Incremental Certified Programming

TOMÁS DÍAZ, University of Chile, Chile
KENJI MAILLARD, Inria, France
NICOLAS TABAREAU, Inria, France
ÉRIC TANTER, University of Chile, Chile

Certified programming, as carried out in proof assistants and dependently-typed programming languages, ensures that a software meets its requirements by supporting the definition of both specifications and proofs. However, proofs easily break with partial definitions and incremental changes because specifications are not designed to account for the intermediate incomplete states of programs. We advocate for proper support for incremental certified programming by analyzing its objectives and inherent challenges, and propose a formal framework for achieving incremental certified programming in a principled manner. The key idea is to define appropriate notions of *completion refinement* and *completeness* to capture incrementality, and to systematically produce specifications that are valid at every stage of development while preserving the intent of the original statements. We provide a prototype implementation in the Rocq Prover, called IncRease, which exploits typeclasses for automation and extensibility, and is independent of any specific mechanism used to handle incompleteness. We illustrate its use with both an incremental textbook formalization of the simply-typed λ-calculus, and a more complex case study of incremental certified programming for an existing dead-code elimination optimization pass of the CompCert project. We show that the approach is compatible with randomized property-based testing as provided by QuickChick. Finally we study how to combine incremental certified programming with deductive synthesis, using a novel incrementality-friendly adaptation of the Fiat library. This work provides theoretical and practical foundations towards systematic support for incremental certified programming, highlighting challenges and perspectives for future developments.

CCS Concepts: • **Theory of computation** → **Type theory**; **Program reasoning**.

Additional Key Words and Phrases: incremental programming, program verification, theorem proving

ACM Reference Format:
Tomás Díaz, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. 2025. Incremental Certified Programming. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 290 (October 2025), 28 pages. https://doi.org/10.1145/3763068

1 Introduction

Every software development is *incremental* in practice; from basic definitions, one progressively refines a program until it satisfies its intended requirements. A common approach in this process is to begin by partially defining functions, leaving some parts unimplemented until later stages. This partiality can be managed for instance by using default values as placeholders, raising "not implemented" exceptions or using a partiality monad. This practice is so common that most programming languages provide built-in support for such exceptions (*e.g.*, Python, Scala, Java Apache Common, C#) and/or placeholders (*e.g.*, Haskell, Scala, Rust).

Authors' Contact Information: Tomás Díaz, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Santiago, Chile, tdiaz@dcc.uchile.cl; Kenji Maillard, Inria, Nantes, France, kenji.maillard@inria.fr; Nicolas Tabareau, Inria, Nantes, France, nicolas.tabareau@inria.fr; Éric Tanter, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Santiago, Chile, etanter@dcc.uchile.cl.

ACM 2475-1421/2025/10-ART290
https://doi.org/10.1145/3763068

It is then natural to apply incrementality to *certified programming* [Chlipala 2013], where one not only writes programs but also specifications and proofs that the programs meet these specifications. Some of the most notable successful cases of mechanized certification include CompCert [Leroy 2009]'s verified C compiler developed with the Rocq Prover [The Rocq Development Team 2024], or the seL4 microkernel formalization [Klein et al. 2009], developed in Isabelle/HOL [Nipkow et al. 2002]. The large scale of these formalization efforts necessarily requires good software engineering practices. In particular, in large certification projects with many complex features and subsystems to formalize, programmers should be able to focus on specific fragments to implement, partially defining their programs and proving the specifications only for the implemented parts of the system. Each increment in implementation can then be accompanied by an increment in the proof. For example, to produce a realistic formalization of the Javascript specification [Bodin et al. 2014] and the R language [Bodin et al. 2018], the implementations use a monad for incomplete code, such that some parts can be progressively implemented.

Unfortunately, a program specification is usually expressed for its complete version and is thus not satisfied by incomplete versions. For example, consider the behavior-preserving specification of an optimization function: `forall p, eval (optimize p) = eval p`. This specification is evidently false for programs where the `optimize` function raises a "not implemented" exception. Therefore, a key challenge is to express specifications that are meaningful at every stage of the incremental development; being valid for both complete and incomplete code. Note that the *interactive* nature of theorem proving is related, as it allows one to work with incomplete terms, such as syntactic "holes" in Agda [Bove et al. 2009] and Idris 2 [Brady 2021] or existential variables in Rocq and Lean [Moura and Ullrich 2021], which represent unknown subterms. However, these mechanisms do not support incremental certification per se: in Rocq and Lean, existential variables must be discharged before a definition is accepted; in Agda and Idris, the syntactic holes later appear as existential variables in the proof cases related to incomplete code, rendering specifications invalid just as with exceptions.

Software engineering considerations related to certified programming are gaining traction as the adoption of proof assistants and dependently-typed programming languages grows [Ringer et al. 2019a], with proposals such as proof generation and proof reuse in the face of code changes [Boite 2004; First et al. 2023; Ringer et al. 2021], proof automation and practices for automation [Agrawal et al. 2023; Czajka and Kaliszyk 2018], proof maintenance and planning for changes [Woos et al. 2016], among others. To the best of our knowledge, there is no work that addresses the complementary aspect of providing sound and stable reasoning about incomplete programs as they are incrementally developed, which we call Incremental Certified Programming (ICP for short).

In this context, the contributions of this work are:

- We analyze the principles, objectives, and challenges of ICP, reviewing existing approaches to partial definitions with a textbook formalization of the simply-typed λ-calculus (§2).
- We present a formal framework for ICP that relies on the definition of a *completion refinement*[1] relation as well as an associated notion of program *completeness* in order to account for incrementality (§3). We characterize proper *incremental reformulations* of logical statements that support ICP, and provide a systematic reformulation procedure of a fragment of logical predicates.
- We describe IncRease (§ 4), a Rocq prototype of the formal ICP framework, which exploits typeclasses for extensibility and automation in generating incremental reformulations, and is

[1]The notion of *refinement* has a long history in programming, starting with the idea of the stepwise derivation of programs from specifications [Wirth 1971]. In general, refinement denotes some kind of "improvement" between programs. For instance in compiler verification, one often uses notions of refinement that can refer to reducing the amount of non-determinism, or the possibilities of going wrong [Dockins 2012; Leroy 2009]. Completion refinement considers program improvement in the sense of completing partial implementations; note that an incomplete program has a well-defined and deterministic behavior in that it can produce a result that denotes incompletion.

independent of the specific mechanism chosen for dealing with incompleteness. We describe instantiations of IncRease using different incompleteness mechanisms (§ 5), namely monads, and direct-style exceptions [Pédrot and Tabareau 2018].

- We study the potential of ICP on the certified programming ecosystem with three case studies that apply IncRease with third-party projects: the incremental certification of a dead-code elimination pass of the CompCert compiler [Leroy 2009] (§ 6); the integration with a mainstream library for randomized property-based testing, QuickChick [Dénès et al. 2014], applied to the running example (§ 7); finally, we describe how to make deductive synthesis compatible with ICP using a novel incrementality-friendly adaptation of the Fiat library [Delaware et al. 2015] (§ 8).

We discuss the broader research agenda for ICP in § 9, based on the current limitations of IncRease and the observations drawn from the case studies, related work in § 10 and conclude in § 11.

Implementation. The implementation of IncRease (in Rocq 8.20.0) and examples are provided as supplementary material [Díaz et al. 2025]. Note that the instantiation of IncRease with exceptions currently relies on the Exceptional Type Theory (ExTT) of Pédrot and Tabareau [2018], which is logically inconsistent because any statement can be proven by raising an exception.[2] As such, the prototype only currently serves as a proof-of-concept for experimentation, but would directly benefit from advances in robust and sound implementations of exceptional type theories.

2 Challenges of Incremental Certified Programming

We start by introducing a running example (§ 2.1) used to illustrate various approaches to program incrementally in proof assistants (§ 2.2) and to reason about incomplete code (§ 2.3). We then summarize the main challenges to support Incremental Certified Programming (ICP).

2.1 Running Example: STLC

As a running example, we consider an incremental Rocq formalization of the simply-typed λ-calculus (STLC) with boolean constants and conditionals, summarized in Fig. 1, and based on the Software Foundations textbook [Pierce et al. 2015].

The formalization includes inductive definitions for terms (`tm`) and types (`ty`), a typechecker (`has_type`), an evaluation function (`eval`), and a `safety` statement. We work in a scenario where `has_type` is completely defined, while `eval` can be incomplete. The `eval` function is defined in big-step style, recursively evaluating each term to a value, *i.e.,* variables (`tm_var`), booleans (`tm_true` and `tm_false`) and functions (`tm_abs`). On values, `eval` behaves as the identity. Meanwhile, `eval` evaluates the adequate branch of a conditional (`tm_if`) when the condition evaluates to a boolean value, and otherwise returns the conditional term. We use the symbol **?** to represent unimplemented code for the branch that handles the application case (`tm_app`).

The specification is the `safety` theorem, which states that a well-typed term in the empty environment `eval`uates to a well-typed term of the same type.

[2] More precisely, ExTT satisfies a *weak* form of logical consistency: any type (including `False`) can be inhabited by raising an exception, but all fake proofs at positive types can be discriminated by pattern matching—as opposed to axioms, which cannot be discriminated. Sound logical reasoning about exceptional programs can be recovered using separate universe hierarchies for programming and for proving, as proposed in RETT [Pédrot et al. 2019]. The CoqRETT implementation of RETT follows a translation-based approach [Boulier et al. 2017], which is inherently unable to *impose* any such restriction on the extended type theory. Recent work on sort polymorphism [Poiret et al. 2025] offers a more promising way to enforce this stratification within the framework of the Rocq Prover.

```
Inductive tm : Type :=
| tm_var : string -> tm
| tm_app : tm -> tm -> tm
| tm_abs : string -> ty -> tm -> tm
| tm_true : tm
| tm_false : tm
| tm_if : tm -> tm -> tm -> tm.

Definition env := list (string * ty).
Definition empty : env = [].
Definition has_type : env -> tm -> ty -> bool.

Theorem safety : forall (t : tm) (T : ty),
    has_type empty t T = true ->
    has_type empty (eval t) T = true.
```

```
Inductive ty : Type :=
| TyBool : ty
| TyArrow : ty -> ty -> ty.

Fixpoint eval (t : tm) : tm :=
  match t with
  | tm_app t1 t2 => ?  (* incomplete *)
  | tm_if c tb fb =>
      match eval c with
      | tm_true => eval tb
      | tm_false => eval fb
      | _ as c' => tm_if c' tb fb
      end
  | _ => t
  end.
```

Fig. 1. Incremental formalization of the STLC, used to discuss the incremental development of eval and its specification, safety. Incomplete code is represented by the symbol ?. has_type is complete and omitted.

2.2 Approaches to Incomplete Code

We now illustrate several common approaches to implement the incomplete eval function: default values, axioms, inductive relations, monads, monad algebra, and exceptions. We will consider their impact on the definition and proof of the safety statement in the following subsection.

Default values. A straightforward approach to defining incomplete code is to ask for a default value to use as a placeholder:

```
  Fixpoint dflt_eval (dflt : tm) (t : tm) : tm := match t with | tm_app _ _ => dflt | ... end.
```

This implementation of the evaluator expects an additional argument dflt of type tm to use as default when a case is not yet implemented. Alternatively, the default could be a variable in scope. Regardless, using an existing term such as tm_false as default can be confusing: if the evaluation returns tm_false, we cannot distinguish between the execution of the incomplete branch and a normal execution that happens to yield tm_false.

Axioms. Another common approach is to use axioms as placeholders for incomplete code. For instance, one can postulate an axiom of type tm:

```
  Axiom dflt : tm.
  Fixpoint ax_eval (t : tm) : tm := match t with | tm_app _ _ => dflt | ... end.
```

Using such an axiom avoids both changing the signature and the confusion induced by non-axiomatic default values. However, axioms introduce stuck terms because they have no computational content: matching on an axiom is itself a stuck term, which can inhabit any type in the system, and cannot be discriminated. In other words, canonicity of the type theory is broken.

