Abstract machines and small-step semantics: a winning ticket for proof automation?

Alain Delaët, Sandrine Blazy, Denis Merigoux

Abstract Machines and Small-step Semantics: a Winning Ticket for Proof Automation?

Alain Delaët
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
alain.delaet@inria.fr

Sandrine Blazy
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
sandrine.blazy@irisa.fr

Denis Merigoux
Inria
Paris, France
denis.merigoux@inria.fr

Abstract

Verifying the correctness of a program using an interactive proof assistant involves first defining in the proof assistant the semantics of the involved programming language. Once mechanized, the semantics serves as the basis for the whole proof development, as it is referred to by all subsequent theorems. Traditionally, semantic judgments are mechanized using recursive inductive predicates, matching the pen-and-paper inference rules of operational semantics. However, the shape of these judgments may make proof engineering and maintenance tedious, requiring complex proof automation frameworks. This problem is especially acute when iterating during the language design period.

In this paper, we highlight a different style of writing and mechanizing semantics that follows the CEK abstract machine. We explain how exactly this style interacts better with basic proof automation tactics than the traditional structural operational semantics (SOS) style for the λ-calculus. As such, we explicitly detail why this alternative semantic style allows for scaling up more efficiently the mechanized proof development, and hint at future improvements based on the same principles. After detailing the inner workings of the interactions between the semantic style and proof automation of common theorems, we perform a case study for the medium-sized Catala domain-specific programming language. All of our examples and case studies are mechanized using the Rocq proof assistant and our development is available as a supplementary material [21].

CCS Concepts

• **Software and its engineering** → **Software verification**; **Semantics**; **Context specific languages**.

Keywords

verified compilation, operational semantics of programming languages

1 Introduction

Proof assistants have come a long way in the last twenty years, to the point where software as complex as realistic compilers has been proven correct [2, 3, 9, 13, 31]. A verified piece of software consists of the software itself, which runs outside of the proof tool (as any other software), together with a machine-checked proof that the software is correct with respect to its specification. When the software implements a program transformation (*e.g.*, a compilation pass), its specification expresses its semantic preservation (and possibly other semantic properties), thus involving the formal semantics of the underlying programming languages.

Semantic reasoning on realistic languages generates large proofs and requires a proof tool. Automatic proof tools discharge logical formulas to logic solvers (that may fail to prove the formulas), but they are not very well adapted to proving the correctness of sophisticated program transformations operating over realistic languages. Indeed, to be able to reason about the semantics, it is necessary to somehow deeply embed it inside the proof tool. Hence, proofs about semantics require tricky reasoning. For this reason, proof assistants (a.k.a. interactive proof tools) are rather used in these situations where user interaction is still needed at each proof step to decide on what to reason about. The price to pay is that proof assistants require writing these elementary proof steps (called *tactics*), which makes the proofs difficult to maintain over time as new features are added or statements are reformulated. What is especially tricky is defining and proving the semantic invariants that hold during the proof of a given theorem, as this may necessitate inventing whole new "phantom" data structures for the sole purpose of the proof.

Nearly all theorems in a mechanized development for verified software will refer back to the semantics of the language, whether one is building a verified compiler [2, 3, 9, 13, 31], a verified abstract interpreter [27], or a verified bug-finder [16]. As such, the encoding of the language semantics inside a proof assistant is a critical and central software component of the mechanized development. From a software engineering point of view, the design patterns and coding style of the language semantics as a program in the proof assistant heavily influence the rest of the mechanized development, as it is a direct dependency to everything else.

Compared to other semantic styles (mainly denotational and axiomatic), operational semantics defines precisely the low-level execution of a program and is a de facto standard for mechanizing semantic reasoning in a proof assistant [28, 44, 46]. For instance, the correctness theorem of a program transformation (from a source to a target program) states that any execution of one of the programs is *simulated* by the execution of the other one, while ensuring some invariants between execution states (which can be tricky to define and mechanize). In compiler verification, because of optimizations, bisimulations do not hold, and forward or backward simulations are rather used to simulate one execution step by zero, one or several execution steps, given an *invariant* between semantic states.

Such subtle proofs require a great deal of precision from the operational semantics w.r.t. details like execution order. While certain styles of operational semantics such as big-step are easier to manipulate inside a proof assistant, they may not always contain all the information necessary for proofs involving complex features (*e.g.*, concurrency, weak memory models), forcing the proof engineer to default to a more tedious small-step operational semantics.

The proof may require iterating between proof development and theorem statement refactoring, in order to define new semantic invariants and intermediate lemmas. Moreover, it may be useful

to debug the semantics, which requires replaying the proof many times over. There are other reasons why proofs need to be replayed (and updated): firstly, programming languages evolve to satisfy users' needs, and secondly, new intermediate representations may be added to refactor the proofs (and split them in smaller and more maintainable pieces).

Furthermore, automating these proofs is not a simple task, as the sequence of elementary proof steps is not straightforward; it depends on the property to prove as well as a precise combination of inductive and case-based reasonings. If the individual proof steps are well understood, as they correspond to specific tactics (mainly induction or case-based reasoning, inversions, or formula simplification), there are a number of ways of combining them. This raises the question of what we can do to improve proof engineering and especially proof automation in mechanized developments involving language semantics, which is the subject of this paper.

The main thesis of this paper is that *using abstract machines defined syntactically to model small-step operational semantics allows for a simple but powerful proof automation, leveraging the unification capabilities of the proof assistant.* If this result has already been hinted at multiple times in the literature and is part of the folklore amongst the connoisseurs, we explain in depth and going back to the basic mechanisms of proof automation *why* abstract machines work better in mechanized developments. We believe this extensive description of the phenomenon might guide future improvements in proof automation leveraging clever encoding of semantics, and bring more scientific method into the quest for scaling up mechanized proofs.

1.1 Related Work

Before detailing our contributions, it is necessary to give some context about abstract machines, semantics, and encoding choices inside proof assistants in major mechanized developments over time.

Abstract machines for describing operational semantics have been the subject of extensive study since the inception of the λ-calculus and subsequent languages. Starting with Landin's SECD machine [32], many theoretical abstract machines have been proposed, including Felleisen and Friedman's CEK machine [23] and Krivine's machine [30] that model the different reduction strategies. These machines then influenced the implementation of modern functional programming languages, for instance through the CAM [18] and later the ZAM [34] from which the initial implementation of Caml was derived.

While being a popular foundation for language implementation, we notice that abstract machines have not become the default style for encoding language semantics inside proof assistants. Notably, the famous Software Foundations manual [44] overwhelmingly used to teach Rocq and language theory describes the small-step operational semantics of the λ-calculus using a recursive judgment and substitution. Indeed, this style of semantics encoding, which we will call structural operational semantics throughout this paper, is the closest to the pen-and-paper presentation of small-step operational semantics, making it the best teaching vehicle. We suppose that, because of its absence in standard teaching material, abstract machine semantics have been neglected in mechanized proof developments presented in the literature. To substantiate this claim, we will survey the semantic encoding choices of major works striving to scale up mechanized program verification.

CakeML [31] is a verified compiler for a substantial subset of Standard ML, large enough to bootstrap itself. The semantics of the CakeML languages use mainly functional big-step style [42] (*i.e.*, functions are used instead of relations to define the semantic rules), but interestingly an abstract machine (small step) semantics in the style of CEK was used in earlier versions of the compiler "for the type soundness proof" [42, p. 2]. In general, the use of big-step semantics avoids the bookkeeping of structural operational semantics and scales up better, but they are not adapted to define in a natural way some semantic features such as unstructured control, diverging and concurrent executions. More generally, "[a]s a rule-of-thumb, the more complex the language's features, or the more semantically intricate the desired theorem, the more likely it is that small-step semantics will be needed" [42, p. 1].

Another classic twist on the structural operational semantics mechanization is the factorization of the recursive part of the judgment inside a context, with an unique contextual rule. This twist has been used for a long time [12], and is reused inside a recent survey of mechanization inside Rocq of reduction strategies for structural operational semantics [7]. This twist, as we will explain later in the paper, does not avoid very manual proofs with inversion lemmas and explicit lemma applications, following the style popularized in [44]. It is not an issue as [7] implements a simple pure λ-calculus, but the syntactic proof of type safety [57] is quadratic in the number of constructors and scales poorly. This pitfall is pointed out by [50] which proposes a fix, extended to non-deterministic languages by [15], but the fix requires a clever re-stating of the type safety theorem, which might be difficult to port to the proof of other properties.

Another formally verified compiler for a functional language is CertiCoq [3], a compiler for the specification language of Rocq. Its intermediate languages use big-step semantics and avoid abstract machines. To end our tour of verified compilers, we can also consider Vélus [14], which proves the correctness of a large subset of Lustre using CompCert. Vélus uses a mix of inductive and co-inductive objects to express relations between streams of values. Its main correctness proofs work by performing induction on an object, inversion on other inductive hypotheses, and reconstructing the proof in each case. Moreover, [14, Section 3.2] highlights the difficulty of discovering the details required to mechanize a proof. These details are typically found while writing the proof and often require modifications to the theorem's hypotheses and semantic definitions.

Outside of verified compilers, domain-specific mechanized developments using language semantics also report difficulties for scaling up their proofs. Benzaken et al. [6] prove the correctness of a SQL to IMP scheduler by defining their semantics using big-step semantics. Similar to Vélus, their proof is not automated and requires keeping the language small to remain tractable. For example, they avoid using `if` in one of their intermediate languages to keep the number of inductive variants low, expressing it through other constructs, though at the cost of increasing the length of the generated code. Another key application of mechanized proofs for

software verification is cryptography. The lines of work around Fiat [20, 22] or the recent Noise* [26] rely on synthesis and metaprogramming inside the proof assistant to generate verified code from a high-level specification. These techniques, like other recent works [25], allow for a very high degree of proof automation, at the expense of a large proof infrastructure that may be tedious to maintain and expand.

