# Deductive runtime certification

Konstantine Arkoudas [1] Martin Rinard [2]

*Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, USA*

**Abstract**

This paper introduces a notion of certified computation whereby an algorithm not only produces a result $r$ for a given input $x$, but also proves that $r$ is a correct result for $x$. This can greatly enhance the credibility of the result: if we trust the axioms and inference rules that are used in the proof, then we can be assured that $r$ is correct. Typically, the reasoning used in a certified computation is much simpler than the computation itself. We present and analyze two examples of certifying algorithms.

We have developed denotational proof languages (DPLs) as a uniform platform for certified computation. DPLs integrate computation and deduction seamlessly, offer strong soundness guarantees, and provide versatile mechanisms for constructing proofs and proof-search methods. We have used DPLs to implement numerous well-known algorithms as certifiers, ranging from sorting algorithms to compiler optimizations, the Hindley-Milner $\mathcal{W}$ algorithm, Prolog engines, and more.

*Key words:* Verification, certifying algorithms, program correctness, proofs, certificates, Athena, DPLs

## 1 Introduction

Complete deductive verification of software systems can be extremely onerous. It is a major challenge to prove mechanically that a complex piece of software will always produce the correct output for any given input. The difficulty is due partly to the fact that deductive technology has not yet reached a sufficiently advanced state of the art, and partly to the inherently high complexity of software. Nevertheless, formal proofs are a superb methodology for increasing reliability, and we would like to find a use for them even when it is not practical to prove a system completely correct.

This paper presents an alternative to complete static verification, namely partial dynamic certification. Instead of statically proving that an algorithm

[1] Email: `arkoudas@lcs.mit.edu`
[2] Email: `rinard@lcs.mit.edu`

produces correct results for all inputs, we express the algorithm as a proof-search procedure that takes an input $x$ and not only produces a result $r$ but also *proves* that $r$ is a correct result for $x$. Such *certifying algorithms* can be viewed as instrumented versions of conventional algorithms, modified to *justify* their results with deductive reasoning. The reasoning is performed at runtime, and applies only to the particular input $x$ and result $r$. The completed proof can be regarded as a *certificate* for the result $r$; once this proof is validated, we may say that $r$ has been "certified."

Runtime certification is less powerful than static verification. It guarantees no more and no less than this: if and when the algorithm produces a result, that result is correct—modulo the logic that specifies what counts as correct. That can still be very useful, since it prevents the program from silently generating a plausible but incorrect result.

The advantage of runtime certification is that usually it is much more practical than static verification. Consider, for instance, the unification example we give in Section 4. A complete verification of a unification algorithm was given by Paulson [28], where he states that the proof "relies on a substantial theory of substitutions, consisting of twenty-three propositions and corollaries... The project has grown too large to describe in a single paper... The proof is not entirely beautiful. A surprisingly diverse series of problems appeared." A more recent correctness proof for a Martelli-Montanari-style unification algorithm using the Boyer-Moore theorem prover runs to thousands of lines [31]. By contrast, expressed as a certifying algorithm, our Martelli-Montanari unification procedure was implemented in less than one page of Athena code. [3] This dramatic difference is an apt illustration of the main tradeoff: static verification gives us peace of mind for all inputs, but is difficult; runtime certification gives us more limited assurance, pertaining only to particular inputs and outputs, but is much more feasible. Another important difference is that static proofs—such as the aforementioned by Paulson—usually verify an abstract model of the software component, not the actual code; whereas in certified computation the theorem refers to the actual result obtained in real time.

We envision runtime certification as a methodology to be applied to selected parts of a software system, not to every part. In some cases it might not be viable to characterize output correctness with mathematical rigor. For other components, such as reactive systems, the important issue is not output correctness but behavioral safety, and in that case other methods and formalisms such as runtime monitoring [8] or I/O automata [18] will be appropriate. Even when correctness has a precise description and is important, certification might not be deemed necessary. For instance, compiler writers are not likely to certify lexical analyzers or parsers. Those parts are so well understood and so routinely automated by highly reliable tools that the effort necessary to certify them would outweigh the benefit. However, we might

[3] Athena is the DPL we have used for all our implementations; we describe it in Section 3.

still want to certify the optimizing part of a compiler's back end, or a type inference algorithm, or some other component whose correctness is crucial.

## 2 An example

Consider Euclid's algorithm for computing the greatest common divisor of two natural numbers $a$ and $b$, denoted $gcd(a, b)$. The algorithm can be stated in pseudocode as follows:

$euclid(a, b) =$ If $b$ is 0, return $a$. Otherwise, return the result of the recursive call $euclid(b, a \bmod b)$.

(We write $a \bmod b$ for the remainder obtained when $a$ is divided by $b$.)

The correctness of this algorithm hinges on the following two results of number theory:

(1) $\forall\, x \mid gcd(x, 0) = x$

(2) $\forall\, x, y \mid y > 0 \Rightarrow gcd(x, y) = gcd(y, x \bmod y)$

As given above, the algorithm relies on these two theorems *tacitly*. In order to prove that the algorithm is correct, the connection must be made explicit. For purposes of static verification, one would use strong induction on $b$ to show that for *any* given $a$ and $b$, $euclid(a, b)$ produces $gcd(a, b)$.