Inductive relations. Using inductive relations is another standard approach to implementing partial definitions in proof assistants: it suffices to not add constructors to the inductive definitions until future increments. For instance, one could define eval as an inductive relation instead of as a function, without providing any constructor for the application case:

```
Inductive i_eval : tm -> tm -> Set :=
| i_eval_tm_if_true : forall (c tb tb' fb : tm), i_eval c tm_true -> i_eval tb tb' ->
  i_eval (tm_if c tb fb) tb'
| i_eval_tm_if_false : forall (c tb tb' fb : tm), i_eval c tm_false -> i_eval fb fb' ->
  i_eval (tm_if c tb fb) fb'
| ... (* no constructor for tm_app *)
```

Note that the absence of constructors for specific cases now has potentially different meanings: it can either mean that the case is intended to be covered in a later increment (as in the tm_app case), or that the case is outside the domain of the partial relation, or that the case is accidentally missing (*e.g.,* the programmer could have forgotten the i_eval_tm_if_false case).

Monads. A favored approach in functional programming languages is to use monads, such as option (or any similar monad), to define partial functions. For instance, one can define a result monad to denote unimplemented code with a dedicated constructor:

```
Inductive result (A : Type) :=
| Success : A -> result A
| NotImplemented : result A.
```

Adopting the result monad means that eval must be rewritten in monadic style, for instance (using the bind notation from the coq-ext-lib library):

```
Fixpoint res_eval (t : tm) : result tm := match t with
  | tm_app _ _ => NotImplemented
  | tm_if c tb fb => c' <- res_eval c ;; match c' with | tm_true => res_eval tb
                                                       | tm_false => res_eval fb
                                                       | _ => ret (tm_if c' tb fb) end
  | _ => ret t end.
```

Monads can effectively discriminate incomplete code, while still preserving the desired computational error-propagating behavior through the bind operator. However, clients of the evaluator must be adjusted to its new return type.

Inlined partiality. Instead of using a dedicated monad like result to handle incompleteness, it is possible to be more local by inlining partiality directly into the inductive type using a dedicated constructor. For instance, the inductive type tm could be extended with an additional tm_not_implemented constructor:

```
Inductive tm : Set := ...
| tm_if : tm -> tm -> tm -> tm
| tm_not_implemented : tm.  (* new constructor *)
```

This encoding preserves the original definition and signature of eval, but any function in the system that matches on terms must now defensively account for this new constructor.

Exceptions. As an alternative to both the monadic and axiomatic approaches, Pédrot and Tabareau [2018] have proposed an exceptional type theory, with exceptions similar to those found in mainstream programming languages:

```
Fixpoint exc_eval (t : tm) : tm := match t with | tm_app _ _ => raise tm | ... end.
```

Exceptions essentially manifest as a novel constructor in each and every inductive type. As in the axiomatic approach, using exceptions leaves the signature and definition of the incomplete function otherwise intact. As in the monadic approach, an exception can be discriminated using a new match-like form (**catch**). Notably, in other elimination forms, such as a **match** or an application, an exception automatically propagates, therefore existing code can be oblivious to exceptions.

2.3 Incremental Reasoning about Incomplete Code

While there are many ways to express incomplete code, *reasoning* about it lacks proper support, regardless of the mechanism used. We illustrate this issue with the running example, by considering the safety statement (Fig. 1) and its proof, as eval is incrementally developed.

Theorems are (usually) not compatible with incompleteness. Like most interesting formal statements, safety involves establishing equalities. Unsurprisingly, such equalities do not hold for incomplete code. For instance, among the different cases to consider with default values, one needs to show that `dflt` has the same type as `tm_app t1 t2`:

```
has_type empty (tm_app t1 t2) T = true -> has_type empty dflt T = true
```

for *any* type `T`, which is not possible in the STLC definition we are considering.

Likewise, with axioms, consider the term `tm_if (tm_app t1 t2) tb fb`, then the goal one has to prove is the following:

```
has_type empty (ax_eval (tm_if (tm_app t1 t2) tb fb)) T = true
```

which, after evaluating `ax_eval`, becomes:

```
has_type empty (match dflt with ... end) T = true
```

Because `dflt` can be any term of type `tm`, there is no hope of proving the goal, since this would basically amount to proving that every term is well-typed.

The problem also manifests with monads. First, the `safety` theorem needs to be reformulated to account for the monadic style:

```
Theorem res_safety : forall (t : tm) (T : ty), has_type empty t T = true ->
    (t' <- res_eval t ;; ret (has_type empty t' T)) = Success true.
```

When the term `t` is an application, `res_eval` evaluates to `NotImplemented`, which propagates through the bind operator, yielding the contradictory goal `NotImplemented = Success true`.

Option 1: admitting statements about incomplete code. When faced with such an incremental certified programming scenario, programmers usually resort to the most readily accessible solution: simply using **`admit`** to discharge impossible goals related to not-yet-implemented functionality. Doing so allows programmers to go on with proving goals related to implemented functionality. The main issue with this state-of-the-practice approach is that it introduces a *disconnect* between the code and its associated proofs. When the programmer performs an implementation increment, all the proofs are still "valid": they are accepted by the proof checker, despite the fact that the implementation increment could violate the specifications. It is the responsibility of the programmer to manually track down the dependencies of the implementation increment to determine exactly which **`admit`**s in which proofs should now be removed and properly tackled. While this might be reasonably simple to do in a small-scale development, it is highly error-prone in the large, especially if a development involves multiple incrementally-defined functions. This disconnection breaks an important property of certified programming that one normally enjoys when editing code: if the associated proofs still go through, then the edits are correct.

Option 2: reformulating the theorem. A robust alternative to admitting statements about incomplete code is to reformulate the formal statement to explicitly account for incompleteness. This alternative is applicable in general with all the approaches to incompleteness we reviewed above, except for default values—because of the potential for confusion—or axioms—because the stuck terms they yield do not satisfy any property.[3]

To illustrate with the monadic approach, one can reformulate the `res_safety` statement by considering the incomplete case explicitly, for instance as follows:

```
Theorem res_safety_inc : forall (t : tm) (T : ty), has_type empty t T = true ->
    match res_eval t with
    | NotImplemented => True
    | Success t' => has_type empty t' T = true
    end.
```

The conclusion of the statement is `True` in the `NotImplemented` case, and faithful to the original statement otherwise. Compared to the use of **`admit`**s, the reformulation of the theorem ensures

[3] *e.g.*, `match dflt with ... end = dflt` does not hold and shows up whenever the incomplete function is recursive.

that when the programmer replaces a non-implemented branch with an actual implementation, the existing proof is not silently and unsafely accepted; instead, the programmer is automatically directed by the proof assistant to the case that requires proving, if needed.[4]

Of course, reformulating a statement to account for incompleteness in such a way can become cumbersome and error-prone depending on the complexity of the original statement.

2.4 Towards Support for ICP

ICP involves two connected facets: the *programming* facet, in which one needs to be able to partially and incrementally define a program; and the *certification* facet, in which one spells out specification and carries out proofs that the incomplete program meets its specification, so far.

Like in standard certified programming, both facets should be *synchronized*, meaning that code and specification/proofs should be connected at all time; we do not consider the use of `admit`s a satisfactory solution for ICP (as explained above), and argue for systematic and robust support.

Programming. The many approaches to express incomplete definitions present different tradeoffs. Adopting a specific mechanism—such as changing type signatures, rewriting code in monadic style, modifying existing functions to handle extra constructors, etc— comes with an *opt-in cost*. Dually, there is an *opt-out cost* to revert changes related to incompleteness once a function is completely defined. Each incompleteness mechanism can additionally be assessed in terms of whether it supports the *distinguishability* of incomplete code, and the *encapsulation* it provides, *i.e.,* if incompleteness can be localized. Also, while most approaches are readily available in proof assistants, exceptions are novel and as-yet not fully supported beyond limited proof-of-concepts implementations. Overall, the choice of a specific mechanism for dealing with incompleteness is tightly coupled to specific intricacies and practical considerations related to the verification tool.

The focus and contribution of this work does *not* depend on a specific incompleteness mechanism. Instead, we propose a general framework for incremental certified programming that can be adopted with various incompleteness mechanisms.

Certification. One of the main challenges of ICP is to express specifications that are provable for incomplete versions of a program. As we have seen, this might imply *reformulating specifications* to explicitly account for incompleteness. In addition, one expects that:

- *formal reasoning is monotone:* establishing the correctness of an implementation increment should be stable in future iterations if that increment code is unchanged;
- *code and proofs are in-sync:* a specification violation introduced in an implementation increment should be automatically detected when rechecking the proofs of the prior iteration;
- *initial specifications are recoverable:* if the initial specifications were adapted to account for incrementality, it should be the case that the reformulated specification implies the initial specification when considering a fully-implemented program.

Next, we present a formal framework to address the challenges of incremental certified programming (§3). We then describe a Rocq prototype Rocq (§4) and instantiations of this framework for different incompleteness mechanisms (§5), illustrating its use through several case studies (§6 to 8).

3 A Formal Framework for Incremental Certified Programming

We now describe a formal framework to support Incremental Certified Programming (ICP). We first introduce the central notion of *completion refinement*, and characterize when a predicate is

[4]The proof could rely on some automation that turns out to work for the new case, which is perfectly fine and safe.

compatible with incrementality (§3.1). Then, we discuss the requirements of incremental reformulations of predicates (§3.2), and analyze the emblematic case of propositional equality (§3.3). Finally, we study how to systematically produce incremental reformulations (§3.4).

3.1 Completion Refinement and Incrementality

Completion refinement. The ICP framework centrally relies on the notion of *completion refinement*, or refinement for short, to relate an incomplete program with some “holes”, with any program where some of these “holes” have been implemented. For instance, in the running example (§2), implementing the `tm_app` case in the definition of `eval` yields a refinement of that program.

Definition 3.1 (Completion Refinement). Completion refinement at type A, noted $\sqsubseteq_A$, is a preorder, *i.e.,* a reflexive and transitive relation on A. *We leave type A implicit in the notation, and write simply $a_1 \sqsubseteq a_2$ to denote that term a_1 refines term a_2, both of type A determined from context.*

This general definition can be instantiated for each specific incompleteness mechanism used to denote “holes”, such as monads, relations, or exceptions. In an exceptional type theory, where incompleteness is denoted by an exceptional term that inhabits each type A, noted $\bullet_A$ below, the refinement relation should specify that the exception is the least refined term at that type, and that refinement is structurally defined over constructors such as `tm_if` (we omit the type index of $\bullet$):

$$\sqsubseteq\text{-INCOMPLETE}\quad \frac{}{a \sqsubseteq \bullet} \qquad\qquad \sqsubseteq\text{-IF}\quad \frac{a_1 \sqsubseteq a_1' \qquad a_2 \sqsubseteq a_2' \qquad a_3 \sqsubseteq a_3'}{\texttt{tm\_if}\ a_1\ a_2\ a_3 \sqsubseteq \texttt{tm\_if}\ a_1'\ a_2'\ a_3'}$$

Likewise, if using the `result` monad to handle incompleteness (§2.2), then `NotImplemented` is the least refined term, and two `Successes` are in the refinement relation if their contents are:

$$\sqsubseteq\text{-NotImplemented}\quad \frac{}{a \sqsubseteq \texttt{NotImplemented}} \qquad\qquad \sqsubseteq\text{-Success}\quad \frac{a \sqsubseteq a'}{\texttt{Success}\ a \sqsubseteq \texttt{Success}\ a'}$$

In contrast to the exceptional approach, not all types admit holes, only monadic types do.

The notion of refinement for functions is defined pointwise, in an extensional manner, and is independent of the specific incompleteness mechanism involved:

Definition 3.2 (Refinement for functions). $f \sqsubseteq g$ when for any term a, we have $f\ a \sqsubseteq g\ a$.

