J. Functional Programming 3 (3): 315–346, July 1993 © 1993 Cambridge University Press

Efficient analyses for realistic off-line partial evaluation

ANDERS BONDORF AND JESPER JØRGENSEN

DIKU, Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
(*e-mail*:anders@diku.dk and knud@diku.dk)

Abstract

Based on Henglein's efficient binding-time analysis for the lambda calculus (with constants and 'fix') (Henglein, 1991), we develop three efficient analyses for use in the preprocessing phase of Similix, a self-applicable partial evaluator for a higher-order subset of Scheme. The analyses developed in this paper are almost-linear in the size of the analysed program. (1) A *flow analysis* determines possible value flow between lambda-abstractions and function applications and between constructor applications and selector/predicate applications. The flow analysis is not particularly biased towards partial evaluation; the analysis corresponds to the closure analysis of Bondorf (1991b). (2) A (monovariant) *binding-time analysis* distinguishes static from dynamic values; the analysis treats both higher-order functions and partially static data structures. (3) A new *is-used analysis*, not present in Bondorf (1991b), finds a non-minimal binding-time annotation which is 'safe' in a certain way: a first-order value may only become static if its result is 'needed' during specialization; this 'poor man's generalization' (Holst, 1988) increases termination of specialization. The three analyses are performed sequentially in the above mentioned order since each depends on results from the previous analyses. The input to all three analyses are constraint sets generated from the program being analysed. The constraints are solved efficiently by a normalizing union/find-based algorithm in almost-linear time. Whenever possible, the constraint sets are partitioned into subsets which are solved in a specific order; this simplifies constraint normalization. The framework elegantly allows expressing both forwards and backwards components of analyses. In particular, the new is-used analysis is of backwards nature. The three constraint normalization algorithms are proved correct (soundness, completeness, termination, existence of a best solution). The analyses have been implemented and integrated in the Similix system. The new analyses are indeed much more efficient than those of Bondorf (1991b); the almost-linear complexity of the new analyses is confirmed by the implementation.

Capsule review

This paper concerns the design and verification of efficient flow analysis algorithms and their use in the Similix partial evaluation system. Specifically, it gives a formal account of four analyses used in the Similix preprocessor, three replacing algorithms already in the distributed version, and a fourth new analysis that improves the termination of partial evaluation.

The success of Similix hinges on the use of sophisticated flow analyses, in particular *binding-time analysis* (bta). Recently, Henglein published an efficient bta algorithm based on constraint solving. This paper shows that Henglein's algorithm scales up and also that the same technique can be used for other flow analyses. The new flow analysis algorithms have much better complexity; the original algorithms had worse than quadratic time complexity

(in the size of the program), but the new analyses are almost linear. This complexity reduction translates into a saving of between 2 and 20 in both time and space for a typical mix of problems.

The paper demonstrates that semantics-based analyses can be applied to realistic problems—yet another example of the fruitful interplay between theory and practice that characterises so much of the work in this area.

1 Introduction

1.1 Binding-time analysis

Off-line partial evaluators basically consist of two phases, a *preprocessing* phase and a *specialization* phase. The preprocessing phase does not know the actual static (known) values with respect to which the source program is specialized; it only knows the text of the program being specialized and a classification of which input values will become static and which will become dynamic (unknown). The preprocessing phase called *binding-time analysis* annotates source expressions as 'always reducible' or 'not reducible'. For example, on the basis of the user supplied static/dynamic classification of input values, the binding-time analysis may determine that some expression `(+ x y)` in the analysed program is always reducible. It may also determine that some other expression `(* a b)` is *not* reducible.

In general, generating a finite and best binding-time annotated source program (where as many operations as possible are reducible) is undecidable because the binding-time analysis cannot know what tests of conditionals evaluate to. Therefore, binding-time analysis must be *approximative*.

Off-line specialization seems to have appeared first in the first self-applicable partial evaluator *Mix* (Jones *et al.*, 1985, 1989). In Mix, preprocessing consisted of one phase only, binding-time analysis. The more recent *Similix* partial evaluator (Bondorf and Danvy, 1991; Bondorf, 1991a, 1991b) has a more complex preprocessing consisting of several (sub-)phases, with binding-time analysis being the central one.

Several partial evaluators have used binding-time analysis (Jones *et al.*, 1985; Bondorf and Danvy, 1991; Consel, 1990; Launchbury, 1991; Gomard and Jones, 1991). The traditional Mix-based approach is to do binding-time analysis by *abstract interpretation*, exemplified by the three distribution systems Mix (Jones *et al.*, 1989), Similix (Bondorf and Danvy, 1991; Bondorf, 1991a) and Schism (Consel, 1990).

Another approach is binding-time analysis based on *type inference* (Schmidt, 1988; Nielson and Nielson, 1988; Gomard and Jones, 1991). The latter approach is both elegant and expressive, but only one partial evaluator has been using it, the experimental Lambda-mix (an elegant system due to its simplicity, but not very useful for practical experiments).

1.2 Binding-time analysis by efficient constraint normalization

Henglein showed how to do type-inference based (monovariant, see Section 1.3) binding-time analysis efficiently (Henglein, 1991) for the language of Lambda-mix, the lambda calculus with constants and an explicit fixed point operator. The inference system is re-expressed as a set of *constraints* that captures local binding-time requirements. Then a constraint solver is used to find a consistent, minimal binding-time classification of variables and expressions. In contrast to the traditional abstract interpretation based methods, the algorithm runs in almost *linear* time; this has been one of the main motivations for the work presented in this paper where we scale up Henglein's ideas to a realistic partial evaluator.

Another motivation is that type inference elegantly expresses binding times that are determined by the *contexts* in which values may occur. This seems much more difficult to express in a forwards analysis (abstract interpretation), as witnessed for instance by the complex 'raise' operation used in the binding-time analysis of Similix (Bondorf, 1991a). The purpose of 'raise' is to annotate lambda-expressions: the annotation of a lambda-expression depends on all possible uses of the expression (to see whether beta-reduction is always possible), so context information is crucial here. Type-inference analyses are bidirectional, so context dependency (backwards flow) is expressed just as easily as forwards flow; this eliminates the need for 'hacks' like the raise operation. The advantages of doing binding-time analysis by efficient constraint normalization are several:

1. Elegance: the analyses are simpler to present to other people. In the abstract interpretation approach, the formulation of an analysis intermingles the *problem* and the *solution*; this makes it hard to understand the analysis. In the constraint-based approach, problem and solution are separated. We believe this gives a much more elegant description.
2. Expressiveness: bidirectionality is easily expressed.
3. Speed of implementation: almost-linear run time.

1.3 Similix

Similix 4.0 is a partial evaluator for a subset of Scheme with higher-order functions and side-effects on global variables. The system uses several preprocessing phases (for an overview, see Bondorf, 1991b). The three bottleneck phases are *closure analysis*, *binding-time analysis* and *evaluation-order dependency analysis* which all run in worse than quadratic time (the closure analysis in worse than cubic time). The closure analysis is used to trace flow of higher-order functions. The bt-analysis classifies operations into static (reducible) and dynamic (non-reducible). The eod-analysis detects potentially evaluation-order dependent expressions (such expressions arise because of the side-effects); this information is used to prevent inverting the order of evaluation-order dependent (residual) expressions by unsafe unfoldings of let-expressions where the actual parameter is evaluation-order dependent. The analyses are *monovariant*: only one annotation is associated with each variable and expression in the program.

In this work we use efficient constraint normalization to redevelop these three analyses. We do not cover the eod-analysis in this paper; we refer to the extended version of the paper (Bondorf and Jørgensen, 1993).

1.4 Separate flow analysis needed

When analysing higher-order functional languages, information on how higher-order values flow is usually needed; this is indeed the case for bt-analysis. In Henglein's bt-analysis, the information about flow of higher-order values is contained in the types: his analysis implicitly performs a flow analysis.

In Henglein's analysis, type information about a function is completely lost if the function becomes dynamic. In Lambda-mix, which Henglein's analysis was designed for, no flow information is needed about dynamic entities, so the information loss is no problem there.

In the context of side-effects, some flow information about a function is needed even if the function is dynamic. For example, when analysing a higher-order application (E_0 E^*) where E_0 is dynamic (implying that the application will not be beta-reduced during specialization), it is necessary to know whether the resulting residual expression may be evaluation-order dependent. Whether this is the case depends on the functional values that E_0 may evaluate to. Thus, not only the bt-analysis but also the eod-analysis needs flow information.

In Similix 4.0, the closure analysis provides such information: it identifies which lambda-expressions may flow into E_0. Hence, by looking at the bodies of the lambda-expressions in this set, it can be decided whether expression (E_0 E^*) is potentially evaluation-order dependent. Notice that whether (E_0 E^*) is evaluation-order dependent is independent of whether E_0 is dynamic or not.

Since knowledge of flow of higher-order values is needed independently of the binding times, we cannot scale up Henglein's analysis directly. We instead perform a separate *flow analysis* followed by what we henceforth refer to as the bt-analysis. Both the bt-analysis and the eod-analysis (which is performed after the bt-analysis) then have access to flow information, but notice that collecting flow information is only done once: in the flow analysis. Neither the bt-analysis nor the eod-analysis needs to collect any flow information.

The flow analysis corresponds to the closure analysis of Similix 4.0: it traces higher-order flow (and, since we here also treat partially static data structures, flow of constructed values). As we shall see later, the flow analysis does not collect sets of closures, but simply equates information from different points of the analysed program, for instance the information associated with the topmost expression of the bodies of all lambdas that may flow together; this information turns out to be sufficient for our purpose. Henglein has also described a flow analysis based on his bt-analysis (Henglein, 1992).

1.5 Poor man's generalization

A major problem in partial evaluation is termination: often, partial evaluation fails to terminate. Holst proposed a way to increase termination called 'poor man's generalization' (Holst, 1988). The idea is to *generalize* (make dynamic) static values that are not used to determine control. Termination is increased this way because the generalized static values could have taken on infinitely many values at specialization time, had they not been generalized. By generalizing, static information is of course lost at specialization time; however, only 'insignificant' static values (not used to determine control) are generalized, so only this insignificant information is lost. By discarding insignificant static information, poor man's generalization in general improves the quality of residual code (independently of the termination issue): different specialized versions of code are only generated when needed.

The method is 'poor man's' because it in no way guarantees termination, it only makes the partial evaluator terminate more often. However, some very typical cases are caught this way, for instance the 'counter problem' where some counter is maintained in a recursion. This problem is mentioned in Katz and Weise (1992) where an idea related to poor man's generalization is used to increase termination for an *on-line* partial evaluator (a partial evaluator without separate preprocessing).

A classical example where poor man's generalization makes specialization terminate is specialization of the following program fragment (written in Scheme syntax (IEE, 1990)) with x dynamic and y static:

```
... (f ... 0) ...
(define (f x y) (if (= x 0) y (f (- x 1) (+ y 1))))
```

A memoizing specializer such as Mix or Similix 4.0 would generate an infinite sequence of specialized functions:

```
... (f-0 ...) ...
(define (f-0 x) (if (= x 0) 0 (f-1 (- x 1))))
(define (f-1 x) (if (= x 0) 1 (f-2 (- x 1))))
(define (f-2 x) ...)
 ⋮
```

Poor man's generalization would generalize y in which case specialization terminates and yields the original program.

It is clear that to obtain poor man's generalization, a bt-analysis must know all contexts in which a static value may be used. It may only generalize the value if all of these contexts are insignificant. This kind of dependency is straightforward to express in the constraint-based analysis framework, and we have exploited this by adding a new *is-used* analysis that tackles the poor man's generalization problem.

1.6 Prerequisites

Knowledge of partial evaluation is needed (for example as described in Jones *et al.*, 1989, or Gomard and Jones 1991). The reader should also know Henglein's paper on efficient bt-analysis (Henglein, 1991). To fully comprehend the technicalities of the paper, Similix-specific knowledge is an advantage: the basics are described in Bondorf and Danvy (1991) and Bondorf (1991a). The manual (Bondorf, 1991b) describes the system in an introductory non-formal way, and the paper (Bondorf, 1992) contains a short (formal) description of the specializer phase of Similix.

2 Preliminaries

2.1 Programming language

A program is a set of recursive procedures (functions) written in the Similix core language, a subset of Scheme (IEE, 1990) augmented with user-defined n-ary constructors; see Figure 1.

An expression is a constant (atomic value or quoted expression) K, a variable V, a conditional (`if` E_1 E_2 E_3), a let-expression (`let` ((V E_1)) E_2), a sequence operation (`begin` E_1 E_2), a primitive (e.g. arithmetic) operation (O E^*), a constructor application (C E^*), a selector application (C-M E_1) (the associated constructor and the field are specified), a predicate application (C? E_1) ('is this a value constructed by constructor C ?'), a first-order call to a top-level defined named procedure (P E^*), a lambda-abstraction (`lambda` (V^*) E) or a higher-order application (E E^*).