In our approach, we express Euclid's method as an algorithm *derive-gcd* that not only produces a result $r$ for two inputs $a$ and $b$ but also *proves* that $r$ is the *gcd* of $a$ and $b$. Accordingly, the output of *derive-gcd* is a *theorem* of the form $gcd(a, b) = r$. The theorem is to be derived from axioms and previously established results—in this case, equations (1) and (2)—using standard inference methods, e.g., universal specialization, modus ponens, reasoning by cases, etc. Specifically, we formulate *derive-gcd* as follows:

*derive-gcd*$(a, b) =$ If $b$ is 0 then infer $gcd(a, b) = a$ from (1). Otherwise, let $m = a \bmod b$ (i). Recursively apply *derive-gcd* to $b$ and $m$ to obtain a theorem of the form $gcd(b, m) = r$ (ii). From (2), (i), and the supposition $b > 0$, we conclude $gcd(a, b) = gcd(b, m)$ (iii). Finally, from (iii), (ii), and the transitivity of equality, we obtain $gcd(a, b) = r$.

We can view *derive-gcd* as a method for arguing that a certain result is correct—as a "justification algorithm." It is a recipe that spells out how to obtain the gcd of two integers, and also how to convince a skeptical observer that the result is indeed correct.[4]

The important point is the reduction in our trusted base. What do we need to trust in order to believe a theorem produced by *derive-gcd*? We need to trust the premises used during the justification process, namely, proposi-

[4] It is noteworthy that Euclid himself formulated his procedure as a certifying algorithm. We quote from the beginning of Proposition II, Book 7 of the Elements: "It is required to find the greatest common measure of AB and CD. If now CD measures AB, since it also measures itself, then CD is a common measure of CD and AB. And it is manifest that it is also the greatest, for no greater number than CD measures CD. But, if CD does not measure AB, then..." The presentation continues in the same style.

tions (1), (2), and the transitivity of equality; and we also need to trust that the primitive arithmetic operations at our disposal for computing remainders and for performing numerical comparisons are correctly implemented. Presumably (1) and (2) were previously derived from more rudimentary axioms and definitions; transitivity of equality can be taken as an axiom; and the correctness of elementary arithmetic operations can reasonably be taken in good faith. But the *control structure* of *derive-gcd* does not need to be trusted in order for the resulting theorems to be credible; and that is the crucial difference between a certifying algorithm such as *derive-gcd* and a conventional algorithm such as *euclid*. For instance, if the conditional analysis that *derive-gcd* performs or the recursive call that it places to itself are misguided, in the sense that they fail to produce the proper outcome, then the proof will not go through and no theorem will be established. But if a theorem is derived then it can be trusted, even if its derivation happened to be a fluke. Certifying algorithms thus enforce a strict separation between logic and control, pushing the latter outside the trusted base.

This becomes more evident if the language in which we implement the certifying algorithm is capable of issuing a *certificate* for every output theorem, i.e., a formal proof for the correctness of the result. Certificates can be obtained automatically in a language such as Athena (see below). In Athena the user can apply *derive-gcd* to two integers, say 12 and 8, and obtain two things, the theorem $gcd(12, 8) = 4$ and a certificate that proves this theorem. In this case, the certificate will be a straight-line proof such as the following:

| | |
|---|---|
| a. $gcd(4, 0) = 4$ | from (1) |
| b. $4 > 0$ | primitive |
| c. $8 \bmod 4 = 0$ | primitive |
| d. $gcd(8, 4) = gcd(4, 0)$ | from (2), b, and c |
| e. $gcd(8, 4) = 4$ | from d, a, and equality transitivity |
| f. $8 > 0$ | primitive |
| g. $12 \bmod 8 = 4$ | primitive |
| h. $gcd(12, 8) = gcd(8, 4)$ | from (2), g, and f |
| j. $gcd(12, 8) = 4$ | from h, e, and equality transitivity |

Observe that this certificate is essentially the evaluation trace of the application of *derive-gcd* to 12 and 8. Also note that the asymptotic complexity of *derive-gcd* and *euclid* are the same. Certificates for theorems of the form $gcd(a, b) = r$ will have $O(\log_2 \min\{a, b\})$ steps.

## 3 Denotational proof languages

Certified computation can be viewed as a language-independent paradigm—as a general style of backing up computation with proof. Nevertheless, in making that approach workable in practice we are confronted with a language problem: we need languages in which justification algorithms such as *derive-gcd* can be fluidly expressed. To be suitable for certified computation, a programming language must offer several features over and above the standard amenities of modern languages. The language must have a built-in rigorous notion of statement and proof, and furthermore:

- There must be a trusted deduction mechanism by which one can derive consequences of axioms and definitions.
- There must be a sound abstraction mechanism for proofs in order to automate tedious steps. Arbitrarily complicated proof-search algorithms (e.g., semantic tableaux or Knuth-Bendix completion) must be expressible in a trusted manner.
- There must be built-in support for natural deduction as practiced in common mathematical reasoning. Assumption and eigenvariable discharge, for example, should be handled automatically.
- If requested, theorem-producing computations must be able to output certificates, which can then be independently checked.

One of our main contributions to certified programming has been the design of DPLs (Denotational Proof Languages) [1]. In particular, we have implemented Athena, a DPL for polymorphic multi-sorted first-order logic that satisfies all of the above criteria. There are two main innovations behind Athena, one syntactic and the other semantic. On the syntax front, Athena has formalized key notions such as assumption scope and eigenvariable scope in novel ways (most notably, without reducing them to variable scope in the typed $\lambda$-calculus as is done in Curry-Howard systems [14]), which has enabled the introduction of syntax forms such as **assume**, **pick-any**, and **pick-witness**, that closely capture the most common and useful idioms of mathematical reasoning. Semantically, the main contribution of DPLs is the concept of assumption bases. The introduction of assumption bases as a fundamental semantic abstraction on a par with lexical environments, stores, and continuations allows for an elegant treatment of proof theory, even for logics that have been difficult to formalize in a non-graphical manner, such as Fitch-style natural deduction, and for a smooth integration of proof checking with computation.