Finally, we can also look at specification efforts for semantics of large, real-world languages. Extensive work has been recently put into formalizing WebAssembly, in Isabelle [55] and in Rocq [47]. Both mechanizations use small-step semantics, but, to Watt's admission [56], "[B]ecause WebAssembly was deliberately designed to be small, simple, and amenable to formalisation, mechanizing the full semantics [...] is more easily achievable". On the JavaScript side, JSCert [10] uses big-step semantics. The K framework [52] has been extensively used to model several high-profile real-world programming languages, but the framework is not a proof assistant and does not allow for the mechanized proofs of meta-properties involving the semantics. Overall, the related work presented above tends to flee from mechanizing large small-step semantics to avoid manual proofs that scale and repair poorly.

Nonetheless, it is sometimes impossible to avoid small-step semantics because the proof relies on the execution order or details missing from big-step semantics. Then, mechanized developments tend to flock back to some kind of abstract machines to support the scaling up, although some fail to mention their use of abstract machines properly. For instance, Appel and Blazy [4] present a small-step semantics using an abstract machine (which they name "*continuations*") for the statements of the `Cminor` language of CompCert. They then define a sound separation logic for their imperative language. They report that the use of this abstract machine "allow an uniform representation of statement execution", without elaborating. It is worth noting that these "continuations" are first-order syntactic data structures representing the control stack of an abstract machine, distinct from the higher-order continuations of continuation-passing style in denotational semantics. More recently, syntactically encoded abstract machines were incorporated in the Skeletal semantics framework [11]. They provide general tooling for defining and manipulating operational semantics. Skeletal semantics were used on toy languages to derive interpreters or static analyzers automatically from operational semantics. To improve automation in skeletal semantics, Khayam and Schmitt [29] define a purely functional meta-language that captures target language features and introduces control stacks to derive abstract machine semantics.

Additionally, Courant et al. [17] use abstract machine semantics for the paper proof of the correctness of their code generator, and claim that they could be "easily mechanized". Going beyond semantics, using syntactic artefacts to efficiently encode judgments in a proof assistant seems to have a broader application scope. Li and Appel [35] encode program transformations in a similar fashion, using MetaCoq [54] to enable proof sharing between different transformations. However, the correctness proof for their transformations still requires significant work: they focus more on extracting efficient implementations of transformations rather than easing their mechanized proof development.

These hints from the literature pointed us to recognize that behind these semantics using "continuations" (i.e., control stacks), there is a revamp of abstract machines for the λ-calculus, inside proof assistants. Their syntactic features seem to help mechanized proof scale up, and lower the barrier to using small-step semantics in large developments. *Why? Can we generalize this insight and leverage it to obtain better proof automation for more theorems involving semantics?* We will answer these questions in the paper by showing the fine-grained interactions between semantic encodings and proof tactics, using first a call-by-value λ-calculus with both a structural operational semantics and a abstract machine semantics following the CEK machine.

This subject arose for us during the development of a new verified compiler for the Catala domain-specific programming language [39]. Catala is a language designed for situations where a given set of laws or regulations has to be turned into computer code for automatic enforcement. In practice, this situation concerns mostly tax or social benefits computations. These strong requirements make these sometimes-thousands-of-lines-of-code-long programs [40] hard to write and maintain, as the source of truth for their behavior is determined by legal interpretation performed by government lawyers. Catala has a set of syntactic and semantic features that facilitates the review of the program source code by lawyers. First, Catala allows for literate programming, locally linking each snippet of computer code to the article of statutes or regulations that specifies it. Second, Catala's semantics is based on default logic [33], a logic that matches how the law lays out its instructions for how to compute things. Because tax- and social-benefits-computing programs are critical for the smooth operation of state apparatus, a high level of assurance is also required of the Catala compiler that translates the source Catala code to target programming languages such as C or Python.

1.2 Contributions

We defined a new intermediate representation λ^δ for Catala, improving upon the original default calculus from [39], and mechanically verified an alternative compilation scheme using an option monad instead of try/catch exceptions. These efforts will thus be used as an empirical evaluation of the proof engineering and automation techniques discussed in this paper. While Catala is not as complex as real-world general-purpose languages like C [36], it is based on a simply typed λ-calculus with classical extensions, enriched with a default calculus involving default terms written as $\langle t^* \mid t \text{ :- } t \rangle$. For instance, the simple default term $\langle t_{\text{ex1}}, t_{\text{ex2}} \mid t_{\text{just}} \text{ :- } t_{\text{cons}} \rangle$ has two exceptions t_{ex1} and t_{ex2}, that defaults to a base case consequence t_{cons} when the base case justification t_{just} is true. This logical structure posits that exceptional cases should apply first, and only by default the base case should apply. It fits very-well to define law and regulations. Moreover, default terms have been evaluated empirically as powerful enough to accurately formalize the entirety of the real-world French housing benefits [38]. The use of small-step for the Catala semantics helps to pin down precisely the execution order and error semantics of the default term. While it may not be completely necessary as the semantics could be simple enough to be converted into big-step, the small-step semantics have been useful during the design phase, when iterating over the semantics

of the default term. While later works like [24, Section 4.1] and [5] have proposed optimizations or simplifications for the semantics, we will stick in this paper to the original small-step style of the Catala semantics that also matches the compiler implementation.

Adding new features (incl. the default calculus) to Catala resulted in the λ^δ language, which exhibits the key characteristics where abstract machine semantics shine for proof automation: syntactic ambiguity arising when multiple reduction rules can apply to a single language construct. Unless noted otherwise, all results presented in this paper have been mechanically verified using the Rocq proof assistant. The complete development is available as supplementary material. Specifically, we claim the following novel contributions.

- **Semantics styles:** We mechanize the proof of equivalence between a structural operational semantics and a abstract machine semantics equivalent to a CEK machine for the call-by-value λ-calculus (prior mechanizations of the CEK machine exist [51], but we have not yet found a mechanization of this equivalence proof with the structural operational semantics).
- **Semantics and tactics:** We explain the different interactions between basic proof automation tactics and semantic judgments, depending on the style of semantics.
- **Proof engineering:** We rely heavily on state-of-the-art proof automation techniques and present powerful but low-tech meta-interpreters based on a couple of primitive tactics, leveraging the proof assistant's syntactic unification.
- **Application:** We illustrate how our findings scale on the proof of a novel compilation pass for the λ^δ domain-specific programming language of Catala.

This paper is organized as follows. First, Section 2 presents two small-step semantics for a call-by-value λ-calculus. Then, Section 3 details how both styles interact with the basic blocks of proof automation in a proof assistant. Finally, Section 4 illustrates how abstract machine semantics allows for better scaling up of proofs in mechanized developments. It introduces λ^δ, a new mechanized semantics for Catala's core intermediate representation, the default calculus, as well as the proofs of various theorems around λ^δ, including the correctness of a non-trivial compilation pass.

2 Mechanizing Small-step Semantics: Choose Your Style

There are many ways to mechanize semantics inside a proof assistant. Coupled with the criticality of the semantics as a central proof component, the choice of semantic styles makes for a hard decision for the proof engineer, as this choice will determine the shape and details of all future proofs, as well as how easy they will be to write, debug and repair. Admittedly, the syntax of terms quite naturally maps to an inductive type inside the proof assistant, without much room for user fantasy. But the semantic judgments can be mechanized in different ways. Indeed, one can choose to mechanize typing and reduction as total or partial functions over terms, thus building an executable typer or interpreter [42]. But this encoding differs quite significantly from semantics written on paper, with standard inference rules.

As the semantics encoding is part of the trusted computing base of the mechanized development, it is better for human review [37, Section 1.2.1] if the encoding matches closely the pen-and-paper rules. Also, since functions have to be proven total, encoding semantics as functions quickly becomes cumbersome. Hence, the recommended way [41, 44] to mechanize semantics is to define them as inductive types, where each type constructor corresponds to an inference rule, which ensures that the inductive reasoning and the case analysis closely follow the pen-and-paper ones.

But even when encoding the semantic judgment inductively, the user is still left with some degrees of freedom, like the choice between small-step or big-step semantics. They are equivalent but they will be expressed with very different rules, and mechanized differently. As stated in the introduction, this paper will focus only on small-step semantics. But additionally, there are several ways of defining small-step semantics, depending on the way semantic states are defined. This is precisely the point of this paper, and we will show two such small-step semantics: first, *structural operational semantics*, using recursive premises in the reduction judgment, and second a *abstract machine semantics*, using syntactically-defined abstract machines to encode the control stack that describe the context in which we step.

This section presents both styles: the first one feels familiar as it represents the most taught and presented style. On the contrary, the second one can feel more artificial and verbose, while very precise and exhaustive. Why bother with the second style? We answer in Section 3, where we dive deep into the issues of proof automation and proof repair, and compare the styles. This section presents the two styles for a simply-typed λ-calculus augmented with conditionals that simplifies the more realistic language λ^δ we detail later in Section 4. A value is a Boolean value or a function abstraction. A term is a variable x, a value v, a function application $t\ t$ or a conditional. Its (standard call-by-value) semantics is defined by the judgments $t \longrightarrow t'$ for traditional small-step semantics, and $s \rightsquigarrow s'$ for abstract machine semantics (see Figure 1).