Completeness. The second key notion for ICP is *program completeness*, which characterizes the end point of the incremental development process. Intuitively, we say that a program is *complete* when it is fully implemented. Similar to refinement, completeness depends on the mechanism used to define incomplete code. For instance, with the `result` monad, a `Success` term is complete if its content is also complete, `NotImplemented` is not complete, terms of type `tm` are always complete, and other type formers may also require recursively checking completeness of their elements. Completeness for functions is semantic preservation of completeness, mapping complete inputs to complete outputs, independent of the incompleteness mechanism. We illustrate the monadic example and the case of functions below, with a type-indexed judgment $\texttt{complete}_\texttt{T}$:

$$\text{Complete-Success}\quad \frac{\texttt{complete}_\texttt{A}\ a}{\texttt{complete}_{\texttt{result A}}\ (\texttt{Success}\ a)} \qquad \text{Complete-tm}\quad \frac{}{\texttt{complete}_\texttt{tm}\ t} \qquad \text{Complete-Fun}\quad \frac{\forall a, \texttt{complete}_\texttt{A}\ a \implies \texttt{complete}_\texttt{B}\ (f\ a)}{\texttt{complete}_{\texttt{A}\to\texttt{B}}\ f}$$

The natural connection between refinement and equality is that completeness is minimality with respect to refinement. We call a type that satisfies this property a *complete-minimal* type:

Definition 3.3 (Complete-minimal types)**.** A type A is complete-minimal when for any complete term $a : A$, we have $a' \sqsubseteq a \implies a' = a$, for any a'.

This is not a universal property because of the extensional behavior of functions. Consider the two functions $f_1 := \lambda\, x : \texttt{unit}.\, x$ and $f_2 := \lambda\, x : \texttt{unit}.\, \texttt{tt}$, in the exceptional setting. These functions constitute a counterexample because: (1) both functions are complete, in that they are literally fully implemented; (2) we have $f_2 \sqsubseteq f_1$ because f_2 always returns the most refined term `tt` (Def. 3.2); (3) the functions are *not* extensionally equal because $f_1 \bullet = \bullet$ and $f_2 \bullet = \texttt{tt}$, and $\bullet \neq \texttt{tt}$.

Likewise, in the monadic setting, with the functions adjusted to be of type `result unit` $\rightarrow$ `result unit` ($f_1 := \lambda\, x : \texttt{result unit}.\, x$ and $f2 := \lambda\, x : \texttt{result unit}.\texttt{Success tt}$), we would have f_1 `NotImplemented` = `NotImplemented` and f_2 `NotImplemented` = `Success tt`. Therefore, while base types are complete-minimal, function types are not.

Incrementality. To achieve a smooth incremental process, one expects predicates to remain provable along every step of development. We call a predicate satisfying such a notion as being *incremental*, and call a non-incremental one a *tight* predicate.

Definition 3.4 (Incrementality)**.** A predicate P is *incremental* when for any two terms a_1 and a_2 such that $a_1 \sqsubseteq a_2$, we have $P\ a_1 \implies P\ a_2$.

Incrementality is crucial because it ensures that if a predicate is not provable for an incomplete definition, then the problem does not lie in the unimplemented part of the code but rather in the parts that are already implemented (or in the specification itself). Programmers can direct their efforts towards fixing the existing code or specification rather than refining the incomplete code. Incrementality corresponds with the expectation that formal reasoning should be monotone (§ 2.4).

3.2 Incremental Reformulations

While incrementality is a desirable property for ICP, propositional equality is not incremental in general. Given its pervasiveness in specifications, it is the principal source of tight predicates in any development. For example, the `safety` theorem (§ 2.1) cannot be proven in its original form using any of the incompleteness approaches discussed earlier, precisely because it uses propositional equality. Fortunately, as we have seen in §2.3, it is usually possible to design alternative incremental formulations of a predicate, such as the `res_safety_inc` statement. The question is how to characterize such reformulations: given a tight predicate P, what properties should a predicate Q satisfy to be a proper incremental reformulation of P? Building on the informal criteria for evaluating reformulations outlined in § 2.4, we now formalize their key properties.

Approximation and Recoverability. Any reformulation Q of a predicate P should satisfy two key properties, beyond being incremental. The first is that whenever P holds, Q should also hold. Indeed, the new predicate Q should only allow us to extend the proof to deal with the incomplete parts of the program, while still remaining valid on the complete code. This requirement is captured by the notion that Q must be an *approximation* of P.

Definition 3.5 (Approximation)**.** A predicate Q is an *(under)approximation* of a predicate P when for any term a, we have $P\ a \implies Q\ a$.

Approximation provides a guarantee similar to that of incrementality (Def. 3.4), in that it ensures that if Q is not provable for some case, then the issue comes from the complete parts of the program or from the (tight) specification P itself, and is not caused by the reformulation.

However, approximation alone is not sufficient to guarantee that Q is a useful reformulation of P. For instance, the always-true predicate is trivially an approximation of any predicate P, but is useless for incremental certification. This leads to the second key property, which is that Q must be precise enough to capture the intended specification of P, that is, ensuring that P can be *recovered* from Q when the program is complete (§2.4).

Definition 3.6 (Recoverability)**.** A predicate P is *recoverable* from a predicate Q when for any *complete* term a, we have $Q\ a \implies P\ a$.

Dually to the true predicate, the false predicate satisfies recoverability, but not approximation.

Incremental reformulation. The proper incremental reformulation of a tight predicate needs to satisfy all three properties of incrementality, approximation, and recoverability. These properties together establish the synchronization between the tight predicate P and its reformulation Q, namely that we can reason incrementally using Q, without being hampered by incompleteness, while still achieving P once the program is complete.

Definition 3.7 (Incremental reformulation)**.** A predicate Q is an *incremental reformulation* of a predicate P when the following conditions hold:
(i) Q is *incremental*: for any two terms a_1 and a_2 such that $a_1 \sqsubseteq a_2$, $Q\ a_1 \implies Q\ a_2$;
(ii) Q is an *approximation* of P: for any term a, $P\ a \implies Q\ a$;
(iii) P is *recoverable* from Q: for any *complete* term a, $Q\ a \implies P\ a$.

3.3 Equality and Incrementality

We now turn to propositional equality: why it is not incremental, and how we can reformulate it. Take a minimalist predicate that uses equality: $0 = t$, where t is the program being incrementally developed. Now consider that the whole program is not yet defined; using the incompleteness notation of the exceptional approach, $t := \bullet$.

Equality is not incremental. Because 0 and $\bullet$ are distinct normal forms, $0 = \bullet$ is not provable, despite the fact that it should hold by incrementality (Def. 3.4). Indeed, from the definition, any two terms a_1, a_2, such that $a_1 \sqsubseteq a_2$, and a predicate P, then $P\ a_1 \implies P\ a_2$. Taking $a_1 := 0$ and $a_2 := \bullet$, where $0 \sqsubseteq \bullet$ (Rule $\sqsubseteq$-INCOMPLETE), and $P := \lambda\, x.\, 0 = x$, where $P\,0$ holds by reflexivity, then $P\,\bullet$ should hold, which is not the case. This shows that propositional equality is not incremental.

Refinement is incremental. For an incremental reformulation Q of the predicate P above, by Def. 3.7 the following should hold: *(i)* $Q\,\bullet$, *(ii)* $P\ 0 \implies Q\,0$, *(iii)* for any complete $a \neq 0$, $\neg(Q\,a)$. Naturally, refinement itself can be used: $Q := \lambda\, x.\, 0 \sqsubseteq x$ satisfies all these conditions.

Symmetry. Although it is tempting to conclude that it suffices to replace propositional equality ($=$) by refinement ($\sqsubseteq$), the former is symmetric while the latter is not. This simple example shows that for an equality where one side is complete and the other may be incomplete, such as $0 = t$, equality must be replaced by refinement directed at the complete side. That is, $0 = t$ must be reformulated as $0 \sqsubseteq t$, while $t = 0$ must be reformulated as $t \sqsupseteq 0$. In the general case where both operands are incrementally defined, such as $t_1 = t_2$, the solution is to split the equality via the existence of a common term ($\exists t.\, t = t_1 \wedge t = t_2$) and then reformulating each equality: $\exists t.\, t \sqsubseteq t_1 \wedge t \sqsubseteq t_2$.

Polarity. For specifications that involve implication, the polarity of equality propositions matters. The reformulation approach described above is valid for a *positive* occurrence, such as a conclusion: $A \rightarrow t = 0$ must be reformulated as $A \rightarrow t \sqsupseteq 0$. This allows dealing with incompleteness seamlessly: if t is $\bullet$, the conclusion ($\bullet \sqsupseteq 0$) is trivial by maximality of $\bullet$. In contrast, for a *negative* occurrence, the approach is reversed: $t = 0 \rightarrow A$ must be reformulated as $t \sqsubseteq 0 \rightarrow A$. Then, if t is $\bullet$, the premise ($\bullet \sqsubseteq 0$) is a contradiction, and A is trivially established by the principle of explosion. A negative occurrence of a general equality, as in $t_1 = t_2 \rightarrow A$, needs to be reformulated with the dual $\exists t.\, t \sqsupseteq t_1 \wedge t \sqsupseteq t_2$, with the additional constraint that t be complete, otherwise the reformulation would be trivial by picking $t := \bullet$.

$$\textsc{IncRef-Const}\quad \frac{a \notin \mathit{free\_vars}(P)}{\text{IncRef } (\lambda\, a : A.\, P)} \qquad \textsc{IncRef-}\longrightarrow\quad \frac{\text{IncRef } P \qquad \text{IncRef } Q}{\text{IncRef } (\lambda\, a : A.\, P\ a \rightarrow Q\ a)} \qquad \textsc{IncRef-}\wedge\quad \frac{\text{IncRef } P \qquad \text{IncRef } Q}{\text{IncRef } (\lambda\, a : A.\, P\ a \wedge Q\ a)}$$

$$\textsc{IncRef-}\vee\quad \frac{\text{IncRef } P \qquad \text{IncRef } Q}{\text{IncRef } (\lambda\, a : A.\, P\ a \vee Q\ a)} \qquad \textsc{IncRef-}\forall\quad \frac{\forall b : B, \text{IncRef } (P\ b)}{\text{IncRef } (\lambda\, a : A.\, \forall b : B, P\ b\ a)} \qquad \textsc{IncRef-}\exists\quad \frac{\forall b : B, \text{IncRef } (P\ b)}{\text{IncRef } (\lambda\, a : A.\, \exists b : B, P\ b\ a)}$$

$$\textsc{IncRef-=}\quad \frac{\text{complete-minimal B} \qquad \text{is\_complete } f \qquad \text{is\_monotone } f \qquad \text{is\_complete } g \qquad \text{is\_monotone } g}{\text{IncRef } (\lambda\, a : A.\, f\ a =_B g\ a)}$$

$$\begin{array}{llrll r}
\mu(\lambda\, a.\, P) & := \lambda\, a.\, P & (\mu\text{-Const}) & \nu(\lambda\, a.\, P) & := \lambda\, a.\, P & (\nu\text{-Const}) \\
\mu(\lambda\, a.\, P\ a \rightarrow Q\ a) & := \lambda\, a.\, (\nu P)\ a \rightarrow (\mu Q)\ a & (\mu\text{-}\longrightarrow) & \nu(\lambda\, a.\, P\ a \rightarrow Q\ a) & := \lambda\, a.\, (\mu P)\ a \rightarrow (\nu Q)\ a & (\nu\text{-}\longrightarrow) \\
\mu(\lambda\, a.\, P\ a \wedge Q\ a) & := \lambda\, a.\, (\mu P)\ a \wedge (\mu Q)\ a & (\mu\text{-}\wedge) & \nu(\lambda\, a.\, P\ a \wedge Q\ a) & := \lambda\, a.\, (\nu P)\ a \wedge (\nu Q)\ a & (\nu\text{-}\wedge) \\
\mu(\lambda\, a.\, P\ a \vee Q\ a) & := \lambda\, a.\, (\mu P)\ a \vee (\mu Q)\ a & (\mu\text{-}\vee) & \nu(\lambda\, a.\, P\ a \vee Q\ a) & := \lambda\, a.\, (\nu P)\ a \vee (\nu Q)\ a & (\nu\text{-}\vee) \\
\mu(\lambda\, a.\, \forall (b : B), P\ b\ a) & := \lambda\, a.\, \forall (b : B), (\mu P\ b)\ a & (\mu\text{-}\forall) & \nu(\lambda\, a.\, \forall (b : B), P\ b\ a) & := \lambda\, a.\, \forall (b : B), (\nu P\ b)\ a & (\nu\text{-}\forall) \\
\mu(\lambda\, a.\, \exists (b : B), P\ b\ a) & := \lambda\, a.\, \exists (b : B), (\mu P\ b)\ a & (\mu\text{-}\exists) & \nu(\lambda\, a.\, \exists (b : B), P\ b\ a) & := \lambda\, a.\, \exists (b : B), (\nu P\ b)\ a & (\nu\text{-}\exists)
\end{array}$$

$$\mu(\lambda\, a.\, f\ a = g\ a) := \lambda\, a.\, \exists (b : B), b \sqsubseteq f\ a \wedge b \sqsubseteq g\ a \qquad (\mu\text{-=})$$

$$\nu(\lambda\, a.\, f\ a = g\ a) := \lambda\, a.\, \exists (b : B), b \sqsupseteq f\ a \wedge b \sqsupseteq g\ a \wedge \text{is\_complete } b \qquad (\nu\text{-=})$$