As a programming language, Athena is a higher-order strict functional language in the tradition of Scheme and ML. Such a language affords distinct advantages for certified computation, e.g., higher-order proof continuations can be freely passed around, and this often comes handy. Nevertheless, this is not essential. Other programming languages, e.g., an object-oriented language such as Java, could just as well be meshed with the abstract syntax and semantics of DPL proofs in a conservative manner (i.e., so that a Java program that does not contain any DPL proofs looks and behaves exactly as prescribed by the Java specification).

In terms of deduction, Athena comes with a small but logically complete collection of very simple inference rules, namely introduction and elimination rules for the logical connectives and quantifiers, and equivalence and congruence rules for equality. This results in a minimal trusted base. More complicated rules can be implemented as methods, whose soundness is ensured by the formal semantics of the language. An automatic theorem prover (e.g., based on resolution) can thus be soundly implemented. Alternatively, a powerful off-the-shelf ATP such as Vampire [32] can be employed as an oracle, and then the proof output by Vampire can be converted into a native deduction using the system's primitive inference rules. Such outsourcing is preferable to "rolling one's own" prover because it leverages the great progress that has been made in the ATP community over the last decade. Either way, having such a prover is important in order to be able to skip tedious steps when writing down proofs or proof algorithms. Athena currently uses a combination of outsourcing (via Vampire) and natively written methods for rewriting and other types of proof automation.

Finally, an important issue for logical frameworks is the size of the overall "trusted base." On one extreme, we could provide a minimal initial collection of axioms and primitive inference rules in a foundational theory that is in principle adequate for all mathematics, such as ZF, and offer only two ways of extending the system: by conservative definitions and by deduction using the provided inference rules. In this way everything is tied back to the initial axioms and inference rules, which are presumably obviously sound, and thus soundness is safeguarded. On the other extreme, the users are allowed to introduce arbitrary axioms, inference rules, and decision procedures, without a small initial base to serve as an anchor. The first approach is superior in principle, as a minimal trusted base maximizes the assurance that deduction provides, but it is unnecessarily stringent in practice.

Athena takes a middle ground: a minimal trusted base is provided and users have the option of tying everything back to that base by extending it only via conservative definitions and deduction; but they are not required to do so. They also have the option of introducing arbitrary axioms, rules, and decision procedures. The choice of where to anchor the proofs is left up to the users, to be decided on an individual basis by the context of each application. For instance, in the unification example of the next section we introduce five inference rules as primitives. They are not the simplest possible rules: in the worst case, applying four of them takes linear time in the size of their inputs, while the fifth takes quadratic time. The rules could be further reduced, expressed as trusted methods in terms of simpler rules and axioms, albeit at the expense of additional work. But even if we leave them as they are, we will already have accomplished a remarkable trust reduction: instead of having to trust the conventional Martelli-Montanari algorithm, which has a complex control structure and exponential complexity, we need only trust five very short and simple inference rules of quadratic complexity at worst.

## 4 Unification via a certifying algorithm

As a more involved example, in this section we will use certifying computation to implement first-order unification. A conventional unification algorithm takes as input two terms $s$ and $t$ and produces an idempotent most general unifier ("mgu") $\theta$ for them, if one exists; otherwise the algorithm fails. By contrast, our unification prover will take $s$ and $t$ as inputs and will not simply produce $\theta$, but will in fact prove that $\theta$ is an idempotent mgu for $s$ and $t$. In other words, the output of our prover will be a theorem of the form $imgu(s, t, \theta)$, asserting that $\theta$ is an idempotent mgu of $s$ and $t$.

The first step is to formulate a sound logic that allows us to derive judgments of this form. Then, based on this logic, we can start to implement a unification procedure as a certifying algorithm. In what follows, by "term" we will mean a first-order Herbrand term over some function symbols and variables; by "equation" we will mean an ordered pair of terms $\langle s, t\rangle$, which will be more suggestively written as $s \approx t$; and by "substitution" we will mean a function from variables to terms that is the identity almost everywhere [2]. We use the letters $x, y$, and $z$ as typical variables; $f$, $g$, and $h$ as function symbols; $s$ and $t$ for terms; and $\theta, \sigma$, and $\tau$ for substitutions. We write $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ for the substitution that maps each $x_i$ to $t_i$ (we assume that $x_1, \ldots, x_n$ are distinct) and every other variable to itself; and we write $\overline{\theta}$ for the unique homomorphic extension of a substitution $\theta$ to the corresponding term algebra [34,2].

The Martelli-Montanari (MM for short) unification algorithm [19] deals with finite *systems* of equations rather than with single equations. By a system of equations we will mean a list of the form

$$E = [s_1 \approx t_1, \ldots, s_n \approx t_n]. \tag{3}$$

We write $E_1, E_2$ for the list obtained by concatenating $E_1$ and $E_2$. For convenience, we sometimes treat a single equation as an one-element list, e.g., writing $E, s \approx t$ instead of $E, [s \approx t]$. Finally, for any $\theta$ and system $E$ of the form (3), we write $\overline{\theta}(E)$ for the system $[\overline{\theta}(s_1) \approx \overline{\theta}(t_1), \ldots, \overline{\theta}(s_n) \approx \overline{\theta}(t_n)]$.

A system of the form (3) is unifiable iff there exists a substitution $\theta$ such that $\overline{\theta}(s_i) = \overline{\theta}(t_i)$ for $i = 1, \ldots, n$. We call $\theta$ a *unifier* of $E$. If $\theta$ is more general than every $\sigma$ that unifies $E$ then we say that $\theta$ is a *most general unifier* of $E$. Most general unifiers are unique up to composition with a renaming, and in that sense we may speak of *the* mgu of some $E$. We write $U(E)$ for the set of all unifiers of $E$, where we might of course have $U(E) = \emptyset$ if $E$ is not unifiable. Thus the traditional unification problem of determining whether two terms $s$ and $t$ can be unified is reducible to the problem of deciding whether the system $[s \approx t]$ is unifiable.

