Technical Report MIT-CSAIL-TR-976, December 16, 2004

Machine-Checkable Correctness Proofs for Intra-procedural Dataflow Analyses

Alexandru Sălcianu and Konstantine Arkoudas

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{`salcianu,arkoudas`}`@csail.mit.edu` *Fax: 617-253-1221*

Abstract

This technical report describes our experience using the interactive theorem prover Athena for proving the correctness of abstract interpretation-based dataflow analyses. For each analysis, our methodology requires the analysis designer to formally specify the property lattice, the transfer functions, and the desired modeling relation between the concrete program states and the results computed by the analysis. The goal of the correctness proof is to prove that the desired modeling relation holds. The proof allows the analysis clients to rely on the modeling relation for their own correctness. To reduce the complexity of the proofs, we separate the proof of each dataflow analysis into two parts: a generic part, proven once, independent of any specific analysis; and several analysis-specific conditions proven in Athena.

Key words: Dataflow analysis, correctness proofs, interactive theorem proving, Athena

1 Introduction

Modern compilers use a variety of dataflow analyses, whose correctness directly affects the correctness of the produced executables. Although the theoretical foundations of dataflow analyses are well understood and described in detail in popular textbooks [19], many such analyses are presented without a formal specification of the properties they compute and without a correctness proof. Even when detailed paper-and-pencil correctness proofs are given, they tend to be very long and mostly tedious. As a consequence, few people ever read and review such proofs, leading to low confidence in them. We do not want to underemphasize the importance of such proofs: The first author wrote a large paper-and-pencil correctness proof for a pointer and escape analysis [23], and, although difficult, that proof was invaluable in understanding (and correcting) the analysis design.

The goal of our research is to use advances in interactive theorem proving to express analysis correctness proofs in a machine-checkable manner. Using

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

an interactive theorem prover has two advantages. First, it forces the analysis designer to be precise in the description of the analysis and in the specification of the properties that the analysis computes. Second, the correctness proof is machine-checkable. Unfortunately, the increased precision and machine-checkability have the drawback of requiring a significant increase in the proof effort. In addition, many machine-checkable proofs are unnatural and hard to read. We use the interactive theorem prover Athena [4, 2] because it has the potential to tackle these problems. Proof readability and writability were primary objectives in the design of Athena. The goal was to allow for high-level structured proofs written in the same style and at the same level of detail as the informal proofs that are given in practice. Athena also achieves significant proof automation, both through user-defined tactics and through the seamless integration of external cutting-edge automated theorem provers (such as Vampire [25] and Spass [26]).

A dataflow analysis computes an *analysis fact* for each program point; the analysis fact conservatively *models* each possible program state at that point. In our approach, we ask the analysis designer to provide a formal specification of the modeling relation, and a correctness proof, i.e., a proof that the computed analysis facts satisfy the intended modeling relation. Each program optimization that uses the analysis results can rely on the modeling relation. Therefore, we decouple the problem of optimization correctness into two parts: the correctness of the underlying analysis, and the correctness of the program transformation. Our work examines only the analysis correctness.

Proving the correctness of an analysis is a daunting task. To simplify it, 1) we focus on the high-level definition of the analysis; and 2) we split the correctness proof into several simpler proofs.

Following classic textbooks [19], we express a dataflow analysis as a fixed-point of a set of dataflow equations. Intuitively, the analysis starts with a special *initial analysis fact* for the beginning of each analyzed procedure, and next uses the *analysis transfer functions* to abstractly interpret [8] the program statements. The analysis facts belong to a *property* lattice; in the control flow join points, the lattice join operator combines the incoming analysis facts. Given a set of monotonic constraints / transfer functions over a lattice with no infinite ascending chains, there are well-understood algorithms for computing the least fixed-point [19]. We consider these fixed-point solvers correct, and do not prove their correctness. Instead, given a set of transfer functions, we focus on proving that any set of analysis results (one result for each program point) that satisfy the transfer functions also satisfy the intended modeling relation associated with the analysis.

We separate the analysis correctness proof into a generic part (proven once, independent of the examined analysis), and three sufficient analysis-specific conditions. For each new analysis, the analysis designer needs to prove these three conditions in Athena. The generic part of the proof is a proof by induction that uses the analysis-specific conditions in its base case and induction step. These conditions require that the analysis fact for the beginning of an analyzed procedure models the concrete state(s) at that point, and that the abstract interpretation of each instruction preserves the modeling relation. [1] In general, the analysis-specific conditions involve the execution of at most one simple instruction; hence, their proofs are significantly easier than the entire correctness proof, and, hopefully, large parts of these proofs can be automatic. Still, as the execution of an invoked procedure may involve many instructions, some sort of user-supplied frame theorem [2] is required in the case of a call instruction.

Notice that we study the correctness of the high-level analysis specification and not the correctness of a particular analysis implementation. Still, if we have a high-level analysis specification, we can automatically generate an implementation that solves the dataflow equations [1,27].

Contributions: This technical report makes the following contributions:

- We present a methodology for doing machine-checkable correctness proofs for dataflow analyses. Our methodology reduces the proof effort by focusing only on a clear set of high-level analysis-specific conditions.
- We present experience in applying our methodology for proving the correctness of three related dataflow analyses in the interactive theorem prover Athena. In general, the proof effort was reasonable, and the resulting proofs are similar to paper-and-pencil proofs. Our proofs are available online from `http://www.mit.edu/~salcianu/df-proofs`.

Paper structure: Section 2 introduces a simple language that we use for the presentation of our ideas. Section 3 formally defines the forward intra-procedural dataflow analyses and their correctness. Section 4 presents the example of a constant propagation analysis. Section 5 presents our correctness proof methodology. Sections 6 and 7 briefly introduce Athena and describe our experience in using it to prove the correctness of three related analyses. Finally, Section 8 discusses related work, and Section 9 concludes.

