Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Electronic Notes in Theoretical Computer Science 132 (2005) 5–17

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Cobalt: A Language for Writing Provably-Sound Compiler Optimizations

Sorin Lerner[1]

Department of Computer Science & Engineering
University of Washington
Seattle, WA, USA

Todd Millstein[2]

Computer Science Department
University of California, Los Angeles
Los Angeles, CA, USA

Craig Chambers[3]

Department of Computer Science & Engineering
University of Washington
Seattle, WA, USA

Abstract

We overview the current status and future directions of the Cobalt project. Cobalt is a domain-specific language for implementing compiler optimizations as guarded rewrite rules. Cobalt optimizations operate over a C-like intermediate representation including unstructured control flow, pointers to local variables and dynamically allocated memory, and recursive procedures. The design of Cobalt engenders a natural inductive strategy for proving the soundness of optimizations. This strategy is fully automated by requiring an automatic theorem prover to discharge a small set of simple proof obligations for each optimization. We have written a variety of forward and backward intraprocedural dataflow optimizations in Cobalt, including constant propagation and folding, branch folding, full and partial redundancy elimination, full and partial dead assignment elimination, and simple forms of points-to analysis. The implementation of our soundness-checking strategy employs the Simplify automatic theorem prover, and we have used this implementation to automatically prove the above optimizations correct. An execution engine for Cobalt optimizations is implemented as part of the Whirlwind compiler infrastructure.

Keywords: Compiler optimization, automated correctness proofs.

doi:10.1016/j.entcs.2005.03.022

1 Cobalt By Example

This section overviews the current status of the Cobalt language. We informally describe Cobalt and our technique for proving Cobalt optimizations sound through a small example. More details on the language, the technique for proving soundness automatically, and the implementation of the language and its soundness checker are provided in an earlier paper [9]. Section 2 describes our current and future directions for increasing the expressiveness of Cobalt while retaining automated soundness checking. Section 3 describes related work, and section 4 concludes.

1.1 Forward Transformation Patterns

The heart of a Cobalt optimization is its *transformation pattern.* For a forward optimization, a transformation pattern has the following form:

ψ_1 ***followed by*** ψ_2 ***until*** $s \ \Rightarrow\ s'$ ***with witness*** $\mathcal{P}$

A transformation pattern describes the conditions under which a statement s may be transformed to s'. The formulas ψ_1 and ψ_2, which are properties of a statement such as "x is defined and y is not used," together act as the guard indicating when it is legal to perform this transformation on a procedure P: s can be transformed to s' if on all paths in P's control-flow graph (CFG) from the start of the procedure to s, there exists a statement satisfying ψ_1, followed by zero or more statements satisfying ψ_2, followed by s. Figure 1 shows this scenario pictorially.

Forward transformation patterns codify a scenario common to many forward dataflow analyses: an *enabling* statement establishes the conditions necessary for a transformation to be performed downstream, and any intervening statements are *innocuous*, i.e., do not invalidate the conditions. The formula ψ_1 captures the properties that make a statement enabling, and ψ_2 captures the properties that make a statement innocuous. The *witness* $\mathcal{P}$ captures the conditions established by the enabling statement that allow the transformation to be safely performed. Witnesses have no effect on the semantics of an optimization; they will be discussed more below in the context of our strategy for automatically proving optimizations sound.

Example 1 *A simple form of constant propagation replaces statements of the form $X := Y$ with $X := C$ if there is an earlier (enabling) statement of the form $Y := C$ and each intervening (innocuous) statement does not modify Y.*

[1] Email: lerns@cs.washington.edu
[2] Email: todd@cs.ucla.edu
[3] Email: chambers@cs.washington.edu

