Languages, Tools, and Interfaces for Safer Code Evolution

A DISSERTATION PRESENTED
BY
ANASTASIYA KRAVCHUK-KIRILYUK
TO
THE HARVARD JOHN A. PAULSON SCHOOL OF ENGINEERING AND APPLIED SCIENCES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
APRIL 2026

Languages, Tools, and Interfaces for Safer Code Evolution

Abstract

Evolution is the essence of software. This work argues that the burden of code evolution should be addressed at the level of languages and tools, not individual programmers. We discuss this goal through the lens of three complementary contributions. First, we present Persimmon, a functional language design with built-in extensibility features, such as extensible variant types and extensible pattern matching, powered by *nested family polymorphism*. Persimmon highlights how the support for code evolution, and specifically extensible programming, can be built into languages by design. Second, we focus on the generative power of large language models to assist in the verification of evolving software. We present *Dafny Sketcher*, a neurosymbolic framework for automated Dafny proof synthesis, and use it to show a scaffolding paradox: building on proof skeletons from reference solutions does not meaningfully improve proof generation. We explore some mitigating strategies to assess the effect of guidance when coupled with a solution sketch. Finally, we consider the human aspects of automated proof synthesis, by exploring how users interact with different tool interfaces during a proof task. Our study highlights some tradeoffs that users contend with between interface usefulness and output reliability. These results inform future designs of tool interfaces enabling co-evolution of code and proofs. Together, these contributions showcase how safer code evolution can live within our languages, tools, and interfaces.

Contents

To my family.

Acknowledgments

To my wonderful advisor, Nada. The joy you bring to research is simply contagious. From our very first meeting, I knew this would be an incredibly fun time. You were brimming with ideas and I did my best to keep up with your boundless energy. Thankfully, you shared many of your thoughts through Google Chat (I believe it was still called Hangouts at that time), so I could go back and reflect on them. Our work together has truly been the best hangout. No matter what I brought into our meetings, we figured out the next steps together, and you made the time to dig into the details with me. Thank you for always listening, and for making it so easy to share about the good things and the hard ones. Thank you for teaching me to take on the challenges before I can overthink them. Thank you for being the best mentor and a great friend.

A huge thank you to my committee members, Stephen Chong and Talia Ringer, for their thoughtful advice and encouragement. I recall our meeting at the end of my fifth year, when it really hit me that things were wrapping up. Your advice to explore fun research threads and to collaborate more in my final year really framed how I approached this time. When working so deeply on a project, it's easy to get stuck in the details, but our conversations and your feedback on my writing have helped me see the bigger picture. I have a much better understanding of how my work fits into the world because of you. Thank you.

I would also like to thank my collaborators, for devoting their time and insights to our shared projects. Through this work, I've had the privilege of meeting so many amazing researchers who have been a source of inspiration and support. It has been such a pleasure to learn from all of you.

A special thank you to my friends. Harvard has brought me some of the best friends anyone could ask for. Thank you for all the group dinners, game nights, lunch dates, and Easter egg hunts. Not

everyone gets to call their lab family, but that's how I've felt the whole time. To my friends beyond Harvard, thank you for staying constant through all the journeys our lives have taken. Seeing you grow and forge your own paths has given me so much courage. Thank you for being the coolest role models.

To my family. Thank you for being impressed by everything I do and encouraging my wildest dreams. I am so grateful that I got my love of learning from you, and that you made it possible for me to follow this path. Thank you for never suggesting I've been a student for too long. Thank you for humoring my attempts to explain my work in our native language. I tried my best but I know it made no sense. Thank you for your quiet support, your vocal support, and for always keeping the door open.

To my husband, George. You have been here every step of the way. Our Boston adventure was a leap of faith, and you have really stepped up to the challenge of making this place feel like home. You have been my biggest fan, my grounding presence, and my safe space. Thank you for staying up late when I needed a pep talk, for getting me through all the deadlines, and for picking up takeout at just the right moment. Thank you for your kind reassurance, especially when the finish line seemed miles away. Thank you for believing I can do anything.

0

Introduction

This work is a culmination of my deep belief that the brunt of ever-evolving software should not fall on the programmers. As programmers, we know that a living codebase is never finished: bug fixes, feature additions, and specification changes can continue ad infinitum. Since there is no hope that software will ever stop changing, it is our job as programming language researchers to support programmers in their evolving mission. Instead of individual programmers, the burdens of code evolution should be delegated to the *tools and languages themselves*. Now, how do we make that happen?

The contributions of this work highlight two ways that languages and tools can support safer code evolution. The first approach is *safety by construction*: safety properties can be baked directly

into language and tool design, so that unsafe programs are eliminated entirely (i.e., cannot be constructed). This approach often manifests in the form of rich type systems, where type checking of programs serves as the static guardian of safety guarantees. Two well-known examples of such languages are Rust with its ownership model for memory safety, and Ada with its built-in bounds checking [Matsakis & Klock, 2014; Taft et al., 2014]. The second approach is *formal verification*: the intended behavior of a program can be formally specified (for example, as a logical proposition, or using Hoare-style pre- and postconditions), and any code changes can be verified against those specifications to ensure that the defined safety properties are preserved. Theorem provers such as Rocq and Lean are essential to this approach, providing the infrastructure for mechanized proof developments that can co-evolve along with the code itself [The Rocq Development Team, 2024; de Moura et al., 2015].

This dissertation meets the challenge of supporting code evolution through three complementary angles: (i) the languages we build to support safer extensible programming by design, (ii) the tools we build to support the automated verification of program properties as code evolves, and (iii) the interfaces through which programmers interact with supportive tools. The first contribution follows the safety-by-construction approach, building the safety guarantees for evolving programs into the language itself. The second and third contributions follow the formal verification approach, addressing both the technical considerations of automated proof synthesis as well as the human aspects of making such automation tools usable in practice. Each contribution is presented in a separate chapter.

Languages for Extensible Programming. The tension between extensible programming and safety is long well-known, particularly in the form of the Expression Problem [Wadler, 1998]. Extensibility in this sense manifests as adding new constructors to data types, along with new func-

tionality over those data types. Existing programming paradigms excel when only one dimension of extensibility is present, but combining both is more complicated. One solution to this problem is building languages in which extensibility is a foundational design principle, as opposed to an afterthought. This is the approach we take in Chapter 1.[1] We present PERSIMMON, a functional language design with built-in extensibility features, powered by *nested family polymorphism*. Families in PERSIMMON are extensible units that relate sets of types and functionality over those types, which can be inherited and specialized in derived families. Since families can be nested arbitrarily, this design supports coordinated evolution of modular software frameworks. The central features of our design are extensible variant types and extensible top-level `cases` constructs, which support type-safe extensibility in both dimensions (types and functionality). We prove the soundness of our system via progress and preservation. This work demonstrates how code evolution can be supported intrinsically in a language design, shifting the burden from programmers to the language itself.

Tools for Verification of Evolving Code. In addition to writing correct code, another challenge is mechanizing the formal proofs of a system's properties, and co-evolving the code and the proofs over time. Interactive and automated theorem provers help in this task, but they are notoriously rigid with respect to verifying evolving systems. Usually, any change in the program yields a specification change in the mechanization, and propagates through the proofs in the form of new proof obligations. Formal verification effort, therefore, is expensive in both labor cost and human patience. The rise of large language models (LLMs) has been a beacon of hope for verification engineers everywhere, offering an opportunity to automate much of the verification process. One day, LLMs might assist with formalization and verification, and keeping proofs up-to-date with

[1]This chapter is based on work previously published as: [Kravchuk-Kirilyuk et al., 2024]

evolving codebases. In order to get there, researchers have been working on figuring out what kind of symbolic help is useful to LLMs in proof generation. One consistent assumption across the literature has been that partial proofs are helpful, and giving an LLM an on-track proof sketch should aid proof synthesis. In Chapter 2, we present *Dafny Sketcher*, a neurosymbolic framework for Dafny proof synthesis, and use it to examine this assumption empirically.[2] We discover *the scaffolding paradox*: providing an LLM with an on-track proof skeleton from a reference solution does not meaningfully improve Dafny proof generation. Given this surprising finding, we explore two mitigation strategies, case repair and process supervision via prompting, to better understand how guidance (along with the partial solution) can affect proof synthesis in this instance. This work investigates how LLMs may interact with more symbolic methods aiding proof synthesis, as we strive to automate the verification of evolving software.