Constructors belong to constructor families, thus effectively providing 'disjoint sum of product' types (sum of constructor types). Constructors belonging to the same family have the same associated type. We shall use the term *constructed values* for values generated by evaluating constructor applications.

Primitive operations may be applied to constructed values and function values, but they may only return such values if they were supplied as arguments; this restriction ensures that the analyses 'know' the sources of constructed values and function values. The different application forms are kept syntactically distinct; the distinction is made automatically during parsing.

$\Pi \in$ Program ; D $\in$ Definition ; E $\in$ Expression ; K $\in$ Constant ; V $\in$ Variable ;
O $\in$ PrimopName ; C $\in$ ConstrName ; M $\in$ Field ; P $\in$ ProcName ;

Π ::= D^*
D ::= (`define` (P V^*) E)
E ::= K | V | (`if` E_1 E_2 E_3) | (`let` ((V E_1)) E_2) |
    (`begin` E_1 E_2) | (O E^*) | (C E^*) | (C-M E_1) | (C? E_1) |
    (P E^*) | (`lambda` (V^*) E) | (E E^*)

Fig. 1. Scheme Subset

2.2 Constraint systems

Every program point will be associated with a number of unique *constraint variables*. As program points we shall use the variables and expressions (expression occurrences) of a program. Constraint variables may range over many different kinds of information, for instance binding times. A constraint system C is a set of constraints $op^{\star}(t_1, \ldots, t_n)$ on terms $t_1, \ldots, t_n$ that (may) contain constraint variables. We use $op^{\star}$ as a *syntactic* operator between constraint variables and op as the corresponding semantic operator between values (Henglein makes a similar distinction by adding a '?' on top of an operator to make it syntactic; our '$\star$' corresponds to Henglein's '?'). What op ranges over varies from analysis to analysis. One operator has a fixed meaning, the equality operator $=^{\star}$.

We are searching for a *solution* to a constraint system. A solution is a *value substitution* ρ. We say that ρ satisfies constraint $op^{\star}(t_1, \ldots, t_n)$ if and only if relation $op(\rho(t_1), \ldots, \rho(t_n))$ holds. Substitution ρ is a solution to constraint system C if and only if ρ satisfies all constraints c in C.

A constraint system is solved by a series of *rewritings* into a *normalized system* that immediately can be solved. Constraints of form $\alpha{=}^{\star}\ldots$ are always solved by substituting $\ldots$ for constraint variable α everywhere in the constraint system; that is, the *elementary* substitution $\sigma = [\alpha \mapsto \ldots]$ is applied to the constraint system. The solution ρ is obtained by composing the value substitution that solves the normalized constraint system with (the composition of) the elementary substitutions generated (and applied) during rewriting.

For each analysis, we specify several parts: definitions of relevant domains for the analysis; definitions of constraint operators $op^{\star}$; constraint generation from the syntax of the analysed program; constraint-system rewrite rules used to solve the generated constraint system.

2.3 *Miscellaneous notation*

To denote that a constraint variable is associated with expression E, we index the variable: α_{E}. Similarly, a constraint variable associated with a program variable V is denoted by α_{V}. We use α_{P} to denote the constraint variable associated with the body expression of top-level defined procedure P, and similarly we use α_{P_i} for formal procedure parameter V_i of P.

In figures displaying constraint generation, we implicitly assume 'for all indices' when writing constraints where some parts have index subscripts; index ranges are implicitly defined by context. For an example, constraint $\alpha_{\mathsf{E}_i}{=}^{\star}\ldots$ actually stands for a series of constraints (separated by commas) parameterized over E_i. The range of E_i is given by context.

We use the notation $\langle\ldots\rangle_i$ to build a tuple. The tuple generated by $\langle\ldots\rangle_i$ has the same length as the size of the range of index i. The range for index i is given by context; for example, i might be defined indirectly through a term E_i whose range in turn is given by context.

Let D be a domain and m a natural number. Then D^m is shorthand for the cartesian product domain $D \times \ldots \times D$ where D appears m times.

In the constraint-normalization rewriting systems, we use $c\ldots$ when we need to refer to constraint $\ldots$ as a whole (we can then simply refer to the variable c). This notation is equivalent to the 'as' symbol in languages with pattern matching (such as ML).

The sum type associated with a constructor C is denoted by τ(C).

3 System overview

Figure 2 contains an overview of the interconnections between the three analyses (flow, bt, is-used). The input is a program Π, the output an annotated program Π'. We use the symbol ϕ for flow constraint variables and we use β for bt-constraint variables.

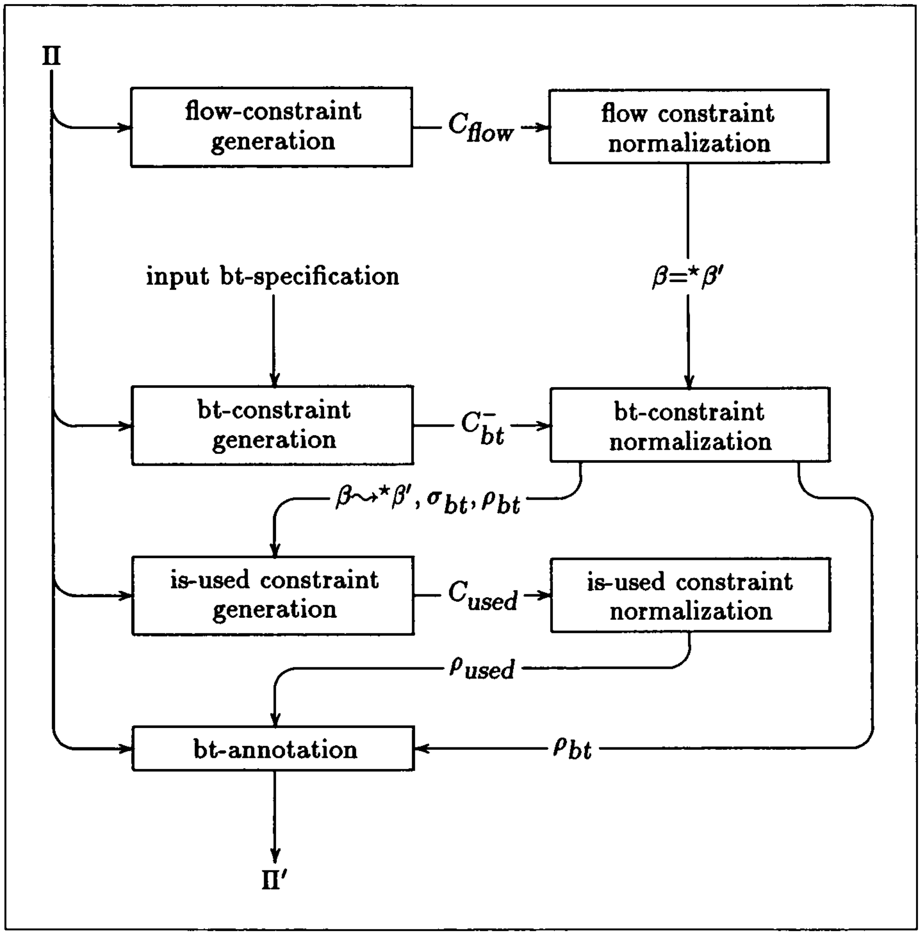
Π
flow-constraint generation
C_{flow}
flow constraint normalization
input bt-specification
$\beta =^* \beta'$
bt-constraint generation
C^-_{bt}
bt-constraint normalization
$\beta \rightsquigarrow^* \beta', \sigma_{bt}, \rho_{bt}$
is-used constraint generation
C_{used}
is-used constraint normalization
ρ_{used}
bt-annotation
ρ_{bt}
Π'

Fig. 2. Overview

The flow constraint set C_{flow} is generated by a one-pass compositional (recursive descent) run over the program. The flow analysis generates two sets of equality constraints, between bt-variables (constraints of form $\beta =^* \beta'$) and between eod-variables (these are not displayed here; see Bondorf and Jørgensen, 1993).

The point is that neither the bt-analysis nor the eod-analysis then needs to do any subsequent flow analysis. The equalities generated by the flow analysis already contain full information about how actual parameters to application expressions may flow to formal parameters of lambda-expressions, which lambda-bodies that may be the result of applications, and which argument expressions to constructor expressions that may be the result of a subsequent selector application. For example, if the analysed program contains an expression

```
E1 = ((if ... (lambda (...) E2) ...) ...)
```

the flow analysis generates an equality constraint between the bt-variables describing the binding time of (the return value of) E_1 and binding time of (the return value of) of E_2; a similar eod-equality constraint is also generated.

The bt-constraint generation phase generates bt-constraints by a one-pass compositional run over the program and it generates additional constraints for the bt-variables associated with the program's input variables. The latter constraints are generated from a user supplied input bt-specification that specifies which program inputs are static and which are dynamic.

The input (referred to as C_{bt} later) to bt-constraint normalization is both the output C^{-}_{bt} from the bt-constraint generation phase and the set of equality constraints generated by the flow analysis. The bt-analysis generates a minimal substitution ρ_{bt} which could be used directly for generating a *safe* annotation. An annotation is safe if and only if the specializer cannot commit any bt-tag projection errors (Gomard and Jones, 1991); this in turn means that the tags can be avoided since they are statically determined at bt-analysis time. There is a close analogy to type security: a well-typed program 'can't go wrong'; therefore, no type tags are needed at run time.

Due to poor man's generalization, it is not the minimal bt-substitution that is desired: static first-order values that are never used in a significant way (to determine control, cf. Section 1.5) will be generalized, that is, made dynamic instead of static. This generalization must be consistent: the resulting bt-substitution must still satisfy the normalized bt-constraint system that was used to find the minimal bt-substitution ρ_{bt}. The constraints that are not trivially satisfied by the generalization process turn out to have form $\beta \rightsquigarrow^{*} \beta'$; also, the substitutions σ_{bt} and ρ_{bt} built during bt-analysis are needed to obtain a consistent generalization. Details will be given in Section 6. The is-used analysis produces a substitution ρ_{used} that is used for deciding what to generalize. The program Π is finally annotated yielding Π'. The complete preprocess of Similix also contains additional analyses, one of which is the eod-analysis; these analyses will not be discussed in this paper.

4 Flow analysis

The purpose of the flow analysis is to trace flow of constructed values and function values. We shall not restrict ourselves to describe only strongly typed programs in the usual sense known from e.g. ML (we are dealing with Scheme, a dynamically typed language), so we want to be able to describe potential flow of several constructed values and several function values into the same program point. This motivates the definition of *FlowVal* in Figure 3: a flow value f is a pair of fields containing information about constructed values and function values.