Fig. 2. The IncRef syntactic fragment of predicates, with monotonization (μ) and antitonization (ν).

This discussion sheds light on how to reformulate specifications that rely on propositional equality, in order to achieve incremental certified programming. But it also highlights the fact that reformulation is quite subtle, and hence error-prone when performed manually. We next turn to systematic, incremental-by-construction reformulation of arbitrary predicates.

3.4 Generating Incremental Reformulations

We present a syntactic fragment of predicates, called IncRef, for which incremental reformulations can be systematically derived (Fig. 2). This fragment covers a broad range of practical specifications, avoiding the difficulties of manual ad-hoc reformulation. We describe the fragment and define a transformation that produces incremental reformulations.

Implication. The impact of the polarity of a predicate, when reformulating an implication, is reflected in the requirements of incrementality, approximation, and recoverability. Since incrementality corresponds to monotonicity of reasoning with respect to refinement (Def. 3.4), a predicate in a positive position is incremental if it is *monotone*, while a negative one is incremental if it is *antitone*. Thus, incremental reformulation is defined with two mutually-recursive functions μ and ν, which produce monotone and antitone reformulations, respectively. An implication can be reformulated if its constituents can be reformulated (IncRef-$\longrightarrow$), and the reformulation functions (μ-$\longrightarrow$, ν-$\longrightarrow$) address the polarity changes in their premises by recursively applying μ and ν appropriately.

Basic logical constructs. Constant predicates can always be incrementally reformulated (IncRef-Const) by simply returning the original predicate (μ-Const, ν-Const). Basic logical connectives, *i.e.,* conjunction (IncRef-$\wedge$), and disjunction (IncRef-$\vee$) can be reformulated if one can produce incremental reformulations for their constituents. Universal and existential quantification over a type (IncRef-$\forall$, IncRef-$\exists$) can be reformulated when their predicate can be pointwise incrementally reformulated. In all these cases, reformulation simply proceeds recursively on subterms.

Propositional equality. Rule IncRef-= captures and generalizes the intuition about the systematic reformulation of equality discussed in §3.3: it restricts equality to be on complete-minimal types—which ensures recoverability from refinement—and generalizes both sides of the equality by functions f and g—defining the notion of *observation* on the term a. Two typical cases of observation are: (1) constant functions that produce some term to use as the specification of some behavior, *e.g.,* an expected output, or (2) applying the term a to some arguments, or use the term a as arguments to other functions. For instance, in the conclusion of the running example, *i.e.,* `has_type empty (eval t) T = true`, the left-hand side of the equality can be instantiated in IncRef-= as `f := fun eval => has_type empty (eval t) T`, applying arguments to `eval` and as an argument to another function, while the right-hand side specifies the expected output `g := fun _ => true`. Completeness of the observations ensures that when a is complete then both f and g produce complete outputs of a complete-minimal type, which allows recovering equality from refinement (Def. 3.3). Monotonicity of the observations preserves the refinement relation for a, which is then used to show incrementality of the reformulation.

The monotonization of equality on complete-minimal types reformulates equality as the existence of a common more refined term b (μ-=), and the antitonization as the existence of a common less refined type that is complete (ν-=), as explained in §3.3. For function types, which are not complete-minimal (§3.1), equality can be reformulated by translating it to pointwise equality using function extensionality, and recursively applying this process until a complete-minimal type is reached.

Inductive predicates. The presentation of IncRef (Fig. 2) can be extended with new forms of predicates, most importantly inductive predicates. In the case of non-indexed parameterized inductive predicates, the reformulation is direct: it suffices to reformulate every proposition that appears as argument of each constructor. In fact, the cases of conjunction, disjunction and existential quantification (Fig. 2) are specific cases of such inductive predicates.

For reformulating indexed inductives types, the implicit equalities that follow from the use of indices need to be made explicit, before reformulating the proposition of each constructor. To illustrate, consider the simple `IsEven` (indexed) inductive predicate:

```
Inductive IsEven : nat -> Prop :=
| IsEven0 : IsEven 0
| IsEvenSS : forall n, IsEven n -> IsEven (S (S n)).
```

The first constructor mandates that the index is equal to 0, and the second constructor requires the index of the resulting type to be equal to the input type index plus 2. `IsEven` is a tight predicate due to these implicit equalities: `IsEven` • is not provable because • is neither equal to 0 nor to `S (S n)`.

Making equalities explicit consists in using the "forded" presentation of `IsEven`, *i.e.,* where indices are replaced with parameters, and equalities turned into explicit arguments of constructors:

```
Inductive IsEven (n : nat) : Prop :=
| IsEven0 : n = 0 -> IsEven n
| IsEvenSS : forall n', n = S (S n') -> IsEven n' -> IsEven n.
```

Then, the reformulation proceeds as usual for each constructor. The CompCert case study (§6) describes a more involved example of an inductive relation in a real-world setting.

Correctness of Incremental Reformulation. We now establish that the reformulation process described in this section indeed produces proper incremental reformulations.

Theorem 3.1. *For any* IncRef *predicate P:*

- *$\mu(P)$ is monotone, and $\nu(P)$ is antitone*
- *$\mu(P)$ is an approximation of P, and P is an approximation of $\nu(P)$;*
- *P is recoverable from $\mu(P)$, and $\nu(P)$ is recoverable from P.*

We show in the Rocq implementation of the ICP framework described in the next section that each step of the transformation is correct by construction, meaning that every inference rule for an IncRef predicate preserves the invariants of Theorem 3.1. These steps are proven constructively, by induction on the derivation of an IncRef predicate. Consequently, the transformation described by μ yields incremental reformulations of predicates (Def. 3.7):

Corollary 3.2. *For any* IncRef *predicate P, $\mu(P)$ is an incremental reformulation of P.*

Directly applying the rules of Fig. 2 to `res_safety` yields the following statement:

```
Theorem res_safety_inc_ref : forall (t : tm) (T : ty), has_type empty t T = true ->
    exists r, r ⊑ (t' <- res_eval t ;; ret (has_type empty t' T)) ∧ r ⊑ Success true.
```

As explained in §3.3, the statement can be simplified, because `Success true` is complete. The Rocq implementation tries to automatically detect these cases, using typeclass instances and priorities, and directly reformulate into the simpler form:

```
  Theorem res_safety_inc_ref : forall (t : tm) (T : ty), has_type empty t T = true ->
      (t' <- res_eval t ;; ret (has_type empty t' T)) ⊒ Success true.
```

4 IncRease: A Framework for ICP in Rocq

We now describe IncRease, a prototype of the formal framework presented in §3 implemented in the Rocq Prover [The Rocq Development Team 2024], using typeclasses for both extensibility and automation. The same approach is applicable to any proof assistant with similar features, such as Lean [Moura and Ullrich 2021]. Other implementation techniques are also possible, for instance using Elpi [Tassi 2018] or other metaprogramming frameworks. We start by explaining the core ICP framework, before presenting the automatic generation of incremental reformulations of predicates. We describe the monadic and exceptional instantiations of IncRease and some applications in the following sections. Note that the proofs associated with instance declarations are omitted here—they can be found in the accompanying artifact.

4.1 Core Framework

The openness of IncRease with respect to specific incompleteness mechanisms is supported by defining the notions of refinement and completeness as typeclasses [Sozeau and Oury 2008].

The `Refinable` typeclass equips a type `A` with a refinement preorder (Def. 3.1):[5]

```
Class Refinable (A : Type) : Type := {
  ⊑ : A -> A -> Prop ;
  is_transitive : forall a1 a2 a3, a1 ⊑ a2 -> a2 ⊑ a3 -> a1 ⊑ a3 ;
  is_reflexive : forall a, a ⊑ a  }.
```

Refinement for function types (Def. 3.2) is then specified with the following instance:[6]

```
Instance refinableFun {A B : Type} `{Refinable B} : Refinable (A -> B) := {
  f ⊑ g := forall a, f a ⊑ g a }.
```

The notion of completeness is modeled as the following class:

```
Class Complete (A : Type) := { is_complete : A -> Prop }.
```

Completeness for functions (§3.1) enforces that they preserve completeness of their arguments:

```
Instance completeFun {A B : Type} `{Complete A} `{Complete B} : Complete (A -> B) := {
    is_complete f := forall a, is_complete a -> is_complete (f a) }.
```

[5]Using a symbol directly as a field name is not allowed in Rocq. We adopt this notation for presentation purposes.
[6]In Rocq, an implicit parameter is indicated in curly braces: `{A : Type}` indicates that parameter `A` is resolved implicitly. If the implicit parameter need not be named, because it is a premise that is unused in the code itself, the name can be omitted and the opening curly brace preceded with a backquote: `` `{Refinable A} `` means that there has to be some term of type `Refinable A` available in scope (here, an instance of the `Refinable` typeclass for `A`).

The `CompleteMinimal` typeclass characterizes types for which completeness implies minimality in the refinement order (Def. 3.3):

```
Class CompleteMinimal (A : Type) `{Refinable A} `{Complete A} : Type := {
  is_complete_minimal : forall a, is_complete a -> forall a', a' ⊑ a -> a' = a  }.
```

Finally, we provide three default instances that describe terms that cannot be made incomplete, *i.e.*, for which the refinement relation is taken to be propositional equality, and for which completeness trivially holds.

```
Instance eqRefinable (A : Type) : Refinable A | 100 := {  ⊑ := eq }.
Instance eqCompleteTrue (A : Type) : Complete A | 100 := { is_complete _ := True }.
Instance eqCompleteMinimalTrue (A : Type)
                  (refEqA := eqRefinable A) (compA := eqCompleteTrue A) : CompleteMinimal A | 100.
```

These instances help reduce boilerplate code because the typeclass search mechanism will use them automatically when no other instance is found. The instances are defined as (ambiguous) defaults and therefore assigned a low priority (the declared "cost" of 100 for each instance above).