Values	v	::=	$\texttt{true} \mid \texttt{false} \mid \lambda\, x.\, t$
Terms	s	::=	$x \mid v \mid t\ t \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t$

2.1 Traditional Small-step Semantics

There are two kinds of rules defined in Figure 1a: contextual rules (4), (5) and (6) have recursive premises, and computation rules (1), (3) and (2). Rule (1) defines the semantics of function application ; the substitution mechanism here is assumed to be capture-avoiding; in practice our mechanized development uses de Bruijn variables and `autosubst` [53]. While computation rules actually encapsulate the core of the semantics, contextual rules encode the recursive nature of the evaluation and its order. In pen-and-paper presentations, it is common to define more compact semantics using reduction contexts to factor out the contextual rules as shown in Figure 1b. However, the actual effects of the context are more difficult to parse. In mechanized semantics, the context C is an inductive type defining different contexts including a hole. Then, a function, named the context application function, is defined to take a term and a context, and return the new term corresponding to the context where the hole has been replaced with the argument term. The

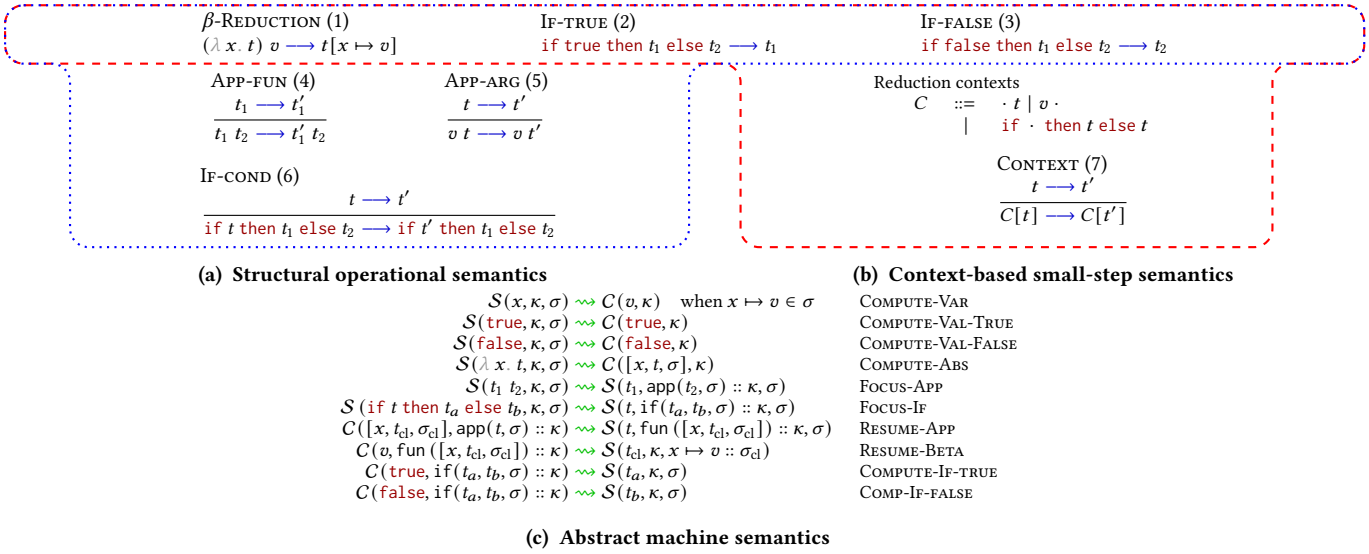


Figure 1: Three small-step semantics for a simply-typed λ-calculus with conditionals

context reduction rule uses this function to state that if a term reduces, then any context applied to this term also reduces.

Both semantics (with or without explicit context) share key characteristics. First, their judgment is recursive. Second, this recursiveness concretely adds a (recursive) premise to rules (4), (5) and (6). In the contextual version of the structural operational semantics, the recursive premise only appears in rule (7). These recursive premises heavily impact the manipulation of the rules during a meta-theoretical proof. Indeed, suppose we want to know which rules apply to a term t to determine t' such that $t \longrightarrow t'$, which is a very frequent proof-pattern. We case analyze each rule to determine whether it can apply or not. The shape of t allows us to discard many rules. However, suppose t is $\texttt{if true then } t_2 \texttt{ else } t_3$. Then, we can narrow down eligible rules to If-true, but also the condition-related contextual rule (If-cond or Context), which might apply because we cannot rule out immediately that the premise $\texttt{true} \longrightarrow t'$ might hold for some t'. Of course, we can further deduce that there is no such t', but this requires an extra proof step. Hence, the determination of which rule can be applied for t requires more than a purely syntactic analysis of t; it also requires a recursive examination of which rules apply for its sub-terms. While this extra consideration is not an issue for pen-and-paper proofs, it matters when automating mechanized proofs. Of course, given suitable lemmas, these steps can be shortened, but in turn these lemmas have to be maintained over updates in the semantics, making the codebase heavier and more tedious to evolve.

2.2 Abstract Machine Semantics

Operational semantics may rely on an abstract machine to model the executions between semantic states. We defined our toy λ-calculus using substitution, with no need for such a machine, as its semantic states are only defined from the syntax of terms. However, operational semantics of realistic languages rely on environments, *e.g.*, to map variables to values. Pushing this idea further, the reduction judgment sometimes features complex semantic elements that encode information about the state.

Our abstract machine semantics relies on a variant of the CEK abstract machine [1, 23]. Its semantic states rely on a control stack that tell the reduction where to resume when the current subterm is fully reduced. This makes the semantics less readable and elegant than traditional small-step semantics, which are a more human-readable and pleasant form of semantics, ideally suited to a pen-and-paper presentation and reviewing. However, we will later argue that it may not be the best way to mechanize semantics.

The judgment is $s \rightsquigarrow s'$, where states s and s' augment the term t being reduced with extra information about the state of its reduction in control stack κ, that represents the remaining steps that need to be done once t is reduced. We distinguish between evaluation states $\mathcal{S}(t, \kappa, \sigma)$ stepping from t, and return states $\mathcal{C}(v, \kappa)$ when a step returns a value v, namely Boolean values and closures. Contrary to the structural operational semantics presented above, we chose to avoid substitutions in our abstract machine semantics and use environments instead. This may appear to bias the comparison, as we could have chosen to base our abstract machine semantics on the CK abstract machine that uses substitution. But using syntactically-defined environments is more aligned with the philosophy of this paper, and helps leverage the same proof automation techniques as the rest of the abstract machines in the semantics. Furthermore, our mechanized development for abstract machine semantics handles variables and environments natively, which we compare to our structural operational semantics mechanized development that uses `autosubst` to discharge substitution-related lemmas, hence levelling the playing field between the two styles.

The environment σ maps variables to values. A control stack κ is a list of frames k together with their environment σ. A frame k remembers where a sub-term is being reduced in a larger term. So, there are as many frames as reduction contexts, and σ is a mapping from the variables in the sub-terms of k. For instance, $\texttt{if}(t_1, t_2, \sigma)$

corresponds to `if` $\cdot$ `then` t_1 `else` t_2, then $\mathsf{app}(t, \sigma)$ corresponds to $\cdot\ t$, and $\mathsf{fun}\,([x, t, \sigma])$ corresponds to $(\lambda\ x.\ t)\ \cdot$.

Values	v	::=	$[x, t, \sigma] \mid$ `true` $\mid$ `false`
Environment	σ	::=	$x \mapsto v, \sigma \mid \cdot$
Control Stack	κ	::=	k^*
Frame	k	::=	$\mathsf{app}(t, \sigma) \mid \mathsf{fun}\,([x, t, \sigma]) \mid \mathtt{if}(t, t, \sigma)$
State	s	::=	$\mathcal{S}(t, \kappa, \sigma) \mid C(v, \kappa)$

There are three kinds of rules in our semantics, depending on whether a rule *computes* a result (and steps to a C state), or *focuses* on a sub-term (and steps from a $\mathcal{S}$ state to another $\mathcal{S}$), or *resumes* a computation from a result (and steps from a C state to a $\mathcal{S}$ state). For instance, Focus-App *focuses* the step from $t_1\ t_2$ to t_1, keeping t_2 along with the current environment σ in the control stack κ. The value of t_1 is a closure $[x, t_{\text{cl}}, \sigma_{\text{cl}}]$, computed by Compute-Abs. Then, the evaluation of $t_1\ t_2$ resumes by popping from the control stack, and switching to the next frame corresponding to the reduction of t_2 along with the saved environment. When t_2 evaluates to v, then Resume-Beta performs the equivalent of the substitution by updating the environment σ with $x \mapsto v$ and resumes the evaluation of the function body. By switching alternatively between $\mathcal{S}$ and C states, the rules carefully decompose the reduction steps in a very predictable manner.

The rules (see Figure 1c) are easy to read, mainly as they do not update σ; they are not recursive and easy to mechanize inductively. In the end, there are more rules (to focus on a sub-term or resume a step). However, in the structural operational semantics, determining which rule can be applied for a given term t requires more than a purely syntactic analysis of t (which is enough here); it also requires a recursive examination of which rules apply for some sub-terms of t. Thanks to control stack, this extra requirement disappears: the syntactic shape of the initial state completely and uniquely determines which rule to apply. In a nutshell, switching to control stack adds new rules, but their structure is harmonized and more regular, leading to a better proof automation.

3 Proof Ergonomics and Repair: Comparing Styles

This section explains why the abstract machine semantics is easier to mechanize than the structural operational semantics. Indeed, the rules of the former do not have recursive premises stepping on sub-terms, thanks to control stack. So, the behavior of common elementary proof-patterns (that are general enough not to be specific to Rocq) becomes much simpler. In Rocq, it relies on tactics such as `constructor`, aimed at proving a conclusion defined inductively by applying one of its constructors (and in particular applying the adequate semantic rule), and `inversion` aimed at exploiting unused hypotheses. Thus, abstract machine semantics enables us to automate large parts of our proofs, as hinted in CompCert [8].