The constructor field is a finite tuple capturing that constructed values constructed

$$
\begin{array}{llll}
f & \in \mathit{FlowVal} & = \mathit{CstrValField} \times \mathit{FctValField} & \\
cf & \in \mathit{CstrValField} & = \mathsf{C}_1\text{-}\mathit{CstrVal}_\perp \times \ldots \times \mathsf{C}_m\text{-}\mathit{CstrVal}_\perp & \\
cv_{\mathsf{C}} & \in \mathsf{C}\text{-}\mathit{CstrVal} & = \mathit{ArgVal}^{\mathit{arity}(\mathsf{C})} & \text{; for all } \mathsf{C}\in\{\mathsf{C}_1\ldots\mathsf{C}_m\} \\
\mathit{ff} & \in \mathit{FctValField} & = 0\text{-}\mathit{FctVal}_\perp \times \ldots \times \mathit{FA}\text{-}\mathit{FctVal}_\perp & \\
fv_{\mathsf{n}} & \in \mathsf{n}\text{-}\mathit{FctVal} & = \mathit{ArgVal}^{\mathsf{n}} \times \mathit{ResultVal} & \text{; for all } \mathsf{n}\in\{0\ldots \mathit{FA}\} \\
av & \in \mathit{ArgVal} & = \mathit{FlowVal} \times \mathit{BtVar} & \\
rv & \in \mathit{ResultVal} & = \mathit{FlowVal} \times \mathit{BtVar} & \\
 & & & \\
\phi & \in \mathit{FlowVar} & & \\
\beta & \in \mathit{BtVar} & & \\
\rho_{\mathit{flow}} & \in \mathit{FlowValSub} & = \mathit{FlowVar} \to \mathit{FlowVal} & \\
\sigma_{\mathit{flow}} & \in \mathit{FlowVarSub} & = \mathit{FlowVar} \to \mathit{FlowVar} & \\
 & & & \\
c & \in \mathit{FlowConstraint} & = \mathit{Flow}\text{-}{=}^* + \mathit{Flow}\text{-}{<}^*_{\mathsf{C}} + \mathit{Flow}\text{-}{<}^*_{\mathsf{n}} & \\
 & \mathit{Flow}\text{-}{=}^* & = \mathit{FlowVar} \times \mathit{FlowVar} & \\
 & \mathit{Flow}\text{-}{<}^*_{\mathsf{C}} & = (\mathit{FlowVar} \times \mathit{BtVar})^{\mathit{arity}(\mathsf{C})} \times & \text{; constructor args.} \\
 & & \quad \mathit{FlowVar} & \\
 & \mathit{Flow}\text{-}{<}^*_{\mathsf{n}} & = (\mathit{FlowVar} \times \mathit{BtVar})^{\mathsf{n}} \times & \text{; function args.} \\
 & & \quad (\mathit{FlowVar} \times \mathit{BtVar}) \times & \text{; function result} \\
 & & \quad \mathit{FlowVar} &
\end{array}
$$

Fig. 3. Flow analysis: domains

by different constructors can flow into the same program point. The tuple domain *CstrValField* contains one field for each constructor $\mathsf{C}_1 \ldots \mathsf{C}_m$ used in the analysed program Π. If no constructors are present in the program at all, the tuple degenerates to a unit-domain with $\perp$ being the only element. For each program Π, we consider a finite family of domains C_1-*CstrVal*...C_m-*CstrVal*. The field corresponding to constructor $\mathsf{C} \in \{\mathsf{C}_1, \ldots, \mathsf{C}_m\}$ in the tuple domain *CstrValField* is of type C-$\mathit{CstrVal}_\perp$.

The values in C-$\mathit{CstrVal}_\perp$ are interpreted as follows: if no value constructed by constructor C can ever flow into the program point being described, then the value is $\perp$ (the least element). If some value constructed by constructor C *may possibly* flow into the program point, then the value is in C-*CstrVal*. At any program point, we are not interested in distinguishing between different values constructed by the same constructor C. Therefore each domain C-*CstrVal* contains only one description for the arguments to the constructor. The domains C-*CstrVal* are domains of tuples of lengths equal to the constructor arities.

The domain *FctValField* is similar to *CstrValField*. Different function arities play the same role in *FctValField* as different constructors do in *CstrValField*: there is one field in the tuple for each function arity 0...*FA* used in the analysed program Π. We write elements of a domain n-*FctVal* as 'arguments $\to$ result', in accordance with standard notation in type inference; but notice that the domains n-*FctVal* are cartesian product domains, not function domains.

Flow of simple values (integers, booleans, etc.) is not traced as we are not interested in that information; such flow could be analysed by extending *FlowVal* to contain more fields.

If we had just been interested in the flow analysis for its own sake, argument and result fields would just be flow variables. However, we want to collect flow information for the bt-analysis: we have therefore added bt-variables to argument- and result-information. In Bondorf and Jørgensen (1993), eod-variables are also present.

A constraint is either equality between flow variables ($=^\star$-constraints) or a constraint that expresses flow of constructor or function values ($\prec^\star_\mathrm{C}$- and $\prec^\star_\mathrm{n}$-constraints). We use infix notation for the three kinds of flow constraints, as exemplified in Figure 4. The definition of the relations $\prec_\mathrm{C}$ for constructor values and $\prec_\mathrm{n}$ for function values follows (where $\downarrow$ is used to project on product domains):

Definition 1 (Flow relations)

1. $cv_\mathrm{C} \prec_\mathrm{C} f \;\equiv\; cv_\mathrm{C} = (f \downarrow \mathit{CstrValField}) \downarrow \mathrm{C}\text{-}\mathit{CstrVal}$
   (it always holds that cv_C has form $av_1 \dots av_{arity(\mathrm{C})}$).
2. $fv_\mathrm{n} \prec_\mathrm{n} f \;\equiv\; fv_\mathrm{n} = (f \downarrow \mathit{FctValField}) \downarrow \mathrm{n}\text{-}\mathit{FctVal}$
   (it always holds that fv_n has form $av_1 \dots av_\mathrm{n} \rightarrow rv$). □

Flow-constraint generation is specified in Figure 4; the figure shows the constraints generated for the different expression forms. The complete set of flow constraints is obtained by generating constraints for every (sub-)expression is the analysed program Π.

Notice from Figure 4 and from Definition 1 that the flow analysis is equality-based, not inclusion-based. Using equality is quite conservative as the flow values of for instance the branches of a conditional then influence each other. A more precise analysis would use inclusion (with appropriate changes of the definitions of *ArgVal* and *ResultVal* to be sets of $(\mathit{FlowVal} \times \mathit{BtVar})$): such an analysis would correspond closely to the closure analysis of Sestoft (1988) (and that of Bondorf, 1991a). Whether it is important to get the improved precision provided by an inclusion-based analysis is not clear to us; equality-based analyses are common in type inference, and the experiments we have done so far indicate that the equality-based analysis suffices in practice. The motivation for using equality is efficiency: the equality-based flow analysis can be done in almost-linear time (see Section 9) whereas an inclusion-based analysis is expected to be at least cubic.

Let us now inspect Figure 4 in more detail. No flow constraints are generated for constant expressions as these generate neither constructed nor function values. If the expression is a variable, the flow value of the expression is simply that of the variable. For a conditional, the flow values of the branches and of the whole conditional must be equal (which, as mentioned above, is conservative). The rules for let- and begin-expressions are straightforward. The rule for primitive operations reflects the restriction on primitives mentioned in Section 2.1: the only possible constructed or function values that may be returned are those given as arguments.

Constructor expressions are the source of constructed values; the overbars on top of some bt-variables will be discussed in Section 5. Selector expressions may return whatever is described by the selected M'th field of the argument value; pairs of fresh 'dummy' nodes $(\phi_\mathrm{i}, \beta_\mathrm{i})/(\phi_\mathrm{j}, \beta_\mathrm{j})$ are generated for all other fields than the

E = K	: {}
E = V	: $\{\phi_E =^* \phi_V\}$
E = (if E₁ E₂ E₃)	: $\{\phi_E =^* \phi_{E_2}, \phi_E =^* \phi_{E_3}\}$
E = (let ((V E₁)) E₂)	: $\{\phi_{E_1} =^* \phi_V, \phi_E =^* \phi_{E_2}\}$
E = (begin E₁ E₂)	: $\{\phi_E =^* \phi_{E_2}\}$
E = (O E₁ ... Eₙ)	: $\{\phi_E =^* \phi_{E_i}\}$
E = (C E₁ ... Eₙ)	: $\{\langle(\phi_{E_i}, \overline{\beta}_{E_i})\rangle_i <^*_C \phi_E\}$
E = (C-M E₁)	: $\{\langle(\phi_i, \beta_i)\rangle_i +\!\!+ \langle(\phi_E, \beta_E)\rangle +\!\!+ \langle(\phi_j, \beta_j)\rangle_j <^*_C \phi_{E_1}\}$ where $1 \leq i \leq M-1$, $M+1 \leq j \leq$ *arity*(C)
E = (C? E₁)	: {}
E = (P E₁ ... Eₙ)	: $\{\phi_E =^* \phi_P, \phi_{E_i} =^* \phi_{P_i}\}$
E = (lambda (V₁ ... Vₙ) E₁)	: $\{\langle(\phi_{V_i}, \beta_{V_i})\rangle_i \rightarrow (\phi_{E_1}, \overline{\beta}_{E_1}) <^*_n \phi_E\}$
E = (E₀ E₁ ... Eₙ)	: $\{\langle(\phi_{E_i}, \overline{\beta}_{E_i})\rangle_i \rightarrow (\phi_E, \beta_E) <^*_n \phi_{E_0}\}$

Fig. 4. Flow analysis: constraint generation

M'th one (++ denotes tuple-concatenation and $\langle(\phi_E, \beta_E)\rangle$ is a one-element tuple). No constraints are generated for predicate expressions: with respect to E, predicate expressions are comparable to constant expressions; with respect to the argument E_1, the treatment is comparable to the treatment of the test of a conditional.

Procedure calls straightforward. Lambda-expressions are the source of function values, and the applications dually 'consume' function values.

4.1 Normalizing the constraints

The rewrite system in Figure 5 specifies the rules used to solve the flow constraint system. Rule 1 solves equality by substitution. The rules 2 and 3 reduce the number of $<^*_C$- and $<^*_n$-constraints and generate flow-equality and bt-equality constraints instead. The intuition behind the rules is simple: $<^*_C$- and $<^*_n$-constraints specify equality with a component of ϕ (cf. Definition 1); the rules follow by transitivity of equality. Of the generated equality constraints, only the flow-equality constraints are solved during flow-constraint normalization. The generated bt-equality constraints are supplied to the bt-analysis as described in Section 3.

In Section 7.1 we prove that the rewrite system in Figure 5 correctly solves *any* flow constraint system (not necessarily generated from Figure 4): the system is *sound* (does not introduce false solutions), *complete* (does not discard any solutions), and *terminating*. Notice from Figure 2 that we are not interested in the minimal solution, a flow substitution ρ_{flow}. A minimal solution exists (provided the $\beta =^* \beta'$ constraints are solved, see Section 7.1), but the interesting output are the generated bt-equality constraints: the net effect that interests us in the end is the bt-annotation of the program Π.

$$
\begin{array}{lll}
F1 & : & C \cup \{\phi =^{*} \phi'\} \Longrightarrow [\phi \mapsto \phi']C \\
F2 & : & C \cup \{c@\langle(\phi_i, \beta_i)\rangle_i <^{*}_{C} \phi,\ \langle(\phi_i', \beta_i')\rangle_i <^{*}_{C} \phi\} \Longrightarrow \\
 & & C \cup \{c\} \cup \{\phi_i =^{*} \phi_i',\ \beta_i =^{*} \beta_i'\} \\
F3 & : & C \cup \{c@\langle(\phi_i, \beta_i)\rangle_i \rightarrow (\phi_0, \beta_0) <^{*}_{n} \phi,\ \langle(\phi_i', \beta_i')\rangle_i \rightarrow (\phi_0', \beta_0') <^{*}_{n} \phi\} \Longrightarrow \\
 & & C \cup \{c\} \cup \{\phi_i =^{*} \phi_i',\ \phi_0 =^{*} \phi_0',\ \beta_i =^{*} \beta_i',\ \beta_0 =^{*} \beta_0'\}
\end{array}
$$

Fig. 5. Flow analysis: constraint normalization

5 Binding-time analysis

Following Henglein (Henglein, 1991), the bt-analysis associates *two* constraint variables with every expression, β and $\overline{\beta}$. The bt-variable β describes the 'internal' bt-value whereas $\overline{\beta}$ describes the 'external' bt-value. As we shall see when generating constraints, β will be determined by the *components* of an expression, and $\overline{\beta}$ will be determined by the *contexts* in which the expression occurs; this also explains the overbars that were used in Figure 4: notice that the β_E's being defined never have overbars whereas the β_{E_i}'s being referred to all have overbars. The values of β and $\overline{\beta}$ are usually equal, but they may differ: $\overline{\beta}$ may be dynamic and β static. When this happens, a *lift*-operation is inserted in the annotated source expression (Gomard and Jones, 1991); lift's argument is static, its result dynamic. At specialization time, the value generated by processing lift's argument is transformed into a residual expression by the specializer.

As in Henglein (1991), we shall only allow lifting first-order constants. This is a natural restriction in the context of Similix as Similix also refuses to lift higher-order values into residual expressions (Bondorf, 1991a): lifting higher-order values and data structures is in general unsafe since it may lead to residual code that exponentially duplicates data structure and closure allocations.

Also as in Henglein (1991), we do not consider 'induced' lift operations that lift one higher-order value into another one. We believe that induced lifts would complicate the analysis significantly, but we do not know whether the complexity would be severely worsened. We briefly discuss the consequences of not considering induced lifts in Bondorf and Jørgensen (1993).

The bt-analysis is done over the partially ordered set given in Figure 6; the diagram defines the ordering $\sqsubseteq$. Notice that the domain is simple in that it consists solely of atomic values; all structure flow has already been resolved in the flow analysis.