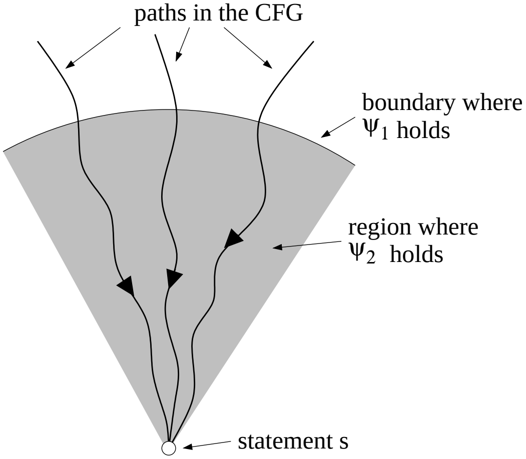

Fig. 1. CFG paths leading to a statement s which can be transformed to s' by the transformation pattern ψ_1 ***followed by*** ψ_2 ***until*** $s \Rightarrow s'$ ***with witness*** $\mathcal{P}$. The shaded region can only be entered through a statement satisfying ψ_1, and all statements within the region satisfy ψ_2. The statement s can only be reached by first passing through this shaded region.

The enabling statement ensures that variable Y holds the value C, and this condition is not invalidated by the innocuous statements, thereby allowing the transformation to be safely performed downstream. This sequence of events is expressed by the following transformation pattern (the witness is discussed in more detail in section 1.3):

$\quad stmt(Y := C)$
followed by
$\quad \neg mayDef(Y)$
until
$\quad X := Y \;\Rightarrow\; X := C$
with witness
$\quad \eta(Y) = C$

The “pattern variables” X and Y may be instantiated with any variables of the procedure being optimized, while the pattern variable C may be instantiated with constants in the procedure.

1.2 Labels

Each node in a procedure's CFG is *labeled* with properties that are true at that node, such as $stmt(\mathtt{x} := \mathtt{5})$ or $mayDef(\mathtt{y})$. The formulas ψ_1 and ψ_2 in an optimization are boolean expressions over these labels.

Users can define a new kind of label by giving a predicate over a statement, referred to in the predicate's body using the distinguished variable *currStmt*. As a trivial example, the $\mathit{stmt}(S)$ label, which denotes that the statement at the current node is S, can be defined as:

$$\mathit{stmt}(S) \triangleq \mathit{currStmt} = S$$

As another example, $\mathit{syntacticDef}(Y)$, which stands for syntactic definition of Y, can be defined as:

$$\begin{array}{lll} \mathit{syntacticDef}(Y) \triangleq & \multicolumn{2}{l}{\textbf{case } \mathit{currStmt} \textbf{ of}} \\ & \quad \texttt{decl } X & \mapsto \quad X = Y \\ & \quad X := \ldots & \mapsto \quad X = Y \\ & \quad \textbf{else} & \mapsto \quad \mathit{false} \\ & \textbf{endcase} & \end{array}$$

The label $\mathit{syntacticDef}(Y)$ holds at a node if and only if the current statement is a declaration of or an assignment to Y. The "**case**" predicate is a convenience that provides a form of pattern matching over the C-like intermediate language that Cobalt optimizations manipulate, but it is easily desugared into an ordinary logical expression. Similarly, pattern variables and ellipses get desugared into ordinary quantified variables.

Given the definition of *syntacticDef*, a conservative version of the *mayDef* label from example 1 can be defined as:

$$\begin{array}{lll} \mathit{mayDef}(Y) \triangleq & \multicolumn{2}{l}{\textbf{case } \mathit{currStmt} \textbf{ of}} \\ & \quad *X := Z & \mapsto \quad \mathit{true} \\ & \quad X := P(Z) & \mapsto \quad \mathit{true} \\ & \quad \textbf{else} & \mapsto \quad \mathit{syntacticDef}(Y) \\ & \textbf{endcase} & \end{array}$$

In other words, a statement may define variable Y if the statement is either a pointer store (since our intermediate language allows taking the address of a local variable), a procedure call (since the procedure may be passed pointers from which the address of Y is reachable), or otherwise a syntactic definition of Y.

