Static and Dynamic Semantics Processing
(Detailed abstract)

Charles Consel and Olivier Danvy
Research Report YALEU/DCS/RR-761
June, 1990

This work is supported by the Darpa grant N00014-88-K-0573.

Static and Dynamic Semantics Processing (Detailed abstract) *

Charles Consel † Olivier Danvy ‡

June 21, 1990

Abstract

This paper presents a step forward in the use of partial evaluation for *interpreting* and *compiling* programs, as well as for *automatically generating a compiler* from executable denotational definitions of programming languages.

We determine the static and dynamic semantics of the programming language, reduce the expressions representing the static semantics, and generate object code by instantiating the expressions representing the dynamic semantics. By processing the static semantics of the language, compiling programs is performed. By processing the semantics of the partial evaluator, generating compilers is achieved. The correctness of the compiler is guaranteed by the correctness of both the executable specification and our partial evaluator.

The results reported in this paper improve on previous work in the domain of compiler generation [14, 28, 37], and solves several open problems in partial evaluation [15]. In essence:

- Our compilation goes beyond a mere *syntax-to-semantics mapping* since the static semantics gets processed at compile time by partial evaluation. It is really a *syntax-to-dynamic-semantics mapping.*
- Because our partial evaluator is self-applicable, *a compiler is actually generated.*
- Our partial evaluator handles *higher-order functions*, and *non-flat binding time domains.* Our source programs are enriched with an *open-ended set of algebraic operators.*

Our experiment parallels the one reported in [22]: starting with the same denotational semantics of an Algol subset, we obtain the same good results, but entirely automatically and using the original semantics only, instead of writing several others, which requires proving their congruence. We are able to compile strongly typed, Algol-like programs and to *process their static semantics at compile-time* (scope resolution, storage calculation, and type checking), as well as to generate the corresponding compiler completely automatically.

Object code is reasonably efficient. It has been found to be about twenty times faster than the interpreted source program. The compiler is well-structured and efficient. The static semantics is still processed at compile time. Compiling using this compiler is twelve times faster than compiling by partially evaluating the interpreter with respect to the source program.

Keywords

Compiler generation, semantics, partial evaluation, self-application, binding times, Algol, Scheme

---

*(Draft). Technical Report 761, Computer Science Department, Yale University.

†Yale University, P.O. Box 2158, New Haven, CT 06520, USA (consel-charles@yale.edu)

‡Kansas State University, Manhattan, KS 66506, USA (danvy@ksuvax1.cis.ksu.edu)

1 Introduction

Existing semantics-directed compiler generators essentially amount to a syntax-to-semantics mapping [14, 32, 28]. They map the representation of programs as abstract syntax trees into the representation of their meaning as lambda-expressions. Lacking a static/dynamic distinction in the semantic specification, it is not clear how to simplify the resulting lambda-expressions [24]. For this reasons compile-time actions are likely to be performed at runtime, thereby impeding the performances of the whole system.

In this perspective, using a self-applicable partial evaluator is an obvious choice. As a static semantics processor, a partial evaluator ensures the static semantics of a program to be processed at compile-time. Viewing the valuation functions as a definitional interpreter [24, 30, 35, 37], compiling a program is achieved by specializing its interpreter. Further, generating a compiler is achieved by specializing the partial evaluator with respect to the interpreter [2]. Of course, the partial evaluator needs to be powerful enough.

We have solved a series of open problems in partial evaluation [15]. This makes it possible to match the requirements that have been found necessary in semantics-directed compiler generation [28]. Our partial evaluator tackles higher-order Scheme programs [29] with an open-ended set of algebraic operators and non-flat binding time domains. Also, it can specialize itself and thus generate compilers automatically.

This paper illustrates this step forward with the compilation of Algol-like programs and the automatic derivation of a stand-alone compiler from an executable specification of this Algol-like language. Compiling includes reducing the expressions representing the static semantics and therefore goes beyond a mere syntax-semantics mapping. The compiler is stand-alone and therefore optimizes the compilation process. The whole static semantics of Algol is processed at compile time: syntax analysis, scope resolution, storage calculation, and type checking. Running the object code is twenty times faster than interpreting the source code. Compiling an Algol program using the stand-alone compiler is twelve times faster than compiling by specializing the definitional interpreter of Algol with respect to the Algol program.