The top value is the bt-value **D** ('dynamic'). The bt-value **S** describes first-order static values. A family of bt-values **Cl** describes static function values; contrasting to the single **Cl**-value used in Bondorf (1991a), we use a value $\mathbf{Cl}_n$ for each function arity n. This gives a more conservative analysis than the one in Bondorf (1991a), but also a safer one: if functions of different arity flow together, they are made dynamic. This ensures that no arity errors occur when beta-reducing static function applications at specialization time. A similar family of bt-values **Ps** describes constructor values, indexed by the sum type $\xi = \tau(\mathrm{C})$ of the constructor C. The domains used in the

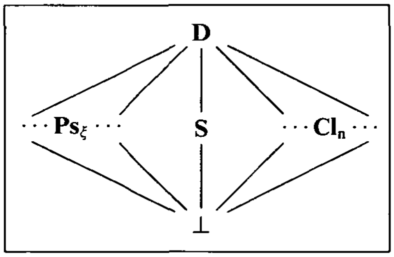
D
$\cdots \mathbf{Ps}_\xi \cdots$
S
$\cdots \mathbf{Cl}_n \cdots$
$\perp$

Fig. 6. *BtVal*-domain

b	$\in$	*BtVal*		; defined in Figure 6
v	$\in$	*BtVal1*	$=$	$BtVal \setminus \{\perp, \mathbf{D}\}$
β	$\in$	*BtVar*		
		BtVarD	$=$	$BtVar + \{\mathbf{D}\}$
ρ_{bt}	$\in$	*BtValSub*	$=$	$BtVar \rightarrow BtVal$
σ_{bt}	$\in$	*BtVarValSub*	$=$	$BtVar \rightarrow BtVarD$
c	$\in$	*BtConstraint*	$=$	$Bt\text{-}{=}^{\star} + Bt\text{-}{\sqsubseteq}^{\star} + Bt\text{-}{\leadsto}^{\star} + Bt\text{-}{\rhd}^{\star}$
		$Bt\text{-}{=}^{\star}$	$=$	$BtVarD \times BtVarD$
		$Bt\text{-}{\sqsubseteq}^{\star}$	$=$	$BtVal1 \times BtVarD$
		$Bt\text{-}{\leadsto}^{\star}$	$=$	$BtVarD \times BtVarD$
		$Bt\text{-}{\rhd}^{\star}$	$=$	$BtVarD \times BtVarD$

Fig. 7. Binding-time analysis: domains

bt-analysis are specified in Figure 7. Infix notation is used when writing constraints. The relations are defined as follows:

Definition 2 (Bt-relations)

1. $\sqsubseteq$ is the partial order on *BtVal*.
2. $b_1 \leadsto b_2 \;\equiv\; (b_1 = b_2) \vee (b_1 = \mathbf{S} \wedge b_2 = \mathbf{D}) \vee (b_1 = \perp)$.
3. $b_1 \rhd b_2 \;\equiv\; (b_1 = \mathbf{D} \Rightarrow b_2 = \mathbf{D})$. □

A constraint $\beta \rhd^{\star} \beta'$ is identical to the one used in Henglein (1991): if β is dynamic, β' must be dynamic too. A 'lift constraint' has form $\beta \leadsto^{\star} \beta'$ ('$\leadsto$' corresponds to Henglein's '$\leq$'); notice that it is satisfied if β and β' get assigned the same value (and it is satisfied if β gets assigned $\perp$), but also if β is **S** and β' is **D**: this latter case corresponds exactly to lifting a static first-order value. Notice that neither $\mathbf{Ps}_\xi \leadsto \mathbf{D}$ nor $\mathbf{Cl}_n \leadsto \mathbf{D}$ holds: we only allow lifting first-order constants. As a consequence of this, if a constructed or function value at any point flows together with something dynamic, then the constructed/function value must (will) be made dynamic right back at its source. On the other hand, a first-order static value may well be used statically some place and at the same time flow together with something dynamic at another place.

Constraint generation is specified in Figure 8; constraints are generated by (recursively) inspecting all (sub-)expressions of the program. In addition to the constraints generated from Figure 8, the following constraints are part of the initial constraint system:

1. For each expression E, a 'lift' constraint $\beta_E \rightsquigarrow^* \overline{\beta}_E$.
2. For each input variable V, either a constraint $\mathbf{S} \sqsubseteq^* \beta_V$ or a constraint $\beta_V =^* \mathbf{D}$, depending on the binding times for inputs supplied by the user.
3. A constraint $\overline{\beta}_E =^* \mathbf{D}$ for the body E of the goal procedure (since partial evaluation always generates residual code, never a value, at the top-level).
4. The equality constraints produced by the flow analysis.

The rules in Figure 8 are derived in a straightforward way by inspecting the specializer as defined in Bondorf (1992): the constraints must ensure that no bt-tag projection error can be committed by the specializer.

$E = K$	: $\{\mathbf{S} \sqsubseteq^* \beta_E\}$
$E = V$	: $\{\beta_E =^* \beta_V\}$
$E = (\texttt{if}\ E_1\ E_2\ E_3)$	: $\{\beta_E =^* \overline{\beta}_{E_2}, \beta_E =^* \overline{\beta}_{E_3}, \overline{\beta}_{E_1} \rhd^* \beta_E, \mathbf{S} \sqsubseteq^* \overline{\beta}_{E_1}\}$
$E = (\texttt{let}\ ((V\ E_1))\ E_2)$	: $\{\overline{\beta}_{E_1} =^* \beta_V, \beta_E =^* \overline{\beta}_{E_2}\}$
$E = (\texttt{begin}\ E_1\ E_2)$	: $\{\beta_E =^* \overline{\beta}_{E_2}\}$
$E = (O^{trans}\ E_1 \ldots E_n)$	: $\{\beta_E =^* \overline{\beta}_{E_i}, \mathbf{S} \sqsubseteq^* \beta_E\}$
$E = (O^{\neg trans}\ E_1 \ldots E_n)$	: $\{\beta_E =^* \overline{\beta}_{E_i}, \beta_E =^* \mathbf{D}\}$
$E = (C\ E_1 \ldots E_n)$	: $\{\mathbf{Ps}_{\tau(C)} \sqsubseteq^* \beta_E, \beta_E \rhd^* \overline{\beta}_{E_i}\}$
$E = (C\text{-}M\ E_1)$	: $\{\mathbf{Ps}_{\tau(C)} \sqsubseteq^* \overline{\beta}_{E_1}, \overline{\beta}_{E_1} \rhd^* \beta_E\}$
$E = (C?\ E_1)$	: $\{\mathbf{Ps}_{\tau(C)} \sqsubseteq^* \overline{\beta}_{E_1}, \overline{\beta}_{E_1} \rhd^* \beta_E, \mathbf{S} \sqsubseteq^* \beta_E\}$
$E = (P\ E_1 \ldots E_n)$	: $\{\beta_E =^* \overline{\beta}_P, \overline{\beta}_{E_i} =^* \beta_{P_i}\}$
$E = (\texttt{lambda}\ (V_1 \ldots V_n)\ E_1)$	: $\{\mathbf{Cl}_n \sqsubseteq^* \beta_E, \beta_E \rhd^* \beta_{V_i}, \beta_E \rhd^* \overline{\beta}_{E_1}\}$
$E = (E_0\ E_1 \ldots E_n)$	: $\{\mathbf{Cl}_n \sqsubseteq^* \overline{\beta}_{E_0}, \overline{\beta}_{E_0} \rhd^* \beta_E\}$

Fig. 8. Binding-time analysis: constraint generation

A constant expression is thus expected to specialize to a static first-order value. Therefore its bt-value must be S; since we have limited the constraints to contain only equality constraints between bt-variables and **D**, we must express the constraint as $\mathbf{S} \sqsubseteq^* \beta_E$ (this is similar to what is done in Henglein (1991)). The treatment of a variable expression is straightforward.

The conditional requires the branches and the whole expression to have the same bt-value. In addition to this, the result is dynamic if the test is: $\overline{\beta}_{E_1} \rhd^* \beta_E$. This, however, depends on the specializer: a specializer that can treat 'dynamic choice of static values' (see e.g. Bondorf 1992) does not have this requirement; for such a specializer, the $\rhd^*$-constraint is simply left out. Finally, the test must be at least static. If, for instance, the test may evaluate to a function value, this last constraint ensures that the test becomes dynamic (since in that case, $\overline{\beta}_{E_1}$ must both be greater than **S** and greater than $\mathbf{Cl_n}$). Let- and begin-expressions are straightforward.

In Similix, the user can specify that a primitive operator is *transparent* or not (Bondorf and Danvy, 1991). Transparent operations are reduced if all arguments are (first-order) static, but non-transparent primitive operations are always dynamic. Therefore there are two rules for primitive operations.

The flow analysis has already determined flow of constructed values and function values, so the remaining operations for constructors and higher-order functions are quite straightforward. Constructor expressions thus generate the bt-value $\mathbf{Ps}_\xi$. Also, if the value is ever made dynamic by some context (in which case the constructor application will not be reduced during specialization), the argument expressions also become dynamic. Selector and predicate expressions are straightforward.

Lambda-expressions are similar to constructor expressions. Again, if the value is made dynamic by some context (in which case residual code will be generated for the lambda-expression when specializing), both the formal parameters and the lambda-body become dynamic too. This is similar to Lambda-mix (Gomard and Jones, 1991); in the traditional Similix bt-analysis, the 'raise' operator ensured the same property. Finally, application is straightforward.

Constraint normalization is specified by the rewrite system in Figure 9. Soundness, completeness and termination theorems are given in Section 7.2: the rewrite system correctly solves any bt-constraint system (not necessarily generated from Figure 8).

$$
\begin{array}{lll}
\mathit{B1} & : & C \cup \{\beta =^{\star} \mathbf{D}\} \Longrightarrow [\beta \mapsto \mathbf{D}]C \\
\mathit{B2} & : & C \cup \{\beta =^{\star} \beta'\} \Longrightarrow [\beta \mapsto \beta']C \\
\mathit{B3} & : & C \cup \{c@\mathbf{S} \sqsubseteq^{\star} \beta,\ c'@\beta \rightsquigarrow^{\star} \beta'\} \Longrightarrow C \cup \{c,\ c',\ \mathbf{S} \sqsubseteq^{\star} \beta'\},\ \text{if } \mathbf{S} \sqsubseteq^{\star} \beta' \notin C \\
\mathit{B4} & : & C \cup \{c@\mathbf{Ps}_\xi \sqsubseteq^{\star} \beta,\ \beta \rightsquigarrow^{\star} \beta'\} \Longrightarrow C \cup \{c,\ \beta =^{\star} \beta'\} \\
\mathit{B5} & : & C \cup \{c@\mathbf{Cl}_n \sqsubseteq^{\star} \beta,\ \beta \rightsquigarrow^{\star} \beta'\} \Longrightarrow C \cup \{c,\ \beta =^{\star} \beta'\} \\
\mathit{B6} & : & C \cup \{v \sqsubseteq^{\star} \beta,\ v' \sqsubseteq^{\star} \beta\} \Longrightarrow C \cup \{\beta =^{\star} \mathbf{D}\},\ \text{if } v \neq v' \\
\mathit{B7} & : & C \cup \{\mathbf{D} \rhd^{\star} \beta\} \Longrightarrow C \cup \{\beta =^{\star} \mathbf{D}\} \\
\mathit{B8} & : & C \cup \{\mathbf{D} \rightsquigarrow^{\star} \beta\} \Longrightarrow C \cup \{\beta =^{\star} \mathbf{D}\} \\
\mathit{B9} & : & C \cup \{\mathbf{D} =^{\star} \beta\} \Longrightarrow C \cup \{\beta =^{\star} \mathbf{D}\}
\end{array}
$$

Fig. 9. Binding-time analysis: constraint normalization

Let C_{btN} be the normalized constraint system. The *output* of the bt-analysis consists of (cf. Figure 2): (1) the $\beta \rightsquigarrow^{\star} \beta'$ constraints in C_{btN}; (2) the substitution σ_{bt} built during constraint normalization (by composing all substitutions generated from rules *B1* and *B2* during normalization); (3) the minimal substitution (solution) ρ_{bt} defined by

$$
\rho_{bt}(\beta) = \begin{cases} \mathbf{D} & \text{if } \sigma_{bt}(\beta) = \mathbf{D} \\ v & \text{if } v \sqsubseteq^{\star} \sigma_{bt}(\beta) \in C_{btN} \\ \bot & \text{otherwise} \end{cases}
$$

This definition of ρ_{bt} is equivalent to the one given in Section 7.2 (Theorem 18) where it is proved that ρ_{bt} is the minimal solution.