4.2 Automatic Generation of Incremental Reformulations

We encode the IncRef fragment together with the monotonization and antitonization functions (`ir_mono` and `ir_anti`, resp.), as well as the proof of Theorem 3.1, using a single typeclass `IncRef`:

```
  Class IncRef {A : Type} `{Refinable A} `{Complete A} (P : A -> Prop) := {
      ir_mono : A -> Prop;
      ir_anti : A -> Prop;
      is_monotone_mono : forall a1 a2, a1 ⊑ a2 -> ir_mono a1 -> ir_mono a2;
      is_antitone_anti : forall a1 a2, a1 ⊑ a2 -> ir_anti a2 -> ir_anti a1;
      approx_mono : forall a, P a -> ir_mono a;
      approx_anti : forall a, ir_anti a -> P a;
      recoverability_mono : forall (a : A), is_complete a -> ir_mono a -> P a;
      recoverability_anti : forall (a : A), is_complete a -> P a -> ir_anti a }.
```

All of the transformations described in § 3.4 are implemented in the Rocq development as instances of this typeclass, with each instance including the corresponding proof of Theorem 3.1. For example, the snippet below shows the implementation of rules μ-= and ν-=:

```
  Instance IncRefEq {B} `{HB : CompleteMinimal B} (f g : A -> B)
    (Hcf : is_monotone f) (Hch : is_monotone g) (Hcf' : is_complete f) (Hcg' : is_complete g)
    : IncRef (fun a => f a = g a) := {
        ir_mono := fun a => exists b, b ⊑ f a ∧ b ⊑ g a ;
        ir_anti := fun a => exists b, f a ⊑ b ∧ g a ⊑ b ∧ is_complete b }.
```

In practice, this means that IncRease is able to derive automatically an instance of `IncRef` for a large class of predicates, and the function `ir_mono` can be used directly to monotonize a predicate. Because the instance `IncRefEq` requires completeness assumptions, IncRease can be instrumented to automatically infer completeness of functions `f` and `g` in practice

An interest of the typeclass-based approach is that users can manually extend IncRease by registering `IncRef` instances as needed and use priorities to guide the reformulation process. This allows extending the IncRef fragment beyond the initial core predicates (§ 3.4), for example to support specific inductive predicates used in a given development, as illustrated later in § 6.

5 Instantiating IncRease

We now describe both monadic and exceptional instantiations of IncRease, which we use in the case studies. We leave the study of incomplete inductive relations in IncRease to future work (§ 9).

5.1 Monadic Instantiations of IncRease

We illustrate a monadic instantiation of IncRease with the `result` monad introduced in § 2:

```
Inductive result (A : Type) :=
| Success : A -> result A
| NotImplemented : result A.
```

The refinement relation is straightforward: `NotImplemented` is the least refined element.

```
Instance refinableResult {A : Type} `{Refinable A} : Refinable (result A) := {
  r1 ⊑ r2 := match r1, r2 with | _, NotImplemented => True
                               | Success x1, Success x2 => x1 ⊑ x2
                               | _, _ => False end }.
```

`NotImplemented` is not complete, and a `Success` is complete if its content is also complete.

```
Instance completeResult {A : Type} `{Complete A} : Complete (result A) := {
  is_complete r := match r with | Success a => is_complete a
                                | NotImplemented => False end }.
```

With this incompleteness mechanism, terms of non-monadic base types (*e.g.*, `tm`, `ty` or `bool`) are by definition complete, so we rely on the default refinement relation to be propositional equality with the total completeness relation (§4.1) for such types.

When inlining partiality directly into an inductive type, by introducing a new specialized constructor, the process is similar and straightforward. For example, if one adds a new `NotImplementedBool` constructor to booleans, then `NotImplementedBool` is the least refined term for this type, and `true` and `false` are the only complete terms for the type. Other base types that are not modified with inlined partiality are complete by definition, *i.e.,* refinement defaults to propositional equality.

5.2 Exceptional Instantiation of Increase

We provide an exceptional instantiation of IncRease based on the exceptional type theory of Pédrot and Tabareau [2018] (ExTT). While the CoqRETT [Pédrot et al. 2019] plugin already implements a prototype of ExTT, we do not use it here because of incompatibility with recent Rocq versions. Instead, we provide a simple prototype based on rewrite rules [Cockx et al. 2021], which have recently been integrated into Rocq [Leray et al. 2024]. This basic implementation serves as a proof-of-concept for experimenting and for initial validations of the theoretical framework.

Technically, we introduce a (type-indexed) exceptional term • for incompleteness:

```
Symbol • : forall (A : Type), A.
```

The propagation semantics of the exceptional term is then straightforward to implement using rewrite rules, but has to be defined for each type, as illustrated below for the case of `tm`:

```
Rewrite Rules tm_red_rew :=
| match • tm as t0 return ?P with | tm_var _ => _ | _ => _ end
  => • (?P@{t0 := • tm})
```

A rewrite rule states that, during reduction, whenever the pattern on the left of the => symbol is encountered, the matched term reduces to the term on the right. Here, a `match` whose discriminee is a `• tm` reduces to a • exception of the correct return type. The return type is captured with pattern variable `?P`, where the variable for the scrutinee `t0` is explicitly substituted by the current one.

We can then define the incomplete `eval` function using `• tm`:

```
Fixpoint eval (t : tm) : tm := match t with | tm_app _ _ => • tm | ... end.
```

Contrary to the monadic case, we cannot rely on the default propositional equality instance (§4.1) to define refinement, because the • exception can inhabit any type. The definitions remain straightforward, with the exception being the least refined element for each type involved in the development, and considering the possible occurrences in the recursive terms. This is simply a variation on relational parametricity that adds $x \sqsubseteq \bullet$ for every element x. The following snippet illustrates the case of type `tm`, as shown in §3.1:

```
Inductive refinement_tm : tm -> tm -> Prop := (* other constructors omitted *)
  | refinement_tm_if : forall {t1 t2 t3 t1' t2' t3' : tm},
      t1 ⊑tm t1' -> t2 ⊑tm t2' -> t3 ⊑tm t3' -> (tm_if t1 t2 t3) ⊑tm (tm_if t1' t2' t3')
  | refinement_tm_unk : forall (t : tm), t ⊑tm • tm
where "t1 ⊑tm t2" := (refinement_tm t1 t2).
```

Completeness must also be defined per case, but is straightforward as well: any term that is not the exception is complete, considering the recursive occurrences. For example, completeness for type `ty` is defined as follows:

```
Inductive is_complete_ty : ty -> Prop :=
  | is_complete_ty_bool : is_complete_ty Ty_Bool
  | is_complete_ty_arrow : forall T1 T2,
    is_complete_ty T1 -> is_complete_ty T2 -> is_complete_ty (Ty_Arrow T1 T2).
```

In the exceptional approach, refinement relations and completeness predicates are boilerplate code that could be automated using metaprogramming frameworks such as MetaCoq [Sozeau et al. 2020] or Elpi [Tassi 2018], enhancing the usability of the approach.

6 Incremental Dead-Code Elimination in CompCert

This case study applies IncRease to reformulate complex inductive predicates in a real-world setting, and to show that incompleteness can be introduced after the fact in a large development, although with associated costs. We use the CompCert C verified compiler [Leroy 2009], which is structured as a sequence of compilation passes, each transforming an intermediate representation. All passes are certified separately and integrated into a global correctness theorem. Specifically, we focus on a single optimization pass, dead-code elimination (DCE), explained in §6.1, and study how to reformulate correctness to be incremental with respect to an incomplete implementation of the optimization (§6.2). We also discuss retrofitting incompleteness in existing developments in §6.3.

6.1 Background on the CompCert DCE Pass

One of the intermediate representations used in CompCert is the Register Transfer Language (RTL), which consists of instructions, functions, and programs. Instructions (of type `instruction`) roughly correspond to elementary instructions of the target processor, storing results in pseudo-registers. Functions (of type `function`) are defined as control-flow graphs (CFG) of instructions, along with metadata such as its signature, entry point in the CFG, and argument registers. A program is defined as a list of global definitions and the "main" function that serves as the program's entry point.

The DCE pass is an optimization over RTL programs, defined with `transf_instr`, which optimizes individual instructions, and `transf_function`, which applies `transf_instr` across the entire CFG of a function. This optimization pass must preserve the semantics of the original function.

The semantics of RTL is described as transitions between states. A state captures the current point of execution of a program, and can be either (1) a *regular state*, which represents execution inside a function, (2) an intermediate *callstate*, which appears during function calls, or (3) an intermediate *return state*, which represents the point when a function terminates and returns to its caller. Regular states track the execution within a function, storing a reference to the current function (`f`), a pointer to the call stack (`sp`), the current program point in the CFG (`pc`), and the register state (`e`). Callstates store a reference to the function definition being called (`f`), and the list of arguments (`args`). Return states store the return value (`v`). All states also maintain the call stack (`s`) and the memory state (`m`).

To formally establish the correctness of the DCE pass, CompCert defines the inductive relation `match_states`, shown below, which connects execution states between unoptimized and optimized functions. The definition of `match_states` follows the structure of RTL states, distinguishing between regular states, call states, and return states. The following snippet shows the case of regular states, focusing on the function transformation aspect:

```
Inductive match_states: state -> state -> Prop :=
| match_regular_states: forall s f sp pc e m ts tf te tm cu ...
      (FUN : transf_function (romem_for cu) f = OK tf),
    match_states (State s f vptr pc e m) (State ts tf vptr pc te tm)
| match_call_states:    ... (* case of call states   *)
| match_return_states : ... (* case of return states *)
```

In match_regular_states, a regular state in the unoptimized program is related to a regular state in the optimized program if they represent the same execution context and the executing function in one state is the optimized version of the other. This is enforced by the premise FUN, which ensures that tf is obtained by applying transf_function to f, over a read-only memory generated for the compilation unit cu. Additional premises are included to ensure that the call stacks coincide (s), as well as memory (m, tm) and registry states (e, te), among others, omitted here for brevity.

Building on the match_states relation, the correctness theorem of the DCE pass can be stated as:

```
  Theorem step_simulation :
    forall S1 t S2, step ge S1 t S2 -> forall S1', match_states S1 S1' -> sound_state prog S1 ->
       exists S2', step tge S1' t S2' ∧ match_states S2 S2'.
```

That is, if a valid initial state S1, in the context of a program prog, can take a transition step to a state S2, in the global environment ge and producing a trace t of system calls, then a related state S1' can also take a step to a state S2' with the optimized function, producing the same trace of system calls t, and preserve the relation between the end states S2 and S2'.

6.2 Reformulating Inductive Propositions for DCE

To simulate the incremental scenario, we leave one case of the optimization function transf_instr incomplete, making transf_function incomplete as well. The problem that arises with incompleteness of the transf_function is that step_simulation becomes tight with respect to the function. Because the optimization function is incomplete for one instruction, there is not enough information to conclude that the transformed state is capable of transitioning to another state, and thus that it simulates the original one. More particularly, the tightness can be pinpointed to the equalities over the incomplete function, namely in the FUN premise of the match_regular_states constructor (likewise for match_call_states).

We cannot directly use the approach from §3.4 to reformulate match_states because match_states is actually not a predicate over the incomplete function, but a relation over states. We must therefore parameterize the inductive definition by transf_function, such that it becomes a non-indexed inductive predicate. Moreover, the match_regular_state constructor is already in a forded presentation, explicitly exhibiting the equality that we need to change. The reformulation can then proceed as expected, changing each proposition in the constructors correspondingly.

To avoid duplication of inductives (original, monotonized and antitonized), we abstract over the equality and parameterize match_states with an additional generic relation R, as shown below.

```
Inductive match_states_pred (R : forall A `{Refinable A} `{Complete A}, A -> A -> Prop)
    (transf_function : romem -> function -> res function) : state -> state -> Prop :=
| match_regular_states_pred: forall s f sp pc e m ts tf te tm cu ...
      (FUN: R (res function) (transf_function (romem_for cu) f) (OK tf)),
      match_states_pred R transf_function (State s f vptr pc e m) (State ts tf vptr pc te tm)
| ...
```

The original match_states definition is obtained by passing eq and transf_function as arguments.