A system $E$ is said to be *in solved form* iff the set of equations that occur in it is of the form $\{x_1 \approx t_1, \ldots, x_k \approx t_k\}$ where the variables $x_1, \ldots, x_k$ are distinct and $x_i$ does not occur in $t_j$ for any $i, j \in \{1, \ldots, k\}$. It is straightforward to show that a system $E$ of this form determines a unique substitution

$\theta_E = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ that is an idempotent most general unifier of $E$.

The MM algorithm attempts to transform a given system into solved form by repeated applications of the following rules:

- *Simplification*: $E_1, t \approx t, E_2 \Rightarrow E_1, E_2$;
- *Decomposition*: $E_1, f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n), E_2 \Rightarrow E_1, s_1 \approx t_1, \ldots, s_n \approx t_n, E_2$;
- *Transposition*: $E_1, t \approx x, E_2 \Rightarrow E_1, x \approx t, E_2$, provided that $t$ is not a variable;
- *Application*: $E_1, x \approx t, E_2 \Rightarrow \overline{\{x \mapsto t\}}(E_1), x \approx t, \overline{\{x \mapsto t\}}(E_2)$, provided that $x$ occurs in $E_1, E_2$ but not in $t$.

For any two systems $E$ and $E'$, we write $E \Rightarrow E'$ to signify that $E'$ can be obtained from $E$ by one of these rules.
The qualification in the transposition rule is needed to guarantee the termination of the transformation process. The same goes for the qualification that $x$ must occur in $E_1, E_2$ in the last rule (the second qualification of that rule ensures that the process does not proceed in the presence of an equation $x \approx t$ where $x$ occurs in $t$, since such an equation is not unifiable).

The idea behind using these transformations as an algorithm for unifying two terms $s$ and $t$ is this: we start with the system $E_1 = [s \approx t]$ and keep applying rules (non-deterministically), building up a sequence $E_1 \Rightarrow E_2 \Rightarrow \cdots \Rightarrow E_k$, until we finally arrive at a system of equations $E_k$ to which no more rules can be applied. It is not difficult to prove termination (i.e., that it is impossible to continue applying rules ad infinitum), and that if $s$ and $t$ are indeed unifiable then the final system $E_k$ will be in solved form, i.e., its equations will comprise a set of the form $E_k = \{x_1 \approx t_1, \ldots, x_n \approx t_n\}$ where the variables $x_1, \ldots, x_n$ are distinct and $x_i$ does not occur in any $t_j$. Accordingly, the substitution $\theta_{E_k} = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is an idempotent mgu of $E_k$. Further, we can show that if $E_i \Rightarrow E_{i+1}$ then $U(E_i) = U(E_{i+1})$, so that any substitution that unifies $E_i$ also unifies $E_{i+1}$ and vice versa. Thus it follows that $\theta_{E_k}$ is also an idempotent mgu of $E_{k-1}, E_{k-2}, \ldots, E_1$, and hence an idempotent mgu of $s$ and $t$. On the other hand, if the final set of equations $E_k$ is *not* in solved form then we may conclude that the initial terms $s$ and $t$ are not unifiable.

We will now set up a calculus for proving that a system of equations is unifiable. We can use such a calculus to show that two given terms $s$ and $t$ can be unified by adducing a proof to the effect that the system $[s \approx t]$ is unifiable. Such a proof would start from axioms asserting that certain systems are evidently unifiable and proceed by applying inference rules of the form "If $E_1, \ldots, E_n$ are unifiable then so is $E$." The MM transformation rules are not appropriate for that purpose because they proceed in the reverse direction: they start from the equations whose unifiability we wish to establish and work their way back to systems whose unifiability is apparent. In that sense, they are *analytic* or "backward" rules: they keep breaking up the original equations into progressively simpler components. By contrast, we want *synthetic* rules that will allow us to move in a forward manner: starting from simple elements, we must be able to build up the desired equations in a finite number of steps.

$$\frac{}{\vdash_U [x_1 \approx t_1, \ldots, x_n \approx t_n] : \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}} \quad [\mathit{Solved\text{-}Form}]$$

provided $[x_1 \approx t_1, \ldots, x_n \approx t_n]$ is in solved form

$$\frac{\vdash_U E_1, E_2 : \theta}{\vdash_U E_1, t \approx t, E_2 : \theta} \quad [\mathit{Reflexivity}]$$

$$\frac{\vdash_U E_1, s \approx t, E_2 : \theta}{\vdash_U E_1, t \approx s, E_2 : \theta} \quad [\mathit{Symmetry}]$$

$$\frac{\vdash_U E_1, s_1 \approx t_1, \ldots, s_n \approx t_n, E_2 : \theta}{\vdash_U E_1, f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n), E_2 : \theta} \quad [\mathit{Congruence}]$$

$$\frac{\vdash_U E_1, x \approx t, E_2 : \theta}{\vdash_U E_1', x \approx t, E_2' : \theta} \quad [\mathit{Abstraction}]$$

provided $\overline{\{x \mapsto t\}}(E_1', E_2') = E_1, E_2$.

Fig. 1. A logic for deducing idempotent most general unifiers.

In fact we will see that the MM algorithm is, in a precise sense, a backward proof-search algorithm for the deduction system we formulate here.