6 Is-used analysis

As briefly outlined in Section 3, the is-used analysis generates a substitution ρ_{used} that is used for deciding what to generalize. Substitution ρ_{used} identifies the values that are used in a significant way, cf. Section 1.5: those values may *not* be generalized. The new non-minimal bt-substitution ρ_{bt1} is then defined in the following way:

$$\rho_{bt1}(\beta) = \begin{cases} \mathbf{D} & \text{if } \rho_{bt}(\beta) = \mathbf{S} \wedge \neg\rho_{used}(\beta) \\ \rho_{bt}(\beta) & \text{otherwise} \end{cases}$$

Notice that ρ_{bt1} sometimes returns **D** when the minimal ρ_{bt} returns **S**. It does so exactly when β is not used in a significant way.

Substitution ρ_{used} is defined as the minimal solution to an *is-used constraint system*. Substitution ρ_{bt1} should be best in the sense that it should make as much dynamic as possible; thus, $\rho_{used}(\beta)$ should hold for as *few* β's as possible. This motivates the ordering *true* $\succeq$ *false* on the boolean domain *UsedVal* (Figure 10); we use the symbol $\succeq$ in order to distinguish it from the ordering on bt-values.

u	$\in$	*UsedVal*		
β	$\in$	*BtVar*		
ρ_{used}	$\in$	*UsedValSub*	$=$	$\mathit{BtVar} \rightarrow \mathit{UsedVal}$
c	$\in$	*UsedConstraint*	$=$	$\mathit{Used\text{-}used}^* + \mathit{Used\text{-}}{\succeq}^*$
		$\mathit{Used\text{-}used}^*$	$=$	*BtVar*
		$\mathit{Used\text{-}}{\succeq}^*$	$=$	$\mathit{BtVar} \times \mathit{BtVar}$

Fig. 10. Is-used analysis: domains

The constraint $\mathit{used}^*(\beta)$ specifies that β is used in a significant way. We define the following relations:

Definition 3 (Is-used relations)

1. $\mathit{used}(u) \equiv u$. □
2. $\succeq$ is the partial order on *UsedVal*.

The minimal solution (Section 7.3, Theorem 22) to a normalized is-used constraint system C_{usedN} is defined by

$$\rho_{used}(\beta) = \begin{cases} \mathit{true} & \text{if } \mathit{used}^*(\sigma_{bt}(\beta)) \in C_{usedN} \\ \mathit{false} & \text{otherwise} \end{cases}$$

The definition of ρ_{bt1} constrains the values that ρ_{used} may assume since ρ_{bt1} must be a solution to the normalized bt-constraint system C_{btN}: it is necessary to propagate used^*-constraints in order to make $\rho_{used}(\beta)$ hold for more β's than those directly specified (by Figure 11). The $\succeq^*$-constraints serve this purpose; how these constraints are added is described in Section 6.1.

$E = K$	: $\{\mathit{used}^*(\sigma_{bt}(\beta_E))\}$	if $\rho_{bt}(\beta_E) = \mathbf{S}$
$E = (\texttt{if}\ E_1\ E_2\ E_3)$	: $\{\mathit{used}^*(\sigma_{bt}(\overline{\beta}_{E_1}))\}$	if $\rho_{bt}(\overline{\beta}_{E_1}) = \mathbf{S}$
$E = (O^{trans}\ E_1 \ldots E_n)$	: $\{\mathit{used}^*(\sigma_{bt}(\beta_E))\}$	if $\rho_{bt}(\beta_E) = \mathbf{S}$
$E = (C?\ E_1)$	: $\{\mathit{used}^*(\sigma_{bt}(\beta_E))\}$	if $\rho_{bt}(\beta_E) = \mathbf{S}$
All other cases	: $\{\}$	

Fig. 11. Is-used analysis: constraint generation

Generation of is-used constraints is specified in Figure 11 (applied recursively to all program (sub-)expressions). Recall that a used^*-constraint should be understood as 'if this value can possibly become static, then make it static': the value is used in a 'significant way'. The generation of used^*-constraints depends on ρ_{bt}: only those bt-variables that are assigned **S** by ρ_{bt} are of interest. In fact, we would in general obtain a greater value for ρ_{used} if we had not been careful only to generate the absolutely necessary used^*-constraints. The reason for this is that, as we shall see below, used^*-constraints get propagated by the is-used constraint normalization. So even though a constraint $\mathit{used}^*(\beta)$ has no effect on the value of ρ_{bt1} if $\rho_{bt}(\beta) = \mathbf{D}$, some derived constraint $\mathit{used}^*(\beta')$ may show up for a variable β' for which $\rho_{bt}(\beta') = \mathbf{S}$. This potentially prevents a desired generalization of β'. Notice that we have applied σ_{bt} whenever referring to bt-variables (indirectly when applying ρ_{bt} since $\rho_{bt} = \rho_{btN} \circ \sigma_{bt}$): this ensures that bt-variables that have been equated by the bt-analysis are treated equally.

In Figure 11, we have specified that constants should not be generalized; constant expressions are the source of static first-order values. Then we specify that if the test of a conditional can become static, then it should be (what determines control should be static if possible).

The used^*-constraint for an expression $(O^{trans}\ E_1 \ldots E_n)$ is not obvious: should one always reduce (transparent) primitive operations if possible? If yes, a constraint should be generated as shown in Figure 11. If no, no such constraint should be generated; for example, the +-operation in the example from Section 1.5 should *not* be reduced since this exactly leads to non-termination as described. In the implementation, we have split the transparent operators into two subclasses such that the user can declare whether the operator should always be reduced if possible; if it is declared that reduction should not necessarily take place, the primitive operation will only be reduced if some context forces it to, for instance if the test of a conditional. In practice, operators that can only be repeatedly applied a finite number of times can safely be declared 'always reduce'; an example is arithmetic test-predicates. Other operators (such as +) should be declared 'do not necessarily reduce'.

Finally, we have specified that constructor testing predicates should always be reduced: predicate expressions provide a source of static first-order values.

Static input variables are also similar to constants, so in addition to the constraints generated from Figure 11, we generate a *used**-constraint for each of the static input variables as well. Also, a constraint $used^*(\beta_{\mathsf{E}})$ is generated for the body E of the goal procedure if $\rho_{bt}(\beta_{\mathsf{E}}) = \mathbf{S}$ (this has effect in the rare cases where the body of the goal procedure is static).

Is-used constraint normalization is specified by the rewrite system in Figure 12. Soundness, completeness and termination theorems are given in Section 7.3: the rewrite system correctly solves any is-used constraint system.

$$\boxed{U1\ :\ C \cup \{used^*(\beta'),\, \beta \succeq^* \beta'\} \Longrightarrow C \cup \{used^*(\beta'),\, used^*(\beta)\}}$$

Fig. 12. Is-used analysis: constraint normalization

6.1 Additional is-used constraints

Let us now return to the criteria that the minimal solution ρ_{used} must fulfill to ensure that ρ_{btl} is a solution to C_{btN}. The outcome of the considerations in this section is the following: for every constraint $\beta \rightsquigarrow^* \beta'$ in C_{btN}, a constraint $\beta \succeq^* \beta'$ must be contained in C_{used}, the initial is-used constraint system. In addition to this, we exploit that the particular constraint system generated from Figure 8 has a certain form regarding the $\rhd^*$-constraints.

We prove later (Section 7.2, Theorem 17) that C_{btN} consists of the following kinds of constraints: (1) 'garbage' constraints (constraints that are trivially satisfied by any substitution), (2) $v \sqsubseteq^* \beta$ constraints, (3) $\beta \rightsquigarrow^* \beta'$ constraints and (4) $\beta \rhd^* \beta'$ constraints; it furthermore holds that substitution ρ_{bt} does not assign $\mathbf{D}$ to any of the variables in the non-garbage constraints. By adjusting ρ_{used}, we must ensure that ρ_{btl} solves all of these constraints. We do a case analysis over the different constraint forms.

Case (1), garbage constraints: trivially satisfied by ρ_{btl}.

Case (2), $v \sqsubseteq^* \beta$ constraints: since $\rho_{bt}(\beta) \sqsubseteq \rho_{btl}(\beta)$ holds for any β, these constraints are trivially satisfied by ρ_{btl} (since all constraints in C_{btN} are satisfied by ρ_{bt}).

Case (3), $\beta \rightsquigarrow^* \beta'$ constraints: we need to look at the cases where ρ_{btl} differs from ρ_{bt} on β and/or β'. This happens when β and/or β' get assigned $\mathbf{S}$ by ρ_{bt} and $\mathbf{D}$ by ρ_{btl}. Case $\rho_{bt}(\beta') = \mathbf{S} \wedge \rho_{btl}(\beta') = \mathbf{D}$: it holds for all bt-values b that $b \rightsquigarrow \mathbf{S} \Rightarrow b \rightsquigarrow \mathbf{D}$, so it is trivial that ρ_{btl} satisfies $\beta \rightsquigarrow^* \beta'$. Case $\rho_{bt}(\beta) = \mathbf{S} \wedge \rho_{btl}(\beta) = \mathbf{D}$: since $\rho_{bt}(\beta') = \mathbf{S}$ (otherwise the constraint $\beta \rightsquigarrow^* \beta'$ would not be satisfied), we must generalize β' such that $\rho_{btl}(\beta') = \mathbf{D}$ holds; otherwise, $\beta \rightsquigarrow^* \beta'$ would not be satisfied by ρ_{btl}. Notice that $\neg\rho_{used}(\beta)$ holds. We can obtain the desired generalization by ensuring that $\neg\rho_{used}(\beta')$ also holds (cf. the definition of ρ_{btl}): $\neg\rho_{used}(\beta) \Rightarrow \neg\rho_{used}(\beta')$ which is equivalent to $\rho_{used}(\beta) \Leftarrow \rho_{used}(\beta')$ and to $\rho_{used}(\beta) \succeq \rho_{used}(\beta')$. Hence, whenever C_{btN} contains a $\beta \rightsquigarrow^* \beta'$ constraint, we must add a constraint $\beta \succeq^* \beta'$ to C_{used};

notice from Figure 12 that is-used information thus in effect flows backwards through $\leadsto^\star$-constraints. In practice, there is no need to generate $\succeq^\star$-constraints for all $\beta \leadsto^\star \beta'$ constraints in C_{btN}; it suffices to generate a $\beta \succeq^\star \beta'$ constraint 'by need' each time there actually is a constraint $used^\star(\beta')$; this turns out to make the implementation simpler and more efficient since the $\leadsto^\star$-constraints need only be represented indirectly by attaching information to the β's (no explicit pass over $\leadsto^\star$-constraints is then needed).

Case (4), $\beta \rhd^\star \beta'$ constraints: we need to look at the cases where ρ_{bt1} differs from ρ_{bt} on β and/or β'. Case $\rho_{bt}(\beta') = \mathbf{S} \wedge \rho_{bt1}(\beta') = \mathbf{D}$: it holds for all bt-values b that $b \rhd \mathbf{S} \Rightarrow b \rhd \mathbf{D}$ so it is trivial that ρ_{bt1} satisfies $\beta \rhd^\star \beta'$. Case $\rho_{bt}(\beta) = \mathbf{S} \wedge \rho_{bt1}(\beta) = \mathbf{D}$: here we cannot know whether $\rho_{bt}(\beta')$ is $\bot$, $\mathbf{S}$, $\mathbf{Cl}_n$ or $\mathbf{Ps}_\xi$ (in contrast to the $\leadsto^\star$-case where we could deduce and exploit the fact $\rho_{bt}(\beta') = \mathbf{S}$). We want to obtain $\rho_{bt1}(\beta') = \mathbf{D}$ in order to make ρ_{bt1} satisfy $\beta \rhd^\star \beta'$, but only when $\rho_{bt}(\beta') = \mathbf{S}$ since only values with bt-value $\mathbf{S}$ are allowed to be poor man's generalized (this case, $\rho_{bt}(\beta') = \mathbf{S}$, can be handled as above under case (3), i.e. by generating $\succeq^\star$-constraints). In the other three cases ($\rho_{bt}(\beta') \in \{\bot, \mathbf{Ps}_\xi, \mathbf{Cl}_n\}$), we must prevent β from being generalized. This can be done by generating an is-used constraint $used^\star(\beta)$. To summarize, whenever C_{btN} contains a constraint $\beta \rhd^\star \beta'$ where $\rho_{bt}(\beta) = \mathbf{S}$, the value of $\rho_{bt}(\beta')$ must be inspected. If this value is $\mathbf{S}$, an is-used constraint $\beta \succeq^\star \beta'$ must be generated; if the value is in $\{\bot, \mathbf{Ps}_\xi, \mathbf{Cl}_n\}$, an is-used constraint $used^\star(\beta)$ must be generated. Alternatively, one could instead generate a constraint $used^\star(\beta)$ indiscriminately of the value of $\rho_{bt}(\beta')$, but this would yield more conservative results (less generalization).