Human Interaction with Automation Tools. We also explore another aspect of automated proof synthesis: human interaction with the tools themselves via tool interfaces. The success of human-in-the-loop automated proof synthesis depends not just on the effectiveness of the tools, but also on how easily humans can convey their intent to the tools and parse their feedback. In Chapter 3, we report our qualitative results from a user study where participants were observed solving proof tasks in Rocq using two helper tools: a web-based LLM interface, Claude, and a search-based proof synthesis tool, Proofster.[3] The different considerations of using both interfaces, such as the unreliability of Claude's output combined with a comfortable natural language interface, versus Proofster's rigid interface that guarantees correctness by construction, raise important questions about user trust, interface accessibility, and verification strategy in presence of automated helper tools. This work informs the design of future tool interfaces that will support the co-evolution of

[2]This chapter is based on work currently under submission: [Kravchuk-Kirilyuk et al., 2026b]

[3]This chapter forms the basis of a manuscript under development: [Kravchuk-Kirilyuk et al., 2026a]

code and proofs.

Together, these contributions paint a picture of a multifaceted approach to safer code evolution: from built-in language mechanisms supporting extensibility, to insights for LLM-powered tools that may one day automate proof maintenance for evolving code, to considerations for the tool interfaces that humans interact with in practice. They also inspire a few broader lessons that cut across the three facets. First, safety is not just a surface feature, it is a built-in system property that needs underlying structural support. One way to create this structure within systems is by carefully choosing expressive, unifying abstractions. Not every abstraction is helpful, but a fitting abstraction can help tame complexity and provide support in reasoning about system properties. Second, the right supportive structures do not eliminate the complexity of expressing and maintaining safety properties, but shift it to different layers of the system. Even automation only shifts this complexity to the management and coordination of automation tools. Finally, whether or not this shift actually reduces the human burden depends heavily on the tool orchestration strategies and the feedback from tool interfaces. It is not enough to build well-intentioned safety features into languages and tools if humans are not able to use them effectively. Overall, these lessons suggest that safer code evolution is not a problem to be solved by clever engineering alone, but rather an ongoing conversation between the human programmers and the tools they rely on.

1

Persimmon: Extensible Programming via Family Polymorphism

1.1 Introduction

A long-standing challenge in programming language design is supporting modular, extensible software *over time*. As requirements change and programs evolve, developers must be able to introduce extensions without pervasive modification to previously written code. The Expression Problem [Wadler, 1998] characterizes this challenge by highlighting a fundamental tension between two directions of extensibility: adding new constructors to data types, and adding new functionality

over data types. Different programming paradigms address this tension in different ways. In the object-oriented programming style, extending data types with new constructors (e.g., as classes) is straightforward, but adding new functions requires sweeping changes to all constructors. In contrast, the functional programming style facilitates the addition of new functions, but adding new type constructors requires sweeping changes to all functions. When such changes to existing code are infeasible, developers must duplicate code; either way, modularity is lost.

This tension has driven language designers to devise new programming abstractions for code reuse and polymorphism. *Family polymorphism* is one such idea that originates in object-oriented programming [Ernst, 2001; Igarashi et al., 2005; Zhang & Myers, 2017]. It allows extension to occur at the level of *families* of mutually related types. All code is *polymorphic* to the family it appears within, therefore any code defined in a base family can be safely inherited and reused as-is by derived families. A family captures a relationship between types that are intended to evolve together. This allows programmers to extend entire families of related constructs in a coordinated way. Virtual classes [Madsen et al., 1993], virtual types [Thorup, 1997], and nested inheritance [Nystrom et al., 2004] are all forms of family polymorphism.

Importantly, *nested* family polymorphism supports the inheritance and further binding of nested families, enabling large-scale extensibility and code reuse [Ernst, 2003]. Complex software systems such as extensible compilers can be expressed using nested family polymorphism, where the source and target languages are represented as extensible nested components (Figure 1.1). Mixins can also be encoded via nested family polymorphism, supporting the *composition* of large nested systems while avoiding the need for a dedicated mixin construct.

However, the expressive potential of nested family polymorphism has yet to be fully realized in functional programming language design. Although associated types [Chakravarty et al., 2005] in Haskell are inspired [Peyton Jones, 2009] by virtual types, they do not provide the same level

of extensibility that nested family polymorphism can offer in the object-oriented setting. FPOP introduces top-level family polymorphism into the functional setting, but does not support the inheritance and extension of nested families [Jin et al., 2023].[1] Compositional programming [Zhang et al., 2021] does support nested inheritance, but limits type declarations to the top-level, lacking the expressive power types can achieve as mutually recursive family members.

Other systems may support nested inheritance, but not extensible variant types [Nystrom et al., 2004; Ernst et al., 2006; Igarashi et al., 2005; Zhang & Myers, 2017]. Variant types (also known as algebraic data types) are central to functional programming; they are the primary way to allow variations in the data representation of a type. The elimination form of variants is pattern matching, which often results in more concise code than achievable in the object-oriented style through the Visitor pattern [Gamma et al., 1994]. We consider it critical that a deeper integration of nested family polymorphism into functional languages should support extensible variant types. A language design should allow a derived family to add new constructors to variant types declared in its base family, and support the extension of pattern match expressions with new cases.

One difficulty in supporting extensible variant types is the tension between extensibility and type safety. Type safe code must check *exhaustivity* of pattern matching – there must exist a match case for each variant. In the presence of extensions, exhaustivity checking involves both the inherited definitions from the base family *and* their extensions in the derived family. Nested inheritance makes exhaustivity checking even harder, as we must allow for the possibility that pattern match expressions inherited from families *nested within* the base family may not be exhaustive in the derived family.

We reconcile this tension by introducing `cases` constructs, nominal pattern matching expressions that are polymorphic to their enclosing families. Since `cases` definitions are family members, they

[1] Rocqet, a recent extension to FPOP, does support nested families [Ebresafe et al., 2025].

can be extended directly in the derived family, mirroring our extensible variant types. Our approach also simplifies exhaustivity checking even in the presence of nested inheritance: `cases` are checked to be exhaustive as part of a well-formedness check for family members. The type of a `cases` construct reflects all handled constructors of the scrutinee type, eliminating the need to access definitions from the base family.

1.1.1 Design Considerations

A variety of approaches have been proposed for extensible functional programming. Our work builds on this line of research by addressing additional design goals that aim to better harness the expressive power of extensibility in systems with nested components. In particular, we pursue a language design that satisfies the following goals, in addition to the classic requirement of **type safety**.

- **Extensibility at scale**. While many solutions focus only on extensibility of *small* code units like classes, traits, and functions, we recognize that module and namespace mechanisms carry a convenient organizational advantage in large software developments. The ability to coevolve components of *arbitrarily large* code units (namely, modules that nest modules) will enable the programmer to create *extensible software frameworks* with ease.

- **Scalable extensibility**. In addition to extensibility *at scale* (i.e., supporting large code bases and nesting), a solution should also be *scalable*. Scalable solutions allow engineers to rapidly develop and extend code bases as software evolves. Little bookkeeping should be required of the programmer before an extension is introduced, and the effort to implement an extension should be proportional to the delta in program functionality. One common way to address extensibility is by explicitly parameterizing a unit of code with extensibility hooks. However, solutions of this flavor tend to require code to be written differently in the absence and presence

of future extensions. They lead to *parameter clutter* for large code units with interdependent components, reducing the scalability of the extensibility mechanism.

➢ **Mutual recursion**. The solution should support unrestricted, mutually recursive references between constructs in different components of the program. Complex systems with nested components – such as extensible compilers – rely on this feature.

➢ **Composable extensions**. In addition to being possible, extensions should be composable. The language should support composing extensions (and even families of extensions).

➢ **Idiomatic functional style**. The programming experience should not feel foreign to the working functional programmer. It should also be friendly to the novice programmer unaware of extensibility concerns.

1.1.2 Contributions

In this chapter, we discuss the following contributions of our work:

- We present a type-safe language design, Persimmon, that supports extensible variant types and pattern matching via nested family polymorphism.[2] The design is based on a simple functional core and is applicable to other functional languages with declared types. The core calculus of Persimmon provides a basis for integration of our family polymorphic mechanism into statically typed functional languages. We prove the soundness of the type system.

- We showcase the expressive power of our design and its applicability to real programming challenges, such as extensible compilers. Our examples show that our language design meets our design goals.

[2]Our implementation of the Persimmon type checker is available at https://github.com/akravc/persimmon.