The judgments of our logic are of the form $\vdash_U E : \theta$, asserting that the substitution $\theta$ is an idempotent most general unifier of $E$. The logic itself comprises one axiom and four unary rules, shown in Figure 1. The axiom [*Solved-Form*] asserts that every system of equations $E = [x_1 \approx t_1, \ldots, x_n \approx t_n]$ in solved form is unifiable, and that $\theta_E = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is an idempotent mgu of $E$. The rules [*Reflexivity*], [*Symmetry*], and [*Congruence*] are self-explanatory, and their soundness should be clear (it is straightforward to prove that all five rules are sound, and indeed complete [1]). Observe that if we read the rules in a forward manner then, in relation to the MM transformations, reflexivity can be viewed as the inverse of simplification, symmetry as the inverse of transposition, and congruence as the inverse of decomposition. We will also see that [*Abstraction*] is the inverse of application. Also notice that these are pure inference rules, in the sense that no control information is embedded in them. Restrictions such as found in the transposition rule of the MM system will instead be relegated to the control structure of a certifying algorithm that automates this logic, keeping the logic itself cleaner.

Finally, consider the abstraction rule. The proviso $\overline{\{x \mapsto t\}}(E_1', E_2') = E_1, E_2$ is the key here. It means that the equations in $E_1', E_2'$ are abstractions of the equations in $E_1, E_2$, obtainable from the latter by replacing certain occurrences of $t$ by $x$. Alternatively, the equations in $E_1, E_2$ are *instances* of the equations in $E_1', E_2'$, obtained from the latter by applying the substitution $\{x \mapsto t\}$. Accordingly, the equations in $E_1', E_2'$ are *more general* than those of $E_1, E_2$.

Let us illustrate with an example. We will show that the substitution

$\theta = \{x \mapsto a, y \mapsto g(h(a)), z \mapsto h(a)\}$ is an idempotent mgu of $f(x, g(z), b, z)$ and $f(a, y, b, h(x))$. The following deduction proves this:

| | | |
|---|---|---|
| 1. | $[x \approx a, y \approx g(h(a)), z \approx h(a)] : \theta$ | [*Solved-Form*] |
| 2. | $[x \approx a, y \approx g(h(a)), b \approx b, z \approx h(a)] : \theta$ | 1, [*Reflexivity*] |
| 3. | $[x \approx a, g(h(a)) \approx y, b \approx b, z \approx h(a)] : \theta$ | 2, [*Symmetry*] |
| 4. | $[x \approx a, g(z) \approx y, b \approx b, z \approx h(a)] : \theta$ | 3, [*Abstraction*] on $z \approx h(a)$ |
| 5. | $[x \approx a, g(z) \approx y, b \approx b, z \approx h(x)] : \theta$ | 4, [*Abstraction*] on $x \approx a$ |
| 6. | $[f(x, g(z), b, z) \approx f(a, y, b, h(x))] : \theta$ | 5, [*Congruence*] |

We note that the only rule that creates—or in any way affects—the substitution $\theta$ of a judgment $\vdash_U E : \theta$ is the axiom [*Solved-Form*]. All of the other rules simply pass along the substitution of the premise unchanged. Thus a substitution is created only once, for a system in solved form, and from that point on it is carried along from system to system via the various rules, until it is finally attached to the original input.

We can now implement a certifying algorithm *unify* that takes an input system $E$ and uses this logic to derive a theorem of the form $\vdash_U E : \theta$, provided that $E$ is unifiable, or fails otherwise. In what follows we will sketch the outlines of such an algorithm at a high level. [5]

Most of the work in *unify* is done by an auxiliary procedure *find-candidate*, which takes a system $E$ and splits it into three parts, a prefix $E_1$, an equation $s \approx t$, and a suffix $E_2$, in such a way that $E_1, s \approx t, E_2$ matches the left-hand side of one of the four MM rules. These three values are returned in a three-element list. If no such decomposition exists, then *find-candidate* returns the empty list. First, *unify* passes its input $E$ to *find-candidate*. If the latter is unable to split $E$ as discussed above and returns the empty list, then *unify* checks to see whether $E$ is in solved form. If so, rule [*Solved-Form*] is used to prove the corresponding theorem; otherwise *unify* fails. On the other hand, if some decomposition $E_1, s \approx t, E_2$ is found using some MM rule $T$, then *unify* places a recursive call to itself with a new input list, appropriately constructed from $E_1, s, t$, and $E_2$, and then uses that result and the corresponding rule of our calculus (the inverse of $T$) to derive a theorem about $E$.

Figure 2 depicts the run-time flow of control when *unify* is applied to the system $[f(x, g(z), b, z) \approx f(a, y, b, h(x))]$. In essence, every application of a primitive rule justifies the work of the corresponding recursive call. By the time the entire proof has been completed, we have validated the original problem decomposition and proof search. Therefore, we do not need to trust *find-candidate* or *unify*, which are by far the most complicated parts of the system. If we are confident in the five primitive inference rules, the result is credible.