2 Program Representation and Semantics

This technical report uses the following notation: S^* is the set of all finite lists with elements from the set S; S^+ is similar to S^*, but contains only non-empty lists. We write $e : l$ for the list obtained by adding the element e at the head of the list l. If f is a function $A \to B$, $f\,[a \mapsto b]$ is the function that behaves exactly like f, except that $f\,(a) = b$. For each relation $\mathcal{R} \subseteq A \times B$, we write

[1] There is also a third correctness condition that we explain later in the paper.

[2] Essentially, a frame theorem is a reduced procedure specification: e.g., a procedure does not change the local variables of its caller.

$a\ \mathcal{R}\ b$ for $(a, b) \in \mathcal{R}$; $\mathcal{R}^*$ denotes the transitive and reflexive closure of $\mathcal{R}$.

We present our ideas in the context of a simple language with recursive procedures and local variables. Figure 1 presents the mathematical objects for the program representation and semantics. A program P is a mapping from a subset of procedure names to the corresponding procedures. Each procedure consists of a list of formal parameters and a list of instructions. Instructions have the expected semantics; e.g., "v `:=` `ct`" loads the constant ct into the local variable v; "`if(`v `== 0) goto` a" jumps to the a-th instruction from the current procedure, etc.

$$
\begin{array}{rcllll}
P & \in & \mathit{Program} & = & \{P' \in A \to \mathit{Proc} \mid A \subseteq \mathit{ProcName},\ \mathtt{main} \in A\} \\
p & \in & \mathit{ProcName} & & \text{procedure names} \\
 & & \mathit{Proc} & = & \mathit{Var}^* \times \mathit{Instr}^* \\
v & \in & \mathit{Var} & & \text{local variables (including formal parameters)} \\
 & & \mathit{Instr} & ::= & v\ \mathtt{:=}\ \mathtt{ct} \mid v_1\ \mathtt{:=}\ v_2 \mid v\ \mathtt{:=}\ v_1\ \mathit{bop}\ v_2 \\
 & & & \mid & \mathtt{if(}v\ \mathtt{==\ 0)\ goto}\ a \\
 & & & \mid & v\ \mathtt{:=\ call}\ p\ \mathtt{(}v_0\mathtt{,}\ v_1\mathtt{,}\ \ldots\mathtt{,}\ v_{k-1}\mathtt{)} \mid \mathtt{return}\ v \\
\mathit{bop} & \in & \mathit{Bop} & = & \{\mathtt{+}, \mathtt{-}, \mathtt{*}, \mathtt{mod}, \mathtt{div}, <, \leq, >, \geq, \mathtt{==}, \wedge, \vee\} \\
a & \in & \mathbb{N} & & \text{addresses inside a procedure}
\end{array}
$$

$$
\begin{array}{rcllll}
c & \in & \mathit{State} & = & \mathit{Stack} & \text{Concrete states} \\
S & \in & \mathit{Stack} & = & (\mathit{VState} \times \mathit{Lab} \times \mathit{Var})^+ & \text{Execution stack} \\
V & \in & \mathit{VState} & = & \mathit{Var} \to \mathbb{Z} & \text{State of local variables} \\
\mathit{lb} & \in & \mathit{Lab} & = & \mathit{ProcName} \times \mathbb{N} & \text{Program labels} \\
 & & c^{\mathit{init}} & = & \langle \lambda v.0, \langle \mathtt{main}, 0\rangle, v_0\rangle & \text{Initial program state}
\end{array}
$$

Fig. 1. Program representation and semantics.

Each instruction has a label $\mathit{lb} \in \mathit{Lab} = \mathit{ProcName} \times \mathbb{N}$: $\langle p, i\rangle$ is the label of the i-th instruction from the procedure named p. $\mathit{instrAt}_P\,(\mathit{lb})$ denotes the instruction from label lb in program P. For most instructions, control goes from label $\mathit{lb} = \langle p, a\rangle$ to $\mathit{next}\,(\langle p, a\rangle) = \langle p, a+1\rangle$. For a jump instruction, control can also go to the jump target. $\mathit{pred}_P\,(\mathit{lb})$ denotes the set of control flow predecessors of label lb, i.e., labels of the instructions that may be executed right before executing the instruction from label lb.

The meaning of our programs is given by a small-step operational semantics, informally called the *concrete semantics*. Currently, a concrete state contains only the execution stack. Each stack frame corresponds to a procedure activation, and contains 1) the state of the local variables of the procedure, 2) the current label inside the procedure, and 3) the caller variable that will receive the returned value. The state of the local variables is a total function from variables to integers; on procedure entry, parameters are initialized with the values of the actual arguments; all other local variables are initialized to 0.[3] All variables have integer values; booleans are encoded as integers in a C-like fashion. The auxiliary function pc takes a concrete state

[3] Only variables mentioned in the program can have a non-zero value; hence, the state of local variables has a finite representation.

$c = \langle V, lb, v_r \rangle : S_{\text{tail}}$, and returns the label of the instruction about to be executed, i.e., lb.

The execution of a program P starts with the first instruction from the distinguished procedure `main`, i.e., $instrAt_P\,(\langle \texttt{main}, 0 \rangle)$. An execution of P is a (possibly infinite) chain of transitions: $c^{init} = c_0 \rightsquigarrow_P c_1 \rightsquigarrow_P \ldots \rightsquigarrow_P c_i \rightsquigarrow_P c_{i+1} \rightsquigarrow_P , \ldots$. The transition $c_i \rightsquigarrow_P c_{i+1}$ executes the instruction at label $pc\,(c_i)$ in program P, i.e., the instruction $instrAt_P\,(pc\,(c_i))$. The transition relation $\rightsquigarrow_P \; \subseteq State \times State$ is defined by a case analysis of the instruction executed in that step. Here is a sample case:

$$\begin{array}{c} \langle V, lb, v_r \rangle : S_{\text{tail}} \rightsquigarrow_P \langle V\,[v \mapsto \text{ct}]\,, next\,(lb)\,, v_r \rangle : S_{\text{tail}} \\ \text{where } \; instrAt_P\,(lb) = \text{“}v \texttt{ := ct}\text{”}. \end{array} \qquad \text{[ldc]}$$

Appendix A presents the complete definition of the transition relation $\rightsquigarrow_P$. A state is final if its stack has a single frame and the instruction about to be executed is “`return` v.” The value of v is the result of the program.