Before diving into the proof engineering comparison, we emphasize that all results presented in this section are backed by a complete mechanization in Rocq comprising approximately 8,789 lines of code (4,635 lines of specifications and 4,154 lines of proofs). Our formalization includes both semantic styles for the simply-typed λ-calculus with and without conditionals presented here (1,277 lines for basic λ-calculus and 2,317 lines for λ-calculus with conditionals), as well as the more realistic language λ^δ detailed in Section 4 (2,654 lines for the Catala implementation). The development itself is available here [21], with detailed correspondences between paper statements and mechanized proofs documented in the accompanying `CLAIMS.md` file.

We gathered the insights presented in this section during the development of the mechanized formalization; but rather than presenting the formalization itself here, we preferred to take a step back and look for the root causes of our claims about proof automation, tactics behavior, and maintenance overhead.

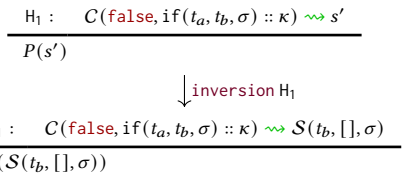
3.1 The Effect of Semantic Style on Common Proof Patterns

The absence of recursive premises in abstract machine semantics improves proof automation in three different ways. First, it makes hypotheses more precise, and avoids introducing inconsistent proof goals when inverting a hypothesis, which structural operational semantics inevitably does (see 3.1.1). Second, it prevents `constructor` from introducing goals that turn out to be false further on in the proof. This makes automation harder as we need to know which rule leads into false proof goals (see 3.1.2). Third, it allows us to switch the target of inductive reasoning away from recursive judgments and towards the very simple list structure of the control stack, for which we can reuse standard-generated induction principles *as is* (see 3.1.3). We illustrate relevant proof goals in Figure 2. The proof goals are a simplified view of the Rocq interface when conducting proofs using one of our semantics. Each proof goal consists of one or more hypotheses at the top, and a conclusion at the bottom separated by a line. The goal is to show the conclusion from the hypotheses using tactics.

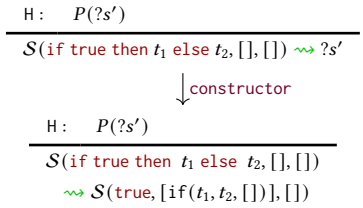
3.1.1 Inverting Judgments. Inverting judgments is a staple of semantic proofs to infer required information, *e.g.*, in Figure 2a, from $C(\texttt{false}, \mathtt{if}(t_a, t_b, \sigma) :: \kappa) \rightsquigarrow s'$, we can deduce that $s' = \mathcal{S}(t_b, \kappa, \sigma)$. The `inversion` tactic unifies a hypothesis with every constructor of its type. But we could also have multiple constructors, and `inversion` would generate a new proof goal for each unification, which may trigger further inversions, as shown in Figure 2c when inverting `H1`: this produces both cases `true` $\longrightarrow t'$ and $t' = t_1$. But inverting a structural operational semantics hypothesis often generates irrelevant proof goals; in our previous examples, `true` $\longrightarrow t'$ is irrelevant because values cannot reduce.

When `inversion` is applied many times in a row, and since each application can generate proof goals, this leads to an exponential growth of the proof cases to analyze and makes the tactic practically unusable in the context of automation. It should be possible to modify the rules of a structural operational semantics to avoid this issue by syntactically separating values from terms. However, this is not satisfactory, as values are still terms, so we would need one more inversion to realize that no rule can be applied.

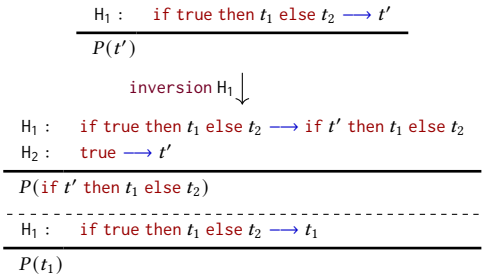
A solution is to add a side condition that t_1 should not be a value, which can be syntactically checked. However, adding non-value premises overloads the semantics in a inconvenient way, and does not fix further proof automation problems. A better solution is to rely on the distinction between evaluation and return states. Hence, `inversion` does not introduce new incoherent proof goals. For instance, inverting $\mathcal{S}(\texttt{if true then } t_1 \texttt{ else } t_2, [], []) \rightsquigarrow s'$, will



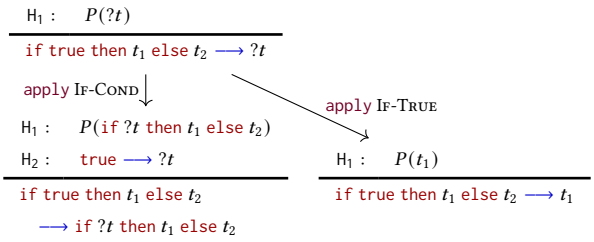
(a) Applying `inversion` on abstract machine semantics. A single proof goal is generated.



(b) Applying `constructor` on abstract machine semantics. The tactic selects the expected Focus-If rule.



(c) Applying `inversion` on structural operational semantics. Two proof goals are generated (separated by the dotted line).



(d) Applying `constructor` on structural operational semantics. It may choose one of the two applicable rules If-Cond or If-True depending on the order of the rules.

Figure 2: Proof goals of the behavior of the `inversion` and `constructor` tactics with both semantics.

only give $s' = \mathcal{S}(\texttt{true}, [\texttt{if}(t_1, t_2, \sigma)], [])$. This helps greatly when scaling the proof to more complex constructs.

Such a unique result of an inversion is a meta-property of the reduction judgment, which we check empirically by trying to invert every possible instance, and assert that at most one unique proof goal is generated. It could also be expressed as a MetaCoq [54] statement and proved accordingly. We leave the illustration of this MetaCoq case study as future work.

3.1.2 Introducing Judgments. A common pattern when trying to prove that a step holds is to use `constructor` to choose the possible (and hopefully single) semantic rule. We need to figure out its outcome, which can be achieved using existential variables, noted $?t$, that represent unknown terms that need to be instantiated. So, we generate a step to an existential variable $?t$ and apply `constructor` to instantiate $?t$ when it finds a fitting rule. For instance, rule If-True matches $\texttt{if true then } t_1 \texttt{ else } t_2 \longrightarrow ?t$ and $?t$ becomes t_1.

Contrary to abstract machine semantics, using `constructor` on a structural operational semantics may generate unprovable proof goals. For example, in Figure 2d, when we use `constructor` on $\texttt{if true then } t_1 \texttt{ else } t_2 \longrightarrow ?t$, both If-Cond and If-True may be applied. The first one will generate $?t'$ (and $\texttt{true} \longrightarrow ?t'$) and instantiate $?t$ as $\texttt{if } ?t' \texttt{ then } t_1 \texttt{ else } t_2$. The proof goal then becomes $\texttt{if true then } t_1 \texttt{ else } t_2 \longrightarrow \texttt{if } ?t' \texttt{ then } t_1 \texttt{ else } t_2$. This goal holds only if $\texttt{true} \longrightarrow ?t'$, which cannot happen. So, `constructor` can pick from two matching rules but only one (If-True) allows us to conclude the proof. More generally, the fix of adding non-value premises to contextual rules leads to more instances of the problem of picking the right match for `constructor`. There are two workarounds but none is satisfactory. The first one is to reorder the semantic rules, so that `constructor` will select the most precise rule. Sadly, this approach is very brittle, as this hinders proof repair [48]. The second workaround is to avoid using `constructor` and apply manually the correct constructor, but this leads to long and manual proofs.

In contrast, in abstract machine semantics, `constructor` almost always either fails, or selects the only applicable rule, without introducing unprovable goals, as for instance in Figure 2b. Still, some rules have premises (*e.g.*, to check whether a variable belongs to σ). But, thanks to the control stack, they are not recursive for the semantics, contrary to the contextual rules in structural operational semantics. It thus becomes possible to automate their proofs. Furthermore, we know that at most one rule is applicable without considering the premises. Therefore, if a premise is unsolvable, the step itself is unsolvable. So, `constructor` does not introduce any unprovable goal. Overall, `constructor` becomes a very powerful yet simple proof automation tool for abstract machine semantics.

3.1.3 Generating and Applying Induction Principles. The last advantage of abstract machine semantics is the improved handling of induction principles, which is paramount in semantic reasoning. Indeed, a machine state consists of the remainder of a command and a control stack that describes the context in which it occurs, namely the control flow that is required by the reasoning. So, it becomes easier to reason inductively only on the structure of the control stack, rather than on a mix of case analysis together with rule induction on semantics.

First, let us describe inductive reasoning on structural operational semantics. When a step is recursive, one may need the induction hypothesis H_{ind} to concern the reduction of a deeper sub-term than the direct sub-terms. For instance, proving a predicate P by induction on $(t_1\ t_2)\ t_3 \longrightarrow (t_1'\ t_2)\ t_3$ may require P to hold on $t_1 \longrightarrow t_1'$.

Obtaining H_{ind} on deeper sub-terms is challenging because the automatically generated induction principle for $\longrightarrow$ is shallow: it only provides H_{ind} for direct sub-terms. Hence, it would merely state that P holds inductively for $t_1\ t_2 \longrightarrow t_1'\ t_2$. To address this issue, we can either write a custom induction principle (which is error prone) or do the induction on a well-founded measure (which may be tricky to define).