[5] The interested reader can find the actual Athena code for this and other examples (along with explanatory comments) at `www.cag.lcs.mit.edu/~kostas/dpls/cc-examples`.

$$
\begin{tikzcd}
\mathit{unify}\ [f(x,g(z),b,z) \approx f(a,y,b,h(x))] \arrow[d, "\text{(Decompose)}"] & & \{f(x,g(z),b,z) \approx f(a,y,b,h(x))\} : \theta \\
\mathit{unify}\ [x \approx a, g(z) \approx y, b \approx b, z \approx h(x)] \arrow[d, "\text{(Apply)}"] \arrow[r, phantom, "\xleftrightarrow{\text{Inverses}}"] & & \{x \approx a, g(z) \approx y, b \approx b, z \approx h(x)\} : \theta \arrow[u, "\mathit{Congruence}"'] \\
\mathit{unify}\ [x \approx a, g(z) \approx y, b \approx b, z \approx h(a)] \arrow[d, "\text{(Apply)}"] & & \{x \approx a, g(z) \approx y, b \approx b, z \approx h(a)\} : \theta \arrow[u, "\mathit{Abstraction}"'] \\
\mathit{unify}\ [x \approx a, g(h(a)) \approx y, b \approx b, z \approx h(a)] \arrow[d, "\text{(Transpose)}"] & & \{x \approx a, g(h(a)) \approx y, b \approx b, z \approx h(a)\} : \theta \arrow[u, "\mathit{Abstraction}"'] \\
\mathit{unify}\ [x \approx a, y \approx g(h(a)), b \approx b, z \approx h(a)] \arrow[d, "\text{(Simplify)}"] & & \{x \approx a, y \approx g(h(a)), b \approx b, z \approx h(a)\} : \theta \arrow[u, "\mathit{Symmetry}"'] \\
\mathit{unify}\ [x \approx a, y \approx g(h(a)), z \approx h(a)] \arrow[rr, "\mathit{Solved\text{-}Form}"'] & & \{x \approx a, y \approx g(h(a)), z \approx h(a)\} : \theta \arrow[u, "\mathit{Reflexivity}"']
\end{tikzcd}
$$

Fig. 2. Control flow, shown counter-clockwise, of *unify* applied to the running example; $\theta$ here is $\{x \mapsto a, y \mapsto g(h(a)), z \mapsto h(a)\}$.

## 5 Related work

Certified computation can be viewed as a generalization of work by Rinard and Marinov at MIT [30] and other researchers elsewhere [29] on logical validation of compiler transformations. There, an optimization procedure—say, for constant propagation—does not only produce a transformed control flow graph but also proves a bisimulation theorem relating it to the original. Here we extend that idea to arbitrary computations.

The idea of using deduction for computational purposes is not new. It is the cornerstone of the school of relational logic programming [17], which dates back at least to the inception of Prolog in the early 1970s. Computation in that setting is described by the well-known slogan of Kowalski: *Computation* = *Logic* + *Control*. The logic part consists of our theory, the axioms and the inference rules—the "what" of the problem. The control part amounts to a theorem-proving strategy—the "how" part.

In logic programming, however, users do not write proofs or proof methods; they only write axioms (expressed as Horn clauses). The control engine, SLD resolution, is baked into the underlying framework and cannot be extended. Users cannot formulate arbitrary proof strategies, custom-tailored for specific problem domains. Certified computation with DPLs is quite different in that users are given unrestricted freedom in structuring the control part. Apart from expressiveness, generality, and extensibility, the complete separation of logic from control is also a modularity boon, as many different control engines can be used with—plugged into—a single logic. This facilitates the interaction of a code consumer with arbitrary code producers.

Another methodology for attaining reliable software is static program verification, and we have already discussed the main tradeoff between it and our approach, namely, generality vs. feasibility. Another advantage of program verification is that a static proof has a fixed cost. Once correctness has been established, the algorithm can be confidently executed arbitrarily many times without additional effort. By contrast, our model has a runtime price: the algorithm has to do extra work to justify itself every time it generates a result. Nevertheless, in our experience runtime certification has never strictly increased the asymptotic complexity of an algorithm.

Software model checking [22,35] is not so much concerned with establishing output correctness for specific inputs as it is with uncovering errors over as large classes of inputs as possible, with an emphasis on concurrency problems such as deadlocks, critical section violations, etc.

In software testing [5] an extensive sample of inputs are presented to the program and the outputs are checked for correctness. While testing remains invaluable in practice, it has serious drawbacks. First, generation of structurally complex input data is difficult to automate [15], and thus test suites end up being quite limited. Second, how are the outputs to be checked for correctness? Inspection by eye is clearly impractical for large-scale experiments. Hence, software must be written to check whether an output is correct for a given input. But such "checkers" are often complicated and difficult to implement, and so their credibility can be just as weak as the credibility of the algorithm we are trying to test. We will elaborate that point shortly. Finally, testing is an empirical, finite process; it cannot test all inputs. All too often, after a program has shipped, it starts generating erroneous results for certain combinations of inputs that simply fell through the cracks during testing.

To overcome the last of the above problems, Wasserman and Blum [33] suggest that "checkers" be permanently attached to programs and deployed at runtime, after a result $r$ has been obtained, to check whether $r$ is correct. As the authors concede, this is only viable for problems for which a result can be checked in asymptotically less time than the time it takes to generate it. That is, it only makes sense to couple an algorithm $A$ with a checker $C$ if the complexity of $C$ is strictly less than that of $A$. They call those "simple checkers."

Unfortunately, simple checkers often don't exist. In many cases, the most efficient way for a checker $C$ to independently determine whether a result $r$ is correct for an input $x$ is to recompute its own result $r'$ and then compare $r$ and $r'$ for equality. The tester will thus be of the same complexity as the algorithm itself, and we will have no reason to trust it more than the original algorithm. Testing in such cases amounts to "redundancy coding"; it is inefficient and of little value in boosting the credibility of our results. Consider sorting, for instance. To check that an output list $L'$ is correct for an input list $L$, a checker first needs to verify that $L'$ is sorted, which is relatively easy—it can be done in linear time in the length of $L'$. But it also needs to verify that $L'$ is a permutation of $L$. The most straightforward algorithm for that takes quadratic time, i.e., it is more expensive than the sorting itself, which presumably took $n \log n$ time. To do the permutation check more with a better worst-case performance, the checker would be reduced to sorting $L$, getting a result $L''$, and then checking $L''$ and $L'$ for equality—thereby achieving zero trust reduction.