3 Forward Intra-Procedural Dataflow Analyses

Definition 3.1 A forward intra-procedural dataflow analysis $\mathcal{A}$ is a function that, for each program P, produces a tuple $\langle \mathcal{L}_P, [\![.]\!]_P, A_P^{init}, \mathcal{M}_P, A_P \rangle$, consisting of:

(i) A *property lattice* $\mathcal{L}_P$, with a join operator $\sqcup_{\mathcal{L}_P}$ and an induced ordering relation $\sqsubseteq_{\mathcal{L}_P}$ (we ignore the subscript $\mathcal{L}_P$ whenever it is obvious from the context).

(ii) A family of monotonic *transfer functions* $[\![.]\!]_P : Lab_P \to \mathcal{L}_P \to \mathcal{L}_P$, where Lab_P denotes the set of labels from program P. Intuitively, for each label lb from P, $[\![lb]\!]_P$ takes the analysis fact for the program point before label lb, and returns the analysis fact for the program point after label lb.

(iii) An *initial analysis fact* $A_P^{init} \in \mathcal{L}_P$ for the entry point of each procedure.

(iv) A *modeling relation* $\mathcal{M}_P \subseteq State \times \mathcal{L}_P$; $c\ \mathcal{M}_P\ l$ iff the analysis fact $l \in \mathcal{L}_P$ models the concrete state c.

(v) A function $A_P : Lab_P \to \mathcal{L}_P$, where Lab_P is the set of labels from P. $A_P\,(lb)$ is the analysis fact for the program point right before label lb. A_P satisfies the *dataflow equations*:

$$\forall lb \in Lab_P \;.\; A_P\,(lb) \sqsupseteq \begin{cases} A_P^{init} & \text{if } lb = \langle p, 0 \rangle \\ \displaystyle\bigsqcup_{lb_2 \in pred_P(lb)} [\![lb_2]\!]_P(A_P\,(lb_2)) & \text{otherwise} \end{cases}$$

The dataflow equations simply state that for each procedure we start with an initial analysis fact for the procedure entry point, and next use the transfer functions to propagate this information along the control flow graph; we use $\sqcup$ in the control flow join points. Strongly connected components in the control flow graph require fixed-points. The dataflow equations use $\sqsupseteq$, instead of equality, to allow aggresive fixed-point approximations.[4]

We are now ready to define the correctness of an analysis. We first introduce a predicate to identify the reachable program states, the only states the analysis cares about:

Definition 3.2 [Reachable States] $\mathit{reachable}_P\,(c) \stackrel{\text{def}}{\leftrightarrow} c^{\mathit{init}} \leadsto_P{}^* \; c.$

Definition 3.3 [Analysis Correctness] Consider a forward intra-procedural analysis $\mathcal{A}$ that assigns to each program P the tuple $\langle \mathcal{L}_P, [\![.]\!]_P, A_P^{\mathit{init}}, \mathcal{M}_P, A_P \rangle$, as required by Def. 3.1. Analysis $\mathcal{A}$ is *correct* iff

$$\forall P \in \mathit{Program}.\ \forall c \in \mathit{State}.\ \mathit{reachable}_P\,(c) \;\rightarrow\; c\ \mathcal{M}_P\ A_P\,(\mathit{pc}\,(c))$$

In plain English, for each reachable concrete state c, $\mathit{pc}\,(c)$ represents the program point reached by the program execution (i.e., the label of the instruction about to be executed); the analysis correctness condition requires that the analysis result for $\mathit{pc}\,(c)$, i.e., $A_P\,(\mathit{pc}\,(c))$, models the concrete state c, with respect to the intended modeling relation $\mathcal{M}_P$.

4 Example: Constant Propagation

Preliminaries: If A is a set, $L = \mathit{Lift}\langle A \rangle$ is the lattice with the elements $\top$, $\bot$, $\mathit{lift}\,(x)$ for any $x \in A$, and the following ordering relation: $\bot$ is smaller than any element, any element is smaller than $\top$, and distinct elements of L are otherwise incomparable. Formally,

$$\begin{array}{l} l_1 \sqsubseteq_L l_2 \stackrel{\text{def}}{\leftrightarrow} (l_1 = \bot_L) \vee (l_2 = \top_L) \vee (l_1 = l_2) \\ l_1 \sqcup_L \bot = l_1;\quad \bot \sqcup_L l_2 = l_2;\quad l \sqcup_L l = l; \text{otherwise, } l_1 \sqcup_L l_2 = \top \end{array}$$

If A is a set, and B is a lattice, $F = A \rightarrow B$ is a lattice with the following element-wise ordering relation and join operator:

$$f_1 \sqsubseteq_F f_2 \stackrel{\text{def}}{\leftrightarrow} \forall a \in A.\ f_1\,(a) \sqsubseteq_B f_2\,(a) \qquad f_1 \sqcup_F f_2 = \lambda a.\ f_1\,(a) \sqcup_B f_2\,(a)$$

Constant Propagation: Figure 2 presents the specification of the constant propagation analysis. The property lattice for the constant propagation analysis is $M = \mathit{Var}_P \rightarrow \mathit{Lift}\langle \mathbb{Z} \rangle$, where Var_P is the set of all variables from the program P. For each label lb, the constant propagation analysis computes a function m that maps each local variable to $\bot$, $\top$, or $\mathit{lift}\,(x)\,, x \in \mathbb{Z}$. The modeling relation (also presented in Fig. 2), requires that for each local variable v that has value $V\,(v) = x$ in the concrete state, $m\,(v)$ is either $\top$ (that approximates all values), or $\mathit{lift}\,(x)$. Therefore, if $m\,(v) = \mathit{lift}\,(x)$ *and the analysis is correct* (according to Def. 3.3), we know that in any reachable execution state

[4] E.g., widening [8, 9]; one can also imagine fixed-point solvers that jump to $\top$ after a certain number of iterations failed to reach a fixed point.