- We show how the powerful extensibility mechanism can be compiled into a functional language without extensible variants via our prototype compiler from Persimmon to Scala.

1.2 Motivation

We begin with an example that shows how powerful (and practical!) the combination of nested family polymorphism and extensible variant types can be. Consider a compiler that can (1) transform simply-typed lambda calculus (STLC) terms into continuation passing style (CPS) and (2) transform CPS-converted terms using closure conversion. The target languages for CPS and closure conversion share some parts, so to achieve better code reuse and modularity both target languages may need to extend a shared, intermediate language. Figure 1.1 shows the components of such a compiler, here named `BaseComp`: the source language (`STLC`), the shared language `IL`, the target language of CPS ($\mathsf{IL_K}$, which extends `IL`), and the target language of closure conversion ($\mathsf{IL_C}$, which extends `IL`). Importantly, these language components are nested within the compiler.

Already, we recognize the need for extensible variant types in this scenario. In the functional programming approach, the types, values, and expressions of the target languages $\mathsf{IL_K}$ and $\mathsf{IL_C}$ are represented as algebraic data types (ADTs). Since both $\mathsf{IL_K}$ and $\mathsf{IL_C}$ extend the intermediate language `IL`, the constructs in $\mathsf{IL_K}$ and $\mathsf{IL_C}$ are *extensions* of constructs in `IL`. If our system does not support extensible ADTs, our capacity for code reuse is limited. For example, functions that operate on values `Val` in `IL` could not automatically adapt to operate on extended values `Val` in $\mathsf{IL_K}$ or $\mathsf{IL_C}$. Extensible variant types are therefore crucial for supporting code reuse across related languages.

In our solution, we implement extensible variant types via family polymorphism, thereby ensuring that constructs defined within each family are polymorphic to the family. This is especially useful when multiple types must co-evolve in a compatible manner as the code is extended. For example,

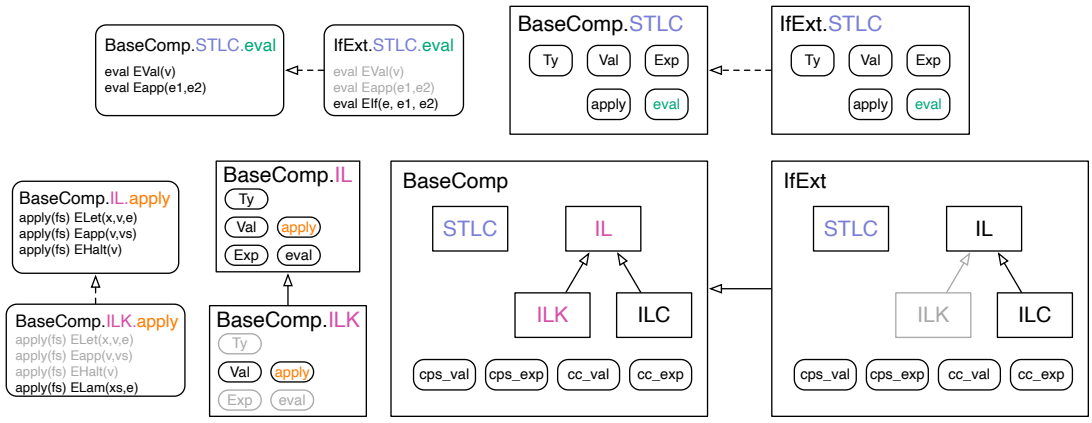

Figure 1.1: Motivating example: extensible compilers with nested inheritance.

the ADT that defines intermediate expressions `Exp` in IL may reference the type `Val`, representing values. However, when the definition for `Val` is extended in IL_K, we need not redefine `Exp`. The inherited definition for `Exp` implicitly refers to the extended type `Val` via a relative path. Family polymorphism thus lets us seamlessly reuse inherited code and co-evolve related types safely.

So far, we have considered a single compiler and its language components. Now, let us examine the case where the source language `STLC` is extended with if-expressions. If nested inheritance of compiler components (such as `STLC`) is not supported, we may need to build a new compiler for each extension of `STLC`, making only minor changes between versions. We would much rather have an extensible compiler instead. Persimmon supports this through *nested family polymorphism*: families containing nested families can be extended, while preserving the structural and hierarchical relationships between them. The benefit of our solution can be seen in Figure 1.1. There are two compilers pictured, the `BaseComp` compiler for base `STLC`, and the `IfExt` compiler for `STLC` with if-expressions. Instead of existing as a fully separate implementation, the compiler `IfExt` extends `BaseComp`, and in turn, its component languages `STLC`, `IL`, and `ILC` extend their counterparts in `BaseComp`. Nested family polymorphism in Persimmon enables extensibility at the scale of large code units with complex structure. We discuss this example in more detail in Section 1.4.2.

We can take our motivating example even further to highlight the importance of composable extensions. With composable extensions, we can create compilers for versions of `STLC` with arbitrary combinations of features (for example `STLC` with if-expressions, let-expressions, and references). Persimmon supports composable extensions via a mixin encoding, further detailed in Section 1.4.3.

1.3 Background

Before diving into a discussion of the language features that support extensibility in Persimmon, we provide the necessary background on the theoretical concepts that underlie those features. We first introduce *family polymorphism*, the programming paradigm that supports extensibility at the level of families of related types. We then discuss *relative path types*, the mechanism we use as part of our type system to realize family polymorphism. The substitution of relative paths is the vehicle by which inherited code adapts to its new context in the derived family. Finally, we discuss *linkages*, the special data structures that Persimmon uses to store family contents. Linkages reflect the nested nature of families in Persimmon, making it easy to perform concatenation of inherited and extended constructs, and to perform path substitution.

1.3.1 Family Polymorphism

Family polymorphism, introduced by Ernst [2001], is a programming paradigm in which mutually dependent types are grouped into *families*, and all code within a family is *polymorphic* to the family in which it appears. This means that when a family is extended, the code from the base family can be safely used in the derived family without modification. Importantly, these mutually dependent types (members of the family) can co-evolve safely over time: when extending one type causes changes to be made to another type, these changes are made in tandem, and type

```
class Ecosystem {
  class Prey {
    void flee() { }
    void eat() { }
  }

  class Predator {
    Prey target;
    void hunt(Prey prey) { }
  }

  class Habitat {
    Predator[] predators;
    Prey[] preyAnimals;
    void simulate() { }
  }
}
```

```
class SavannaEcosystem extends Ecosystem {
  (* This is a savanna−specific Prey (zebra, gazelle, etc.) *)
  class Prey {
    void flee() { (* Run across plains *) }
    void eat() { (* Graze on grass *) }
    void camouflageInTallGrass() { }
  }

  (* This is a savanna−specific Predator (lion, cheetah, etc.) *)
  class Predator {
    void hunt(Prey prey) {
      (* Chase across open terrain *)
      (* Use pack tactics *)
    }
    void roar() { }
  }

  class Habitat {
    void simulate() {
      (* Dry and wet season cycles *)
      (* Watering hole interactions *)
    }
  }
}
```

Figure 1.2: A base class `Ecosystem` with related nested classes, `Prey` and `Predator` (left). An extension `SavannaEcosystem`, which further specifies `Prey` and `Predator` behavior (right).

safety is maintained by only allowing compatible family members to interact with one another. By treating the family as the unit of extension, family polymorphism supports compatible extension of interdependent types and functions.

Ernst [2001] formalized family polymorphism in an object-oriented setting, where each family is represented as an enclosing object, and the members of the family are represented as object attributes. On the other hand, Persimmon follows the class-based approach to family polymorphism, formalized in .FJ by Igarashi et al. [2005], where a family is associated with a class rather than an object instance. Family members are represented as nested classes. Let us walk through a short example of class-based family polymorphism to get an intuition for how it supports type safe co-evolution of related types.

Consider the code in Figure 1.2. On the left hand side, we have our base class `Ecosystem`. Within the `Ecosystem`, we have related types, such as `Predator` and `Prey`. These two types are related,

```
class OceanEcosystem extends Ecosystem {
  (* This is ocean-specific Prey (fish, squid, etc.) *)
  class Prey {
    void flee() { (* Swim in schools *) }
    void eat() { (* Filter plankton *) }
    void diveDeep() { }
  }

  (* This is ocean-specific Predator (shark, orca, etc.) *)
  class Predator {
    void hunt(Prey prey) {
      (* Use electroreception *)
      (* Ambush from below *)
    }
    void swimInCurrents() { }
  }

  class Habitat {
    void simulate() {
      (* Tidal patterns *)
      (* Coral reef interactions *)
    }
  }
}
```

Figure 1.3: An extension `OceanEcosystem`, which specifies different `Prey` and `Predator` behavior as compared to `SavannaEcosystem`.

because predators hunt prey. We also have a type for `Habitat`, which depends on `Predator` and `Prey`. This general definition of `Ecosystem` does not yet define the specific behavior of each type of `Ecosystem`, but serves as a blueprint for future extensions.