We compile Algol into low level Scheme, with explicit store and properly typed operators. Figure 1 displays the source and object code of the factorial program. The Algol source program is written with a while loop and an accumulator. There is no explicit type declaration. The target program is a specialized version of the interpreter with respect to the source program. It is written in Scheme because the interpreter is written in Scheme. It computes the factorial of 5 because the source program computes the factorial of 5. The main procedure is passed a store and updates it during the computation. The

```
block                    (letrec ([evProgram1 (lambda (s)
  { n int 5; r int 1;}                          (loop2 (intUpdate 1 5 (intUpdate 0 1 s))))]
  { while n > 0                  [loop2 (lambda (s)
    do                                  (if (gtInt (fetchInt 1 s) 0)
      r := n * r;                           (let ([s1 (intUpdate 0 (mulInt (fetchInt 1 s) (fetchInt 0 s)) s)])
      n := n - 1;                             (loop2 (intUpdate 1 (subInt (fetchInt 1 s1) 1) s1)))
    od; }                                   (initCcont s)))])
end                       evProgram1)
```

Figure 1: Source and object code of the factorial program.

while loop has been mapped to a tail-recursive procedure iterating on the store. All the continuations of the original continuation semantics except the initial one have disappeared: the target program is in direct style. All the locations have been computed at compile-time (variables `r` and `n` at locations 0 and 1, respectively). There is no type-checking at run-time: all the injection tags have disappeared and all the operators are properly typed. Essentially we obtain a front-end compiler, mapping syntax to a lambda-expression representing the dynamic semantics. Representing this lambda-expression with assembly language instructions is out of scope here, but is naturally achieved by a realistic program transformation such as the one reported in [20].

Our experiment parallels the one reported in [22], where a compiler is derived by hand, which necessitates (1) to introduce three semantics and proving their congruence to make it possible to process the static semantics and (2) to introduce combinators and using the compiling algorithm of [36] to generate object code. In contrast, self-applicable partial evaluation offers a unified framework for semantics-directed compiler generation. The static and dynamic semantics are determined by analyzing the *binding times* [17] of the executable specification. Compile time and run time combinators are automatically extracted, based on the binding time information [10]. The compiling algorithm is provided by the partial evaluator. We can experiment with the executable specification before turning it into a compiler. We can experiment with compiling by specializing the same executable specification. We generate a compiler by self-application with respect to the same executable specification.

Three reasons motivate this subset of Algol. It is small enough for its complete description to fit in a paper, and yet significant enough to highlight the effectiveness of our treatment. It is precisely the same as in [22] and thus our treatment can be compared directly with the one in [22]. It is very simple to extend, *e.g.*, with procedures, and we have actually done it. The semantics gets more voluminous and the results are still as good.

This paper is organized as follows. Section 2 presents an overview of the source language specification. Section 3 describes how this specification is analyzed, and how its static and dynamic semantics are processed. Section 4 addresses the actual compiling process. Section 5 describes the generation of a stand-alone compiler. Section 6 compares our results with related works. Finally our approach is put into perspective.

2 Semantic Definition: An Executable Specification

Denotational semantics definitions can be seen as executable specifications by transliterating their valuation functions into functional programs that act as definitional interpreters [15, 30, 37]. This makes it convenient to experiment with these specifications in a purely functional framework before turning them into compilers. Our specification language is a side-effect free dialect of Scheme. In particular, because we consider continuation semantics, the transliterations are evaluation-order independent [30] and thus are not tailored to run in Scheme only.

However, Scheme is not enough: its capacity for data abstraction is limited by its ground data types (pairs, vectors, *etc.*) unless we use Church-like data abstractions, which is sound but often cumbersome. We want to specify our data types algebraically. For example, figure 2 displays the abstract syntax of our Algol subset and its concrete definition.