Analysis Property Lattice: $M = Var_P \to Lift\langle\mathbb{Z}\rangle$

Analysis Modeling Relation:

$$c \; \mathcal{M}_P \; m \;\overset{\text{def}}{\leftrightarrow}\; \exists V \in VState.\, \exists lb \in Lab.\, \exists v_r \in \mathbb{N}.\, \exists S_{\text{tail}} \in Stack.$$
$$(c = \langle V, lb, v_r\rangle : S_{\text{tail}}) \wedge$$
$$(\forall v \in Var_P \,.\; (m(v) = \top) \vee (m(v) = lift\,(V(v)))$$

Initial Analysis Fact for Procedure Entry Points: $A_P^{init} = \lambda v.\, \top$.

Transfer Functions:

$instrAt_P\,(lb)$	$[\![lb]\!]_P(m)$
v `:=` `ct`	$m\,[v \mapsto lift\,(\text{ct})]$
v_1 `:=` v_2	$m\,[v_1 \mapsto m\,(v_2)]$
v `:=` v_1 bop v_2	$m\,[v \mapsto \top]$
v `:=` `call` p `(`v_0`,` …`,` v_{k-1}`)`	$\lambda v'.\, \top$
otherwise (`if` and `return`)	m (unchanged)

Fig. 2. Specification of a simple constant propagation analysis.

at label lb, v has value x. A program optimization can use this guarantee to safely replace any use of v at label lb with the constant x.

The transfer functions map v to "$lift\,($`ct`$)$" for a "v `:=` `ct`" instruction and propagate constants across "v_1 `:=` v_2" copy instructions. The transfer functions for binary operations and for calls are very conservative; we discuss more precise transfer functions in Section 7.

5 Analysis Correctness Proof Methodology

This section presents three analysis-specific conditions. As we prove in Theorem 5.2, these conditions are sufficient for correctness.

Condition 1 *Upper approximations preserve the modeling relation:*[5]

$$\forall P \in Program.\ \forall c \in State.\ \forall l_1, l_2 \in \mathcal{L}_P.$$
$$reachable_P\,(c) \wedge (c\; \mathcal{M}_P\; l_1) \wedge (l_1 \sqsubseteq l_2) \to (c\; \mathcal{M}_P\; l_2)$$

Condition 2 *Initial analysis facts are correct:*

$$\forall P \in Program.\ \forall c \in State.\ \forall p \in \mathbb{N}.$$
$$reachable_P\,(c) \wedge (pc\,(c) = \langle p, 0\rangle) \to c\; \mathcal{M}_P\; A_P^{init}$$

The next condition uses the *intra-procedural transition relation* $\twoheadrightarrow_P$; $\twoheadrightarrow_P$ is similar to the transition relation $\rightsquigarrow_P$ except that, in the case of a call instruction, $\twoheadrightarrow_P$ relates the program states before and after the call by "skipping" over all the instructions from the invoked procedure and its transitive callees.

[5] This corresponds to our choice that in the property lattice smaller should mean "more precise" and bigger should mean "safer" (the opposite choice is also possible). Here is a definition of these terms: according to $\mathcal{M}_P$, an analysis fact $l \in \mathcal{L}_P$ models several concrete states. The fewer states l models, the more precise and less safe l is.

Condition 3 *Commuting diagram (instructions preserve modeling):*
$\forall P \in \mathit{Program}.\ \forall c_1, c_2 \in \mathit{State}.\ \forall l \in \mathcal{L}_P.$
$\mathit{reachable}_P\,(c_1) \wedge (c_1\ \mathcal{M}_P\ l) \wedge (c_1 \twoheadrightarrow_P c_2) \rightarrow\ \ c_2\ \mathcal{M}_P\ [\![pc\,(c_1)]\!]_P(l)$

Definition 5.1 $c_1 \twoheadrightarrow_P c_2$ iff one of the following conditions is true:

(i) The instruction about to be executed in c_1 is not a call or a return instruction, and $c_1 \rightsquigarrow_P c_2$; OR

(ii) The instruction about to be executed in c_1 is a call, and c_2 is the concrete state immediately after the return from that call, i.e.,

$$\exists c_a, c_b \in \mathit{State}.\ (c_1 \rightsquigarrow_P c_a) \wedge (c_a\ \ (\rightsquigarrow_P^{|c_a|})^*\ c_b) \wedge (c_b \rightsquigarrow_P c_2) \wedge (|c_1| = |c_2|)$$
$$\text{where}\ \ c \rightsquigarrow_P^k c' \overset{\text{def}}{\leftrightarrow} (c \rightsquigarrow_P c') \wedge (|c| \geq k) \wedge (|c'| \geq k)$$
$$|c| \text{ denotes the height of the stack in state } c$$

In plain English, c_a is the state immediately after call, c_b is the state immediately before the return from the invoked procedure, and none of the transitions between c_a and c_b return from the invoked procedure, i.e., all transitions from c_a to c_b keep the stack at least as high as the stack from c_a.

Note: Condition 3 can be further split into simpler parts, by specializing it for each kind of instructions.

The next theorem is our only paper-and-pencil proof and shows that conditions 1, 2, and 3 are sufficient for the correctness of our analysis:

Theorem 5.2 *If an analysis $\mathcal{A}$ satisfies conditions 1, 2, and 3, then $\mathcal{A}$ is correct.*

Proof. Let's pick an arbitrary program P, and an arbitrary state $c \in \mathit{State}$, such that $\mathit{reachable}_P\,(c)$. We shall prove that $c\ \mathcal{M}_P\ A_P\,(pc\,(c))$.