This appears to be the case for most polynomial-time algorithms: checkers for them are either as expensive as the algorithms themselves or only marginally better. Algorithms for intractable (NP-complete) problems do have dramatically simpler checkers capable of verifying *positive* answers. That follows directly from the definition of the class NP. Consider, for instance, an algorithm $H$ for determining whether a graph is Hamiltonian. We can efficiently check "yes" answers from $H$, provided of course that the latter produces actual Hamiltonian cycles as evidence. However, we cannot efficiently check *negative* answers from $H$. Indeed, under the assumption that NP $\neq$ co-NP (which is a widely held belief [25]), it is easily proved that no NP-complete problem can be in co-NP, which means that there are no efficient checkers capable of verifying negative answers to NP-complete problems. Therefore, truly simple and complete checkers seem to exist only for problems in NP $\cap$ co-NP that are also believed to lie outside P, such as integer factorization.

Blum's important contribution was showing that in some cases one may employ probabilistic techniques [6] to perform certain checks more efficiently—but we then give up the peace of mind that a complete guarantee would give. For instance, in the sorting example above we can check whether $L' = [y_1, \ldots, y_m]$ is a permutation of $L = [x_1, \ldots, x_n]$ by using a deterministic hash function $h$ to compute the sums $s_1 = h(x_1) + \ldots + h(x_n)$ and $s_2 = h(y_1) + \cdots + h(y_m)$. This can be done in linear time, and if $L'$ is indeed a permutation of $L$, then $s_1 = s_2$; otherwise the two sums will *probably* differ. The probability of error can be made arbitrarily small, but more and more checks are required at runtime for convergence, further penalizing runtime performance. It is also unclear how widely applicable such techniques are; they seem to be better suited for numerical problems.

So contrary to the folk wisdom in Computer Science which holds that checking a result is easier than generating one, oftentimes checking a result is just as difficult as producing one. In fact in some cases it is even more difficult, or even altogether impossible. A prime example arises in compilation: although it is possible—and relatively easy—to mechanically *find* a machine-language executable that simulates a given program in the source language, it is impossible to mechanically *check* whether a given machine-language program respects the semantics of a given source. No such checker exists, as the problem is undecidable. As another example, consider the execution of a

Prolog query. How can we check a "yes/no" query answer? It is impossible to build a complete, correct checker, because the problem is undecidable: a bogus "yes" answer might send our checker into an infinite loop. But both of these—and many other—problems are readily amenable to our approach. Athena has been used to implement credible compilation as well as a certifying Prolog system that backs up its answers with extremely simple natural deduction reasoning.

The "runtime assertions" of Meyer's contract-programming paradigm [20] are another related approach. Executable assertions are useful for dynamically performing sanity checks and for ensuring that certain simple pre- and post-conditions hold, but are generally too weak to guarantee output correctness. Simple assertions (containing only built-in primitives such as arithmetic operations and comparisons, boolean combinations thereof, and bounded quantification) are trustworthy but cannot express output correctness for many useful algorithms. For instance, such assertions cannot ascertain whether a Gauss-Jordan elimination algorithm has correctly determined that a given system of linear equations has no solutions; that a symbolic integration procedure has correctly integrated a given function; that a string has correctly matched a regular expression pattern; and so on. Even when such assertions can capture output correctness, they often do so in a naive declarative style and thus their runtime execution becomes impractical. For instance, there are many problems—such as shortest-path graph problems—that can be procedurally solved with efficiency, e.g., using dynamic programming techniques, but whose declarative output specification would take exponential time to check. Of course it is possible to allow specification assertions to contain arbitrary executable code (e.g., assertions in Jass [4] can contain arbitrary side-effect-free Java code). In that case one can perform complete correctness checks for arbitrary computations, but the trust issue for the checking code resurfaces intact.

We view techniques such as monitor-oriented programming ("MoP" [8]) and runtime verification [16,13] as orthogonal to certified computation. For many problems, guaranteeing output correctness in such frameworks is difficult or impossible due to the same tension between specification simplicity and trust that we discussed in the preceding paragraph. For instance, the pre- and post-conditions of ASML [3] and JML [7] are simple and easy to trust, but use bounded non-deterministic choice and quantification and hence are either unable to express correctness for interesting problems or else are too inefficient to execute. We can remedy this by writing ASML and JML specifications as "model programs" that maintain variables, update state, perform loops, etc., but then the specifications become too operational.[6] Nevertheless, techniques such as MoP and runtime trace verification are very useful for observing and

[6] For example, a declarative correctness postcondition for a gcd algorithm would be grossly inefficient to execute. Alternatively, we could express Euclid's algorithm as an ASML program, but then we would have little reason to trust it more than an implementation.

steering a system's dynamic behavior using combinations of inline and offline monitoring in ways that certified computation cannot, most notably for the purpose of ensuring safety properties of system states, detecting and reacting to protocol violations, raising exceptions of the appropriate type, and so on. Accordingly, we envision using such techniques in conjunction with certified computation in order to ensure that software component implementations conform to different aspects of their specifications.

The proof-carrying code (PCC) of Lee and Necula [23] is primarily a compilation methodology concerned with producing programs that satisfy some security policy and are thus "safe" to execute. That requires general verification, as the program must be proven safe for all possible traces. How difficult this is depends on the particular notion of safety involved. Certified computation, by contrast, is a general paradigm for computation, concerned with ensuring correctness for particular input-output pairs rather than safety properties for all inputs. Of course, a "certifying compiler" that outputs proofs of memory or type safety [24,26] can be viewed as a certifying algorithm in our sense.