Figure 3 displays two algebraic specifications defining the locations and the store. This makes it possible to target their actual representations and to limit our transformations to *what* these operators

```
⟨Program⟩    ::= ⟨Block⟩
⟨Block⟩      ::= block { ⟨DeclList⟩ } { ⟨StmtList⟩ } end
⟨DeclList⟩   ::= empty | ⟨Declaration⟩ ; ⟨DeclList⟩
⟨Declaration⟩::= ⟨Ident⟩ ⟨Expr⟩
⟨StmtList⟩   ::= empty | ⟨Stmt⟩ ; ⟨StmtList⟩
⟨Stmt⟩       ::= ⟨Block⟩ | ⟨Ident⟩ := ⟨Expr⟩
                 | while ⟨Expr⟩ do ⟨StmtList⟩ od
                 | if ⟨Expr⟩ then ⟨Stmt⟩ else ⟨Stmt⟩
⟨Expr⟩       ::= ⟨Constant⟩ | ⟨Ident⟩
                 | ⟨Expr⟩ ⟨AritBinop⟩ ⟨Expr⟩
                 | ⟨Expr⟩ ⟨RelBinop⟩ ⟨Expr⟩
⟨Constant⟩   ::= int ⟨Int⟩ | real ⟨Real⟩ | bool ⟨Bool⟩
⟨AritBinop⟩  ::= + | - | / | *
⟨RelBinop⟩   ::= < | > | =
```

```
(defineType Program block)

(defineType Declaration ident expr)

(defineType Expr
  (Int value) (Real value) (Bool value)
  (Identifier ident)
  (AritBinop op expr1 expr2)
  (RelBinop op expr1 expr2)
  (RelUnop op expr))

(defineType Statement
  (Block declList stmtList)
  (While expr stmtList)
  (If expr stmt1 stmt2)
  (Assign ident expr))
```

Figure 2: The abstract syntax and its concrete declaration

do (not interfering with *how* they are implemented). In all the figures, domains and type constructors are overlined, underlined or accented with a tilde depending on their static properties, as explained in section 3.

Figure 4 presents the definition of the semantic domains and the types of the valuation functions. The valuation functions are displayed in figures 5 and 8.

In essence, our source specification is the denotational semantics of [22]. It is runnable, modular, and expressed in a side-effect free subset of Scheme extended with abstract data types.

3 Determining Static and Dynamic Semantics

Traditionally, to derive a compiler from a semantic definition, one first has to determine its static semantics. Then, the soundness of the static semantics is proved manually. This process is generally agreed to be very difficult and error prone [18, 28].

In contrast, our approach consists in automatically analyzing the semantic definition to determine its static properties. This process is achieved by *binding time analysis* [5, 8, 17, 26]. Essentially, this phase splits the definition of a language into two parts: a safe approximation of the static semantics (the usual compile time actions) and the dynamic semantics. By analogy with the traditional approach, binding time analysis can be seen as a theorem prover: for each syntactic construct, a set of inference rules is defined to infer the static properties; the starting axioms state that the program is static (available at compile time) and the store is dynamic (not available until run time).

Before giving an example of the kind of reasoning performed by the binding time analysis, let us

Domain $m \in \overline{Mappings}$
Operations

initMappings	$: \overline{Mappings}$
addMapping	$: \overline{Ident \times Location \times Mappings \rightarrow Mappings}$
fetchLoc	$: \overline{Ident \times Mappings \rightarrow Location}$

Domain $s \in Store$
Operations

initStore	$: \underline{Store}$
intUpdate	$: \overline{IntLoc} \times \underline{Int} \times \underline{Store} \rightarrow \overline{Store}$
realUpdate	$: \overline{RealLoc} \times \underline{Real} \times \underline{Store} \rightarrow \underline{Store}$
BoolUpdate	$: \overline{BoolLoc} \times \underline{Bool} \times \underline{Store} \rightarrow \underline{Store}$
fetchInt	$: \overline{IntLoc} \times \underline{Store} \rightarrow \underline{Int}$
fetchReal	$: \overline{RealLoc} \times \underline{Store} \rightarrow \underline{Real}$
fetchBool	$: \overline{BoolLoc} \times \underline{Store} \rightarrow \underline{Bool}$

Figure 3: Location and Store algebras.