Let $pc\,(c) = \langle p, a\rangle$, i.e., c is about to execute the a-th instruction from the p-th procedure. As c is reachable, there exists an execution $c^{init} = c_0 \rightsquigarrow_P c_1 \rightsquigarrow_P \ldots \rightsquigarrow_P c_k = c$. Let s be the largest $i, 1 \leq i \leq k$, such that $|c_i| = |c_k|$, and $|c_{i-1}| = |c_k| - 1$, or 0 if no such i exists. In both cases, c_s is the concrete state right at the beginning of procedure p's invocation that c is still executing.

The chain of transitions $c_s \rightsquigarrow_P c_{s+1} \rightsquigarrow_P \ldots \rightsquigarrow_P c_k$ contains: 1) transitions for the instructions from procedure p, and 2) transitions for the instructions from procedures invoked by p. We "skip" over the latter transitions by using the intra-procedural transition relation: $c_s = c_{s_0} \twoheadrightarrow_P c_{s_1} \twoheadrightarrow_P \ldots \twoheadrightarrow_P c_{s_t} = c_k$. Each transition $c_{s_i} \twoheadrightarrow_P c_{s_{i+1}}$ corresponds to either 1) a non-call instruction from procedure p or 2) a call from p to a procedure p', the instructions from p' and its transitive callees, and the return back into p.

We prove by induction that $\forall i.\ 0 \leq i \leq t\ \rightarrow\ c_{s_i}\ \mathcal{M}_P\ A_P\,(pc\,(c_{s_i}))$, and next instantiate i with t to prove the final result.

Base case: $i = 0$. As $c_{s_0} = c_s$ is the state at the beginning of p, $A_P\,(pc\,(c_{s_0})) \sqsupseteq A_P^{init}$ (see dataflow equations in Definition 3.1). By Cond. 2, $c_{s_0}\ \mathcal{M}_P\ A_P^{init}$. By Cond. 1, $c_{s_0}\ \mathcal{M}_P\ A_P\,(pc\,(c_{s_0}))$.

Induction step: Suppose $c_{s_i}\ \mathcal{M}_P\ A_P\,(pc\,(c_{s_i}))$, and let $l_2 = [\![pc\,(c_{s_i})]\!]_P(A_P\,(pc\,(c_{s_i})))$. By Cond. 3, $c_{s_{i+1}}\ \mathcal{M}_P\ l_2$. As $pc\,(c_{s_i}) \in pred_P\left(pc\left(c_{s_{i+1}}\right)\right)$, by the dataflow equations from Def. 3.1, $A_P\left(pc\left(c_{s_{i+1}}\right)\right) \sqsupseteq l_2$. By Cond. 1, $c_{s_{i+1}}\ \mathcal{M}_P\ A_P\left(pc\left(c_{s_{i+1}}\right)\right)$. This completes our proof. □

Additional proofs: This paper is focused on partial correctness. So far, we did not discuss the proofs that $\mathcal{L}_P$ is really a lattice, nor the proof of termination. Usually, $\mathcal{L}_P$ is obtained by standard lattice constructors: e.g., the product of two lattices, the powerset of a set, etc., that are guaranteed to produce a lattice. For termination, we have to prove that $\mathcal{L}_P$ does not have any infinite ascending chain (usually proven by a finiteness argument), and that all transfer functions are monotonic.

Backward analyses: Our methodology can easily be adapted to handle backward analyses too: Cond. 2 will refer to the procedure exit points, and Cond. 3 will propagate the modeling relation "backward:"

$$\begin{aligned}&\forall P \in \mathit{Program}.\ \forall c_1, c_2 \in \mathit{State}.\ \forall l \in \mathcal{L}_P.\\ &\quad \mathit{reachable}_P\,(c_1) \wedge (c_2\ \mathcal{M}_P\ l) \wedge (c_1 \twoheadrightarrow_P c_2) \rightarrow\ c_1\ \mathcal{M}_P\ [\![pc\,(c_1)]\!]_P(l)\end{aligned}$$

6 Brief Description of Athena

Athena [4,3] is a new interactive theorem proving system for multi-sorted first-order logic that has facilities for structured proof representation and proof checking, automated theorem proving, and model generation. Athena also provides a Scheme-like higher-order functional programming language, and a proof abstraction mechanism for expressing arbitrarily complicated inference *methods* in a way that guarantees soundness, akin to the tactics and tacticals of LCF-style systems such as HOL [12] and Isabelle [20]. Proof automation is achieved in two ways: first, through user-formulated proof methods; and second, through the seamless integration of state-of-the-art ATPs such as Vampire [25] and Spass [26] as primitive black boxes for general reasoning. For proof representation and checking, Athena uses a block-structured Fitch-style natural deduction calculus with novel syntactic constructs and a formal semantics based on the abstraction of *assumption bases* [2]. Fitch-style natural deduction [21] is a way of structuring proofs so that they mirror the proofs presented by mathematicians in practice; special emphasis is placed on modeling hypothetical reasoning and keeping track of the scope of assumptions.

The assumption base contains the propositions that are known to be valid at a specific point in the proof. Each (sub)proof adds the proven proposition to the assumption base. To prove propositions of the form $P_1 \rightarrow P_2$, Athena provides special constructs that add P_1 to the assumption base *during* the dynamic scope of P_2's proof. A proof consists of either the application of primitive inference rules (i.e., modus ponens), or the invocation of an external ATP. If the external ATP does not succeed in a certain time bound, we do a few steps of the proof, and next try the ATP again on a simpler proposition.

Common proofs can be abstracted into user-defined methods.

Among other applications, Athena has been used to implement parts of a proof-emitting optimizing compiler [17] and to verify the core operations of a Unix-like file system [5]. [4] contains a list of applications, along with a tutorial on Athena's syntax and semantics.

7 Experience

We used Athena to formalize and prove the correctness of three related dataflow analyses. For each analysis, we proved the three conditions from Section 5 and the monotonicity of the transfer functions.