Now, on the right hand side in Figure 1.2, we have a derived class `SavannaEcosystem`, which extends the base `Ecosystem`, and further specifies the behavior of its family members. For example, prey can `camouflageInTallGrass()`, and predators can `roar()`. This is behavior specific to the `SavannaEcosystem`. If we did not plan on having parallel extensions to `Ecosystem`, it may have been enough to have a subtyping relationship here, where an instance of `SavannaEcosystem` could be used anywhere that `Ecosystem` was expected. However, when parallel extensions are at play, things get a little more complicated.

Consider a parallel extension to `Ecosystem`, `OceanEcosystem`, shown in Figure 1.3. The predators and prey of this ecosystem have different specialized behaviors, such as `diveDeep()` and

```
OceanEcosystem ocean;
OceanEcosystem.Predator orca;
OceanEcosystem.Prey fish;

orca.hunt(fish);   (* This works! Orca can hunt a fish. *)

SavannaEcosystem savanna;
SavannaEcosystem.Prey zebra;

orca.hunt(zebra);   (* Compile time error: Orca can't hunt a zebra on land. *)
```

Figure 1.4: Example of a compile time error.

`swimInCurrents()`. While `OceanEcosystem` and `SavannaEcosystem` exist at the same time, we do not expect the predators and prey of these ecosystems to interact with their counterparts. The defined behavior for `Predator` and `Prey` only makes sense within the intended ecosystem (the type family).

Ultimately, family polymorphism strives to prevent interactions between incompatible family members. Consider the code snippet shown in Figure 1.4. Family polymorphism ensures that the `Prey` and `Predator` types that are interacting through the `hunt()` function call are compatible family members: `OceanEcosystem.Prey` with `OceanEcosystem.Predator`, or `SavannaEcosystem.Prey` with `SavannaEcosystem.Predator`. When they are not compatible, such as on the last line of the code snippet, the error is detected at compile time, not at runtime. Finally, it is important to note that in this setting, a subtyping relationship between the base class and a derived class is no longer safe. If `SavannaEcosystem.Prey` was a subtype of `Ecosystem.Prey`, then an instance of `SavannaEcosystem.Prey` could be passed in place of `Ecosystem.Prey` as a parameter to `hunt()`. This would allow undesirable interactions to happen at runtime, since expressions such as `orca.hunt(zebra)` would not throw a compile time error.

1.3.2 Relative Path Types

A classic approach to implementing class-based family polymorphism is through the use of relative path types, as formalized in .FJ [Igarashi et al., 2005]. A relative path type uses a special keyword (such as `self`) within the type path to refer to the "current" family, instead of naming a concrete family. Since the path is relative, any code containing this type that is inherited into a derived family will automatically resolve the path to refer to the (possibly extended) version of this type in the derived family. Consider the following code snippet, which is an annotated version of the class `Ecosystem` in Figure 1.2.

```
class Ecosystem {
  class Prey { ... }
  class Predator {
    self(Ecosystem).Prey target;
    void hunt(self(Ecosystem).Prey prey) { ... }
  }
}
```

Note that within the `Predator` class, the type `Prey` has an implied relative path prefix highlighted in orange: `self(Ecosystem)`. This means that instances of `Predator` are only allowed to interact with instances of `Prey` from the same enclosing class (or family). Within the class `Ecosystem`, the relative path prefix `self(Ecosystem)` resolves simply to `Ecosystem`. However, when this code is inherited into `SavannaEcosystem`, the `self(Ecosystem)` path now refers to `SavannaEcosystem`, which is the "current" extension of `Ecosystem`. This automatic adaptation prevents unsound interactions between incompatible extensions.

The mechanism that enables resolution of relative paths to the appropriate concrete path prefix in our system is path substitution. When code is inherited from a base family into a derived family, all relative paths that refer to the base family are substituted with relative paths referring to the derived family. This substitution happens as many times as needed along the inheritance chain. Path

substitution is also complicated by the presence of nested families: a substitution applied to an outer family needs to propagate recursively into nested families. Since path substitution is an essential component for extensibility in our system, it's important that the underlying representation for families and their contents naturally supports substitution. In Persimmon, we represent families using linkages.

1.3.3 Linkages

Linkages are supporting data structures that are similar to global class tables. In object oriented systems, a class table is a common choice for a mapping of class names to the definitions within that class, such as its fields, and methods. Class tables enable the lookup of type-level information during type checking, and the lookup of definitions during evaluation. They work well in a setting where class members can be resolved by name globally. For example, a class `A` and a nested class `X` within `A` would both be at the top-level of a class table, distinguished by the class name.

$$\texttt{A} \mapsto \{fields_a, methods_a\}$$
$$\texttt{A.X} \mapsto \{fields_x, methods_x\}$$

However, in a language with family polymorphism and nested inheritance, names aren't global: types depend on paths, and path-dependent references are resolved through substitution. Furthermore, class tables are flat maps, and are thus not a fitting representation for nested hierarchies where nested members must be inherited and specialized together. The relationships between enclosing and nested families are lost in a class table.

Linkages, on the other hand, are designed to handle the additional complexity introduced by nesting and relative path types [Zhang & Myers, 2017]. Each class (or family) is represented by a linkage, which is a map of maps representing the contents of that family. The contents include any nested families as well, and so each nested family maps to its own sub-linkage within this enclosing

linkage. The linkage for a class `X` nested in `A` would appear within the linkage for `A`.

$$\texttt{A} \mapsto \{\texttt{X} \mapsto \{fields_x, methods_x\}, fields_a, methods_a\}$$

This nested linkage structure perfectly reflects the nested structure of families, and naturally supports the recursive operations that are necessary for inheritance in this setting. Consider a family `B` that extends `A`, at the same time extending the nested family `X`. When family members from `A` are inherited into `B`, the linkages for the base and derived families can be recursively merged together, modeling inheritance as well as extension and overwriting. Path substitution is naturally supported by propagating the operation to any nested family linkages.

$$\texttt{A} \mapsto \{\texttt{X} \mapsto \{fields_x, methods_x\}, fields_a, methods_a\}$$
$$\texttt{B} \mapsto \{\texttt{X} \mapsto \{fields_{x'}, methods_{x'}\}, fields_b, methods_b\}$$

1.4 Language Features

In this section, we present the key language features of Persimmon via case studies. We highlight the support for extensible variant types, extensible pattern matching, nested families, and mixins in Persimmon. The case studies presented here also serve as a gentle introduction to our language. We hope to develop an intuition behind the different features of our language and how they are expressed in the type system before formally introducing our language design in Section 1.5.

1.4.1 Extensible Variant Types and Extensible Pattern Matching

First, we focus on the essentials of our extensibility solution: extensible variant types and pattern matching. Extensible pattern matching in Persimmon sets our solution apart from other family polymorphic systems with nesting such as Familia, which does not support extensible pattern matching [Zhang & Myers, 2017]. We introduce these features with the classic example of a base

```
Family STLCBase {
  type Ty = TUnit | TNat | TArr(t1: Ty, t2: Ty)
  type Val = Unit | Var(x: Str) | Lam(x: Str, e: Exp)
  type Exp = EVal(v: Val) | EApp(e1: Exp, e2: Exp)

  def eval : Exp -> Option Val =
    case EVal(v) = Some v;
    case EApp(e1, e2) =
      (eval e1) >>= (λ v -> apply(e2) v)

  def apply(e2: Exp) : Val -> Option Val =
    case Lam(x, e) = eval (subst x e2 e);
    case _ = None

  ... (* subst, tc, print, etc. *)
}
Family STLCIf extends STLCBase {
  type Ty += TBool
  type Val += True | False
  type Exp += EIf(e: Exp, e1: Exp, e2: Exp)

  def eval : Exp -> Option Val +=
    case EIf(e, e1, e2) =
        (eval e) >>= (λ v -> branch(e1, e2) v)

  def branch(e1: Exp, e2: Exp) : Val -> Option Val =
    case True = eval e1;
    case False = eval e2;
    case _ = None

  ... (* extensions to subst, tc, print, etc. *)
}
```