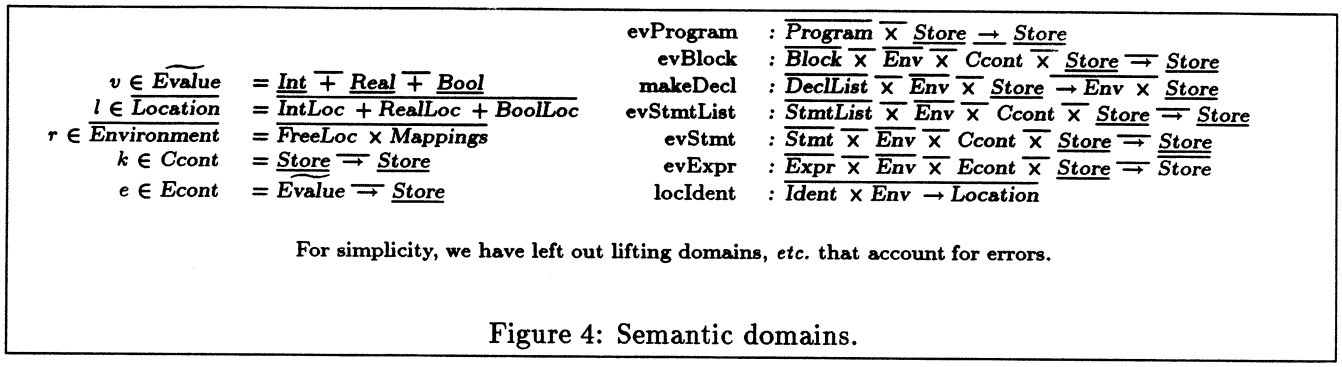

Figure 4: Semantic domains.

point out that the domains and the type constructors, displayed in figure 4, are overlined or underlined depending on how they account for static or dynamic computations, respectively. These constructors are accented with a tilde when they represent partially static data, *i.e.*, structured data made of both static and dynamic parts (or recursively of partially static data). For simplicity, we have not annotated continuations, though they are a good example of partially static, higher-order values. The annotations represent the deductions achieved by the binding time analysis; they will support the following reasoning.

Property 1 *Every valuation function is static in its program argument.*

Initially the program is static. Because the semantic definition respects the denotational assumption [32], *i.e.*, it is compositional, the meaning of a sentence is solely defined in terms of its proper subparts. The original motivation for the denotational assumption was to enable structural induction over abstract syntax trees. Presently, compositionality implies staticness: no abstract syntax tree is ever built. Furthermore, assuming that the arguments of the valuation functions are used consistently, *i.e.*, a variable is uniquely bound to elements of the same domain, no dynamic argument interfere with a program argument. Therefore, the staticness of a program argument is guaranteed in every valuation function. □

Property 1 is at the basis of compiling by partial evaluation. It has numerous consequences.

Property 2 *Storage calculation and location types are completely static.*

All the identifiers of the program are static since they are part of the program (by property 1). Because storage calculation only depends on identifiers, this operation is completely static (*cf.* function locIndent in figure 4). □

Property 3 *In the representation of an expressible value, the injection tag is static.*

Let us consider the domain *Evalue*, defined as a disjoint sum of basic values. As such, this sum is used to perform type checking. Operationally, this domain is implemented as a cartesian product. Its elements hold the injection tag and the actual value. The injection tag is static because it is induced by the program text (static by property 1) or the location type (static by property 2). □

Once static properties of language definitions have been automatically determined, we can perform static and dynamic processing. In addition to the present application, static properties are useful both from a language design and from an implementation point of view: they give precise and safe bases to reason about this language.

4 Processing Static and Dynamic Semantics

The static and dynamic properties of a language definition determine *what* to process at compile time and at run time. This section focuses on *how* this is done. We describe specializing the executable specification with respect to a program, based on the binding time information.

4.1 Compiling binding time information

According to the binding time information, this phase determines which partial evaluation *action* (that is, program transformation) is to be performed during specialization for each expression of the definition. This greatly simplifies and improves the specialization phase, as introduced and discussed in [10]. Indeed, the binding time information of a given expression does not have to be analyzed repeatedly to determine which partial evaluation action to perform. This has been done statically in the present phase.

The main partial evaluation actions denote the following treatments. Standard evaluation – *Ev*: the expression only manipulates available data. Reproduction of the expression verbatim – *Id*: no data is available. Reduction – *Red*: the outermost syntactic construct can be reduced. Rebuilding – *Reb*: the outermost syntactic construct has to be rebuilt, but sub-components have to be partially evaluated.