The first analysis is the simple constant propagation analysis from Section 4. The second analysis extends constant propagation with constant folding: The transfer function for a "v `:=` v_1 *bop* v_2" statement computes the result of the binary operation if the analysis already knows that both operands are constants. The third analysis improves over the second one by using a more precise transfer function for call statements of the form "v `:= call` p `(...)`" that maps only v to $\top$ (instead of all local variables). The correctness proof of the third analysis requires the proof of a frame condition, stating that the execution of the transitively invoked procedures do not change the caller's local variables, except for the variable that stores the result of the call.

The table below presents an overview of the formalization and proof effort (including the proofs of all intermediate results, e.g., the frame condition). During the proofs for the simple constant propagation, the language formalization went though several debugging and simplification iterations. Therefore, it is impossible to separate the time spent on the first two entries of the table below.

	Formalization	Proofs	Total	Human
	[# non-commented, non-empty lines]			Effort
Language + semantics	457	164	621	15 days
Simple ct. propagation	174	262	436	(together)
+ constant folding	+50	+71	+121	3 hours
+ more precise transfer function for call	+4	+685	+689	5 days

The rest of this section gives a brief overview of our work in Athena. All proofs are available online from

http://www.mit.edu/~salcianu/df-proofs

Our correctness conditions are universally quantified over all programs. To prove them, we pick one unknown program P, formalize the structure of P, its semantics, and the analysis for P, and prove (in Athena) the correctness conditions instantiated for P; next, we generalize over P.

We introduce Athena sorts (similar to types in a programming language) for the sets from the program representation and semantics. We also introduce function symbols; each relation/predicate is modeled as a function with boolean values. For each function, we declare its signature and a few axioms. The signatures allows Athena to do Hindley-Milner-like sort-inference.

Language formalization: We formalize the program structure and semantics only once for all the analyses. We declare the sort `VarP` for P's variables (i.e., the set Var_P), the sort `ProcNameP` for P's procedure names, and the sort `Instr` for instructions.[6] The analyzed program is declared as an (uninterpreted) function from procedure names to procedures:

```
(domain VarP)             # Sort: variables from P.
(domain ProcNameP)        # Sort: procedure names in P.
(declare main ProcNameP)  # Name of the main procedure (element of the sort ProcNameP)
(datatype Instr           # Sort: instructions from P.
  (ldc VarP Num)          #   Constructors correspond to different
  (copy VarP VarP)        #   kinds of instructions.
  ...)
(datatype Proc   # Sort: procedures; one procedure = list of parameters +
  (proc (List-Of VarP) (List-Of Instr)))         # list of instructions.
(declare P (-> (ProcNameP) Proc))  # The analyzed program.
```

The formalization of the operational semantics introduces additional sorts, axioms for the transition relation `step` (i.e., $\rightsquigarrow_P$), and many auxiliary axioms. The Athena code closely matches the definitions from Section 2 (except that Athena uses prefix notation):

```
(datatype StackFrame (stackFrame VState Label VarP))  # VState, Label definitions ommited.
(datatype State (state (List-Of StackFrame)))
# ...
(declare step (-> (State State) Boolean))  # Operational semantics transition relation.
(define step-axiom-ldc  # Axiom: transitions for "v := ct" statements.
  # Identifiers starting with "?" denote variables in the object logic.
  (forall ?vstate ?label ?vr ?tail ?cs2 ?v ?n
          (let ((cs1 (state (Cons (stackFrame ?vstate ?label ?vr) ?tail))))
            (if (currentInstr cs1 (ldc ?v ?n))
                (iff (step cs1 ?cs2)
                     (= ?cs2  (state (Cons (stackFrame (updateVS ?vstate ?v ?n)
                                                       (next ?label)
                                                       ?vr)
                                           ?tail)) ))))))
(assert step-axiom-ldc)  # Add this axiom to the assumption base.
```

Analysis formalization: We introduce a polymorphic sort for lattices of the form $Lift\langle S\rangle$ (that can be instantiated for any set S), and a sort for the analysis lattice $Var_P \rightarrow Lift\langle \mathbb{Z}\rangle$; for each sort, we define the corresponding ordering relations:

```
# Sort: polymorphic datatype for lifted domains
(datatype (Lift S)
    bottomLift  (lift S)  topLift)
# Definition of orderLift ommited for brevity.
# Sort for the analysis lattice. To encode Var_P → Lift⟨Z⟩ in the first-order logic
```

[6] There are several Athena keywords for introducing sorts. The simplest is `domain`; `datatype`/`structure` introduce a sort too, but they also introduce function symbols for the datatype constructors; a structural induction mechanism; axioms stating that each element of the sort is obtained by using a constructor; and, in the case of `datatype`, axioms stating that the domain is freely generated. For non-`datatype` domains, the user can specify a different equality relation.

```
# of Athena, we use a representation similar to a list of association pairs.
(structure M
    allTop                        # allTop encodes λv. ⊤
    (updateM M VarP (Lift Num)))  # (updateM m v x) encodes m[v ↦ x]
# The axioms for  updateM (ommited for brevity) state that (lookUpM ?x ?m) returns the
# first association for ?x in the mapping ?m, or topLift if no such association exists.
# Order relation for the analysis lattice.
(declare orderM (-> (M M)  Boolean))
(define orderM-axiom
  (forall ?m1 ?m2
    (iff (orderM ?m1 ?m2)
         (forall ?x  (orderLift (lookUpM ?x ?m1)
                                (lookUpM ?x ?m2))))))
```

The definitions for the modeling relation and the transfer functions closely correspond to the definitions from Section 4. The predicate (`model` c m) holds iff $c\ \mathcal{M}_P\ m$; similarly, (`tf` lb m_1 m_2) holds iff $[\![lb]\!]_P(m_1) = m_2$.