Figure 1.5: A base lambda calculus (`STLCBase`) and an extension (`STLCIf`) in extended Persimmon syntax.

lambda calculus (STLC) and an extension to STLC, shown in Figure 1.5. For convenience, we use an extended Persimmon syntax in this example.[3]

Let us break down the contents of the program in Figure 1.5. Family `STLCBase` contains the base calculus with natural numbers and unit. Within the family, we declare algebraic data types (ADTs) with the keyword `type`. The ADTs `Ty`, `Val`, and `Exp` represent types, values, and expressions of the base calculus. Each ADT is defined as a set of constructors, where each constructor may have input fields specified in parentheses. For example, the constructor `Var` of type `Val` has a single field `x` of type `Str`, while the constructor `Unit` has no fields. Each function declared within the family has a name, an arrow type, and a definition. If the function involves pattern matching (for example,

[3]We add a base type `Str` for strings, add option types, and omit type annotations on constructor variables within pattern match cases (these annotations could be inferred).

the function `eval` on line 5), we specify each match case separately using the keyword `case`.[4] We support wildcard pattern match cases (marked with _) that match any constructors of the given ADT definition in the current family, and do not apply in a blanket fashion to the extensions of that ADT. Note that the base code looks quite ordinary. This is one advantage of PERSIMMON: no prior setup is required in base families to enjoy extensibility in the derived families. Our code follows the functional programming style familiar to the user.

Family `STLCIf` in Figure 1.5 highlights the elegance of extensible variant types and pattern matching in PERSIMMON. `STLCIf` is an extension that adds booleans and if-expressions to the base calculus. For example, the type `Val` is extended with constructors `True` and `False`, and a new case for the if-expression, `EIf`, is added to function `eval`. All extensions use our extensibility marker `+=` for clarity. PERSIMMON ensures exhaustivity of pattern matching *at definition*: the new case for `EIf` must be specified in `eval`, otherwise the pattern match in the derived family will not pass the well-formedness check. We can also add new types and functionality in derived families, such as the function `branch` in `STLCIf`.

Note that PERSIMMON code is highly modular: the base family only contains the base code, while the derived family only contains the extension code. Base code does not include any scaffolding for future extensions, and is not duplicated in the derived family. This *parsimonious* approach to extensible, user-written code is one inspiration for the name of our language, PERSIMMON.

Our minimalist approach to extensibility relies on *relative path types*. Relative path types ensure that all code is polymorphic to the enclosing family. Each type in Figure 1.5 has an implicit path prefix, which is inferred to be the path to the immediate enclosing family. When code is inherited, any path prefix referring to the base family is substituted by the path prefix referring to the derived family. For example, consider type `Ty` which is inherited by family `STLCIf` and extended with an

[4]We support the in-line case syntax for user convenience, while the underlying representation involves our extensible `cases` constructs, detailed in Section 1.5.

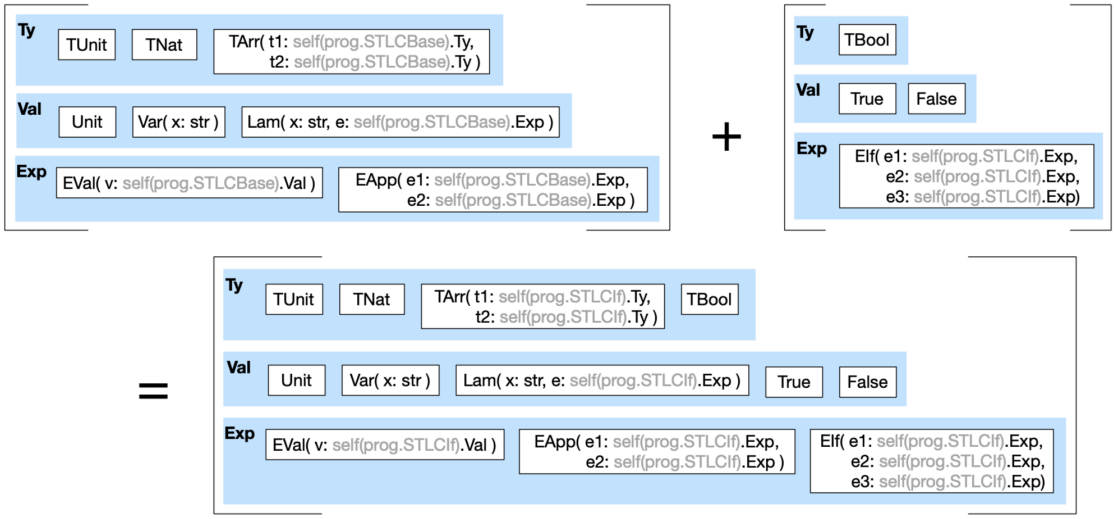

Figure 1.6: Example of linkage concatenation for types, combining both inherited and extended definitions.

extra constructor, `TBool`. Figure 1.6 shows what happens under the hood: the type definition for `Ty` from family `STLCBase` (top left) is *concatenated* with the extension for `Ty` (top right), resulting in the full definition (bottom) for `Ty` in family `STLCIf`. Any self-referencing paths in the base code that refer to the parent family (for example, `self(prog.STLCBase)` in constructor `TArr`) are substituted with paths to the derived family, `self(prog.STLCIf)`. We substitute path prefixes in all inherited code, with the help of map-like data structures called *linkages* (detailed further in 1.5.4).

Our approach has multiple advantages. Relative path types and path substitution support type-safe code reuse in Persimmon. Pattern matching is ensured to be exhaustive at definition. Persimmon code is modular and readable due to a minimal code overlap between the base and derived families. Most constructs in Persimmon are built-in extensibility hooks, eliminating parameter clutter. Finally, Persimmon reduces user effort associated with the advance setup of extensible frameworks, compared to traditional design-pattern approaches.

```
Family BaseComp {
  Family STLC extends STLCBase {}

  (* base intermediate language *)
  Family IL {
    type Ty = TUnit | TCont(ts: List Ty)
    type Val = Unit | Var(x: Str)
    type Exp = ELet(x: Str, v: Val, e: Exp) |
         EApp(v: Val, vs: List Val) | EHalt(v: Val)
    type Fun = MkFun(n: Str, xs: List Str, e: Exp)

    def eval(fs: List Fun): Exp -> Option Val =
      case ELet(x, v, e) = eval(fs) (subst x v e)
      case EApp(v, vs) = apply(fs, vs) v
      case EHalt(v) = Some v

    def apply(fs: List Fun, vs: List Val): Val -> Option Val =
      case _ = None
    ... (* subst, tc, print, etc. *)
  }

  (* target language of CPS *)
  Family ILK extends .IL {
    type Val += Lam(xs: List Str, e: Exp)

    def apply(fs: List Fun, vs: List Val): Val -> Option Val +=
      case Lam(xs,e) = eval(fs) (subst xs vs e)
    ... (* subst, tc, print, etc. *)
  }

  (* target language of closure conversion *)
  Family ILC extends .IL {
    type Ty += TVar(a: Str) | TExist(a: Str, t: Ty)
    type Val += Pack(t: Ty, v: Val) | Name(n: Str)
    type Exp += EUnpack(a: Str, x: Str, v: Val, e: Exp)

    def eval(fs: List Fun): Exp -> Option Val +=
      case EUnpack(a,x,v,e) = unpack(fs,a,x,e) v

    def unpack(fs: List Fun, a: Str, x: Str, e: Exp):
      Val -> Option Val =
        case Pack(t, v) = eval(fs) (subst x v (subst a t e))
        case _ = None

    def apply(fs: List Fun, vs: List Val): Val -> Option Val +=
      case Name(n) =
        let (xs, e) = lookup(fs, n) in eval(fs) (subst xs vs e)
      case Pack(t, v) = None
    ... (* subst, tc, print, etc. *)
  }

  (* CPS translation *)
  def cps_val(k: ILK.Val): STLC.Val -> ILK.Exp =
    ... (* cps_val cases *)
  def cps_exp(k: ILK.Val): STLC.Exp -> ILK.Exp =
    ... (* cps_exp cases *)
  (* closure conversion *)
  def cc_val: ILK.Val -> (List ILC.Fun, ILC.Val) =
    ... (* cc_val cases *)
  def cc_exp: ILK.Exp -> (List ILC.Fun, ILC.Exp) =
    ... (* cc_exp cases *)
} (* end of BaseComp family *)

(* Compiler extension: add if-then-else to STLC and ILs *)
Family IfExt extends BaseComp {
  Family STLC extends STLCIf {}

  Family IL {
    type Ty += TBool
    type Val += Bool(b: B)
    type Exp += EIf(v: Val, e1: Exp, e2: Exp)

    def eval(fs: List Fun): Exp -> Option Val +=
      ... (* new EIf case *)

    def apply(fs: List Fun, vs: List Val):
      Val -> Option Val += ... (* new Bool case *)
    ... (* subst, tc, print, etc. *)
  }

  Family ILC extends .IL {
    def unpack(fs: List Fun, a: Str, x: Str, e: Exp):
      Val -> Option Val +=
        case Bool(b) = None
    ... (* subst, tc, print, etc. *)
  }

  def cps_val(k: ILK.Val): STLC.Val -> ILK.Exp +=
    ... (* new cases *)
  def cps_exp(k: ILK.Val): STLC.Exp -> ILK.Exp +=
    ... (* new cases *)
  def cc_val: ILK.Val -> (List ILC.Fun, ILC.Val) +=
    ... (* new cases *)
  def cc_exp: ILK.Exp -> (List ILC.Fun, ILC.Exp) +=
    ... (* new cases *)
}
```