The actions are defined for each syntactic construct of the meta-language. Those described above capture the usual program transformations for partial evaluation of first order functional programs as described in [7, 33]. This set of actions is extended in [9] to handle higher order functions and structured data.

4.2 Extracting static and dynamic combinators

Actions can be exploited further for extracting two sets of combinators representing the *purely static* and *purely dynamic* semantics of a language definition. An *Ev*-combinator is extracted from an expression solely annotated with *Ev*, and an *Id*-combinator is extracted from an expression solely annotated with *Id* [10].

These combinators are actually capturing an instruction set to compile – the *Ev*-combinators – and an instruction set to execute a program – the *Id*-combinators. In the present specification, an *Ev*-combinator will perform compile time storage calculation and some *Id*-combinators will perform run time storage management.

4.3 Specialization

At this stage, we have reduced specialization to executing the partial evaluation actions. The specializer is implemented as a simple processor for these actions. This processor is perfectly suited for self-application.

4.4 Summary

Separating the static and dynamic semantics of partial evaluation has been found to be crucial for generating compilers by self-application, not only for generating a compiler and but also for running a generated compiler [18]. With respect to simplicity, orthogonality, and efficiency, our treatment as given in this section largely improves earlier results. For example, extracting combinators usually reduces the size of source programs sharply, yielding smaller programs that are faster to specialize and, correspondingly, smaller and faster compilers [10]. This observation has been confirmed again in this Algol experiment.

5 Generating a Compiler

Generating a compiler is achieved by specializing the partially evaluator with respect to the executable specification. This is realized straightforwardly because our partial evaluator is self-applicable. What we specialize is the processor for partial evaluation actions presented in section 4.3. We specialize it with respect to the preprocessed executable specification, encoded with partial evaluation actions.

In essence, all the static semantics of the specializer is processed at compiler generation time, yielding a residual program dedicated to processing the static semantics of the language and emitting object code representing its dynamic semantics – in other terms: a stand-alone compiler. Beyond its theoretical interest and its conceptual elegance, self-application pays off: compiling using the compiler is twelve times faster than compiling by specializing the executable specification.

6 Comparison with related work

6.1 Semantics-directed compiler generation

[35] asserts it firmly: denotational semantics specifications may have the format of a program, but programs they are not – they are mathematical objects. Showing less certainty, [23] derived a compiler from an interpreter using Peter Landin's Applicative Expressions. On the observation that λ-expressions *represent* mathematical functions, the first semantics-directed compiler generator was built [24]. From then on, a number of systems were developed [14].

From the point of view of partial evaluation, which in essence processes static semantics [32], existing semantics-directed compiler generators suffer from the same problem of not processing the static semantics for fear of looping [24]. The call-by-need strategy of Paulson's compiler generator [27] often delays compile-time computations until runtime, which clearly is unsatisfactory. The toolbox approach illustrated by SPS [37] stresses the problem of mere composition of tools: they make compilers compilers without actual speedups. What is expected from a compiler is that it processes the static semantics of a language. Yet as sound as it is, PSI [25] does not unfold static fixpoints and has no partially static structures. In contrast, the micro and macro semantics of MESS [21] characterize the static and the dynamic semantics of a language and ensure the static semantics to be processed at compile time, even though distinguishing between micro and macro semantics relies on the initiative of the user. More recently, [38] introduces a log Γ recording static information about earlier static reductions. The corresponding change in the semantics (type information located in the log instead of the runtime store) can be propagated directly using partial evaluation.

Semantics-directed compiler generation systems share the same goal, and ultimately use the same methods. The trends, as analyzed in Uwe Pleban's POPL'87 tutorial [28], are to use semantic algebras instead of λ-terms; to improve the software engineering of systems; to manage a tradeoff between generality and efficiency; and to choose existing functional languages as specification languages. Since this tutorial, the open problem of including specifications of flow analysis and optimization transformations has been solved [26].

On the other hand, the very format of denotational semantics may be criticized [31]. This is not our point here. We want to illustrate progress in partial evaluation with the classical example of compiling and compiler generation.