Proofs: We prove the first two correctness conditions automatically, using Athena's interface to Vampire [25]. Cond. 2 is the easiest: Vampire proves that Cond. 2 follows from the set of all axioms; for Cond. 1, we had to pass only a subset of the axioms (Vampire takes too much time if we give it the full set of axioms). Condition 3 requires significantly more effort. The Athena definition of Cond. 3 closely follows the definition from Section 5:

```
(define commuting-diagram  # Condition 3.
  (forall ?cs1 ?cs2 ?m1 ?m2 ?lab
    (if (and wfProg  # Analyzed progran is well-formed; e.g., no invalid jumps.
             (reachableState ?cs1)  # reachable_P(?cs1)
             (model ?cs1 ?m1)       # ?cs1 M_P ?m1
             (ipStep ?cs1 ?cs2)     # ?cs1 ↠_P ?cs2
             (pc ?cs1 ?lab)         # pc(?cs1) = ?lab
             (tf ?lab ?m1 ?m2))     # [[?lab]]_P(?m1) = ?m2
        (model ?cs2 ?m2))))  # ?cs2 M_P ?m2
```

To prove `commuting-diagram`, we perform a case analysis on the instruction from `?lab`, and prove each case as a separate theorem. The modeling relation requires a certain condition to hold for each local variable v (see Fig. 2); accordingly, most of the proofs do a case analysis on whether v is the variable being modified by the instruction or not. The proofs are a combination of manual and automatic sub-proofs. The entire proof scripts are available online.

8 Related Work

Compiler correctness has always been an active research area. [13] presents a paper-and-pencil correctness proof for an entire Scheme compiler; small parts of the proof were later formalized in Isabelle [7]. The compiler examined by [13] consists mostly of syntax-directed conversion steps. By comparison, we focus on machine-checkable correctness proofs for dataflow analyses.

The Verifix project [11] uses program checkers to dynamically check the correct compilation of a given program. Formal methods can later be used to prove the correctness of the program checkers. The Credible Compilation framework [22, 17] allows a compiler optimization to output, in addition to the optimized program, a proof that the optimized program is semantically equivalent to the original one. The proof can be checked by a small trusted proof checker; if the proof does not check, the compiler can simply ignore the problematic optimization. The Translation Validation Infrastructure (TVI) [18] attempts to accomplish the same goals as Credible Compilation, but without any assistance from the compiler. TVI attempts to discover an equivalence proof (instead of just checking a proof produced by the compiler). When applied to optimization stages of the GNU C Compiler compiling real applications, TVI generates many simulation invariants and the custom-built theorem prover manages to prove almost, but not all of them. [7] A parallel, similar project, Translation Validation, succeeded in handling several aggressive optimizations that do not preserve the loop structure of the program [28].

The correct assumption behind the aforementioned four projects is that it is much easier to check the correctness of an optimization on a particular program than for all programs. Also, these approaches can detect errors in the implementation of conceptually correct analyses. Still, we believe that proving the correctness of an analysis for all possible programs is very important for the high-level design of the analysis, and can be a useful complement for translation validation approaches.

Cobalt [15] is a framework for defining syntax-directed analyses and optimizations. Cobalt requires the analysis designer to specify an analysis invariant, and next uses the theorem prover Simplify [10] to prove the correctness of the optimizations. However, Cobalt does not deal with classical dataflow analyses: It does not allow the definition of analysis property lattices, transfer functions, etc. Instead, Cobalt supports analyses expressed as reachability conditions on the control flow graph. [8]

Very close to our research is the work from [14] and [6]. [14] presents a correctness proof in Isabelle [20] for the Java Bytecode Verifier (that includes a dataflow analysis for computing the stack typing at various program points); [6] presents a constructive proof in Coq [24] for a dataflow analysis for JavaCard Bytecode. Both of these papers present work of excellent quality, and the complete resulting proofs are available online. [9] They differ from our work in several respects. First, we place a heavier emphasis on proof readability. We aim at allowing analysis designers to write readable proofs, structurally similar to the ones they would write on paper, but with the advantage of machine-checkability; we invite the interested readers to

[7] E.g., as explained in [18, Section 6], for the case of `gcc` compiling itself, 3.5% of the constraints generated for the common-subexpression-elimination (CSE) optimization are not simplified, i.e., automatically proven by the theorem prover.

[8] A forthcoming publication [16] describes Rhodium, a successor of Cobalt that allows the definition of dataflow analyses that use a restricted lattice (a powerset of all user-defined analysis facts). Our work aims at proving correctness of dataflow analyses that use a wider range of lattices.

[9] From `http://www.doclsf.de/papers/tcs02.html` for [14], respectively from `http://www.irisa.fr/lande/pichardie/CarmelCoq/Esop04` for [6].

contrast our proofs (in terms of readability) with the proofs from [14] and [6]. Second, Athena allows significant automation that reduces the overall proof effort. This is done by using Fitch-style tactics capable of incorporating arbitrary computation into the proof-search process; and through the seamless integration of state-of-the-art automated theorem provers. To the best of our knowledge, the official versions of Coq and Isabelle are not interfaced with external ATPs yet. Third, we are more focused on the presentation of a clear framework for proving the correctness of a broad class of dataflow analyses, instead of getting very focused on one specific analysis.

9 Conclusion

This technical report presents our experience with dataflow analysis proofs in the interactive theorem prover Athena. Our experience indicates that such proofs are possible, and that modern automated theorem provers increase the level of automation. Still, the state-of-the-art in theorem proving is very far from full automation, and significant human effort is required.

Acknowledgments

We are grateful to Darko Marinov for many useful discussions on analysis correctness, credible compilation, and theorem proving. This research was supported by the DARPA Cooperative Agreement FA 8750-04-2-0254, DARPA Contract 33615-00-C-1692, the Singapore-MIT Alliance, and the NSF Grants CCR-0341620, CCR-0325283, and CCR-0086154.

References

[1] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *Proc. 2nd International Static Analysis Symposium*, pages 33–50, 1995.

[2] K. Arkoudas. *Denotational Proof Languages*. PhD thesis, Massachusetts Institute of Technology, 2000.