Figure 1.7: A base STLC compiler and an extension in extended Persimmon syntax.

1.4.2 Nested Families and Inheritance

Next, we explore the powerful interaction between nested families and inheritance in Persimmon. We show how we use nested families to implement the previously discussed motivating example of extensible compilers. We include the partial Persimmon code for this example in Figure 1.7, and an inheritance diagram of all nested components in Figure 1.1.[5]

Persimmon supports arbitrary nesting of families and preserves the nested structure upon inheritance. In Figure 1.7, family `BaseComp` represents the base compiler. The nested families within `BaseComp` represent the language components of the compiler: (1) the source language `STLC`, (2) the

[5]The partial implementation assumes built-in option types, `let` expressions, and pairs for convenience.

base intermediate language `IL`, (3) the target language of CPS, `ILK`, and (4) the target language of closure conversion, `ILC`. Both target languages `ILK` and `ILC` extend the intermediate language `IL`. `ILK` adds nested, open lambdas, while `ILC` adds existentials for abstracting closure environments.

The CPS translation from `STLC` to `ILK` is performed via functions `cps_val` and `cps_exp` on lines 44–47. Since the types of these functions are family polymorphic, we can safely reuse them in any extension to `BaseComp` that further binds families `STLC` or `ILK`, as long as any new pattern match cases are specified in the extension. We also get the guarantee that types from incompatible families will not be mixed – both `STLC` and `ILK` must belong to the same enclosing compiler family.

On line 56, family `IfExt` represents the compiler for `STLC` extended with if-expressions. All unchanged constructs, including those inside nested families, are inherited as-is. Only the new constructs needed for the extension are specified in the extension. For example, the nested family `IfExt.IL` further binds `BaseComp.IL` and adds boolean types, if-expressions, and the new match cases for `eval` and `apply` (omitted from figure). Family `IfExt.ILC` must be extended in turn, since it defines a pattern match on the extended type `IfExt.IL.Val`. Nested family `BaseComp.ILK` is inherited as-is to become `IfExt.ILK`. Finally, the translation functions are extended with new pattern match cases (omitted from figure). Figure 1.1 shows a detailed breakdown of all constructs that are inherited unchanged (in light grey) and constructs that are extended (in black).

The extensible compilers example in Figure 1.7 shows how Persimmon combines the benefits of nesting with the benefits of family polymorphism. Persimmon allows us to define and inherit nested components, and to preserve the structural and hierarchical relationships between the components in the derived family. At the same time, family polymorphism in Persimmon provides safety guarantees: inherited code is type safe for use in a derived family, and interactions between members of incompatible families are prohibited. For example, we could not call the function `BaseComp.IL.eval` on an instance of type `IfExt.IL.Exp` due to a family mismatch in the path

prefix of type `Exp`. Note that even though `IfExt.IL` extends `BaseComp.IL`, subtyping does not apply: the family paths are mismatched due to distinct in-scope constructors for `Exp`.

1.4.3 Support for Mixins

```
1  Family IfExt {
2    Family Base extends STLCBase {}
3    Family Derived extends Base {
4      (* ... contents of mixin IfExt, paths substituted *)
5    }
6  }
7  Family ArithExt {
8    Family Base extends STLCBase {}
9    Family Derived extends Base {
10     (* ... contents of mixin ArithExt, paths substituted *)
11   }
12 }
13 Family IfExtBuild extends IfExt {
14   Family Base extends STLCBase {}
15 }
16 Family ArithExtBuild extends ArithExt {
17   Family Base extends IfExtBuild.Derived {}
18 }
19 (** This family contains both:
20     if-expressions and arithmetic. **)
21 Family STLCIfArith extends
22   ArithExtBuild.Derived {}
```

Figure 1.8: An encoding of mixins in Persimmon.

In addition to linear extensions, Persimmon also supports composable extensions – *mixins* – with a simple encoding shown in Figure 1.8. Mixins allow us to compose functionality from parallel extensions without creating new inheritance relationships between the extensions themselves. By supporting mixins through an encoding, we avoid needlessly complicating the type system and duplicating language features. Nested family polymorphism in Persimmon is powerful enough to encode mixins, obviating the need for a native mixin construct. Consider the following example which uses the mixin syntax we would like to encode:

```
1 Mixin IfExt extends STLCBase { (* ... contents of mixin IfExt *) }
2 Mixin ArithExt extends STLCBase { (* ... contents of mixin ArithExt *) }
3 Family STLCIfArith extends STLCBase with IfExt, ArithExt {}
```

Suppose we want to extend `STLCBase` with multiple parallel features, such as if-expressions `IfExt` and arithmetic `ArithExt` (shown above). Ideally, we would define an extension for each feature only once (as on lines 1 and 2 above), and then *compose* those extensions (as on line 3) to

yield arbitrary combinations of features. We use this example as a roadmap for our PERSIMMON encoding.

We encode mixins in PERSIMMON as shown in Figure 1.8 by combining straightforward linear extension with a flexible base for extension. Each family representing a mixin, such as `Family IfExt`, contains two nested families: a `Base` family, and a `Derived` family. The `Base` family can be further bound, which allows extensions to build on any version of `STLC`. The `Derived` family extends `Base` and contains the code that implements the actual extension. This nested family structure ensures that the dependencies between extensions are flexible, and that extensions are composable.

Lines 13–22 in Figure 1.8 show how we encode the composition of extensions in PERSIMMON. This code is a direct translation of our roadmap example. For each subsequent extension, we further bind the `Base` family, effectively "stacking" the extensions on top of each other. `IfExtBuild.Base` extends base STLC, and `ArithExtBuild.Base` extends STLC with if-expressions. Finally, family `STLCIfArith` extends `ArithExtBuild.Derived`, including both features.

While we do encode mixin composition using linear extension, our linearization follows the order of overwriting that is generally imposed by mixins. No inheritance relationship is created between the two parallel extensions, `IfExt` and `ArithExt`; they can be freely composed with other extensions. Our mixin encoding highlights the *parsimony of features* in our language: nested families in PERSIMMON are powerful enough to encode mixins, eliminating the need for a separate mixin construct. Finally, the encoding is completely automated – the programmer can enjoy the same convenient mixin syntax as in the roadmap example.

1.5 Language Design

In this section, we give a comprehensive overview of the Persimmon language design, as follows. First, we discuss the syntax of Persimmon, highlighting the constructs that facilitate nested family polymorphism in our language: relative path types, nested families, and our extensible `cases` constructs (Section 1.5.1).

Next, we present our type system (Section 1.5.2), which is based on static linkages: the map-like data structures that store type-level information about each family (further detailed in Section 1.5.4). Static linkages are a generalization of global class tables found in many type systems. Our type system directly references the contents of computed linkages, and thus is fairly straightforward. Our operational semantics is also streamlined by linkages; however, a different type of linkage is used that also stores definitions (Section 1.5.3).