In paper-and-pencil proofs, the induction is often performed on the size of the judgment. Transposing this into proof assistants requires reflection capabilities on the terms, which in turn requires the proof to operate at a meta-level. Concretely, in Rocq, we would have to define our judgments as `Type` instead of `Prop` since we cannot write a function from `Prop` to `nat`. But working with `Type` instead of `Prop` is not recommended since the standard library lemmas on `Prop` cannot be reused, leaving the proof engineer to reprove all trivial results in the meta world. This increases the complexity of the proof engineering effort, making it an undesirable strategy to adopt. Performing induction on the size of terms is the most common solution, as it only requires defining a decreasing measure on terms. Such proofs generally start with performing induction on the size of terms and use `inversion` on the induction hypothesis featuring the semantic judgment. However, this adds an indirection between the induction and the semantic judgment, increasing the size of predicates and intermediate hypotheses. Moreover, it also requires using `inversion`, and thus, issues related to `inversion` may arise again. Finally, each application of the induction hypothesis requires checking that the size is indeed smaller, which may not always be the case. For instance, the β-Reduction rule might increase the size of the terms, requiring the measure to take into account the effects of yet-to-be-performed substitutions.

The other workaround is to write a custom induction principle, but it requires a deep understanding of the proof before writing it, and it is difficult to maintain [48] since it is the synchronized combination of the induction principle and the induction predicate that makes the proof go through. Since the induction principles are not yet defined at this stage, they must be written manually, often requiring low-level tactics or basic lemmas. As a result, any update of the semantics will require tedious manual restating and reproving of the induction principles. By contrast, abstract machine semantics handles induction principles smoothly. Since the judgment is not recursive, inductive reasoning boils down to case analysis. Interestingly, we found that properties such as determinism, progress, and even the correctness of a translation pass require no more than a case analysis of the semantic judgment.

Nevertheless, a stronger truly recursive induction principle is needed to prove some more complex theorems, as illustrated by the proof of the semantic equivalence between $\longrightarrow$ and $\rightsquigarrow$. We found that using well-founded induction on the control stack is sufficient in those cases, particularly due to the contextual reduction property it admits (formalized in Lemma 3.2). Each implication is a theorem proved by a standard simulation of one semantics by the other, involving an invariant $\equiv$ defining the matching between a semantic state and a term. For instance, the state $\mathcal{S}(\texttt{if true then } t_1 \texttt{ else } t_2, [], [])$ matches with the term $\texttt{if true then } t_1 \texttt{ else } t_2$, and the state $\mathcal{C}(\texttt{true}, [\texttt{if}(t_1, t_2, \cdot)])$ with the term $\texttt{if true then } t_1 \texttt{ else } t_2$. As there are more rules in the abstract machine semantics, a step $\longrightarrow$ may need to be simulated by several steps $\rightsquigarrow$, hence the multi-step execution $\rightsquigarrow^+$ and the 1-n simulation. To utilize the well-founded induction hypothesis, the proof of Theorem 3.1 relies on Lemma 3.2; it states that for each step $\rightsquigarrow$, there is a similar step, but with an extended control stack κ_x to the current state s (written $s +\!\!+ \kappa_x$). For example, $\mathcal{S}(t, \kappa, \sigma) +\!\!+ \kappa_x$ is $\mathcal{S}(t, \kappa +\!\!+ \kappa_x, \sigma)$. This allows us to apply the induction hypothesis on the smaller control stack κ when proving properties about the extended stack $\kappa +\!\!+ \kappa_x$.

Theorem 3.1. *Let s, t, t' be states such that $t \longrightarrow t'$ and $s \equiv t$. Then, there exists s' such that $s \rightsquigarrow^+ s'$ and $s' \equiv t'$.*

Lemma 3.2. *Let s_1, s_2 be states and κ_x a control stack. If $s_1 \rightsquigarrow s_2$, then $s_1 +\!\!+ \kappa_x \rightsquigarrow s_2 +\!\!+ \kappa_x$.*

It is not possible to generate such a compact lemma for $\longrightarrow$, even for two levels of reduction *i.e.*, two frames in κ_x, as it would require a number of distinct lemmas that is quadratic in the number of contextual rules. One way to mitigate this is to add a context in an unified contextual rule. We could then derive a similar lemma but it would still suffer from previous issues related to `inversion` and `constructor`. Lemma 3.2 is very useful in verified compilation proofs as it allows for reusing local results into larger contexts, for instance showing implications between different kinds of simulation diagrams for the same semantics; hence its usefulness outside the narrow scope of internal proofs for the abstract machine semantics.

So, structural induction on the control stack is particularly convenient for handling complex steps as it allows to introduce intermediary steps. Thanks to Lemma 3.2, we replace the control stack with an empty one and apply the induction hypothesis, prove the intermediary needed property, and go back to the control stack inductive cases to generalize our result. While easy proofs can be handled with straightforward case analysis like determinism and progress, induction on the control stack scales remarkably well, for instance for correctness proofs of compilation passes, independently of the level of sub-term inspection required to complete the proof.

3.2 Comparing Semantic Styles and Proof Automation

This section shows that choosing a semantic style with good interaction towards the building blocks of the previous subsection allows for easy, powerful proof automation. Furthermore, our techniques are resilient to semantic changes, facilitating proof maintenance and repair.

Most proofs involving semantics can be structured using the following generic proof outline. The proof begins with an induction on one hypothesis. Then, an inversion on the remaining judgments extracts the necessary information, and the proof proceeds by case analysis. Each case requires applying appropriate tactics and leveraging hand-crafted lemmas. This last phase is the most time consuming and challenging. Hence, one way to mitigate the complexity of the proof is to limit the number of cases where manual intervention is needed, by automating away the handling of a maximum number of cases.

Having a consistent proof outline during proof maintenance greatly simplifies the process. A study on the activity of Rocq proof engineers [49] shows that most of the maintenance time is spent

updating definitions, followed by repairing the impacted proofs. By keeping the proof outline unchanged during such changes, only the specific affected cases need to be updated. This ensures that proofs often do not need to be adjusted.

Although heavier automation techniques exist, they often require a deep understanding of the proof assistant's inner workings, which can make maintenance challenging [49]. Instead, we focus on lightweight automation that operates effectively with a well-suited semantic style. This approach works better with the abstract machine semantics style, due to its efficient interaction with `constructor` and `inversion` tactics. We now show how to leverage these benefits in two case studies, where we strive to simplify proofs and reduce their complexity.

3.2.1 Determinism and Progress. Our first case study is the proof that our semantics are deterministic: if $t \longrightarrow t_1$ and $t \longrightarrow t_2$, then $t_1 = t_2$. Its proof is simpler in abstract machine semantics thanks to its interaction with `inversion`. Indeed, the number of cases left to handle manually afterwards differs, and in structural operational semantics, it is quadratic in the number of judgments for each language construct. In contrast, with abstract machine semanitcs, the proof size is constant: basic automation solves all cases. In a more complex setting, we believe that the remaining cases will involve language features where determinism is actually difficult to prove.

In Section 3.1.1, we explained that a partial fix to the problems with the `inversion` is to add extra premises to the structural operational semantics contextual reduction rules specifying that the sub-terms are not values. This partial fix also improves complexity of the determinism proof: the number of cases to handle manually is linear, one for each contextual reduction.

Our second case study is the proof of progress; in structural operational semantics it states that if term t has type τ (written $\vdash t : \tau$), then either there exists t' such that $t \longrightarrow t'$, or t is a value. In abstract machine semantics, it states a similar property, but it relies on the typing of states $\vdash_{\text{state}} s : \tau$ and control stack $\vdash_{\text{control}} \kappa : \tau_1 \Rightarrow \tau_2$ (a trivial extension of term typing).

For structural operational semantics, we first perform induction on typing. Then, `constructor` handles cases where reduction occurs, and syntactic checks identify values. After basic automation, several cases remain, which are inverted and solved using `constructor` or manually (for a few of them). The number of cases scales linearly with the number of constructors, but the proof is brittle: changes to the order of constructors or a missing case analysis can break the proof automation, leading to expensive manual intervention.

For abstract machine semantics, we start by a case analysis on states. For $\mathcal{S}(t, \kappa, \sigma)$, we further perform case analysis on t to handle the different constructors. For $\mathcal{C}(v, \kappa)$, we examine the shapes of v and the top control unit in κ. We then invert the typing judgments, which does not introduce inconsistent goals thanks to the abstract machine semantics. Most of the proof cases left are solved using basic automation, except for a case corresponding to the handling of variables that requires a little extra effort. Overall, the proof is a lot simpler in abstract machine semantics. Moreover, if we forget to perform case analysis (*e.g.*, on an if condition), additional goals are added without introducing inconsistencies contrary to the structural operational semantics proof, as the application of the `constructor` tactic will fail instead of choosing the wrong constructor to instantiate. Hence, using abstract machine semantics is more robust with respect to proof maintenance and repair.

So, maintaining typing definitions and theorems requires effort linear to the number of language constructs. However, abstract machine semantics is more robust: while structural operational semantics can break easily with changes to the order of constructors or missing inductions, abstract machine semantics manages changes better through its handling of control units, supported by custom tactics. This makes these semantics more resilient and easier to maintain when proving progress.