[3] K. Arkoudas. Specification, abduction, and proof. In *Second International Symposium on Automated Technology for Verification and Analysis*, Taiwan, October 2004.

[4] K. Arkoudas and others. Athena. `http://www.pac.lcs.mit.edu/athena`.

[5] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. Verifying a file system implementation. In *Sixth International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, Seattle, Nov 8-12, 2004 2004.

[6] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyzer in constructive logic. In *Proc. 13th ESOP*, 2004.

[7] B. Ciesielki and M. Wand. Using the theorem prover Isabelle-91 to verify a simple proof of compiler correctness. Technical Report NU-CCS-91-20, Northeastern University College of Computer Science, December 1991.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*, 1977.

[9] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Lecture Notes in Computer Science*, pages 269–295, 1992.

[10] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto, 2003.

[11] G. Goos and W. Zimmermann. Verification of compilers. In *Correct System Design*, 1999. LNCS 1710.

[12] M. J. C. Gordon and T. F. Melham. *Introduction to HOL, a theorem proving environment for higher-order logic*. Cambridge University Press, Cambridge, England, 1993.

[13] J. Guttman, J. Ramsdell, and M. Wand. VLISP: a verified implementation of Scheme. *Lisp and Symbolic Computing*, 8(1–2):33–110, Mar. 1995.

[14] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3), April 2003.

[15] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proc. ACM PLDI*, pages 220–231. ACM Press, 2003.

[16] S. Lerner, T. Milstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proc. 32nd ACM POPL*, 2005. To appear.

[17] D. Marinov. Credible compilation. Master's thesis, MIT Laboratory for Computer Science, 2000.

[18] G. C. Necula. Translation validation for an optimizing compiler. In *Proc. ACM PLDI*, pages 83–95, Vancouver, British Columbia, Canada, June 2000.

[19] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[20] L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Number 828 in LNCS. Springer-Verlag, 1994.

[21] F. J. Pelletier. A Brief History of Natural Deduction. *History and Philosophy of Logic*, 20:1–31, 1999.

[22] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[23] A. Salcianu. Pointer analysis and its applications to Java programs. Master's thesis, MIT Laboratory for Computer Science, 2001.

[24] The Coq Development Team; INRIA LogiCal Project. The Coq proof assistant - official website. `http://coq.inria.fr`.

[25] A. Voronkov et al. The anatomy of Vampire (implementing bottom-up procedures with code trees). *Journal of Automated Reasoning*, 15(2):237–265, 1995.

[26] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.

[27] K. Yi and W. L. Harrison III. Automatic generation and management of interprocedural program analyses. In *20th ACM POPL*, 1993.

[28] L. Zuck, A. Pnueli, Y. Fang, B. Goldberg, and Y. Hu. Translation and run-time validation of optimized code. In *Workshop on Run-Time Verification*, 2002.

A Transition Relation for Concrete Semantics

This section presents the remaining cases from the definition of the operational semantics transition relation $\rightsquigarrow_P$:

$$\begin{array}{c} \langle V, lb, v_r\rangle : S_{\text{tail}} \rightsquigarrow_P \langle V[v_1 \mapsto V(v_2)], next(lb), v_r\rangle : S_{\text{tail}} \\ \text{where } \ instrAt_P(lb) = \text{“}v_1 \texttt{ := } v_2\text{”} \end{array} \qquad \text{[copy]}$$

$$\begin{array}{l} \langle V, lb, v_r\rangle : S_{\text{tail}} \rightsquigarrow_P \langle V[v \mapsto x], next(lb), v_r\rangle : S_{\text{tail}} \\ \quad \text{where } \ instrAt_P(lb) = \text{“}v \texttt{ := } v_1 \ bop \ v_2\text{”} \\ \qquad\qquad \otimes(bop, V(v_1), V(v_2), x) \end{array} \qquad \text{[binop]}$$

$\otimes(bop, x_1, x_2, x)$ holds iff x is the result of binary operation bop on x_1 and x_2

$$\begin{array}{ll} & \langle V, lb, v_r\rangle : S_{\text{tail}} \rightsquigarrow_P \langle V, lb_2, v_r\rangle : S_{\text{tail}} \\ \text{where} & instrAt_P(lb) = \text{“}\texttt{if(}v\texttt{ == 0) goto } a_t\text{”} \\ & lb = \langle p, a\rangle, \quad lb_2 = \begin{cases} \langle p, a_t\rangle & \text{if } V(v) = 0 \\ next(lb) & \text{if } V(v) \neq 0 \end{cases} \end{array} \qquad \text{[jz]}$$

$$\begin{array}{ll} & \langle V, lb, v_r\rangle : S_{\text{tail}} \rightsquigarrow_P \langle V_{\text{callee}}, \langle p, 0\rangle, v\rangle : \langle V, next(lb), v_r\rangle : S_{\text{tail}} \\ \text{where} & instrAt_P(lb) = \text{“}v \texttt{ := call } p \texttt{ (}v_0\texttt{,...,}v_{k-1}\texttt{)}\text{”} \\ & \text{The } p^{\text{th}} \text{ procedure of } P \text{ has parameters } v'_0, v'_1, \ldots, v'_{k-1} \\ & V_{\text{callee}} = (\lambda v.0)\left[v'_0 \mapsto V(v_0), v'_1 \mapsto V(v_1), \ldots, v'_{k-1} \mapsto V(v_{k-1})\right] \end{array} \qquad \text{[call]}$$

$$\begin{array}{c} \langle V, lb, v_r\rangle : \langle V_2, lb_2, v_{r2}\rangle : S_{\text{tail}} \rightsquigarrow_P \langle V_2[v_r \mapsto V(v)], lb_2, v_{r2}\rangle : S_{\text{tail}} \\ \text{where } \ instrAt_P(lb) = \text{“}\texttt{return } v\text{”} \end{array} \qquad \text{[ret]}$$