On the issue of deductive technology, we note that a certifying algorithm could in principle be implemented in any language, e.g. in C, as long as it eventually produces a formal proof (say, in LF [12] or Coq [9] or Athena form) which can then be independently checked. But that would obfuscate the use of deduction for documentation purposes, and would also be unduly cumbersome. Algorithm certification is inextricably linked with proof engineering, and is thus greatly facilitated by systems that make provisions for building and managing formal theories, performing proof search, and constructing and validating formal proofs (see Section 3). Thus LCF-style [10] systems such as HOL [11] or Isabelle [27] could certainly be used for certified computation. However, our certifying algorithms are not related to Milner's seminal "tactics" and "tacticals" [21]. Tactics are a mechanism for goal decomposition in backward proof search (specifically, a tactic is a function that takes a goal and returns a list of subgoals; a tactical is a combinator that composes tactics). We are more concerned with forward proof presentation, and in particular with enabling programmers to express proof methods such as *derive-gcd* in a perspicuous style. In that respect DPLs offer distinct advantages. Most notably, the introduction of assumption bases as a fundamental semantic abstraction obviates the need to use sequents (as is necessary in HOL-like systems), and this results in proof and proof methods that are much closer to the natural deduction format that mathematicians use in practice.

## 6 Conclusions

We have espoused a notion of certified computation where program results are derived deductively rather than merely generated by arbitrary processes. We have illustrated this approach with examples, and shown that it can result in dramatic reductions of the trusted computing base without being inordinately difficult. We have demonstrated that it is a realistic methodology by developing a key enabling technology for it: DPLs, a class of languages that significantly facilitate the formulation of algorithms as theorem provers. We have extensively investigated the theoretical foundations of DPLs, and we have built an efficient implementation of Athena, a DPL that integrates a higher-order functional language with natural deduction. We have successfully used Athena to express many interesting algorithms as theorem provers. Other programming languages could also be conservatively extended to include DPL-style proofs and proof methods.

On the spectrum of formal methods, certified computation is not as light-weight as, say, assertion annotations with automatic code instrumentation; but it is not as heavy-weight as total verification either. Total verification is complicated by the fact that we need to reason about all possible inputs and all possible control-flow paths. That is not necessary when implementing a certifying algorithm. Invariants do not have to be explicitly formulated and proved. Techniques for handling infinite-state systems such as structural induction are not necessary. Induction is essentially replaced by more concrete recursive calls. We believe that the constructive nature of proof in certified computation (with deduction being performed by algorithms that can be executed, observed, debugged, etc.), will be more appealing to programmers than the more abstract techniques required for static verification.

*Acknowledgments:* We would like to thank Viktor Kuncak, Darko Marinov, and Olin Shivers for several helpful insights and suggestions.

## References

[1] K. Arkoudas. Denotational Proof Languages. PhD dissertation, MIT, 2000.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] Mike Barnett and Wolfram Schulte. Spying on Components: A Runtime Verification Technique. In *Proceedings of the OOPSLA 2001 Workshop on Specification and Verification of Component-Based Systems*, 2001.

[4] D. Bartetzko et al. Jass - Java with assertions. In *Proceedings of the 2001 Workshop on Run-Time Result Verification*, 2001.

[5] B. Beizer. *Software testing techniques*. Van Nostrand Reinhold Co., New York, NY, 1990.

[6] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, 1995.

[7] Lilian Burdy et al. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *Electronic Notes in Theoretical Computer Science*, volume 80. Elsevier, 2003.

[8] Feng Chen and Grigore Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In Oleg Sokolsky and Mahesh Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[9] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[10] M. J. Gordon, A. J. Miller, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[11] 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.

[12] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[13] Klaus Havelund and Grigore Rosu. Monitoring Java programs with Java PathExplorer. In Klaus Havelund and Grigore Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier, 2001.

[14] W. A. Howard. The formulae-as-types notion of construction. In J. Hindley and J. R. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalisms*, pages 479–490. Academic Press, 1980.

[15] S. Khurshid. Generating structrually complex tests from declarative constraints. PhD thesis, MIT, 2003.

[16] Moonjoo Kim et al. Java-MaC: a Run-time Assurance Tool for Java Programs. In Klaus Havelund and Grigore Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier, 2001.

[17] R. Kowalski. *Logic for Problem Solving*. North-Holland, Amsterdam, 1979.

[18] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[19] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[20] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[21] R. Milner. The use of machines to assist in rigorous proof. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, International Series in Computer Science, pages 77–87. Prentice Hall, 1985.

[22] Madanlal Musuvathi et al. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[23] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[24] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *1998 PLDI Proceedings*, pages 333–344, 1998.

[25] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.

[26] Maria-Cristina Patron and Dexter Kozen. Certification of Compiler Optimizations using Kleene Algebra with Tests. Technical Report TR99-1779, Cornell University, Computer Science Department, 1999.

[27] L. Paulson. *Isabelle, A Generic Theorem Prover*. Lecture Notes in Computer Science. Springer-Verlag, 1994.

[28] Lawrence C. Paulson. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5(2):143–169, 1985.

[29] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151+, 1998.

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

[31] J. L. Ruiz-Reina et al. Mechanical verification of a rule-based unification algorithm in the Boyer-Moore theorem prover. In *AGP'99 Joint Conference on Declarative Programming*, pages 289–304, 1999.

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

[33] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.

[34] W. Wechler. *Universal Algebra for Computer Scientists*. Springer-Verlag, 1992.

[35] J. Wing and V. Mandana. A case study in model checking software systems. *Science of Computer Programming*, 28:273–299, 1997.