1.3 Soundness

A transformation pattern is *sound* if all the transformations it allows are semantics-preserving, for all possible intermediate-language procedures. Forward transformation patterns have a natural approach for understanding their soundness. Consider a statement s transformed to s' in procedure P. Any execution trace of P that contains s' will at some point execute an enabling statement, followed by zero or more innocuous statements, before reaching

s'. As mentioned earlier, executing the enabling statement establishes some conditions at the subsequent state of execution. These conditions are then preserved by the innocuous statements. Finally, the conditions imply that s and s' have the same effect at the point where s' is executed. As a result, the original program and the transformed program have the same semantics.

Our automatic strategy for proving optimizations sound is based on the above intuition. As part of the code for a forward transformation pattern, optimization writers provide a *forward witness* $\mathcal{P}$, which is a (possibly first-order) predicate over an execution state, denoted η. The witness plays the role of the conditions mentioned in the previous paragraph and is the intuitive reason why the transformation pattern is correct. Our strategy attempts to prove that the witness is established by the enabling statement and preserved by the innocuous statements, and that it implies that s and s' have the same effect. More specifically, for each forward transformation pattern ψ_1 ***followed by*** ψ_2 ***until*** $s \Rightarrow s'$ ***with witness*** $\mathcal{P}$, we ask an automatic theorem prover to discharge the following obligations:

(i) If ψ_1 holds at an arbitrary node n in an arbitrary CFG and the statement at that node is executed from an arbitrary execution state, then $\mathcal{P}$ will hold in the resulting execution state.

(ii) If ψ_2 holds at an arbitrary node n in an arbitrary CFG and the statement at that node is executed from an execution state satisfying $\mathcal{P}$, then $\mathcal{P}$ will also hold in the resulting execution state.

(iii) If s and s' are each executed from the same execution state satisfying $\mathcal{P}$, then the resulting execution states will be identical.

We have proven that if these obligations hold for any particular optimization, then that optimization is sound [9].

In example 1, the forward witness $\eta(Y) = C$ denotes the fact that the value of Y in execution state η is C. Our implementation automatically discharges the above obligations: the witness $\eta(Y) = C$ is established by the statement $Y := C$, preserved by statements that do not modify the contents of Y, and implies that $X := Y$ and $X := C$ have the same effect. Therefore, the constant propagation transformation pattern is automatically proven to be sound.

1.4 Other Features of Cobalt

Cobalt includes several other features that increase its expressiveness, while still preserving fully automatic soundness reasoning.

1.4.1 Backward Transformation Patterns

Cobalt has a notion of a *backward transformation pattern*, which naturally allows the expression of backward optimizations like dead-assignment elimination. A backward transformation pattern is similar to a forward one, except that the direction of the flow of analysis is reversed:

ψ_1 ***preceded by*** ψ_2 ***since*** $s \;\Rightarrow\; s'$ ***with witness*** $\mathcal{P}$

The backward transformation pattern above says that s may be transformed to s' in a procedure P if on all paths in P's CFG from s to the end of the procedure, there exists a statement satisfying ψ_1, *preceded* by zero or more statements satisfying ψ_2, *preceded* by s.

1.4.2 Profitability Heuristics

For some optimizations, including our constant-propagation example, all legal transformations are also *profitable.* However, in more complex optimizations, such as code motion and optimizations that trade off time and space, many transformations may preserve program behavior while only a small subset of them improve the code. To address this distinction between legality and profitability, a Cobalt optimization is written in two pieces. The transformation pattern only defines which transformations are legal. An optimization separately describes which of the legal transformations are also profitable and therefore should be performed; we call this second piece of an optimization its *profitability heuristic.*

This way of factoring optimizations into a transformation pattern and a profitability heuristic is critical to our ability to prove optimizations sound automatically, since only an optimization's transformation pattern affects soundness. Transformation patterns tend to be simple even for complicated optimizations, with the bulk of an optimization's complexity pertaining to profitability. Profitability heuristics can be written in any language, thereby removing any limitations on their expressiveness. Profitability heuristics are completely ignored by the soundness checker. Without profitability heuristics, the extra complexity added to transformation patterns to express profitability information would prevent automated correctness reasoning.