The IncRef instance is defined by using the monotone and antitone definitions of equality for R, named mono_eq (resp. anti_eq) as defined in μ-= (resp. ν-=):

```
Instance IncRefMatchStates S1 S2 : IncRef (fun tf => match_states_pred eq tf S1 S2) := {
  ir_mono tf := match_states_pred  mono_eq  tf S1 S2 ;
  ir_anti tf := match_states_pred  anti_eq  tf S1 S2 }.
```

A second inductive relation over the call stack, `match_stackframes`, which depends as well on `transf_function`, needs to be reformulated similarly. Once this is done, the incremental reformulation of the `step_simulation` theorem can be automatically computed by the framework using the `ir_mono` function (§4.2), where `step_simulation_pred` is the predicate form of `step_simulation` over `transf_function`, similar to how `match_states_pred` corresponds to `match_states`.

```
Theorem inc_step_simulation: ir_mono step_simulation_pred transf_function
```

The proof of the resulting proposition for the incomplete case follows easily from the property of incomplete terms and the other cases are unchanged from the non-incremental scenario.

This example illustrates how complex inductive predicates can be reformulated to avoid code duplication by parametrizing over the relation that connects indices. This relation can later be instantiated with equality (for the original tight case), or with its monotonization or antitonization to define its reformulation. Currently, this transformation must be performed manually as described here; however, it seems possible to achieve automation with metaprogramming frameworks such as MetaCoq [Sozeau et al. 2020] or Elpi [Tassi 2018].

6.3 Introducing Incompleteness in DCE

The process of reformulating the inductive predicates and correctness theorem discussed in the previous section is independent of the specific mechanism used to introduce incompleteness.

Introducing incompleteness in an existing Rocq development like CompCert is itself a non-trivial challenge. Each approach (monads, inlined partiality, exceptions) presents opt-in and opt-out costs that make retrofitting an existing codebase challenging in terms modularity, code duplication, and/or effort. Introducing a new monad requires substantial effort to adjust function signatures and integrate with existing monadic code. Inlining partiality directly into the original inductive forces the whole development to account for the new constructor. One can duplicate the inductive to localize incompleteness, at the expense of increasing code size and requiring some boilerplate between the two versions. Conversely to mainstream programming languages, the current realization of exceptions in type theory lacks the ability to localize their handling, just like inlined partiality.

In the case study implementation, we chose the inlined partiality approach by duplicating the `instruction` inductive type and extending the duplicate with a constructor to denote incompleteness. This duplicated type and its associated definitions live in a separate `RTL_Incomplete` namespace, allowing us to preserve encapsulation and isolate the changes to the DCE pass only.[7] However, as a result, the transformation functions and preservation proofs between the original and extended types amount to a nearly 50% size increase in the file containing the correctness theorem.

Our experience with CompCert also shows that the existing codebase could be better parameterized to support a more lightweight integration of incompleteness. We believe further research is needed to better understand patterns, and devise mechanisms, for introducing incompleteness in certified developments. We return to this topic in §9.

7 ICP Meets Property-Based Testing

Randomized property-based testing, as provided in Rocq by QuickChick [Dénès et al. 2014], is complementary to certified programming as it can eagerly invalidate properties before one works on a (possibly tedious and challenging) formal proof.

[7]The code of §6.2 in the accompanying artifact contains minor adjustments to account for this duplication and namespace.

In this case study, we apply the QuickChick plugin to the running example of an incomplete STLC implementation (§2.1), using the monadic instantiation of IncRease with the `result` monad (§5.1). We then compare what happens when testing the tight `res_safety` property and its incremental reformulation, with (a) the incomplete but correct `res_eval` function (§ 2.2), and (b) a buggy variant `buggy_res_eval` that incorrectly evaluates conditional expressions, recursively evaluating the condition `c` again instead of the true branch `tb` whenever `c` yields `true`.

This case study shows that using tight predicates defeats the purpose of randomized testing by exhibiting incomplete parts of the code; instead, using incremental predicates allows for tests to pass on incomplete code and to fail only on actual bugs. This illustrates how the formal principles we propose for incremental certified programming also apply effectively to the domain of testing.

7.1 Testing Safety with QuickChick

We follow the expected QuickChick workflow, by first defining generators and shrinkers for types (`ty`) and well-typed terms (`tm`). These generators and shrinkers are based on examples found in QuickChick's repository and the Software Foundations Volume 4 textbook [Pierce et al. 2015]. We then define an abstract checker, `stlc_checker`, which generates well-typed terms, parametrized over the predicate to be tested (`pred`) and the size of the generated type (`ty_size`):

```
Definition stlc_checker pred ty_size :=
  forAll (arbitrarySized ty_size) (fun (T : ty) =>
    forAllShrink (genSizedTypedTm T) (liftShrink (shrinkTypedTm T))
      (fun (mt : option tm) => match mt with Some t => pred T t | None => false end)).
```

Following the structure of the `res_safety` predicate, we first universally quantify over the types using the `forAll` checker, which expects a generator and a predicate, and checks that every generated term satisfies the predicate. The `arbitrarySized` generator for `ty` is automatically derived by QuickChick and produces a term of type `ty` of size `ty_size`. Given the generated type `T`, we then generate well-typed terms `tm` using the generator `genSizedTypedTerm`, corresponding to the premise of `res_safety`. The `forAllShrink` checker is similar to `forAll` but receives an additional shrinker function, `shrinkTypedTm`, which builds smaller well-typed terms when a test fails, in order to narrow down the problematic cases. Finally, we test that the well-typed term satisfies the predicate `pred`.

To exercise the tight and incremental formulations of `res_safety`, we implement two concrete checkers that use these predicates (simplifying the predicate for `incr_safety_checker` as in §3.4):

```
Definition tight_safety_checker eval ty_size := stlc_checker (fun (T : ty) (t : tm) =>
  (t' <- eval t ;; ret (has_type map_empty t' T)) = Success true ?) ty_size.

Definition incr_safety_checker eval ty_size := stlc_checker (fun (T : ty) (t : tm) =>
  (t' <- eval t ;; ret (has_type map_empty t' T)) ⊒ Success true ?) ty_size.
```

7.2 Testing Tight vs Incremental Formulations of Safety

The table below summarizes a test execution for the two interpreters, applied to the tight and incremental checkers, showing the counterexamples exhibited by QuickChick.

	Tight safety predicate (=)	Incremental safety predicate (⊒)
`res_eval` (correct)	`(λ (1 : Bool) => (λ _ => 1)) false` ✗ Counterexample triggers incomplete code	– ✓ Passes all 10.000 tests
`buggy_res_eval`	`(λ (1 : Bool) => (λ _ => 1)) true` ✗ Counterexample triggers incomplete code	`if true then (λ _ => true) else (λ _ => true)` ✓ Counterexample triggers bug

As expected, the tight checker fails as soon as it hits incompleteness (function applications), thereby hiding both the erroneous behavior of `buggy_res_eval` and the correctness of the implemented parts of `res_eval`. In contrast, the incremental checker is oblivious to incompleteness, allowing QuickChick to pass all tests for `res_eval`, while hitting the actual problematic conditional terms in `buggy_res_eval`. To check the coverage of the randomly generated terms, we use the `collect` interface from QuickChick to categorize the results and display their occurrences. For instance, we check the distribution of results that `res_eval` produces, with `NotImplemented` accounting for around 1,500 out of 10,000 cases. This shows the benefits of integrating ICP with property-based testing, to ensure that incompleteness does not pollute test results, and that programmers can enjoy the benefits of testing actually-implemented functionality.

8 ICP Meets Deductive Synthesis

We now show an application to deductive synthesis through a novel adaptation of the Fiat library [Delaware et al. 2015]. Fiat supports synthesizing optimized code using a stepwise *specification refinement* infrastructure with respect to a naive reference implementation, as we briefly illustrate in §8.1. But Fiat itself does not support *completion refinement*, because a specification refinement can only be established by providing a complete program. In this case study, we observe that we can adequately reformulate the specification refinement of Fiat to enable a seamless combination of both methodologies (§8.2), allowing the synthesis of incomplete programs (§8.3).

8.1 Background on Fiat

We briefly illustrate Fiat using a simple queue example from the Fiat tutorial (Fig. 3). The queue abstract data type is described by `queue_signature`, where the notation `ADTsignature` generates a record with the type signatures of the ADT operations. In this signature, `rep` is the type of the internal state of the queue, and `data` corresponds to the type of elements stored in the queue. The queue supports three operations: (1) constructing an empty queue ("`empty`"), (2) inserting an element at the tail of the queue ("`enqueue`"), and (3) retrieving the head of the queue ("`dequeue`"). `Constructor` and `Method` are Fiat notations to define the types and parameters of these operations.

A concrete implementation of the queue, `naive_implementation`, uses a list for the internal representation `rep`. The `Def ADT` notation translates to a record that conforms to the signature described in `queue_signature`. For example, `Def Constructor0` specifies a constructor with no arguments, and `Def Method1` and `Def Method0` describe methods with one and zero arguments, respectively. Internally, Fiat uses a *computation* monad to represent the set of possible values satisfying the specification, hence the use of a `return` operation in the constructor and methods.

Starting from this naive queue implementation, Fiat makes it possible to progressively hone it to an optimized implementation, for instance one with constant-time enqueue and amortized constant-time dequeue that uses two lists internally. To do so, the programmer states a relational invariant between both internal representations, and then synthesizes the optimized implementation via a sequence of applications of lemmas and rewrites that establish the preservation of the relational invariant. For instance, the relation between the naive and optimized representations of the queue is given by `rel` in Fig. 3: `naive` is equal to the first list of `opt` appended with its second list reversed.

Formally, an abstract data type (ADT) A is described by an internal representation r, and methods m taking a (self) representation r and an input i as arguments, and producing an updated representation r' and an output o as result. An ADT B is a (*specification*) *refinement* of another ADT A if each method $B.m$ of B refines $A.m$, defined as [Delaware et al. 2015]:

$$\begin{aligned}\text{spec\_refine}(B.m, A.m) := \quad & \forall r_A\, r_B.\, r_A \approx r_B \to \forall i\, r'_B\, o.\ B.m(r_B, i) \rightsquigarrow (r'_B, o) \\ & \to \exists r'_A\, o'.\ A.m(r_A, i) \rightsquigarrow (r'_A, o') \land o = o' \land r'_A \approx r'_B\end{aligned}$$

```
Definition queue_signature : ADTSig := ADTsignature { Constructor "empty" : rep,
                                         Method "enqueue" : rep * data -> rep,
                                         Method "dequeue" : rep -> rep * (option data) }.

Definition naive_implementation : ADT queue_signature := Def ADT {
  rep := (list data),
  Def Constructor0 "empty" : rep := ret nil,,
  Def Method1 "enqueue" (self : rep) (d : data) : rep := ret (self ++ d :: nil),
  Def Method0 "dequeue"(self : rep) : rep * (option data) := ... (* omitted for brevity *) }.

Definition rel (naive : list data) (opt : list data * list data) :=  (* ≈ relation *)
  naive = fst opt ++ rev (snd opt).
```

Fig. 3. Stepwise refinement of a queue ADT, starting from a naive implementation that uses a single list into an optimized version with two lists. Relation rel captures the invariant between the representations.

Above, $\approx$ is the binary relation capturing the invariant between the representation types of A and B (rel in Fig. 3). The reduction $c \rightsquigarrow v$ means that v is a possible result value of the computation c. This is because Fiat allows stepwise refinement of non-deterministic computations with ones that are possibly more deterministic. Note that in the definition above, we are making explicit the equality constraint ($o = o'$), which is implicit in [Delaware et al. 2015] via the non-linear use of a single metavariable. The notion of specification refinement for constructors is similar.