For the particular C_{btN} that we generate, it is easy to see that $\rhd^\star$-constraints do not need any consideration at all in the is-used analysis: first notice (Figure 9) that no $\rhd^\star$-constraints are generated during normalization, so any $\rhd^\star$-constraint in C_{btN} will have form $\sigma_{bt}(\beta) \rhd^\star \sigma_{bt}(\beta')$ where $\beta \rhd^\star \beta'$ is contained in the initial C_{bt}. By inspecting Figure 8, we can see that $\rho_{bt}(\beta)$ can only become S for $\rhd^\star$-constraints originating from the `if`-rule (in which case $\overline{\beta}_{\mathrm{E}_1}$ plays the role of β); for all other $\rhd^\star$-constraints in Figure 8, constraints $\mathbf{Ps}_{\tau(\mathrm{C})} \sqsubseteq^\star \beta$ or $\mathbf{Cl}_n \sqsubseteq^\star \beta$ prevent $\rho_{bt}(\beta) = \mathbf{S}$ from holding (so these $\rhd^\star$-constraints need not be considered). However, notice from Figure 11 that an appropriate $used^\star$-constraint is always generated for the `if`-rule; hence we have a case corresponding to the 'more conservative' situation from above, that is, we can safely ignore the value of $\sigma_{bt}(\beta')$. Summarizing, for the particular constraint-generation systems in Figure 8 and Figure 11, $\rhd^\star$-constraints in C_{btN} can be completely ignored by the is-used analysis. We have exploited this in the implementation; this implies that the implementation only works for initial bt-constraint systems whose $\beta \rhd^\star \beta'$ constraints fulfill that $\rho_{bt}(\beta)$ can only become S if there also exists a constraint $used^\star(\sigma_{bt}(\beta))$.

7 Correctness proofs

In this section we give correctness proofs for the normalization algorithms. For each analysis, we prove that constraint normalization is sound and complete, that it terminates, and we constructively prove existence of a minimal solution (substitution) to the initial constraint system. We first give some general definitions and we prove some lemmas that are common to all analyses; we state and prove a general theorem (Theorem 10) that we then use to prove soundness and completeness of the analyses. Then we give analysis-specific proofs in the following sections.

We consider two kinds of substitutions:

Definition 4 (Substitutions)

$$\begin{array}{llll} \alpha & \in & Var & \\ v & \in & Val & \\ \rho & \in & ValSub = Var \rightarrow Val & \text{; Value substitutions} \\ \sigma & \in & VarValSubst = Var \rightarrow Var + Val & \text{; Variable/value substitutions} \end{array}$$ □

These definitions are more general than the ones used for the particular analyses (compare with the definitions of the ρ- and σ-substitution domains in the figures 3, 7, 10). We show some properties that hold for these more general definitions; hence they also hold for the particular domains used in the analyses.

We lift the domain of substitutions to include terms in the usual componentwise way: applying a substitution to a term t yields a new term identical to t except that any subterm of t that is a variable in the domain of the substitution has been replaced by the value obtained by applying the substitution to the variable. We may therefore always *compose* a substitution ρ and a substitution σ to get a new value substitution $\rho' = \rho\circ\sigma$. We define

Definition 5 (Solutions)

1. $\rho \models c \Leftrightarrow op(\rho(t_1), \ldots, \rho(t_n))$ where $c = op^\star(t_1, \ldots, t_n)$.
2. $\rho \models C \Leftrightarrow \forall c \in C : \rho \models c$. □

where $op \in Val^\star \rightarrow Boolean$ is the (semantic) relation corresponding to the (syntactic) constraint operator $op^\star$. We say that a constraint c is *satisfied* by a value substitution ρ if and only if $\rho \models c$ holds. We say that ρ is a *solution* to a constraint set C if and only if $\rho \models C$ holds.

Definition 6 (Set of solutions) Given a constraint system C, we define the set of solutions $Sol(C)$ to C by

$$Sol(C) = \{\rho \mid \rho \models C\}$$ □

The following lemma states that ρ is a solution to $\sigma(C)$ if and only if the composition $\rho\circ\sigma$ is a solution to C itself:

Lemma 7 Let σ be a variable/value substitution and ρ a value substitution, then

$$\rho \models \sigma(C) \Leftrightarrow \rho\circ\sigma \models C$$

Proof: Let $c = op^\star(t_1, \ldots, t_n)$ be an arbitrary element in C. Then

$$\rho \models \sigma(c) \Leftrightarrow \rho \models op^\star(\sigma(t_1), \ldots, \sigma(t_n)) \Leftrightarrow op(\rho\circ\sigma(t_1), \ldots, \rho\circ\sigma(t_n)) \Leftrightarrow \rho\circ\sigma \models c$$

Since c was picked arbitrarily from C, the lemma follows. □

Notice that we use the precedence rule that function composition binds stronger than function application: $f \circ g(x) = (f \circ g)(x)$. The next lemma is used later for compositionally proving preservation of substitutions:

LEMMA 8 Let C, C_1 and C_2 be constraint sets, then

$$Sol(C_1) = Sol(C_2) \Rightarrow Sol(C \cup C_1) = Sol(C \cup C_2)$$

PROOF: Let ρ be an arbitrary element in $Sol(C \cup C_1)$. Assume that the premise holds. Then

$$\rho \models C \cup C_1 \Leftrightarrow \rho \models C \wedge \rho \models C_1 \Leftrightarrow \rho \models C \wedge \rho \models C_2 \Leftrightarrow \rho \models C \cup C_2. \qquad \square$$

DEFINITION 9 (Constraint-system rewrite rules) A set $\mathcal{R}$ of *constraint-system rewrite rules* is a set containing the following two kinds of rules:

1. Substitution rules:
   (a) $C \cup \{\alpha =^* \alpha'\} \Longrightarrow [\alpha \mapsto \alpha']C$
   (b) $C \cup \{\alpha =^* v\} \Longrightarrow [\alpha \mapsto v]C$
2. Proper rules:
   (a) $C \cup C_1 \Longrightarrow C \cup C_2$ $\square$

We thus consider two kinds of substitution rules, those binding a variable to another variable and those binding a variable to a value (cf. the definition of σ-substitutions in Definition 4). The proper rules replace some constraints by others, but they do not introduce substitutions.

We use the notation $R : C_A \Longrightarrow \sigma(C_B)$ to express that the rule named R rewrites C_A to $\sigma(C_B)$. Rules may have side-conditions that limit the applicability of a rule (e.g. Rule *B3* in Figure 9). We have ignored side-conditions here since they do not influence soundness and completeness; they do, however, influence termination of constraint normalization (in particular, Theorem 16). For proper rules, σ is the identity substitution and we may leave σ out and just write $R : C_A \Longrightarrow C_B$.

THEOREM 10 (Soundness and completeness) Consider a constraint system C_A and a set of constraint system rewrite rules $\mathcal{R}$. If (for all ρ) $\rho \models C_1 \Leftrightarrow \rho \models C_2$ holds for all proper rules $R : C \cup C_1 \Longrightarrow C \cup C_2$ of $\mathcal{R}$, then the system is sound and complete, i.e. for all rules $R : C_A \Longrightarrow \sigma(C_B)$ the following holds:

$$Sol(C_A) = \{\rho \circ \sigma \mid \rho \in Sol(\sigma(C_B))\}$$

PROOF: By case analysis over the different kinds of rewrite rules (cf. Definition 9).

Case: First substitution rule. Here $\sigma = [\alpha \mapsto \alpha']$. First we observe that for any variables α and α' and any value substitution ρ the following holds: $\rho(\alpha) = \rho(\alpha') \Leftrightarrow \exists \rho' : \rho = \rho' \circ [\alpha \mapsto \alpha']$. To see this, note that we can always choose ρ' to be ρ.

$$Sol(C_A) = Sol(C \cup \{\alpha =^* \alpha'\}) \overset{\text{def 6}}{=} \{\rho \mid \rho \models C \cup \{\alpha =^* \alpha'\}\} \overset{\text{def 5}}{=}$$

$$\{\rho \mid \rho \models C \wedge \rho \models \alpha =^* \alpha'\} \overset{\text{def 5}}{=} \{\rho \mid \rho \models C \wedge \rho(\alpha) = \rho(\alpha')\} =$$

$$\{\rho \mid \rho \models C \wedge \exists \rho' : \rho = \rho' \circ [\alpha \mapsto \alpha']\} = \{\rho' \circ [\alpha \mapsto \alpha'] \mid \rho' \circ [\alpha \mapsto \alpha'] \models C\} \overset{\text{renaming}}{=}$$

$$\{\rho\circ[\alpha\!\to\!\alpha'] \mid \rho\circ[\alpha\!\to\!\alpha'] \models C\} \overset{\text{lem 7}}{=} \{\rho\circ[\alpha\!\to\!\alpha'] \mid \rho \models [\alpha\!\to\!\alpha']C\} \overset{\text{def 6}}{=}$$
$$\{\rho\circ[\alpha\!\to\!\alpha'] \mid \rho \in Sol([\alpha\!\to\!\alpha']C)\} = \{\rho\circ\sigma \mid \rho \in Sol(\sigma(C_B))\}$$

Case: Second substitution rule. Similar to the case above.

Case: Proper rules. Because σ is the identity substitution, we have to prove: $Sol(C_A) = Sol(C_B) \Leftrightarrow Sol(C \cup C_1) = Sol(C \cup C_2)$. We have assumed that for all value substitutions ρ and all proper rules, $\rho \models C_1 \Leftrightarrow \rho \models C_2$ holds; we therefore have $Sol(C_1) = Sol(C_2)$ and then by Lemma 8, $Sol(C \cup C_1) = Sol(C \cup C_2)$. □

Thus, if we can show that $\rho \models C_1 \Leftrightarrow \rho \models C_2$ for all proper rules of constraint rewriting system $\mathscr{R}$, then we know that the system is sound and complete: for any rewriting step, a substitution ρ is a solution to the rewritten system if (soundness) and only if (completeness) ρ is a solution to the original system.

THEOREM 11 (Solution to constraint system) If a constraint-normalization system is sound and complete, then a (possibly minimal) solution ρ to a constraint system $\mathscr{R}$ can be found by composing a (possibly minimal) solution ρ' to the normalized version of $\mathscr{R}$ with the composition of the σ-substitutions generated by the rewrite steps during normalization.

PROOF: By Theorem 10, transitivity of equality of solution sets and associativity of function composition. □

7.1 Flow analysis

THEOREM 12 (Soundness and completeness of flow-constraint normalization) The set of rewrite rules for flow-constraint normalization, shown in Figure 5, is sound and complete.

PROOF: By Theorem 10 we have to prove that for all proper rules of $\mathscr{R}$ and all value substitutions $\rho \in FlowValSub$, $\rho \models C_1 \Leftrightarrow \rho \models C_2$ holds. The proof is by case analysis over the two proper rules. We only give the proof for Rule *F3*; the proof for Rule *F2* is similar.

$$\rho \models C_1 \overset{\text{def 5}}{\Leftrightarrow}$$
$$\rho \models \langle(\phi_i,\beta_i)\rangle_i\!\to\!(\phi_0,\beta_0) <^*_n \phi \wedge \rho \models \langle(\phi_i{}',\beta_i{}')\rangle_i\!\to\!(\phi_0{}',\beta_0{}') <^*_n \phi \overset{\text{def 5}}{\Leftrightarrow}$$
$$\langle(\rho\phi_i,\rho\beta_i)\rangle_i\!\to\!(\rho\phi_0,\rho\beta_0)<_n\rho\phi \wedge \langle(\rho\phi_i{}',\rho\beta_i{}')\rangle_i\!\to\!(\rho\phi_0{}',\rho\beta_0{}')<_n\rho\phi \overset{\text{def 1}}{\Leftrightarrow}$$
$$\langle(\rho\phi_i,\rho\beta_i)\rangle_i\!\to\!(\rho\phi_0,\rho\beta_0)<_n\rho\phi \wedge$$
$$\rho\phi_i = \rho\phi_i{}' \wedge \rho\phi_0 = \rho\phi_0{}' \wedge \rho\beta_i = \rho\beta_i{}' \wedge \rho\beta_0 = \rho\beta_0{}' \wedge \rho\varepsilon_0 = \rho\varepsilon_0{}' \overset{\text{def 5}}{\Leftrightarrow}$$
$$\rho \models \langle(\phi_i,\beta_i)\rangle_i\!\to\!(\phi_0,\beta_0) <^*_n \phi \wedge \rho \models \phi_i =^* \phi_i{}' \wedge \rho \models \phi_0 =^* \phi_0{}' \wedge$$
$$\rho \models \beta_i =^* \beta_i{}' \wedge \rho \models \beta_0 =^* \beta_0{}' \wedge \rho \models \varepsilon_0 =^* \varepsilon_0{}' \overset{\text{def 5}}{\Leftrightarrow}$$
$$\rho \models C_2$$
□

Notice that for all β, $\rho\,\beta = \beta$. Thus, a flow-constraint system will only rarely have a solution at all since β-variables must be identical to be considered equal. However, a flow-constraint system can always be solved modulo a substitution σ_{bt} (see below).