1.4.3 Pure Analyses

In addition to optimizations, Cobalt allows users to write pure analyses that do not perform transformations. These analyses can be used to compute or verify properties of interest about a procedure and to provide information to be consumed by later transformations. A pure analysis defines a new label, and the result of the analysis is a labeling of the given CFG. For instance, a does-not-point-to analysis can be defined, which results in nodes of the CFG being annotated with labels of the form $\mathit{doesNotPointTo}(X, Y)$. The new label can then be used by other analyses, optimizations, or label definitions. For example, it can be used to make the definition of *mayDef* in section 1.2 less conservative in the face of pointer stores.

A forward pure analysis is similar to a forward optimization, except that it does not contain a rewrite rule or a profitability heuristic. Instead, it has a ***defines*** clause that gives a name to the new label. The strategy for proving soundness of a pure analysis is a slight variant on the strategy for proving soundness of a forward transformation pattern.

2 Current and Future Work

Our current work on Cobalt is geared toward further increasing expressiveness while maintaining the ability to automatically reason about soundness. We are designing a second version of the language, Cobalt V2, that incorporates the lessons we have learned from the first version, Cobalt V1, while significantly enhancing its capabilities. In this section we use a simple example to give a flavor for what Cobalt V2 will look like. We focus on pure analyses, which best illustrate the advantages of Cobalt V2, but transformations can be easily adapted to the Cobalt-V2 style as well.

A pure analysis in Cobalt V1 states under what conditions a node should be annotated with a particular label, which we also refer to as a *dataflow fact*. The conditions are *global* in that they talk about all paths in the CFG leading to the node in question. Global conditions are appealing because in one shot they concisely describe the nodes that should be annotated with a given dataflow fact. However, in order to automate soundness reasoning, restrictions have to be imposed on these global conditions, and in Cobalt V1 only one stylized form of global condition is supported. Instead of providing the analysis writer with this one stylized global conditions, the main idea in Cobalt V2 is to provide the analysis writer with stylized *local* conditions, which can then be combined in flexible ways to achieve many kinds of global conditions.

As an example, consider a simple does-not-point-to analysis in Cobalt V1:

```
  $\mathit{stmt}(X := \&Z) \wedge Y \neq Z$
followed by
  $\neg \mathit{mayDef}(X)$
defines
  $\mathit{doesNotPointTo}(X, Y)$
with witness
```

$$\eta(X) \neq \eta(\&Y)$$

This analysis says that a node n can be labeled with $doesNotPointTo(X, Y)$ if on all CFG paths to n, there is a point at which X is assigned the address of a variable different from Y, and then X is not modified until n. This global condition can easily be expressed with two *local* rules in Cobalt V2:

$$\begin{array}{ll} \textbf{\textit{if}} & \textit{stmt}(X := \&Z) \wedge Y \neq Z \\ \textbf{\textit{then}} & \textit{doesNotPointTo}(X, Y)@\textit{cfg\_out} \end{array}$$

$$\begin{array}{ll} \textbf{\textit{if}} & \textit{doesNotPointTo}(X, Y)@\textit{cfg\_in} \wedge \neg \textit{mayDef}(X) \\ \textbf{\textit{then}} & \textit{doesNotPointTo}(X, Y)@\textit{cfg\_out} \end{array}$$

The first rule says that the outgoing CFG edge of an assignment $X := \&Z$ should be annotated with $\textit{doesNotPointTo}(X, Y)$ for each variable Y different from Z. The second rule says that if $doesNotPointTo(X, Y)$ appears on the incoming edge of a node that does not modify X, then $doesNotPointTo(X, Y)$ should also appear on the outgoing edge.