For example, consider the refinement of the enqueue method from the naive single-list implementation to the optimized two-list implementation. The implementation is not given directly but rather synthesized during Fiat's stepwise refinement process using concrete terms obtained while proving preservation of the relation. In this case, the new optimized definition is specified by means of a lemma showing the preservation of rel, namely:

```
Lemma rel_enqueue : forall (d : data) (naive : list data) (opt : list data * list data),
  (* corresponding to $r_A \approx r_B$ in spec_refine *)
  rel naive opt ->
  (* corresponding to $r'_A \approx r'_B$ in spec_refine *)
  rel (naive ++ d :: nil) (fst opt, d :: snd opt) (* right-hand side is the new definition *)
```

Proving spec_refine for enqueue requires showing that the updated internal representations are related ($r'_A \approx r'_B$) using rel_enqueue. Finally, given that the outputs (o and o') are related by equality, the new enqueue method must produce the same value as the naive one.

8.2 Adapting Fiat for ICP

Fiat makes it possible to synthesize specification refinements based on *complete* implementations, such as for the enqueue method above, not incomplete ones. For instance, starting from an incomplete naive implementation of enqueue using exceptions, *e.g.*, ret •, the optimized version cannot refine it with an actual complete behavior. Indeed, refining the naive implementation means proving rel_enqueue, which in turn translates to proving that • = fst opt ++ rev (d :: snd opt). And this is only provable if the optimization raises an exception as well, thus being incomplete. Conversely, if the naive implementation is complete but the optimized version is incomplete, then refinement is again unprovable because the proof of rel_enqueue reduces to showing that the complete code naive ++ d is equal to an incomplete term.

These limitations imply that one cannot exhibit incompleteness in the synthesis steps of Fiat, thus preventing incremental implementations. This tightness stems from the definition of spec_refine itself, namely from the equality of outputs ($o = o'$), and from the chosen invariant relation ($\approx$) that typically uses equality internally, such as rel.

We can adapt Fiat to work with an incremental reformulation of spec_refine, extending its deductive synthesis methodology to incremental certified programming. Following the rules from §3.4, spec_refine is reformulated as:

$$\begin{aligned}\text{spec\_refine\_inc}(B.m, A.m) := \quad & \forall r_A\, r_B.\ \boxed{r_A \approx_\nu r_B} \to \forall i\, r'_B\, o.\ B.m(r_B, i) \rightsquigarrow (r'_B, o) \\ & \to \exists r'_A\, o'.\ A.m(r_A, i) \rightsquigarrow (r'_A, o') \wedge \boxed{o =_\mu o'} \wedge \boxed{r'_A \approx_\mu r'_B}\end{aligned}$$

In this definition, observe that the abstract relation in the premise is replaced by its antitone reformulation ($\approx_\nu$), while it is replaced by its monotone reformulation ($\approx_\mu$) in the conclusion. Likewise, the equality in the conclusion is replaced by its monotone reformulation $o =_\mu o'$.

Going back to the example, the invariant `rel` (Fig. 3) is reformulated with rules μ-= and ν-=.

```
Definition rel_mono (naive : list data) (opt : list data * list data) :=
  exists rep, rep ⊑ naive ∧ rep ⊑ fst opt ++ rev (snd opt).
Definition rel_anti (naive : list data) (opt : list data * list data) :=
  exists rep, rep ⊒ naive ∧ rep ⊑ fst opt ++ rev (snd opt) ∧ is_complete rep.
```

This incremental reformulation makes it possible to establish specification refinements with incomplete implementations, as illustrated next.

8.3 Incremental Specification Refinement

We now illustrate how the adaptation `spec_refine_inc` allows solving one of the two problematic cases discussed above: refining a complete naive implementation of the enqueue method with an incomplete optimized version, using exceptions as the incompleteness mechanism (§5.2). As illustrated in §8.1, we specify the incomplete implementation of enqueue through the `rel_enqueue` lemma that captures the preservation of `rel`. In this case, the tight lemma is the following, noting that we leave the handling of the second list incomplete:

```
Lemma rel_enqueue : forall (d : data) (naive : list data) (opt : list data * list data),
  rel naive opt -> rel (naive ++ d :: nil) (fst opt, • (list data) )
```

We adapt this definition to fit the incremental reformulation, obtaining the following definition:

```
Lemma ir_rel_enqueue : forall (d : data) (naive : list data) (opt : list data * list data),
  rel_anti naive opt -> rel_mono (naive ++ d :: nil) (fst opt, • (list data))
```

We define completion refinement and completeness specifically for types `list`, `pair` and `data`, which are the only types involved in the specification refinement of `naive_implementation`. The definitions follow the same pattern presented for the running example (§5.2). The specification refinement process then follows directly from the Fiat infrastructure. In particular, the concrete optimized implementation of enqueue is given by proving lemma `ir_rel_enqueue` (similar to §8.1), which reduces to the following goal (solved using general refinement lemmas):

```
fst opt ++ rev (snd opt) ⊑ fst opt ++ • (list data)
```

The remaining specification refinement of the `empty` constructor and the dequeue method must also account for the new incremental reformulation. However, the required proofs remain almost untouched, thanks to `rel_anti` ruling out incomplete inputs and by reflexivity of completion refinement. The original specification refinement proof script remains mostly unchanged.

This case study shows that the theoretical foundation of IncRease allows accommodating incompleteness in the stepwise refinement approach of Fiat, by adequately reformulating the notion of specification refinement to make it compatible with ICP.

9 Perspectives for Incremental Certified Programming

This work represents a foundational first step towards systematic and practical support for ICP. The current limitations of IncRease as well as the observations drawn from the case studies should motivate further developments and enhancements, in several directions.

Incompleteness mechanisms. So far, we have studied both monadic and exceptional approaches to incompleteness. While these arguably represent the most used approaches in incremental development in standard programming, there are other mechanisms that could be explored. In particular, using inductive relations is a common technique to model partiality in proof assistants, so it would be particularly important to explore how to deal with them in IncRease. Regarding exceptions, we have focused on the Exceptional Type Theory [Pédrot and Tabareau 2018] and a prototype implementation in Rocq using rewrite rules [Cockx et al. 2021] (also available in Agda). The lack of a production-quality implementation of exceptions in type theory currently limits the ergonomics of this approach, however the recent introduction of sort polymorphism [Poiret et al. 2025], now available in Rocq, opens the door to considerable progress on this front. Other approaches to exceptions, such as algebraic effects for dependent types [Brady 2013], could provide interesting alternatives for ICP given a solid integration in proof assistants.

Opt-in/out costs and encapsulation. Important evaluation criteria for incompleteness mechanisms that directly affect the usability of ICP are the opt-in and opt-out costs of incompleteness, and the possibility to locally encapsulate their use. If a development adopts an infrastructure that accommodates incompleteness from the beginning, for instance by using monadic types everywhere, the opt-in/out costs are avoided by construction; otherwise retrofitting incompleteness is challenging, as discussed in the CompCert case study (§ 6.3). A key aspect of the opt-out cost is obtaining a function of the original type, as one would have without employing incremental programming. The challenge is that either the code must be rewritten to restore the original types, which can be costly, or it must be defined in terms of code that uses an incompleteness mechanism. In this case, proving additional properties of the complete function still requires falling back to the incompleteness mechanism. Overall, it would be worthwhile to explore new incompleteness mechanisms or extend existing ones—such as the universe-based approach to isolate exceptions to certain types explored by Pédrot et al. [2019]—to better support the ICP workflow with lower opt-in/out costs and more flexible encapsulation.

Automation. The ICP formal framework presented here is general, compatible with the whole dependently-typed theory, and independent from automation support. In some cases, one might prefer working with handcrafted statements instead of automatically-generated ones. The framework identifies the properties that the statement should satisfy: monotonicity wrt refinement and the coincidence with the original statement once the program is complete. In IncRease one can even combine both manual and automatic formulations, as once a handcrafted predicate is equipped with its corresponding `IncRef` instance, it can be combined with other constructors from the fragment with automation, and will be handled automatically, if so desired. So far, we have only been able to specify and implement automatic incremental reformulation of logical statements for the IncRef fragment (§ 3.4). IncRef is already more expressive than higher-order logic, and supports ad hoc extension for inductive predicates, as shown in the CompCert case study (§ 6). We envision that the expressiveness of this fragment can be pushed further in future work to enhance automation, regarding both dependencies and inductive predicates.

Expressiveness. The IncRef fragment is expressive enough to cover the examples and case studies we have presented, but it does not cover all possible predicates.[8] Additionally, while the case studies deal with meaningful properties from a variety of scenarios, this currently covers but a small sample of the space of properties one can be interested in establishing while working incrementally. A particularly challenging perspective is to study how to handle *hyperproperties* [Clarkson and Schneider 2008], *i.e.,* properties about multiple runs of a program, like noninterference [Goguen and Meseguer 1982] and parametricity [Reynolds 1983]. Additionally, many advanced developments make use of expressive custom logics on top of the core type theory, such as the Iris framework for higher-order concurrent separation logic [Jung et al. 2018]; studying ICP in general, and IncRease in particular, in such settings would be an important step forward.

10 Related Work

Partial programs. Agda [Bove et al. 2009] provides holes for the incremental editing of programs. From an ICP perspective, these holes are analogous to the axiomatic approach illustrated in §2.2. Indeed, these holes are usually represented as yet unfilled existential variables that are opaque to verification of properties [Yuan et al. 2023; Zhao et al. 2024]. Hazel [Omar et al. 2017] is another programming language that comes with a structural editor that supports incremental programming with holes. However, Hazel is not intended for certified programming. The holes in Hazel behave similar to those in Agda and they would need to be adjusted if Hazel were extended to support incremental certified programming. On the semantic side, the denotation of partial programs have been at the heart of Dana Scott's denotational semantics. Even though that branch of research mostly focused on recursion theory, this led to the study of various notions of partiality and corresponding monads [Escardó and Knapp 2017; Fiore 1994]. Di Liberti et al. [2021] investigate the notion of partial universal algebra and partial equational theories, accounting for undefined values.

Extensibility. Incremental development of a program through successive extensions of complete steps is a hard problem, not unrelated to the *expression problem* coined by Wadler [1998] and studied in a variety of contexts. In particular, the expression problem has been studied in the context of proof assistants, adding the proof engineering dimension to the original software engineering concern [Delaware et al. 2013; Forster and Stark 2020; Jin et al. 2023; van der Rest et al. 2022]. The exploration of the expression problem is mostly concerned with devising, exploiting or evaluating modularity mechanisms, while the need for ICP manifests independently of the modularity mechanisms provided by a programming language or proof assistant. This work used a tiny PL metatheory running example without using any advanced modularity technique or mechanism; it would be interesting to study this relation further.

Refinement calculi and deductive synthesis. Deriving correct-by-construction programs by step-wise refinement from specifications was first introduced by Wirth [1971] and further developed in a variety of settings [Morgan 1994]. The specification need not cover all cases, providing a proxy for a partial program. A Rocq incarnation has been proposed by Boulmé [2007], and Fiat [Delaware et al. 2015], which focuses on the refinement of abstract data types. As shown in §8, completion refinement in ICP and specification refinement are orthogonal and complementary.

[8] An issue arises when defining a property over a function that operates on a term with specific preconditions. Consider a head function defined for lists of strictly positive size. If the `size` function is incomplete, the size-related predicate must be reformulated, hence weakening the constraint that ensures the list is not empty. As a result, an application of head may no longer be well-typed, requiring extending the definition of head itself to cover these additional lists. While such an extension appears feasible in principle, automating the process remains a highly challenging direction for future research.

Gradual dependent types and type theory. Gradual typing blends static and dynamic type checking by introducing an unknown type ? that serves as an optimistic placeholder for more precise types. Bringing gradual typing to dependent types [Eremondi et al. 2019; Lennon-Bertrand et al. 2022] also introduces an unknown term $?_A$ at every type A that can be adequately employed to represent a partial program that has yet to be implemented. However these gradual theories by themselves do not offer a consistent logic to reason on such programs since every type is inhabited by an unknown term. Maillard et al. [2022] investigate sound and consistent reasoning on such gradual programs at the price of inherently breaking the monotonicity of the language with respect to precision. The confinement refinement relation considered in this work is similar to the precision relation of gradual dependent type theories, although precision in gradual typing is semantically tied to concerns of runtime checks and errors [New and Ahmed 2018], which are not necessary ingredients for ICP. Our initial exploration of ICP considered the use of a gradual dependent type theory but found that these additional concerns obscure and complicate matters when one wants to tackle ICP. Nevertheless, the monotone reformulation of equality (Eq. (μ-=) of §3.4) coincides with a precision-based characterization of the consistency relation in gradual typing.