3.2.2 Correctness of a Peep-hole Optimization. Our third case study is the correctness of an optimization (denoted $[\![\cdot]\!]$) that is trickier to prove, as illustrated by several attempts to conduct the proof while reusing already solved sub-proofs. It simplifies nested `if` commands:

$$[\![\texttt{if (if } b \texttt{ then false else true) then } t_{\text{then}} \texttt{ else } t_{\text{else}}]\!] = \texttt{if } b \texttt{ then } t_{\text{else}} \texttt{ else } t_{\text{then}}$$

We first try to prove the semantics preservation of this optimization pass with a standard 1-1 simulation diagram, but relying on an invariant $\sim$ of our own. The proof reveals that this simulation does not always hold (multiple steps are needed to reduce the original nested `if` *vs.* one step after the optimization), so we generalize the diagram into this 1-n simulation (if $t_1 \longrightarrow t_2$ and $t_1 \sim [\![t_1]\!]$, then there exists $[\![t_2]\!]$ such that $t_2 \sim [\![t_2]\!]$ and $[\![t_1]\!] \longrightarrow^* [\![t_2]\!]$) and make progress in our proof while reusing some proofs from the previous attempt. However, this diagram still does not hold, because the optimization only applies to nested `if` patterns from the initial program (t_1), and not to nested `if` patterns introduced by reduction steps during evaluation (t_2).

Our final solution is to generalize the invariant together with the simulation, leading to trying to prove the following and final simulation diagram: if $t_1 \longrightarrow t_2$ and $t_1 \sim t_1'$, then there exists t_3 and t_3' such that $t_3 \sim t_3'$, $t_2 \longrightarrow^* t_3$ and $t_1' \longrightarrow^* t_3'$. This diagram is iteratively discovered through careful examination of all cases, which may add new cases in the invariant. Efficiency is achieved when existing proofs are reused and straightforward cases are automated, enabling complex scenarios to be given proper attention. The ease with which these proofs are constructed is significantly impacted by the semantics.

The proof of this simulation with structural operational semantics can be done with two alternative strategies. First, an induction on $\sim$ gives very good induction hypotheses, both strong and precise, but this is because we hand-crafted $\sim$ with the relevant information for the proof. Overall, this proof strategy is sound and straightforward, although each of the cases still needs to be manually solved, and the issues raised in Section 3.1 still apply. However, this strategy only works if you have cracked the correct $\sim$ beforehand, and therefore is not advised when starting a verified compilation endeavor. Furthermore, it is impractical for realistic applications because $\sim$ is not always an inductive predicate.

The second strategy is to start by an induction on $\longrightarrow$. It is standard but cumbersome, as it also involves the previous issues of the structural operational semantics. Indeed, the induction principle

only gives the induction hypothesis for sub-reductions of depth one, while the rule Trans-If manipulates sub-reductions of depth two. Still, it is possible to make the proof work in this setting, but it requires clever thinking. Indeed, the proof case showing a sub-reduction of depth two involves the optimised nested `if` pattern; we need to know how the nested `if` b `then false else true` reduces to make progress in the proof. Fortunately here, we do not need the full power of the recursive hypothesis to know how this expression reduces (either to `true`, or to `false`, or b takes a step inside); it thus suffices to state and prove the right lemma about this sub-reduction to complete the proof. Hence, using this proof strategy requires deep analysis of the problem at hand and is not solvable with a systematic methodology. Alternatively, one could define a specialized induction principle of depth two for the reduction. However, defining a new induction principle for the needs of each proof does not scale. It is precisely these proof engineering inefficiencies that the use of abstract machine semantics addresses.

Abstract machine semantics enables a more systematic yet flexible proof strategy to show the simulation diagram by performing a well-founded induction on the control stack. This generalizes both previous approaches, offering significant advantages over induction on the semantics (the induction hypothesis is not limited to sub-reductions of depth one), while maintaining more flexibility than invariant-based induction (you do not have to have cracked the correct $\sim$ before starting the proof).

To go on with the proof in abstract machine semantics, we must first adapt the invariant $\sim$ from terms to semantic states, as we did for typing proofs. Then, we start the proof with a well-founded induction on the control stack, and invert both the invariant and the semantics. Applying this strategy leads to a quadratic number of cases: we must consider each possible rule combined with each possible invariant. Moreover, the semantics picks frames from the top of the control stack while the invariant picks frames from the bottom of the stack, which makes reconciliation difficult in each proof case. But this difficulty can be dealt with systematically with the following lemma about reduction refining the progress theorem:

Lemma 3.3. *Suppose that $s_1 +\!\!+ \kappa \rightsquigarrow s_2$ and s_1 is well-typed. Then either there is some v such that $s_1 = C(v, [])$, or there is some s_3 such that $s_1 \rightsquigarrow s_3$ and $s_2 = s_3 +\!\!+ \kappa$.*

With this lemma, the effects of the reduction can be propagated through the control stack stack, helping the reconciliation with the invariant on each proof case.

Then, we can automate most of the reasoning steps of this proof for all the quadratic cases, with a tactic that pattern matches on the current goal to select the right hypothesis or lemma to apply. Finally, we leverage `constructor` to automatically apply reduction steps when needed. By combining the use of abstract machine semantics as well as a pattern-matching-based proof automation by tactic, we claim that our proof methodology is robust to changes in the simulation diagram, semantics or invariants, as most of the proof cases are automated, and the manual cases left to solve do not need to come up with clever lemmas like in structural operational semantics.

4 Scaling Up Proof Automation: Application to Catala and λ^δ

This section shows how the proof engineering differences between semantic styles shown so far scale up with Catala [39], a more realistic language targeted for industrial usage inside government agencies, for computer programs in charge of estimating the amount of taxes and social benefits. These programs have to follow a legal specification, and Catala has special semantic features to account for the peculiar logical structure of the law. Here, the context and details of the Catala compilation chain are irrelevant, as we focus on the core intermediate representation λ^δ of Catala and its default calculus.

4.1 The Intermediate Representation λ^δ

The syntax of our novel mechanization λ^δ of Catala's default calculus is presented in Figure 3. It is a standard λ-calculus, enriched with a *default term*. For instance, let x, y : `bool` be two variables and consider default terms $a = \langle \mid x \text{ :- } \langle 1 \rangle \rangle$, $b = \langle \mid y \text{ :- } \langle 2 \rangle \rangle$ and $c = \langle a, b \mid \texttt{true} \text{ :- } \langle 0 \rangle \rangle$. a and b do not have any exceptions (left of "$\mid$"), hence they behave as conditionals on x and y: if x is `true`, then $a = \langle 1 \rangle$, otherwise $a = \varnothing$ (nothing applies), and likewise for b. Note that the consequence of a default term must have a default type $\langle$`integer`$\rangle$, while 1 has type `integer`. Transforming 1 into a pure default term $\langle 1 \rangle$ solves this problem. However, c is a default term with two exceptions, a and b. If one of the exceptions applies (for instance, $a \neq \varnothing$), then it is returned (and $c = a$). If none apply, then $c = \langle 0 \rangle$ because the justification of the default term (left of " : - ") is `true`. If a and b apply ($x = y =$ `true`), then there is a conflict of exceptions and the computation crashes, returning $c = \circledast$.

In practice and in the Catala computer code, a, b and c are partial definitions of the value of a legal variable, that are activated by a boolean justification and related to each other by exceptional and base case relationships.

A traditional small-step semantics was presented previously for the default calculus in [39]. Its judgment is $t_1 \longrightarrow t_2$. For λ^δ, we modified the semantics in the three following ways (see the detailed rules in the Rocq development). Firstly, we extend the semantics with classic rules for integer values, arithmetic operators, optional constructors and pattern matching, and a fold operator. Secondly, we changed the computation of defaults. Consider $d = \langle \varnothing, t_2 \mid t_{\text{just}} \text{ :- } t_{\text{cons}} \rangle$, where the first exception evaluates to $\varnothing$. In the original semantics, when $t_2 \longrightarrow t_2'$, then d reduces to $\langle \varnothing, t_2' \mid t_{\text{just}} \text{ :- } t_{\text{cons}} \rangle$. Note that $\varnothing$ exceptions are kept while evaluating the subsequent ones. These extra $\varnothing$ were an unpleasant source of bookkeeping for little semantic added value, since they do not influence the final value. Hence, we decided to tweak the default evaluation which leads to d reducing to $\langle t_2 \mid t_{\text{just}} \text{ :- } t_{\text{cons}} \rangle$. Third, we added a `crash_if_empty` t operator that aborts if the given term is $\varnothing$ and is the identity otherwise, transforming type $\langle t \rangle$ into t. The last modification concerns the typing of defaults and the introduction of a pure default term, that better tracks through typing the program points where $\varnothing$ might appear and propagate.

Types	τ	::=	`bool` | `integer` | `unit`	atomic types
		|	$\tau \to \tau$ | `option` τ	function and option type
		|	$\langle \tau \rangle$	default type
Values	v	::=	$\langle v \rangle$	pure default
		|	$n \in \mathbb{N}$ | `true` | `false`	integers and booleans
		|	None | Some v	option
Term	t	::=	x | v	variable, value
		|	t op t | fold t t^* t	binary operator, fold
		|	$\lambda\, x.\ t$ | $t\ t$	λ-calculus
		|	`match` t `with` | None $\to$ t | Some $x \to t$	match
		|	None | Some t	option constructors
		|	d	default term
Default	d	::=	$\langle t^* \mid t$ `:-` $t \rangle$	default term
		|	$\langle t \rangle$	pure default term
		|	`crash_if_empty` t	crash if empty
		|	$\circledast$ | $\varnothing$	conflict and empty error term

Figure 3: Syntax of λ^δ (t^* denotes a list of terms).

$$
\begin{array}{rcl}
[\![\varnothing]\!] & := & \texttt{None} \qquad [\![\circledast]\!] := \circledast \qquad [\![\langle t \rangle]\!] := \texttt{Some}\ [\![t]\!] \\
[\![\texttt{crash\_if\_empty}\ t]\!] & := & \texttt{match}\ [\![t]\!]\ \texttt{with} \mid \texttt{None} \to \circledast \mid \texttt{Some}\ v \to v \\
[\![\langle t_1, t_2, \ldots, t_n \mid t_j\ \texttt{:-}\ t_c \rangle]\!] & := & \texttt{let}\ \text{merge\_option a b} = \texttt{match}\ a, b\ \texttt{with} \\
 & & \quad \mid \texttt{Some}\ \_, \texttt{Some}\ \_ \to \circledast \\
 & & \quad \mid \texttt{None}, \texttt{Some}\ v \mid \texttt{Some}\ v, \texttt{None} \to \texttt{Some}\ v \mid \texttt{None}, \texttt{None} \to \texttt{None} \\
 & & \texttt{in match fold merge\_option}\ [\ [\![t_1]\!], [\![t_2]\!], \ldots, [\![t_n]\!]\ ]\ \texttt{None with} \\
 & & \mid \texttt{None} \to \texttt{if}\ [\![t_j]\!]\ \texttt{then}\ [\![t_c]\!]\ \texttt{else None} \mid \texttt{Some}\ v \to \texttt{Some}\ v
\end{array}
$$

Figure 4: Selected compilation rules for eliminating the default term by using option-typed terms. For the sake of clarity, we use `let ... in` . In our mechanized development, the `let ... in` are inlined as abstractions and application.