In addition to these two rules, the analysis writer must also provide the *meaning* of the does-not-point-to fact, which plays an analogous role to the witness in Cobalt V1:

$$\begin{array}{l} \textbf{\textit{define edge fact}}\ \textit{doesNotPointTo}(X, Y) \\ \textbf{\textit{with meaning}}\ \eta(X) \neq \eta(\&Y) \end{array}$$

The meaning of a dataflow fact D is a predicate, with the intent that whenever D appears on an edge, the meaning of D should hold in all concrete stores that can appear on that edge. Our goal is to check automatically that this intent becomes reality, and we can do this by checking that each if-then rule is sound separately. For each rule we ask the theorem prover to show that if the meaning of the antecedent holds at the incoming edge, then the meaning of the consequent holds at the outgoing edge. For example, in the second rule above we would ask the theorem prover to show that if a statement satisfying $\neg \textit{mayDef}(X)$ is executed in a state satisfying $X \neq \&Y$, then the resulting state will also satisfy $X \neq \&Y$.

The local if-then rules of Cobalt V2 offer several advantages over the global condition in Cobalt V1. First, the local rules are more flexible and more expressive. Because the rules operate at the granularity of a node, our does-not-point-to analysis can now be easily extended with many new rules, for example:

$$\begin{array}{ll} \textbf{\textit{if}} & \textit{stmt}(X := Z) \wedge \textit{doesNotPointTo}(Z,\, Y)@\textit{cfg\_in} \\ \textbf{\textit{then}} & \textit{doesNotPointTo}(X, Y)@\textit{cfg\_out} \end{array}$$

$$\begin{array}{ll} \textbf{\textit{if}} & \mathit{stmt}(*A := B) \wedge \mathit{mustPointTo}(A, X)@\mathit{cfg\_in} \wedge \\ & \mathit{doesNotPointTo}(B, Y)@\mathit{cfg\_in} \\ \textbf{\textit{then}} & \mathit{doesNotPointTo}(X, Y)@\mathit{cfg\_out} \end{array}$$

The second rule above assumes the existence of a dataflow fact $\mathit{mustPointTo}(X, Y)$, which signifies that X definitely points to Y. This dataflow fact can be defined using another set of if-then rules.

Another advantage of Cobalt V2 is that the local if-then rules can be seen as flow functions: given some incoming dataflow facts, they determine which outgoing dataflow facts to propagate. This interpretation of if-then rules provides an explanation of Cobalt in terms of concepts already familiar to many analysis writers, thus making the language easier to adopt. Furthermore, because the flow-function formalism matches the commonly used formalism in the analysis community, much previous work on dataflow analysis should adapt seamlessly to Cobalt V2. For example, we will attempt to incorporate into V2 some of our previous work on composing and staging dataflow analyses [8,18] and on deriving interprocedural analyses from intraprocedural ones [1].

3 Related Work

The idea of analyzing optimizations written in a domain-specific language was introduced by Whitfield and Soffa [28]. By analyzing optimizations expressed in a language called Gospel, their system can automatically determine if one optimization helps or hinders another one. This information can then be used to determine an order in which to run optimizations. Their framework also includes a tool, Genesis [27], for executing optimizations written in Gospel. The main difference between our work and the Gospel work is in the properties of interest: we explore soundness whereas Whitfield and Soffa explore optimization dependencies. Despite the different focus, our languages have similarities: both Gospel and Cobalt optimizations consist of a rewrite rule (an ACTION clause in Gospel) guarded by some condition (a PRECONDITION clause in Gospel). The Gospel rewrite rules are more flexible than the Cobalt ones, since they allow moving statements. On the other hand, Gospel has dataflow dependencies as primitives in the language, whereas Cobalt allows the user to define such dependencies with dataflow facts.