Higher-order programming constructs match the expressive power needed for in semantics-directed compiler generation (intensional reason). Partial evaluation captures semantics-directed compiler generation from interpretive specifications (extensional reason). Therefore it makes sense to use the new generation of partial evaluators to solve problems like the turning of non-trivial interpreters into realistic compilers.

6.2 Partial evaluation

A thorough overview of partial evaluation can be found in [2, 13]. On the side of functional programming, the activity has been concentrated on strenghtening self-applicable specializers, with the notable exception of [1], where a powerful specializer is reported. It treats arbitrary numeric-heavy scientific programs, in contrast with our carefully formulated specifications of programming languages [19]. This partial evaluator is not self-applicable and is targeted for "data independent programs", *i.e.*, it stops specializing at dynamic conditional expressions.

Building on top of strengthened self-applicable specializers [6, 7, 4], the barrier of higher-orderness has been teared down in 1989 [16, 3, 8]. Further progress in tackling higher-order constructs and non-flat binding time domains on a unified basis [8] and enhancing the actual treatment of the static and dynamic semantics [10] has led us to the results reported here.

7 Conclusion and Issues

The general mechanism of compilation and compiler generation – reducing expressions representing the static semantics and emitting residual expressions representing the dynamic semantics – was captured extensionally in the definition of a partial evaluator: partially evaluating an interpreter with respect to a program amounts to compiling this program. Previous experiences in semantics-directed compiler generation established the need for modularity, for algebras, for handling higher-order constructs, *etc.* in source specifications.

This paper illustrates the intensional use of partial evaluation where source programs are specified modularly and algebraically with higher-order constructs and non-flat binding time domains. The example is an executable specification of a block-structured, strongly typed language. We automatically derive a compiler where all the static semantics – scope resolution, storage calculation, type checking, *etc.* – is reduced at compile-time. The compiler inherits the structure of the partial evaluator, and the object programs inherit the structure of the interpreter. Our whole systems run in Scheme: partial evaluators, interpreters, compilers, and object programs (though they are parameterized by the semantic algebras of the specializer and of the interpreter).

This work was possible due to a series of breakthroughs starting with separating the static and dynamic semantics of specialization and including: binding time analysis for higher-order constructs and non-flat binding time domains [8] and combinator extraction [10].

Present works address tackling pattern matching [12] and Prolog [11]. Future works will include developing a better programming environment; better extensional criteria for the quality of a source specification and their implementation; parameterizing post-optimizers; and the automatic generation of congruence relations and correctness proofs, both from the binding time analysis and from the actual specialization.

Acknowledgements

To Neil D. Jones for his scientific insight as to how to tame self-application. To Mitchell Wand and Margaret Montenyohl for providing the Algol example and for their encouragements. Thanks are also due to Karoline Malmkjær, David Schmidt, Austin Melton, François Bodin, Pierre Jouvelot, Siau Cheng Khoo, Paul Hudak, and Andrzej Filinski for their thoughtful comments.

References

[1] A. Berlin. Partial evaluation applied to numerical computation. In *ACM Conference on Lisp and Functional Programming*, 1990.

[2] D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation.* North-Holland, 1988.

[3] A. Bondorf. Automatic autoprojection of higher order recursive equations. In N. D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[4] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. Diku Research Report 90/04, University of Copenhagen, Copenhagen, Denmark, 1990. To appear in Science of Computer Programming.

[5] A. Bondorf, N. D. Jones, T. Mogensen, and P. Sestoft. Binding time analysis and the taming of self-application. Diku report, University of Copenhagen, Copenhagen, Denmark, 1988.

[6] C. Consel. New insights into partial evaluation: the Schism experiment. In H. Ganzinger, editor, *ESOP'88, 2nd European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 236–246. Springer-Verlag, 1988.

[7] C. Consel. *Analyse de Programmes, Evaluation Partielle et Generation de Compilateurs.* PhD thesis, Université de Paris VI, Paris, France, 1989.

[8] C. Consel. Binding time analysis for higher order untyped functional languages. In *ACM Conference on Lisp and Functional Programming*, 1990.

[9] C. Consel. *The Schism Manual.* Yale University, New Haven, Connecticut, USA, 1990. Version 1.0.

[10] C. Consel and O. Danvy. From interpreting to compiling binding times. In N. D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.