4.2 Proof Engineering with λ^δ

We proved the classic meta-theoretical results about λ^δ, in both styles: determinism, progress, preservation. We also proved a bi-directional result about the equivalence of the reduction judgments between both semantics. Additionally, we proved in both styles the correctness of a non-trivial compilation pass on λ^δ, an optimization to eliminate default terms in favor of matching option-typed terms where the None case represents the $\varnothing$ value.

This non-trivial compilation pass is illustrated in Figure 4. The translation is straightforward for basic cases: the empty term $\varnothing$ becomes None, the conflict value $\circledast$ remains $\circledast$, and wrapped terms $\langle t \rangle$ translate to Some $[\![t]\!]$. The `crash_if_empty` t operator compiles to a pattern match that returns $\circledast$ when $[\![t]\!]$ reduces to None and preserves the value v when $[\![t]\!]$ reduces to Some v. The most complex translation handles default terms $\langle t_1, t_2, \ldots, t_n \mid t_j$ `:-` $t_c \rangle$, which use a fold operation and a merge_option function to combine multiple option values into one, returning $\circledast$ when two Some _ are found. If no exceptions are found, the justification t_j conditionally returns the conclusion t_c; otherwise the selected exception is returned. This compilation pass systematically transforms $\langle \tau \rangle$ types into `option` τ types, preserving the original typing structure while making the handling of potentially absent values explicit through standard option type operations. This improves the older compilation scheme of [39] that used try/catch exceptions in the target language.

As λ^δ is a realistic language, the number of proof cases becomes significant. To manage proofs, we developed automated proof scripts that handle most cases. For each of the main proofs, an automation script is co-developed in Ltac [19], Rocq's tactic language. This script is continually updated with the following methodology: if the current version of our automated script fails on some individual case, we can check this case, determine which hypothesis or lemma is missing to solve the goal automatically, and add it to our proof script. To illustrate this methodology, let us revisit the different parts of a proof with a proof engineering eye. For instance, for the proof of equivalence (Theorem 3.1 page 8), the induction of $t \longrightarrow t'$ followed by the inversion on $s \equiv t$ results in approximately 2760 cases. To handle this bulk of cases, we use proof automation techniques like hypothesis saturation [45] to apply induction principles as well as inversion lemmas. This eliminates most of the inconsistent goals, reducing the number of cases to 344. An example of such an inconsistent goal in this proof would be to try proving that the term reconstructed from state $\mathcal{S}(t_1, \kappa +\!\!+ [\texttt{if}(t_{\text{then}}, t_{\text{else}}, \sigma_2)], \sigma)$ is of the shape $\lambda\ x.\ t'$: because the frame at the bottom of the control stack is $\texttt{if}(t_{\text{then}}, t_{\text{else}}, \sigma_2)$, the reconstruction of the term from the state above necessarily results in a top-level conditional term, and not a λ-term. Our tactics detect such discrepancies and discard these inconsistent goals by contradiction.

From the remaining consistent goals, we step automatically to the remaining cases. This is where we take the most advantage of the automation-friendliness of abstract machine semantics. Indeed, a lot of the goals to solve look like $\exists s', s \rightsquigarrow^* s' \wedge P(s')$, where P is a predicate we want to prove and s a starting state. The proof then consists in figuring out how s reduces to be able to prove P on one of the products of its reduction. This proof pattern requires the ability to reduce the terms inside the proof goal. If our reduction judgment was encoded in Rocq like a function, we could apply it and try to simplify the resulting term, but remember that our judgments are encoded as inductive types. So we need a way to "apply" the inductive type in our proof goal; which in our setting is as simple

as the Ltac tactic `solve[constructor; eauto]`. The tactic starts by performing one reduction step in the proof goal, transposing to a meta level what an executable interpreter would do on a term. We can then chain reduction steps in the proof goal with the transitivity lemma for reduction to get full meta-interpretation. This is why we call this kind of tactic a *meta-interpreter*. We improve this automation with the ability to perform steps or full interpretations when we only have partial but sufficient information in our hypotheses. Indeed, a lot of the proofs follow this pattern. Moreover, we can run such a meta-interpreter deep inside our terms and under quantifiers to avoid introducing extra existential variables before goal simplification.

Overall, we use dozens of these meta-interpreters, each fitting in a few lines of proof script. We make sure the subgoals they generate are proved on the spot by an ad-hoc automation script using basic simplification tactics, so we do not introduce proof goals that do not have the same shape as the ones we are trying to prove. This proof pattern proves to be robust to changes, precisely because it is low-tech: instead of factorizing our proof scripts into complex tactics with a lot of arguments, we copy-paste snippets of a few lines of code as we go, changing in them the names of the lemmas and simplification steps to call depending on the goals at hand. Having the certainty that applying a constructor on a abstract machine semantics term will either correctly reduce, or not reduce at all permits us to confidently use these automation techniques without risking getting stuck in a computation. Hence, abstract machine semantics allows us to benefit almost for free from a very powerful meta-interpretation system that vastly automates semantic proofs. We believe this technique to be the killer application of abstract machine semantics, for scaling up mechanized proof developments.

Another automation technique on which we rely extensively is a smart inversion tactic. Indeed, the `inversion` tactic only works on a specific judgment, meaning that if the proof is updated, then we need to make sure the right judgments are still being inverted. This makes proof repair tedious, especially in compiler verification proofs where we want to invert all judgments depending on terms, such as reduction, invariant and typing judgments. What we would really want is a smart `inversion` tactic for inverting all inductive judgments that concern a specific term on which we have prior information (*e.g.*, t is a function application). Sadly, if we naively repeat `inversion` on all inductive hypotheses, we get a case analysis that loops because of the recursive nature of judgments in structural operational semantics. Our solution is to implement a smart `inversion` tactic using Ltac2 [43] (Rocq's improved tactic language with reflection capabilities) which greatly improves proof repair and helps us develop our proofs in an iterative way. This smart inversion only applies `inversion` when the inductive judgment features a term on which we have prior information (usually, we know the term is in a specific constructor coming from a prior case analysis). For instance it applies `inversion` on $\Gamma \vdash t_1\ t_2 : \tau$ but not on $\Gamma \vdash t : \tau$. We believe the smart inversion trick could be applied to many Rocq developments.

This automation discipline allows us to quickly deal with the trivial aspects of proofs, and dive in specifically to remaining goals. While they might be automated as well, there is a tradeoff between spending time automating them, or simply treating them manually. Empirically, we find that manually handling a dozen cases is the best way to balance this tradeoff. Nevertheless, there are some limitations to our approach. Most importantly, the derivation of induction principles resists automation (see 3.1.3): the presence of both a `list` and an `autosubst` binder in λ^{δ}'s syntax suffices to break the automatic induction principle derivation, which forces us to write some induction principles by hand.

We report proof figures on our most difficult proof, namely the equivalence between both semantics. The part that generalizes Theorem 3.1 relies on structural operational semantics and counts 606 lines of specification and 609 lines of proof. Compare this to 213 lines of specification and 219 lines of proof for the other side, that relies on abstract machine semantics. The case study of the first side helped us understand how we put our automation discipline to work and the pitfalls of traditional small-step semantics in a mechanized development.

5 Conclusion and Future Work

The shape and details of semantic judgments inside a proof assistant directly influence the proof experience for all the theorems involving the semantics. While traditional operational semantics use a recursive inductive judgment, handling it for a small-step semantics becomes tedious in a proof assistant. Specifically, automating proofs about structural operational semantics requires a significant effort in building ad-hoc proof automation frameworks, that themselves can become tricky to maintain. Instead, we make explicit in this paper the advantages of abstract machine small-step semantics, relying on the control stack to represent recursivity in the judgments. This style gives inductive judgments with no recursive premises, and a much more regular shape of inference rules. While harder to parse and compare to a pen-and-paper presentation, abstract machine semantics plays along very well with the basic tactics of proof assistants like Rocq, and allow for building cheap-yet-powerful automation that unlock scaling up the development.

If the use of abstract machine semantics in mechanized developments is not new, the rigorous and principled analysis of the consequences of the semantic style on proof automation presented in this paper opens up new research questions for future work. On a meta level, for instance using MetaCoq, checkers could be developed to test if a given reduction judgment is proof-automation-friendly, before diving into the hardships of the proofs themselves. Second, the trick of syntactically encoding semantic structures to leverage the proof assistant's unification can be ported to other judgments and generalized. Of course, using a lot of syntactic encodings will widen the gap between pen-and-paper formalizations and their mechanized encodings, which should be filled by equivalence proofs so as not to widen the trusted computing base.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Uppsala, Sweden) *(PPDP '03)*. Association for Computing Machinery, New York, NY, USA, 8–19. https://doi.org/10.1145/888251.888254

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1807–1823. https://doi.org/10.1145/3133956.3134078

[3] Abhishek Anand, Zoe Paraskevopoulou, Andrew Appel, Greg Morrisett, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2021. CertiCoq: A verified compiler for Coq. In *CoqPL*.