THEOREM 13 (Termination of flow-constraint normalization) Flow-constraint normalization terminates.

PROOF: Rule *F2* and rule *F3* can only be applied finitely many times since each application of removes one of finitely many $\prec^{\star}_{\mathsf{C}}/\prec^{\star}_{\mathsf{n}}$ constraints. Each application of the rules *F2* and *F3* adds a finite number of equality constraints. Hence rule *F1* can only be applied finitely many times. Normalization therefore terminates. □

THEOREM 14 (Form of normalized flow-constraint system) A normalized flow-constraint system C_{flowN} contains the following kinds of constraints:

1. $\langle(\phi_i,\beta_i)\rangle_i \prec^{\star}_{\mathsf{C}} \phi$ and $\langle(\phi_i,\beta_i)\rangle_i \rightarrow (\phi_0,\beta_0) \prec^{\star}_{\mathsf{n}} \phi$.
2. $\beta =^{\star} \beta'$.

Furthermore, for every constructor C, C_{flowN} contains at most one constraint of the form $\langle(\phi_i,\beta_i)\rangle_i \prec^{\star}_{\mathsf{C}} \phi$ for each variable ϕ. Similarly, for every function arity, C_{flowN} contains at most one constraint of the form $\langle(\phi_i,\beta_i)\rangle_i \rightarrow (\phi_0,\beta_0) \prec^{\star}_{\mathsf{n}} \phi$ for each variable ϕ.

PROOF: By inspection of the constraint-normalization algorithm. □

Theorem 14 ensures that all flow equalities $\phi_i =^{\star} \phi_i'$ have been resolved. We are not interested in a minimal solution ρ_{flow} to the initial system C_{flow}, but it does exist modulo that the $\beta =^{\star} \beta'$ constraints are solved by a substitution σ_{bt}. We refer to Bondorf and Jørgensen (1993) for details.

We note that the bt-analysis introduces additional $\beta =^{\star} \beta'$ constraints, both during constraint generation (Figure 8) and during constraint normalization (Figure 9). Hence σ_{bt}, as generated during bt-constraint normalization, will typically equate more variables than those required to be equal by the flow analysis. This of course influences the resulting minimal solution ρ_{flow}.

7.2 Binding-time analysis

THEOREM 15 (Soundness and completeness of bt-constraint normalization) The set of rewrite rules for bt-constraint normalization, shown in Figure 9, is sound and complete.

PROOF: By Theorem 10 we have to prove that for all proper rules of $\mathscr{R}$ and all value substitutions $\rho \in BtValSub$, $\rho \models C_1 \Leftrightarrow \rho \models C_2$ holds. The proof is by case analysis over the proper rules. We use the abbreviations $\rho(\beta) = b$ and $\rho(\beta') = b'$.

Case: Rule *B3*. $\rho \models C_1 \overset{\text{def 5}}{\Leftrightarrow} \mathsf{S} \sqsubseteq b \wedge b \rightsquigarrow b' \Leftrightarrow \mathsf{S} \sqsubseteq b \wedge b \rightsquigarrow b' \wedge \mathsf{S} \sqsubseteq b' \overset{\text{def 5}}{\Leftrightarrow} \rho \models C_2$.

Case: Rule *B4*. $\rho \models C_1 \Leftrightarrow \mathbf{Ps}_{\xi} \sqsubseteq b \wedge b \rightsquigarrow b' \Leftrightarrow \mathbf{Ps}_{\xi} \sqsubseteq b \wedge b = b' \Leftrightarrow \rho \models C_2$.

Case: Rule *B5*. $\rho \models C_1 \Leftrightarrow \mathbf{Cl}_{\mathsf{n}} \sqsubseteq b \wedge b \rightsquigarrow b' \Leftrightarrow \mathbf{Cl}_{\mathsf{n}} \sqsubseteq b \wedge b = b' \Leftrightarrow \rho \models C_2$.

Case: Rule *B6.* $\rho \models C_1 \Leftrightarrow v \sqsubseteq b \wedge v' \sqsubseteq b \wedge v \neq v' \Leftrightarrow b = \mathbf{D} \Leftrightarrow \rho \models C_2$.
Case: Rule *B7.* $\rho \models C_1 \Leftrightarrow \mathbf{D} \rhd b \Leftrightarrow b = \mathbf{D} \Leftrightarrow \rho \models C_2$.
Case: Rule *B8.* $\rho \models C_1 \Leftrightarrow \mathbf{D} \sqsubseteq b \Leftrightarrow b = \mathbf{D} \Leftrightarrow \rho \models C_2$.
Case: Rule *B9.* $\rho \models C_1 \Leftrightarrow \mathbf{D} = b \Leftrightarrow b = \mathbf{D} \Leftrightarrow \rho \models C_2$. □

THEOREM 16 (Termination of bt-constraint normalization) Bt-constraint normalization terminates.

PROOF: We show that each of the rules of Figure 9 can only be applied a finite number of times; this is done by case analysis over all constraint-normalization rules. We refer to Bondorf and Jørgensen (1993) for details. □

THEOREM 17 (Form of normalized bt-constraint system) A normalized bt-constraint system C_{btN} contains the following kinds of constraints:

1. Garbage: $\beta \rightsquigarrow^* \mathbf{D}$, $\mathbf{D} \rightsquigarrow^* \mathbf{D}$, $\beta \rhd^* \mathbf{D}$, $\mathbf{D} \rhd^* \mathbf{D}$, $v \sqsubseteq^* \mathbf{D}$, $\mathbf{D} =^* \mathbf{D}$.
2. $v \sqsubseteq^* \beta$; C_{btN} never contains two constraints $v \sqsubseteq^* \beta$ and $v' \sqsubseteq^* \beta$ where $v \neq v'$.
3. $\beta \rightsquigarrow^* \beta'$.
4. $\beta \rhd^* \beta'$.

Furthermore, for any β, neither constraints $\beta \rightsquigarrow^* \beta'$ and $\mathbf{Ps}_\xi \sqsubseteq^* \beta$ nor constraints $\beta \rightsquigarrow^* \beta'$ and $\mathbf{Cl}_n \sqsubseteq^* \beta$ co-exist in C_{btN}.

PROOF: By inspection of the constraint-normalization algorithm. □

THEOREM 18 (Minimal solution) There exists a minimal solution ρ_{btN} to a normalized bt-constraint system C_{btN} (and hence there also exists a minimal solution $\rho_{bt} = \rho_{btN} \circ \sigma_{bt}$ to the original system C_{bt}, cf. Theorem 11). The minimal solution ρ_{btN} is found by first solving C_{btN}'s $v \sqsubseteq^* \beta$ constraints by $\rho_{btN}(\beta) = v$, and then letting $\rho_{btN}(\beta) = \perp$ for all remaining unbound constraint variables β.

PROOF: The minimal solution to the $v \sqsubseteq^* \beta$ constraints is the one obtained by letting $\rho_{btN}(\beta) = v$. We must show that the other constraints in C_{btN} then also are satisfied. The garbage constraints are always satisfied. The $\beta \rhd^* \beta'$ constraints are satisfied because no variable is assigned $\mathbf{D}$. By considering the possible values that can be assigned to β and β', it can be verified that the $\beta \rightsquigarrow^* \beta'$ constraints are also satisfied (see Bondorf and Jørgensen, 1993 for details.) □

7.3 *Is-used analysis*

Proofs for the following theorems can be found in Bondorf and Jørgensen (1993).

THEOREM 19 (Soundness and completeness of is-used constraint normalization) The set of rewrite rules for is-used constraint normalization, shown in Figure 12, is sound and complete. □

Theorem 20 (Termination of is-used constraint normalization) Is-used constraint normalization terminates. □

Theorem 21 (Form of normalized is-used constraint system) A normalized constraint system C_{usedN} contains the following kinds of constraints:

1. $used^{*}(\beta)$.
2. $\beta \succeq^{*} \beta'$.

Furthermore, for any β', constraints $used^{*}(\beta')$ and $\beta \succeq^{*} \beta'$ do not co-exist in C_{usedN}. □

Theorem 22 (Minimal solution) There exists a minimal solution ρ_{usedN} to a normalized is-used constraint system C_{usedN} (and hence there also exists a minimal solution $\rho_{used} = \rho_{usedN} \circ \sigma_{used} = \rho_{usedN}$ to the original system C_{used}, cf. Theorem 11). The minimal solution ρ_{usedN} is defined by, for all β's, if C_{usedN} contains a constraint $used^{*}(\beta)$, then $\rho_{usedN}(\beta) = true$; otherwise, $\rho_{usedN}(\beta) = false$. □

8 Implementation

In this section we give a short description of the implementation of the three analyses. The analyses have all been implemented in Scheme (IEE, 1990) and integrated into the Similix system; the implementation will be publically available in Similix version 5.0. The implementation is based on Henglein (1991). We thus also use a union/find-based algorithm that operates on a *term graph representation* of the constraints systems.

The main difference between Henglein's and our algorithm is that we do not keep one global list of constraints to be solved. We instead solve constraints in a particular order; this order is determined from observations of which constraints are generated during normalization. For example, notice that in the bt-constraint normalization algorithm from Figure 9, no $\rightsquigarrow^{*}$-constraints are ever generated. This can be exploited by always handling the (initial) $\rightsquigarrow^{*}$-constraints before $\sqsubseteq^{*}$-constraints. Then only one piece of code for discovering matches between the two kinds of constraints is needed (the code that implements rewriting by one of the rules *B3*, *B4*, *B5*).

Avoiding the global constraint list speeds up constraint normalization by saving some bookkeeping, but the main benefit is that the code is simpler: we found it much easier to convince ourselves that no cases had been overlooked. The disadvantage is that the implementation cannot immediately be used incrementally. Notice that this is not a limitation of the analyses, only of the implementation: nothing in constraint generation, constraint normalization nor in correctness proofs requires constraints to be solved in any specific order.

In the following we will describe how the bt-analysis was implemented; the other two analyses were implemented in a similar way. Each constraint variable is represented by a variable node in the term graph. To each variable node we associate an *equivalent class representative* (henceforth called 'ecr') which is also a variable node. The ecrs represent equivalence classes of variable nodes; ecrs implement the substitution σ generated during constraint normalization. To each ecr, we associate two dependency lists, one for $\triangleright^\star$-dependencies and one for $\rightsquigarrow^\star$-dependencies. We also associate a so-called *leq*-field representing $\sqsubseteq^\star$-constraints (cf. (Henglein, 1991)). In general, when a constraint is examined, the constraint is removed from the constraint set and the information it conveyed about constraint variables is either combined with (using one of the rules of the rewrite system) or added to information associated with the involved variable nodes. If, say, a constraint $\beta \triangleright^\star \beta'$ is examined, the $\triangleright^\star$-dependency field of β's ecr is updated to contain β'. Equality constraints between variables are solved right away by unifying the two corresponding variable nodes (this implements Rule *B1*); one of these nodes becomes the new ecr. Any information associated with the node that 'loses' its ecr-status is then combined with the information of the new ecr.