Our work is also related to the use of temporal logic for expressing and reasoning about analyses [24,25,22,23,7], since the guards of Cobalt's transformation patterns can be viewed as restricted kinds of temporal-logic formulas. Our language was in fact inspired by recent work in the temporal-logic direction by Lacey et al. [7]. Lacey describes a language for writing optimizations as guarded rewrite rules evaluated over a labeled CFG, and our transformation patterns are modeled on this language. Lacey's intermediate language lacks several constructs found in realistic languages, including pointers, dynamic memory allocation, and procedures. Lacey describes a general strategy, based on relating execution traces of the original and transformed programs, for manually proving the soundness of optimizations in his language. Three example optimizations are shown and proven sound by hand using this strategy. Unfortunately, the generality of this strategy makes it difficult to automate.

Lacey's guards may be arbitrary CTL formulas, while our guard language can be viewed as a strict subset of CTL that codifies a particularly common idiom. However, we are still able to express more precise versions of Lacey's three example optimizations (as well as many others) and to prove them sound automatically. Further, Lacey's optimization language has no notion of profitability heuristics or of pure analyses. Therefore, expressing optimizations that employ profitability or pointer information (assuming Lacey's language were augmented with pointers) would instead require writing more complicated guards, and some optimizations we support may not be expressible by Lacey.

A significant amount of work has been done on manually proving optimizations correct [10,11,2,3,5,17,4]. Transformations have also been proven correct mechanically, but not automatically: the transformation is proven sound using an interactive theorem prover, which requires user involvement. For example, Young [29] has proven a code generator correct using the Boyer-Moore theorem prover enhanced with an interactive interface [6]. In contrast, Cobalt's proof strategy is fully automated.

Instead of proving that the compiler is always correct, translation validation [19,15] and credible compilation [21,20] both attack the problem of checking the correctness of a given compilation run. Therefore, a bug in an optimization only appears when the compiler is run on a program that triggers the bug. Our work allows optimizations to be proven correct before the compiler is even run once. However, to do so we require optimizations to be written in a special-purpose language. Our approach also requires the Cobalt execution engine to be part of the trusted computing base, while translation validation and credible compilation do not require trust in any part of the compiler.

Proof-carrying code [14], certified compilation [16], typed intermediate languages [26], and typed assembly languages [12,13] have all been used to prove properties of programs generated by a compiler. However, the kinds of properties that these approaches have typically guaranteed are type safety and memory safety. In our work, we prove the stronger property of semantic equivalence between the original and resulting programs.

4 Conclusion

We have overviewed the Cobalt project, an approach for automatically proving the correctness of compiler optimizations. Our technique provides the optimization writer with a domain-specific language for writing optimizations. Cobalt is both reasonably expressive and amenable to automated correctness reasoning. Using our technique we have proven correct implementations of several optimizations over a realistic intermediate language. Aside from helping to ensure the reliability of compilers, Cobalt is a promising step toward the goal of *user-extensible* compilers, which would allow programmers to easily *and safely* add unusual or domain-specific analyses and optimizations.

Acknowledgments

This research is supported in part by NSF grants CCR-0073379 and ACI-0203908, a Microsoft Graduate Fellowship, an IBM Faculty Development Award, and by gifts from Sun Microsystems.

References

[1] Chambers, C., J. Dean and D. Grove, *Frameworks for intra- and interprocedural dataflow analysis*, Technical Report UW-CSE-96-11-02, University of Washington (1996).

[2] Cousot, P. and R. Cousot, *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles CA, 1977, pp. 238–252.

[3] Cousot, P. and R. Cousot, *Systematic design of program analysis frameworks*, in: *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, San Antonio TX, 1979, pp. 269–282.

[4] Cousot, P. and R. Cousot, *Systematic design of program transformation frameworks by abstract interpretation*, in: *Conference Record of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland OR, 2002.

[5] Guttman, J., J. Ramsdell and M. Wand, *VLISP: a verified implementation of Scheme*, Lisp and Symbolic Compucation **8** (1995), pp. 33–110.

[6] Kauffmann, M. and R. Boyer, *The Boyer-Moore theorem prover and its interactive enhancement*, Computers and Mathematics with Applications **29** (1995), pp. 27–62.