Proof engineering. One important area of research is on the automatic generation of proofs and proof reuse [Agrawal et al. 2023; Boite 2004; Czajka and Kaliszyk 2018; First et al. 2023; Ringer et al. 2021, 2019b]. In the context of ICP, being able to fix proofs in the face of changes that appear in the iterative process is key. The tools provided by IncRease could provide additional ways to consider incompleteness as valid solutions, and still preserve the proofs and specifications. In a separate line, Jain et al. [2020] explore mutation analysis to check for completeness and adequacy of specifications. Similar to QuickChick, IncRease could provide a principled way to do mutation analysis in the presence of incompleteness such that the analysis does not degenerate. Finally, Ho and Pit-Claudel [2024] explore incremental development in the setting of Dafny [Leino 2010], by defining module-based inductive principles to maintain stability and maintenance of proofs. Although it is a different context, some of the principles and characterizations we make in this work could help design alternative solutions.

11 Conclusions

This work is a first step towards supporting incremental certified programming, an area that we believe deserves full support to extend the practical reach of certified programming. We have analyzed the principles, objectives, and challenges of ICP, providing a formal framework based on completion refinement and completeness to support incremental development. We have illustrated how the IncRease prototype, implemented in Rocq, can be applied to a large project (CompCert), as well as to random property-based testing of partial programs (QuickChick). Additionally, we have demonstrated how to make deductive synthesis compatible with ICP using a novel incrementality-friendly adaptation of the Fiat library. There are still many avenues for future work related to ICP, including developing new incompleteness mechanisms, analyzing their opt-in/out costs and suitability to different settings, improving automation, studying more advanced properties and reasoning frameworks, as well as further integrating these techniques with program synthesis and proof generation, among others.

Data-Availability Statement

The Rocq formalization of IncRease, as well as the different examples and case studies discussed in this article, are available in the archived version at Zenodo [Díaz et al. 2025].

Acknowledgments

This work is partially funded by Inria Équipe Associée GRAPA and ANID/Doctorado Nacional/2022-21221100.

References

Arpan Agrawal, Emily First, Zhanna Kaufman, Tom Reichel, Shizhuo Zhang, Timothy Zhou, Alex Sanchez-Stern, Talia Ringer, and Yuriy Brun. 2023. Proofster: Automated formal verification. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 26–30. doi:10.1109/ICSE-Companion58688.2023.00018

Martin Bodin, Arthur Chargueraud, 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 (POPL 2014)*. ACM Press, San Diego, CA, USA, 87–100. doi:10.1145/2535838.2535876

Martin Bodin, Tomás Diaz, and Éric Tanter. 2018. A Trustworthy Mechanized Formalization of R. In *Proceedings of the 14th ACM Dynamic Languages Symposium (DLS 2018)*. ACM Press, Boston, MA, USA, 13–24. doi:10.1145/3276945.3276946

Olivier Boite. 2004. Proof reuse with extended inductive types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 50–65. doi:10.1007/978-3-540-30142-4_4

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*, Y. Bertot and V. Vafeiadis (Eds.). ACM Press, Paris, France, 182–194. doi:10.1145/3018610.3018620

Sylvain Boulmé. 2007. Intuitionistic Refinement Calculus. In *Typed Lambda Calculi and Applications, 8th International Conference (Lecture Notes in Computer Science, Vol. 4583)*, Simona Ronchi Della Rocca (Ed.). Springer, Paris, France, 54–69. doi:10.1007/978-3-540-73228-0_6

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda - A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, Munich, Germany, 73–78. doi:10.1007/978-3-642-03359-9_6

Edwin Brady. 2013. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 133–144. doi:10.1145/2500365.2500581

Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs))*, Manu Sridharan and Anders Møller (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Aarhus, Denmark, 9:1–9:26. doi:10.4230/LIPIcs.ECOOP.2021.9

Adam Chlipala. 2013. *Certified Programming with Dependent Types.* MIT Press. doi:10.7551/mitpress/9153.001.0001

Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*. IEEE Computer Society Press, Pittsburgh, Pennsylvania, 51–65. doi:10.3233/JCS-2009-0393

Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. 2021. The taming of the rew: a type theory with computational assumptions. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 60:1–60:29. doi:10.1145/3434341

Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for dependent type theory. *Journal of automated reasoning* 61 (2018), 423–453. doi:10.1007/s10817-018-9458-4

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013. Meta-theory à la carte. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013.* ACM Press, Rome, Italy, 207–218. doi:10.1145/2429069

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*. ACM Press, Mumbai, India, 689–700. doi:10.1145/2676726.2677006

Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2014. QuickChick: Property-Based Testing for Coq. In *Coq Workshop.*

Ivan Di Liberti, Fosco Loregiàn, Chad Nester, and Pawel Sobocinski. 2021. Functorial semantics for partial theories. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. doi:10.1145/3434338

Robert W. Dockins. 2012. *Operational Refinement for Compiler Correctness.* Ph. D. Dissertation. Princeton University.

Tomás Díaz, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. 2025. Incremental Certified Programming. doi:10.5281/zenodo.16913455 Archived version of the submitted artifact.

Joseph Eremondi, Éric Tanter, and Ronald Garcia. 2019. Approximate Normalization for Gradual Dependent Types. *Proceedings of the ACM on Programming Languages* 3, ICFP (Aug. 2019), 88:1–88:30. doi:10.1145/3341692

Martín Hötzel Escardó and Cory M. Knapp. 2017. Partial Elements and Recursion via Dominances in Univalent Type Theory. In *26th EACSL Annual Conference on Computer Science Logic, CSL 2017, August 20-24, 2017, Stockholm, Sweden (LIPIcs, Vol. 82)*, Valentin Goranko and Mads Dam (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Stockholm, Sweden, 21:1–21:16. doi:10.4230/LIPICS.CSL.2017.21

Marcelo P. Fiore. 1994. *Axiomatic domain theory in categories of partial maps.* Ph. D. Dissertation. University of Edinburgh, UK. https://hdl.handle.net/1842/406

Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1229–1241. doi:10.1145/3611643.3616243

Yannick Forster and Kathrin Stark. 2020. Coq à la carte: a practical approach to modular syntax with binders. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*. New Orleans, Louisiana, USA, 186–200. doi:10.1145/3372885.3373817

Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. 11–20. doi:10.1109/SP.1982.10014

Son Ho and Clément Pit-Claudel. 2024. Incremental Proof Development in Dafny with Module-Based Induction. *arXiv preprint arXiv:2401.16233* (2024). doi:10.48550/arXiv.2401.16233

Kush Jain, Karl Palmskog, Ahmet Celik, Emilio Jesús Gallego Arias, and Milos Gligoric. 2020. MCoq: Mutation analysis for Coq verification projects. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 89–92. doi:10.1145/3377812.3382156

Ende Jin, Nada Amin, and Yizhou Zhang. 2023. Extensible Metatheory Mechanization via Family Polymorphism. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1608–1632. doi:10.1145/3591286

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *Journal of Functional Programming* (2018). doi:10.1017/S0956796818000151

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 207–220. doi:10.1145/1629575.1629596

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370. doi:10.1007/978-3-642-17511-4_20

Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. 2022. Gradualizing the Calculus of Inductive Constructions. *ACM Transactions on Programming Languages and Systems* 44, 2 (June 2022). doi:10.1145/3495528

Yann Leray, Gaëtan Gilbert, Nicolas Tabareau, and Théo Winterhalter. 2024. The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant. In *Proceedings of the 15th International Conference on Interactive Theorem Proving (ITP 2024)*, Y. Bertot, T. Kutsia, and M. Norrish (Eds.). Leibniz International Proceedings in Informatics (LIPIcs), 26:1–26:18. doi:10.4230/LIPIcs.ITP.2024.26

Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. doi:10.1007/s10817-009-9155-4

Kenji Maillard, Meven Lennon-Bertrand, Nicolas Tabareau, and Éric Tanter. 2022. A Reasonably Gradual Type Theory. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 931–959. doi:10.1145/3547655

Carroll Morgan. 1994. *Programming from specifications, 2nd Edition.* Prentice Hall, Oxford, United Kingdom.

Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*. Springer, 625–635. doi:10.1007/978-3-030-79876-5_37

Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. 73:1–73:30 pages. doi:10.1145/3236768

Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer. doi:10.1007/3-540-45949-9

Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. ACM Press, Paris, France, 86–99. doi:10.1145/3009837.3009900

Pierre-Marie Pédrot and Nicolas Tabareau. 2018. Failure is Not an Option - An Exceptional Type Theory. In *Proceedings of the 27th European Symposium on Programming Languages and Systems (ESOP 2018) (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer-Verlag, Thessaloniki, Greece, 245–271. doi:10.1007/978-3-319-89884-1_9

Pierre-Marie Pédrot, Nicolas Tabareau, Hans Fehrmann, and Éric Tanter. 2019. A Reasonably Exceptional Type Theory. *Proceedings of the ACM on Programming Languages* 3, ICFP (Aug. 2019), 108:1–108:29. doi:10.1145/3341712

Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. 2015. *Software Foundations*. Electronic textbook. http://www.cis.upenn.edu/ bcpierce/sf.

Josselin Poiret, Gaëtan Gilbert, Kenji Maillard, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau, and Éric Tanter. 2025. All Your Base Are Belong to Us: Sort Polymorphism for Proof Assistants. *Proceedings of the ACM on Programming Languages* 9, POPL (Jan. 2025), 76:1–76:29. doi:10.1145/3704912

John C. Reynolds. 1983. Types, abstraction, and parametric polymorphism. In *Information Processing 83*, R. E. A. Mason (Ed.). Elsevier, 513–523.

Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, Zachary Tatlock, et al. 2019a. QED at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages* 5, 2-3 (2019), 102–281. doi:10.1561/2500000045

Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. 2021. Proof repair across type equivalences. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 112–127. doi:10.1145/3453483.3454033

Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2019b. Ornaments for proof reuse in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ITP.2019.26

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 (2020), 947–999. doi:10.1007/s10817-019-09540-0

Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher-Order Logics*. Montreal, Canada, 278–293. doi:10.1007/978-3-540-71067-7_23

Enrico Tassi. 2018. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect). In *The Fourth International Workshop on Coq for Programming Languages*. Los Angeles (CA), United States.

The Rocq Development Team. 2024. *The Rocq prover reference manual*. https://rocq-prover.org/refman Version 8.20.

Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter Mosses. 2022. Intrinsically-typed definitional interpreters à la carte. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022). doi:10.1145/3563355

Philip Wadler. 1998. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

Niklaus Wirth. 1971. Program development by stepwise refinement. *Commun. ACM* 14, 4 (April 1971), 221–227. doi:10.1145/362575.362577

Doug Woos, James R Wilcox, Steve Anton, Zachary Tatlock, Michael D Ernst, and Thomas Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*. 154–165. doi:10.1145/2854065.2854081

Yongwei Yuan, Scott Guest, Eric Griffis, Hannah Potter, David Moon, and Cyrus Omar. 2023. Live Pattern Matching with Typed Holes. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 609–635. doi:10.1145/3586048

Eric Zhao, Raef Maroof, Anand Dukkipati, Andrew Blinn, Zhiyi Pan, and Cyrus Omar. 2024. Total Type Error Localization and Recovery with Holes. *Proc. ACM Program. Lang.* 8, POPL (2024), 2041–2068. doi:10.1145/3632910

Received 2025-03-25; accepted 2025-08-12