Society Press.

[97] Jens Palsberg and Mitchell Wand. CPS transformation of flow information. Unpublished manuscript, available at `http://www.cs.purdue.edu/~palsberg/publications.html`, June 2001.

[98] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[99] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.

[100] M. V. S. Ramanath and Marvin H. Solomon. Jump minimization in linear time. *ACM Transactions on Programming Languages and Systems*, 6(4):527–545, 1984.

[101] John C. Reynolds. On the relation between direct and continuation semantics. In Jacques Loeckx, editor, *2nd Colloquium on Automata, Languages and Programming*, number 14 in Lecture Notes in Computer Science, pages 141–156, Saarbrücken, West Germany, July 1974. Springer-Verlag.

[102] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, 1993.

[103] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[104] Mads Rosendahl. Higher-order chaotic iteration sequences. In Maurice Bruynooghe and Jaan Penjam, editors, *Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming*, number 714 in Lecture Notes in Computer Science, pages 332–345, Tallinn, Estonia, August 1993. Springer-Verlag.

[105] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, December 1993.

[106] Amr Sabry and Matthias Felleisen. Is continuation-passing useful for data flow analysis? In Vivek Sarkar, editor, *Proceedings of the ACM SIG-*

PLAN'94 Conference on Programming Languages Design and Implementation, SIGPLAN Notices, Vol. 29, No 6, pages 1–12, Orlando, Florida, June 1994. ACM Press.

[107] Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, 19(6):916–941, November 1997.

[108] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1&2):131–170, October 1996.

[109] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[110] David A. Schmidt. Natural-semantics-based abstract interpretation. In Mycroft [85], pages 1–18.

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

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

[113] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[114] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, 2000. Reprint of the technical monograph PRG-11, Oxford University Computing Laboratory (1974).

[115] 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.

[116] Frank Tip and Jens Palsberg. Scalable propagation-based call graph cconstruction algorithms. In *Proceedings of the Fifteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 281–293, Minneapolis, Minnesota, October 2000. Published as ACM SIGPLAN Notices, volume 35, number 10.

[117] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit. DIKU Rapport 97/12, University of Copenhagen, Copenhagen, Denmark, 1997.

[118] Philip Wadler. The essence of functional programming (invited talk). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

[119] Mitchell Wand. Embedding type structure in semantics. In Mary S. Van Deusen and Zvi Galil, editors, *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 1–6, New Orleans, Louisiana, January 1985. ACM Press.

[120] Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics*, number 598 in Lecture Notes in Computer Science, pages 294–311, Pittsburgh, Pennsylvania, March 1991. Springer-Verlag.

[121] Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(32):365–387, 1993.

[122] Mark N. Wegman and F. Ken Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 3(2):181–210, 1991.

[123] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.

[124] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.

[125] Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, 1998.

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

[127] Steve Zdancewic and Andrew Myers. Secure information flow and CPS. In David Sands, editor, *Proceedings of the Tenth European Symposium on Programming*, number 2028 in Lecture Notes in Computer Science, pages 46–61, Padova, Italy, April 2001. Springer-Verlag.

Recent BRICS Dissertation Series Publications

DS-01-5 **Daniel Damian.** ***On Static and Dynamic Control-Flow Information in Program Analysis and Transformation*****. August 2001. PhD thesis. xii+111 pp.**

DS-01-4 **Morten Rhiger.** ***Higher-Order Program Generation*****. August 2001. PhD thesis. xiv+146 pp.**

DS-01-3 **Thomas S. Hune.** ***Analyzing Real-Time Systems: Theory and Tools*****. March 2001. PhD thesis. xii+265 pp.**

DS-01-2 **Jakob Pagter.** ***Time-Space Trade-Offs*****. December 2001. PhD thesis. xii+83 pp.**

DS-01-1 **Stefan Dziembowski.** ***Multiparty Computations — Information-Theoretically Secure Against an Adaptive Adversary*****. January 2001. PhD thesis. 109 pp.**

DS-00-7 **Marcin Jurdziński.** ***Games for Verification: Algorithmic Issues*****. December 2000. PhD thesis. ii+112 pp.**

DS-00-6 **Jesper G. Henriksen.** ***Logics and Automata for Verification: Expressiveness and Decidability Issues*****. May 2000. PhD thesis. xiv+229 pp.**

DS-00-5 **Rune B. Lyngsø.** ***Computational Biology*****. March 2000. PhD thesis. xii+173 pp.**

DS-00-4 **Christian N. S. Pedersen.** ***Algorithms in Computational Biology*****. March 2000. PhD thesis. xii+210 pp.**

DS-00-3 **Theis Rauhe.** ***Complexity of Data Structures (Unrevised)*****. March 2000. PhD thesis. xii+115 pp.**

DS-00-2 **Anders B. Sandholm.** ***Programming Languages: Design, Analysis, and Semantics*****. February 2000. PhD thesis. xiv+233 pp.**

DS-00-1 **Thomas Troels Hildebrandt.** ***Categorical Models for Concurrency: Independence, Fairness and Dataflow*****. February 2000. PhD thesis. x+141 pp.**

DS-99-1 **Gian Luca Cattani.** ***Presheaf Models for Concurrency (Unrevised)*****. April 1999. PhD thesis. xiv+255 pp.**

DS-98-3 **Kim Sunesen.** ***Reasoning about Reactive Systems*****. December 1998. PhD thesis. xvi+204 pp.**