Finally, we discuss linkage operations and the benefits of linkages in detail (Section 1.5.4). Each family in a given program (and the program itself) has a corresponding linkage. Linkages – as opposed to other data structures, such as an abstract syntax tree – make it easy to substitute paths inside inherited code so that they refer to the derived family. Persimmon supports nested inheritance, further binding of families, and extensible data types and pattern matching through nested *linkage concatenation*, a recursive operation that combines linkages for a base family and a derived family.

Underlying the linkage operations and the type system is the unifying notion of well-formedness: well-formed family definitions parse into well-formed linkages, and well-formedness of linkages is preserved by concatenation. Exhaustivity of pattern matching is a well-formedness condition, checked at program definition (Section 1.5.2).

$$
\begin{array}{llll}
\textbf{Family Name} & A & \textbf{Relative Path} & sp ::= \texttt{prog} \mid \texttt{self}(a.A) \\
\textbf{Program Path} & \texttt{prog} & \textbf{Path} & a ::= sp \mid a.A
\end{array}
$$

$$
\begin{array}{lrcl}
\textbf{Type} & T,\ T' & ::= & \texttt{N} \mid \texttt{B} \mid a.R \mid T \to T' \mid \{(f_i : T_i)*\} \\
\textbf{Expression} & e,\ g & ::= & n \mid b \mid x \mid a.m \mid a.c \mid g\ e \mid e.f \mid \lambda(x:T).e \\
 & & & \mid \{(f_i = e_i)*\} \mid a.R(\{(f_i = e_i)*\}) \mid a.R(C\ \{(f_i = e_i)*\}) \\
 & & & \mid \texttt{if}\ e\ \texttt{then}\ g\ \texttt{else}\ g' \mid \texttt{match}\ e\ \texttt{with}\ a.c\ \{(f_{arg} = e_{arg})*\} \\
\textbf{Value} & v & ::= & n \mid b \mid \lambda(x:T).e \mid \{(f_i = v_i)*\} \mid a.R(\{(f_i = v_i)*\}) \\
 & & & \mid a.R(C\ \{(f_i = v_i)*\})
\end{array}
$$

$$
\textbf{Path Context}\ \mathrm{K} ::= [] \mid sp :: \mathrm{K} \qquad \textbf{Linkage}\ L ::= L_S \mid L_D
$$

$$
\begin{array}{lrl}
\textbf{Program} & \mathrm{p} & ::= \mathrm{famdef}*\ e \\
\textbf{Definition} & \mathrm{def} & ::= \mathrm{famdef} \mid \mathrm{typdef} \mid \mathrm{adtdef} \mid \mathrm{fundef} \mid \mathrm{casesdef}
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{famdef} & ::= & \textbf{Family}\ A\ (\textbf{extends}\ a.A')?\ \{\mathrm{famdef}*\ \mathrm{typdef}*\ \mathrm{adtdef}*\ \mathrm{fundef}*\ \mathrm{casesdef}*\} \\
\mathrm{typdef} & ::= & \textbf{type}\ R = \{(f_i : T_i)*\} \mid \textbf{type}\ R \mathrel{+}= \{(f_i : T_i = v_i)*\} \\
\mathrm{adtdef} & ::= & \textbf{type}\ R\ (+)?= \overline{C_j\ \{(f_i : T_i)*\}} \\
\mathrm{fundef} & ::= & \textbf{val}\ m : T \to T' = \lambda(x:T).e \\
\mathrm{casesdef} & ::= & \textbf{cases}\ c\ \langle a.R \rangle : \{(f_i : T_i)*\} \to \{(C_j : T_j \to T)*\}\ (+)?= \\
 & & \lambda(x : \{(f_i : T_i)*\}).\{(C_j = \lambda(y_j : T_j).e_j)*\}
\end{array}
$$

Figure 1.9: The PERSIMMON syntax.

1.5.1 SYNTAX

We show the full syntax of PERSIMMON in Figure 1.9. We follow a classic approach to family extension with relative path types [Igarashi et al., 2005]. Essentially, when an expression with a relative path type is inherited into a derived family, the relative path is automatically substituted to point to the version of the same type in the derived family. In this section, we highlight how our syntax supports extensibility, including our use of relative path types as well as the special shape of our `match` expressions, which enables extensible pattern matching in PERSIMMON by forcing the use of our extensible `cases` constructs within the match. We also introduce the syntax of linkages: the underlying helper structures we use in our semantics that store the necessary information about each family.

Paths, Types, and Expressions. In the context of a program, each family has a unique, fully qualified path that specifies the nesting depth of the family with respect to the program path, `prog`. The path `prog` is the prefix to all family paths. For example, family A' nested within a top-level family A is located at the path `prog`.$A.A'$.

Relative paths use the keyword `self` to refer to the current family and automatically adapt when that family is extended. For example, the relative path `self`(`prog`.A) resolves to different fully qualified paths depending on the family in which it appears. Inside the concrete family `prog`.A, it refers to `prog`.A, but when inherited into a derived family `prog`.A'', the same relative path resolves to `prog`.A''. This behavior ensures that inherited code remains up to date and compatible with the most recent extension.

Path types are represented in Persimmon using syntax $a.R$, where a is the path to the family in which type name R is defined. The path a can be relative or concrete, but relative path types, such as `self`(`prog`.A).T, have a relative path prefix. When a relative path type is inherited, we update the relative path prefix to refer to the derived family. A type's path prefix may also specify the concrete family in which the type appears, such as `prog`.$A.T$. Finally, there is no raw ADT type in Persimmon – all ADTs in our system are path types with ADT definitions (for example, the ADT `Exp` on line 7 in Figure 1.7). This restriction means that type checking never depends on unfolding ADT definitions or reasoning about recursive equivalence, but instead reduces to syntactic comparison of paths and lookup of fixed ADT definitions.

Some expressions in Persimmon include a family path as part of their syntax. For example, function calls ($a.m$) specify the path a to the family in which function m is defined. Similarly, `cases` calls ($a.c$) specify a `cases` definition c defined at the family path a. We can view `cases` definitions as special function definitions, with a rigidly structured output type and the ability to extend the body of the definition, which is essential for extensible pattern matching. By separating

our `cases` definitions from their uses, we ensure that `cases` are easily extensible as direct family members, regardless of how deeply their uses are nested.

In turn, our `match` expressions have a special shape due to the use of separate, extensible `cases` constructs. For an example, see the function `ev` and the corresponding `cases` construct `evc` on lines 5–7 in Figure 1.23. Inside a `match` expression, `match` e `with` $a.c$ $\{(f_{arg} = e_{arg})*\}$, the appropriate `cases` definition $a.c$ is applied to a record of arguments. The arguments represent a *match context* – any additional information needed for the pattern match, such as any variables referenced within the body. We do not generalize the left-hand side of the application (to any expression e) in order to simplify the translation of our `match` expressions to other languages that do not have separate `cases` constructs.

Finally, we can create instances of path types ($a.R$) via instance expressions – $a.R(\{(f_i = e_i)*\})$ for named record types, and $a.R(C\ \{(f_i = e_i)*\})$ for ADTs, where C is a valid constructor of the type. To create a record type instance $a.R(\{(f_i = e_i)*\})$, the user must specify an input e_i for each field f_i. If a field is omitted, it must have a pre-defined default value, otherwise the instance will not type-check. We require default values for all extensions of record types to guarantee type safety of inherited instance expressions.

Programs and Definitions. A PERSIMMON program contains an arbitrary number of family definitions and a main expression. Family definitions can contain nested families, record types, ADTs, functions, and `cases` constructs. Extensions for inherited record types, ADTs, and `cases` constructs can be specified with marker +=, as opposed to the new definition marker =.[6] Extensions to record types must provide default values v_i for each field. For readability of ADT definitions, we use an overline symbol instead of a Kleene star to represent a set of ADT constructors C_j with

[6]The += syntax is a user convenience, but not a requirement. Since we do not allow overwriting of existing types, any named type that appears in the base family and in the derived family will be considered an extension in the derived family, regardless of the marker.

$$\left\{\begin{array}{lcl} \texttt{self} & = & a \\ \texttt{super} & = & a.A? \\ \texttt{NEST} & = & \{\ (A \mapsto L_S)*\ \} \\ \texttt{TYPES} & = & \{\ (R \mapsto \{(f_i : T_i)*\}\ )*\ \} \\ \texttt{DEFS} & = & \{\ (R \mapsto \overline{(f_k)*}\ )*\ \} \\ \texttt{ADTS} & = & \{\ (R \mapsto \overline{C_j\ T_j})*\ \} \\ \texttt{FUNS} & = & \{\ (m \mapsto T)*\ \} \\ \texttt{CASES} & = & \{\ (c \mapsto (\langle T\rangle,\ T'))*\} \end{array}\right\}_{L_S} \quad \left\{\begin{array}{lcl} \texttt{self} & = & a \\ \texttt{super} & = & a.A? \\ \texttt{NEST} & = & \{\ (A \mapsto L_D)*\ \} \\ \texttt{TYPES} & = & \{\ (R \mapsto \{(f_i : T_i)*\}\ )*\ \} \\ \texttt{DEFS} & = & \{\ (R \mapsto \{\overline{(f_k = v_k)*}\}\ )*\ \} \\ \texttt{ADTS} & = & \{\ (R \mapsto \overline{C_j\ T_j})*\ \} \\ \texttt{FUNS} & = & \{\ (m \mapsto (T,\ e))*\ \} \\ \texttt{CASES} & = & \{\ (c \mapsto (\langle T\rangle,\ T',\ e))*\} \end{array}\right\}_{L_D}$$

Figure 1.10: Linkage syntax for static linkages L_S (left) and dynamic linkages L_D (right).

the corresponding input fields and their types.