[4] Andrew W. Appel and Sandrine Blazy. 2007. Separation Logic for Small-Step CMinor. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4732)*, Klaus Schneider and Jens Brandt (Eds.). Springer, 5–21. https://doi.org/10.1007/978-3-540-74591-4_3

[5] Bob Atkey. 2023. Simple semantics for defaults in Catala. https://bentnib.org/posts/2023-01-16-catala.html

[6] Véronique Benzaken, Évelyne Contejean, Mohammed Houssem Hachmaoui, Chantal Keller, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. 2022. Translating Canonical SQL to Imperative Code in Coq. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 83 (April 2022), 27 pages. https://doi.org/10.1145/3527327

[7] Małgorzata Biernacka, Witold Charatonik, and Tomasz Drab. 2022. The Zoo of Lambda-Calculus Reduction Strategies, And Coq. In *13th International Conference on Interactive Theorem Proving (ITP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 237)*, June Andronick and Leonardo de Moura (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:19. https://doi.org/10.4230/LIPIcs.ITP.2022.7

[8] Sandrine Blazy. 2024. From Mechanized Semantics to Verified Compilation: the Clight Semantics of CompCert. In *Fundamental Approaches to Software Engineering - 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14573)*, Dirk Beyer and Ana Cavalcanti (Eds.). Springer, 1–21. https://doi.org/10.1007/978-3-031-57259-3_1

[9] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288. https://doi.org/10.1007/s10817-009-9148-3

[10] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A trusted mechanised JavaScript specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 87–100.

[11] Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. 2018. Skeletal Semantics and their Interpretations. *Proceedings of the ACM on Programming Languages* 3 (2018), 1 – 31. https://api.semanticscholar.org/CorpusID:52846325

[12] Olivier Boite and Catherine Dubois. 2001. Proving type soundness of a simply typed ML-like language with references. *Supplemental Proceedings of TPHOL* 1 (2001), 69–84.

[13] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A Formally Verified Compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 586–601. https://doi.org/10.1145/3062341.3062358

[14] Timothy Bourke, Basile Pesin, and Marc Pouzet. 2023. Verified Compilation of Synchronous Dataflow with State Machines. *ACM Transactions on Embedded Computing Systems* 22, 5s (Sept. 2023), 137:1–137:26. https://doi.org/10.1145/3608102 ESWEEK special issue including presentations at the 23rd Int. Conf. on Embedded Software (EMSOFT 2023).

[15] Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2023. Omnisemantics: Smooth handling of nondeterminism. *ACM Transactions on Programming Languages and Systems* 45, 1 (2023), 1–43.

[16] Arthur Correnson and Dominic Steinhöfel. 2023. Engineering a Formally Verified Automated Bug Finder. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1165–1176. https://doi.org/10.1145/3611643.3616290

[17] Nathanaëlle Courant, Antoine Séré, and Natarajan Shankar. 2020. The correctness of a code generator for a functional language. In *Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16–21, 2020, Proceedings 21*. Springer, 68–89.

[18] Guy Cousineau, P-L Curien, and Michel Mauny. 1987. The categorical abstract machine. *Science of computer programming* 8, 2 (1987), 173–202.

[19] David Delahaye. 2000. A Tactic Language for the System Coq. In *Logic for Programming and Automated Reasoning: 7th International Conference, LPAR 2000 Reunion Island, France, November 6–10, 2000 Proceedings 7*. Springer, 85–95.

[20] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. *Acm Sigplan Notices* 50, 1 (2015), 689–700.

[21] Alain Delaët, Sandrine Blazy, and Denis Merigoux. 2025. *Artifact for "Abstract Machines and Small-step Semantics: a Winning Ticket for Proof Automation?"*. https://doi.org/10.5281/zenodo.15583348

[22] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2020. Simple High-Level Code for Cryptographic Arithmetic: With Proofs, Without Compromises. *ACM SIGOPS Operating Systems Review* 54, 1 (2020), 23–30.

[23] Matthias Felleisen and Daniel P. Friedman. 1986. *Control Operators, the SECD Machine, and the λ-Calculus*. Technical Report 197. Department of Computer Science, Indiana University.

[24] Pierre Goutagny, Aymeric Fromherz, and Raphaël Monat. 2025. CUTECat: Concolic Execution for Computational Law. In *European Symposium on Programming*. Springer, 31–61.

[25] Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. 2023. Practical and Sound Equality Tests, Automatically: Deriving eqType Instances for Jasmin's Data Types with Coq-Elpi. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, Boston MA USA, 167–181. https://doi.org/10.1145/3573105.3575683

[26] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. 2022. Noise: A Library of Verified High-Performance Secure Channel Protocol Implementations. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 107–124.

[27] Jacques-Henri Jourdan. 2016. *Verasco: a Formally Verified C Static Analyzer*. Ph. D. Dissertation. Universite Paris Diderot-Paris VII.

[28] G. Kahn. 1987. Natural semantics. In *STACS 87*, Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.). Springer, Berlin, Heidelberg, 22–39.

[29] Adam Khayam and Alan Schmitt. 2022. A Practical Approach for Describing Language Semantics. *Submitted for publication* (2022).

[30] Jean-Louis Krivine. 2007. A call-by-name lambda-calculus machine. *Higher-order and symbolic computation* 20 (2007), 199–207.

[31] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*. ACM Press, 179–191. https://doi.org/10.1145/2535838.2535841

[32] P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (01 1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308 arXiv:https://academic.oup.com/comjnl/article-pdf/6/4/308/1067901/6-4-308.pdf

[33] Sarah B. Lawsky. 2018. A Logic for Statutes. *Florida Tax Review* (2018).

[34] Xavier Leroy. 1990. *The ZINC experiment: an economical implementation of the ML language*. Ph. D. Dissertation. INRIA.

[35] John M. Li and Andrew W. Appel. 2021. Deriving Efficient Program Transformations From Rewrite Rules. *Proc. ACM Program. Lang.* 5, ICFP, Article 74 (Aug. 2021), 29 pages. https://doi.org/10.1145/3473579

[36] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert NM Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. *ACM SIGPLAN Notices* 51, 6 (2016), 1–15.

[37] Denis Merigoux. 2021. *Proof-Oriented Domain-Specific Language Design for High-Assurance Software*. Theses. Université Paris sciences et lettres. https://tel.archives-ouvertes.fr/tel-03622012

[38] Denis Merigoux. 2023. Experience Report: Implementing a Real-World, Medium-Sized Program Derived from a Legislative Specification. In *Programming Languages and the Law 2023 (affiliated with POPL)*. Boston (MA), United States. https://inria.hal.science/hal-03933574

[39] Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. 2021. Catala: A Programming Language for the Law. *Proc. ACM Program. Lang.* 5, ICFP, Article 77 (Aug. 2021), 29 pages. https://doi.org/10.1145/3473582

[40] Denis Merigoux, Raphaël Monat, and Jonathan Protzenko. 2021. A Modern Compiler for the French Tax Code. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/3446804.3446850

[41] Tobias Nipkow and Gerwin Klein. 2016. Concrete Semantics. *with Isabelle/HOL* 29 (2016).

[42] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *European Symposium on Programming*. https://api.semanticscholar.org/CorpusID:4864472

[43] Pierre-Marie Pédrot. 2019. Ltac2: Tactical Warfare. In *The Fifth International Workshop on Coq for Programming Languages, CoqPL*, Vol. 2019.

[44] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2010. *Software Foundations*. Webpage:http://www.cis.upenn.edu/bcpierce/sf/current/index.html

[45] Clément Pit-Claudel. 2016. *Compilation Using Correct-by-Construction Program Synthesis*. Master's thesis. Massachusetts Institute of Technology. https://doi.org/1721.1/107293

[46] Gordon Plotkin. 2004. A Structural Approach to Operational Semantics. *J. Log. Algebr. Program.* 60-61 (07 2004), 17–139. https://doi.org/10.1016/j.jlap.2004.05.001

[47] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 151 (June 2023), 25 pages. https://doi.org/10.1145/3591265

[48] Talia Ringer. 2021. *Proof Repair*. Ph. D. Dissertation. University of Washington.

[49] Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. 2020. REPLica: REPL Instrumentation for Coq Analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA)

(CPP 2020). Association for Computing Machinery, New York, NY, USA, 99–113. https://doi.org/10.1145/3372885.3373823

[50] Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 624–641.

[51] Wojciech Krzysztof Rozowski. 2021. *Formally verified derivation of an executable and terminating CEK machine from call-by-value lambda-p-calculus.* Technical Report. University of Southampton.

[52] Grigore Roșu and Traian Florin Șerbănuță. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012 Membrane computing and programming.

[53] Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015 (LNAI)*, Xingyuan Zhang and Christian Urban (Eds.). Springer-Verlag.

[54] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *Journal of Automated Reasoning* 64, 5 (01 Jun 2020), 947–999. https://doi.org/10.1007/s10817-019-09540-0

[55] Conrad Watt. 2018. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 53–65. https://doi.org/10.1145/3167082

[56] Conrad Watt. 2021. *Mechanising and evolving the formal semantics of WebAssembly: the Web's new low-level language.* Ph. D. Dissertation.

[57] A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (Nov. 1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Received 3 June 2025