[7] Lacey, D., N. D. Jones, E. V. Wyk and C. C. Frederiksen, *Proving correctness of compiler optimizations by temporal logic*, in: *Conference Record of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland OR, 2002.

[8] Lerner, S., D. Grove and C. Chambers, *Composing dataflow analyses and transformations*, in: *Conference Record of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland OR, 2002.

[9] Lerner, S., T. Millstein and C. Chambers, *Automatically proving the correctness of compiler optimizations*, in: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (2003), pp. 220–231.

[10] McCarthy, J. and J. Painter, *Correctness of a compiler for arithmetic expressions*, in: T. J. Schwartz, editor, *Proceedings of Symposia in Applied Mathematics*, 1967.

[11] Morris, F. L., *Advice on structuring compilers and proving them correct*, in: *Conference Record of the 1st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Boston MA, 1973.

[12] Morrisett, G., K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich and S. Zdancewic, *TALx86: A realistic typed assembly language*, in: *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, Atlanta GA, 1999, pp. 25–35.

[13] Morrisett, G., D. Walker, K. Crary and N. Glew, *From System F to Typed Assembly Language*, ACM Transactions on Programming Languages and Systems **21** (1999), pp. 528–569.

[14] Necula, G. C., *Proof-carrying code*, in: *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, 1997.

[15] Necula, G. C., *Translation validation for an optimizing compiler*, in: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, Canada, 2000, pp. 83–95.

[16] Necula, G. C. and P. Lee, *The design and implementation of a certifying compiler*, in: *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, 1998.

[17] Oliva, D. P., J. Ramsdell and M. Wand, *The VLISP verified PreScheme compiler*, Lisp and Symbolic Computation **8** (1995), pp. 111–182.

[18] Philipose, M., C. Chambers and S. Eggers, *Towards automatic construction of staged compilers*, in: *Conference Record of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland OR, 2002.

[19] Pnueli, A., M. Siegel and E. Singerman, *Translation validation*, in: *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, Lecture Notes in Computer Science **1384**, 1998, pp. 151–166.

[20] Rinard, M., *Credible compilation*, Technical Report MIT-LCS-TR-776, Massachusetts Institute of Technology (1999).

[21] Rinard, M. and D. Marinov, *Credible compilation*, in: *Proceedings of the FLoC Workshop Run-Time Result Verification*, 1999.

[22] Schmidt, D. A., *Dataflow analysis is model checking of abstract interpretations*, in: *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego CA, 1998.

[23] Schmidt, D. A. and B. Steffen, *Data flow analysis as model checking of abstract interpretations*, in: G. Levi, editor, *Proceedings of the 5th International Symposium on Static Analysis (SAS)*, Lecture Notes in Computer Science (LNCS) **1503** (1998), pp. 351–380.

[24] Steffen, B., *Data flow analysis as model checking*, in: T. Ito and A. Meyer, editors, *Theoretical Aspects of Computer Science (TACS), Sendai (Japan)*, Lecture Notes in Computer Science (LNCS) **526** (1991), pp. 346–364.

[25] Steffen, B., *Generating dataflow analysis algorithms for model specifications*, Science of Computer Programming **21** (1993), pp. 115–139.

[26] Tarditi, D., G. Morrisett, P. Cheng, C. Stone, R. Harper and P. Lee, *TIL: A type-directed optimizing compiler for ML*, in: *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia PA, 1996.

[27] Whitfield, D. and M. L. Soffa, *The design and implementation of Genesis*, Software Practice and Experience **24** (1994), pp. 307–325.

[28] Whitfield, D. L. and M. L. Soffa, *An approach for exploring code improving transformations*, ACM Transactions on Programming Languages and Systems **19** (1997), pp. 1053–1084.

[29] Young, W. D., *A mechanically verified code generator*, Journal of Automated Reasoning **5** (1989), pp. 493–518.