Let us now look more specifically at the different bt-constraints. Equality constraints between constraint variables are always handled immediately at creation time. In particular, the bt-equality constraints generated by the flow analysis are solved already during the flow analysis; no other bt-constraints have been examined at this point, so the two corresponding bt-variable nodes can be unified without considering (the still empty) associated information such as the $\triangleright^\star$-dependency field. The rest of the constraints are examined in the following order:

1. $\beta \rightsquigarrow^\star \beta'$ constraints. The $\rightsquigarrow^\star$-dependency fields are initialized.
2. $\mathbf{S} \sqsubseteq^\star \beta$ constraints. If the leq-field of β has not been updated, it is updated to **S**. If furthermore the $\rightsquigarrow^\star$-dependency of β is non-empty, Rule *B3* is applied by adding a constraint $\mathbf{S} \sqsubseteq^\star \beta'$ for each β' in β's $\rightsquigarrow^\star$-dependency list.
3. $\mathbf{Ps}_\xi \sqsubseteq^\star \beta$ constraints. These are handled quite similarly to the $\mathbf{S} \sqsubseteq^\star \beta$-constraints, namely by updating leq-fields. If the leq-field of β has already been updated to something different from $\mathbf{Ps}_\xi$, a $\beta =^\star \mathbf{D}$ constraint is generated. This implements one usage of Rule *B6*. If furthermore the $\rightsquigarrow^\star$-dependency of β is non-empty, Rule *B4* is applied by generating (and immediately solving) a constraint $\beta =^\star \beta'$ for each β' in β's $\rightsquigarrow^\star$-dependency list.
4. $\mathbf{Cl}_n \sqsubseteq^\star \beta$ constraints. These are treated similarly to the $\mathbf{Ps}_\xi \sqsubseteq^\star \beta$-constraints, except that Rule *B5* is used instead of Rule *B4*. If the leq-field of β has already been updated to something different from $\mathbf{Cl}_n$, a $\beta =^\star \mathbf{D}$ constraint is generated.
5. $\beta \triangleright^\star \beta'$ constraints. The $\triangleright^\star$-dependency fields are initialized.
6. $\beta =^\star \mathbf{D}$ constraints. These are handled by first unifying the variable node corresponding to β with a special *dynamic node* and then, for each β' in β's dependency lists (both $\triangleright^\star$and $\rightsquigarrow^\star$-dependencies), adding a new constraint $\beta' =^\star \mathbf{D}$. These actions implement the rules *B2*, *B7* and *B8*.

When all constraints have been examined (including those generated during constraint normalization), the bt-value associated with each constraint variable, hence also with each program point, is found from the associated variable node in the following way: if the variable node is in the same equivalence class as the special dynamic node, then the bt-value is **D**. If not, we look at the variable node's leq-field; if this field has been updated, the bt-value is the value of this field. Otherwise, the bt-value is $\perp$. This way, the minimal solution (cf. Section 5) is found. The desired bt-solution is found after is-used analysis by also taking the then updated is-used fields of variable nodes into account.

9 Complexity

Let us now reason about the complexity of the flow analysis.

First ignore the operations on constructed data structures (construction, selection, predicate testing). For this limited language, the number of generated initial constraints is linearly bounded by N where N is the (textual) size of the analysed program; this is easy to see since the constraint generation algorithm is compositional and since it generates a constant number of constraints for each syntactic construct.

The equality constraints have constant size. The size of the $\prec^{*}_{n}$-constraints is varying, but the size of each $\prec^{*}_{n}$-constraint is linearly bounded by the size of the source expression from which it was generated (the constraint grows as the function arity grows, but so does the source expression). Hence the total size of the initial constraint system (where the varying size of each constraint is taken into account) is also linearly bounded by the size of the program text.

Normalization Rule *F3* generates new equality constraints that were not in the initial constraint system. However, the number of such generated equality constraints is linear in the size of $\prec^{*}_{n}$-constraint that is removed by Rule *F3*. Since each $\prec^{*}_{n}$-constraint can at most be removed once and since no $\prec^{*}_{n}$-constraints are generated during normalization, the total number of generated constraints, the initial ones plus the ones generated during normalization, is also linearly bounded by the size of the program text.

All operations performed during normalization can be done in constant time, except those related to unification and those that search for matching arities to check applicability of Rule *F3*. This search is (if implemented naively) at least bounded by $1 + FA$ where FA is the maximum function arity of any lambda-expression in the analysed program. Unification adds a factor $\alpha(N, N)$ due to the union/find operations (α is an inverse of Ackermann's function, see Henglein (1991) for further details). The time complexity for the flow analysis *without* constructor operations thus becomes $O(N \cdot \alpha(N, N) \cdot (1 + FA))$ for the flow analysis.

The factor $\alpha(N, N)$ is less than 4 for all practical purposes (Henglein, 1991), so in practice the time complexity is linearly bounded by N. For hand-written source programs, the factor FA is also small in practice. However, machine generated programs might look quite different (but in particular, the ones generated by Similix do not), so an efficient implementation of the search operation to reduce the FA-factor might become crucial. Notice further that if the analysed language had been strongly typed instead of dynamically typed (such as Scheme), clashing function arities would generate a type error. Hence no search would be needed, and thus the factor FA would vanish. Notice that the complexity would then be equal to that of Henglein (1991).

Now consider the full language including constructor operations. Here the constraint-generation rule for selector expressions (C-M E_l) breaks one linearity: the size of the generated constraint depends on the constructor arity and is thus *not* linear in the size of the expression (which always has just one argument). This implies that the total number of generated constraints is linearly bounded by $M = N \cdot max(1, CA)$ where CA is the maximum constructor arity. Additionally, search time increases as a search is needed to check applicability of Rule *F2*. This search is (if implemented naively) at least bounded by N_c where N_c is the number of constructors used in the analysed program. The time complexity then becomes $O(M \cdot \alpha(M, M) \cdot (1 + max(FA, N_c)))$. Again, CA and N_c are typically small for both hand-written and Similix-generated programs, but for machine generated programs in general, that might not be the case.

By similar considerations, it can be deduced that the time complexities of the bt- and is-used analyses are bounded linearly by N, modulo the small factor $\alpha(\ldots, \ldots)$ (Bondorf and Jørgensen, 1993). The linearity claim is supported by the experimental results presented in Section 10.

Without getting into details, we mention that in the implementation it holds for all three analyses that the amount of storage allocated is linearly bounded by the size of the total number of constraints generated by the analyses. Therefore space complexity can be expected to be linear in the size of the program modulo a small factor. This claim is also supported by the experiments (see Section 10).

10 Performance

This section contains performance results. We compare the performance of the *new* Similix preprocessor based on the analyses described in this paper with that of the *old* preprocessor that used abstract interpretation based analyses; the old system is the one described in Bondorf (1991b) extended with partially static data structures. The tests were run on a Sparc Station 2/Sun OS 4.1 using Chez Scheme Version 3.2. Run times are for the full preprocessing time (of which by far the most takes place in the flow-, bt- and eod-analyses).

Run times do not include garbage collection (but storage allocation is measured separately): the time spent on garbage collection depends on many irrelevant factors (the method used for garbage collection, the system configuration, etc.). Garbage collection time may be significant in practice; for example, the old preprocessor garbage collected 192 times when preprocessing one of the test programs, Cogen; the new preprocessor only garbage collected 9 times.

Figure 13 gives the time and space consumption of preprocessing performed on four selected programs. The first column contains the names of the programs. All four programs are realistic examples: BAWL0 and BAWL1 are interpreters for a substantial lazy functional language (Jørgensen, 1992), Specializer and Cogen are the specializer and the automatically generated compiler generator of Similix. The second column gives the size of the programs (measured as the number of 'cons' cells needed to represent the program as an S-expression). The last two columns show the time and space consumption of the two sets of analyses.

Figure 14 shows complexity results for the two sets of analyses. The figures show

Program	Size/cells	Old		New	
		Time/s	Storage/Kb	Time/s	Storage/Kb
BAWL0	1420	2.6	674	1.3	320
Specializer	2112	4.1	683	1.9	494
BAWL1	4506	12.8	3,420	3.8	978
Cogen	10881	249.	50,043	9.2	2,554

Fig. 13. Performance: benchmarks

Program	Old		New	
	Time/Size	Storage/Size	Time/Size	Storage/Size
BAWL0	1.86	.47	0.91	.23
Specializer	1.93	.32	0.88	.23
BAWL1	2.83	.76	0.84	.22
Cogen	22.9	4.60	0.85	.23

Fig. 14. Performance: complexities

the ratios time/size and storage/size. The tests confirm that the new preprocessor in practice runs linearly in the size of the analysed program while the old preprocessor performs much worse. In particular, the figures for Cogen are very large for the old preprocessor.

11 Conclusion

We have successfully scaled up Henglein's efficient bt-analysis and integrated it into the Similix system. We do a separate flow analysis prior to the bt-analysis; the flow analysis traces flow of constructed and function values. The bt-analysis therefore becomes quite simple: no structured bt-values are needed. A new is-used analysis implements 'poor man's generalization' which improves termination of specialization. We have shown that this new analysis can be formulated elegantly in the same framework as the flow and bt-analyses. In Bondorf and Jørgensen (1993), also the eod-analysis of Similix is formulated in the framework.

Experiments have confirmed dramatic speedups compared to the traditional abstract interpretation based analyses (Bondorf, 1991a, 1991b). Previously, the analyses have never in practice been a major bottleneck though, but that is probably mostly due to the fact that we have never specialized programs that were substantially larger than the BAWL interpreters (Jørgensen, 1992). The experiments in this paper show that if significantly larger programs (of a size comparable to the Cogen-program of Similix) are to be partially evaluated, the new preprocessing proves very useful.

This paper does not prove that the initial constraint systems are correctly specified: the specifications must ensure that the specializer cannot commit any bt-tag projection errors. Such proofs exist for specializers for the pure lambda calculus

(Wand, 1993; Palsberg, 1993); the Scheme subset treated in this paper is of course much more complex.

Acknowledgements

Special thanks to Fritz Henglein for many inspiring discussions. Thanks to Neil Jones and the rest of DIKU's TOPPS/Semantics-group for providing an excellent working environment. Also thanks to the anonymous referees for their constructive suggestions for improvements.

References

Bondorf, Anders. (1991a) Automatic autoprojection of higher order recursive equations. *Science of computer programming,,* **17**(1-3), 3–34. Revision of paper in ESOP'90, LNCS 432, May 1990.

Bondorf, Anders. (1991b) (Sept.). *Similix Manual, system version 4.0.* DIKU, University of Copenhagen, Denmark.

Bondorf, Anders. (1992) (June). Improving binding times without explicit cps-conversion. *Pages 1–10 of: 1992 ACM Conference on Lisp and Functional Programming. San Francisco, California. LISP Pointers V, 1.*

Bondorf, Anders, & Danvy, Olivier. (1991) Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of computer programming,* **16**, 151–195.

Bondorf, Anders, & Jørgensen, Jesper. (1993) *Efficient analyses for realistic off-line partial evaluation: extended version.* Tech. rept. 93/4. DIKU, University of Copenhagen, Denmark.

Consel, Charles. (1990) (June). Binding time analysis for higher order untyped functional languages. *Pages 264–272 of: 1990 ACM Conference on Lisp and Functional Programming. Nice, France.*

Gomard, Carsten K., & Jones, Neil D. (1991) A partial evaluator for the untyped lambda-calculus. *Journal of functional programming,* **1**(1), 21–69.

Henglein, Fritz. (1991) Efficient type inference for higher-order binding-time analysis. *Pages 448–472 of:* Hughes, John (ed), *Conference on Functional Programming and Computer Architecture, Cambridge, Massachusetts. Lecture Notes in Computer Science 523.* Springer-Verlag.

Henglein, Fritz. (1992) (Mar.). *Simple closure analysis.* Working note.

Holst, Carsten Kehler. (1988) (Aug.). *Poor man's generalization.* Working note.

IEEE standard for the Scheme programming language. IEEE Std 1178-1990. (1990) May.

Jones, Neil D., Sestoft, Peter, & Søndergaard, Harald. (1985) An experiment in partial evaluation: the generation of a compiler generator. *Pages 124–140 of:* Jouannaud, Jean-Pierre (ed), *Rewriting Techniques and Applications, Dijon, France. Lecture Notes in Computer Science 202.* Springer-Verlag.

Jones, Neil D., Sestoft, Peter, & Søndergaard, Harald. (1989) MIX: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and symbolic computation,* **2**(1), 9–50.

Jørgensen, Jesper. (1992) (Jan.). Generating a compiler for a lazy language by partial evaluation. *Pages 258–268 of: Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico.*

Katz, Morry, & Weise, Daniel. (1992) (June). Towards a new perspective on partial evaluation. *Pages 29–37 of: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California. Yale University, report YALEU/DCS/RR-909.*

Launchbury, John. (1991) *Projection factorisations in partial evaluation.* Distinguished Dissertations in Computer Science. Cambridge University Press.

Nielson, Hanne R., & Nielson, Flemming. (1988) Automatic binding time analysis for a typed λ-calculus. *Pages 98–106 of: Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. San Diego, California.*

Palsberg, Jens. (1993) Correctness of binding time analysis. *Journal of functional programming, special issue on partial evaluation.*

Schmidt, David A. (1988) Static properties of partial evaluation. *Pages 465–483 of:* Bjørner, Dines, Ershov, Andrei P., & Jones, Neil D. (eds), *Partial Evaluation and Mixed Computation.* North-Holland.

Sestoft, Peter. (1988) (Student Report 88-7-2, October). *Replacing function parameters by global variables.* M.Phil. thesis, DIKU, University of Copenhagen, Denmark.

Wand, Mitchell. (1993) Specifying the correctness of binding-time analysis. *Journal of functional programming, special issue on partial evaluation.*