Each `cases` construct c in our system is a function from a match context to a record of "case handlers" for each constructor C_j of the scrutinee type $a.R$. Each constructor C_j is assigned a handler function, which takes as input the fields of that constructor. The output type of a `cases` construct explicitly names each handled constructor and the type of the corresponding handler function. While we use this detailed syntax for ease of type checking in our system, users can enjoy the convenient in-line syntax shown earlier.

Linkages. We differentiate between two kinds of linkages: *static linkages* L_S that store type-level information only, and *dynamic linkages* L_D that store both type- and definition-level information (Figure 1.10). Both kinds of linkage store the current family path (`self`), the parent family path (`super`), a map of record type definitions (`TYPES`), and a map of ADT definitions (`ADTS`). Static linkages keep track of all record type fields that have defaults, while dynamic linkages additionally store the default values for those fields. Static linkages store only function and `cases` signatures, while dynamic linkages also store the bodies of those constructs. Later on, we discuss how linkages – which are a snapshot of the contents that appear directly in each family – are concatenated to reflect the inherited and extended constructs in a derived family.

1.5.2 Type System

Our type system depends on static linkages, which are a generalization of global class tables. We give a more detailed view of linkage computation and concatenation in Section 1.5.4. During type checking, we simply compute a static linkage L_S for any family path a on demand, and retrieve any desired type definitions or function signatures from this linkage. We retrieve definitions from *complete* linkages, which already include all inherited, extended, and overwritten components for the family path at hand. For example, while the incomplete linkage for family path `prog.STLCIf` in Figure 1.5 maps the type `Ty` to the sole constructor `TBool`, the complete linkage for this family path maps `Ty` to four constructors: the inherited constructors `TUnit`, `TNat`, and `TArr`, as well as the extension `TBool`. Complete linkages are computed on demand by concatenation of all incomplete linkages in the inheritance chain. By delegating the heavy-duty handling of extensibility to linkage computation, the type checking relation is simplified.

It is important to note that the on-demand approach to linkage computation has implications for the efficiency and modularity of type checking. Our theory may require a linkage for the same family path to be recomputed multiple times throughout the type checking process, negatively affecting performance. This is why in our implementation we cache the computed linkages for efficiency. Furthermore, the on-demand approach poses a conflict for separate type checking and compilation of *program fragments* as defined by Cardelli [1997]. A family in Persimmon is an example of a program fragment. Modular type checking of each such fragment (family) would require a more sophisticated dependency analysis than the on-demand approach allows, along with a pre-computation of linkages for each family. The original version of Persimmon does not support modular type checking, but we describe how it could be supported in Section 1.11.

$\boxed{\mathrm{K};\Gamma \vdash e : T}$

$$\frac{}{\mathrm{K};\Gamma \vdash n : \mathtt{N}}\ \text{(T-Num)} \qquad \frac{}{\mathrm{K};\Gamma \vdash b : \mathtt{B}}\ \text{(T-Bool)} \qquad \frac{x : T \in \Gamma}{\mathrm{K};\Gamma \vdash x : T}\ \text{(T-Var)}$$

$$\frac{\mathrm{K} \vdash \mathtt{WF}(T) \qquad \mathrm{K};(x : T, \Gamma) \vdash e : T'}{\mathrm{K};\Gamma \vdash \lambda(x : T).e : T \to T'}\ \text{(T-Lam)} \qquad \frac{\mathrm{K};\Gamma \vdash e : T \qquad \mathrm{K};\Gamma \vdash g : T \to T'}{\mathrm{K};\Gamma \vdash g\ e : T'}\ \text{(T-App)}$$

$$\frac{\forall i,\ \mathrm{K};\Gamma \vdash e_i : T_i}{\mathrm{K};\Gamma \vdash \{(f_i = e_i)*\} : \{(f_i : T_i)*\}}\ \text{(T-Rec)} \qquad \frac{\mathrm{K};\Gamma \vdash e : \{(f_i : T_i)*\} \qquad f : T \in (f_i : T_i)*}{\mathrm{K};\Gamma \vdash e.f : T}\ \text{(T-Proj)}$$

$$\frac{\mathrm{K};\Gamma \vdash e : T' \qquad \mathrm{K} \vdash T' <: T}{\mathrm{K};\Gamma \vdash e : T}\ \text{(T-Subs)} \qquad \frac{\mathrm{K};\Gamma \vdash e : \mathtt{B} \qquad \mathrm{K};\Gamma \vdash g : T \qquad \mathrm{K};\Gamma \vdash g' : T}{\mathrm{K};\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ g\ \mathtt{else}\ g' : T}\ \text{(T-If)}$$

$$\frac{\mathrm{K} \vdash \mathtt{WF}(a) \qquad a \rightsquigarrow L_S \qquad m \mapsto (T \to T') \in L_S.\mathrm{FUNS}}{\mathrm{K};\Gamma \vdash a.m : T \to T'}\ \text{(T-FamFun)}$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash \mathtt{WF}(a) \qquad a \rightsquigarrow L_S \qquad R \mapsto \{(f_j : T_j)*\} \in L_S.\mathrm{TYPES} \qquad R \mapsto (f_k)* \in L_S.\mathrm{DEFS} \\ \forall i,\ \exists j,\ f_i = f_j \wedge \mathrm{K};\Gamma \vdash e_i : T_j \qquad \forall j,\ f_j \in (f_i)* \vee f_j \in (f_k)*\end{array}}{\mathrm{K};\Gamma \vdash a.R(\{(f_i = e_i)*\}) : a.R}\ \text{(T-Constr)}$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash \mathtt{WF}(a) \qquad a \rightsquigarrow L_S \qquad R \mapsto \overline{C_j\ \{(f_i : T_i)*\}} \in L_S.\mathrm{ADTS} \\ C\ \{(f_k : T_k)*\} \in \overline{C_j\ \{(f_i : T_i)*\}} \qquad \forall k,\ \mathrm{K};\Gamma \vdash e_k : T_k\end{array}}{\mathrm{K};\Gamma \vdash a.R(C\ \{(f_k = e_k)*\}) : a.R}\ \text{(T-ADT)}$$

$$\frac{\mathrm{K} \vdash \mathtt{WF}(a) \qquad a \rightsquigarrow L_S \qquad c \mapsto (\langle a'.R \rangle, \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}) \in L_S.\mathrm{CASES}}{\mathrm{K};\Gamma \vdash a.c : \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}}\ \text{(T-Cases)}$$

$$\frac{\begin{array}{c}\mathrm{K};\Gamma \vdash e : a'.R \qquad a' \rightsquigarrow L_S \qquad R \mapsto \overline{C_j\ \{(f_i : T_i)*\}} \in L_S.\mathrm{ADTS} \\ \mathrm{K};\Gamma \vdash a.c : \{(f_{arg} : T_{arg})*\} \to \{(C_j : \{(f_i : T_i)*\} \to T)*\} \\ \mathrm{K};\Gamma \vdash \{(f_{arg} = e_{arg})*\} : \{(f_{arg} : T_{arg})*\}\end{array}}{\mathrm{K};\Gamma \vdash \mathtt{match}\ e\ \mathtt{with}\ a.c\ \{(f_{arg} = e_{arg})*\} : T}\ \text{(T-Match)}$$

Figure 1.11: Type checking of Persimmon expressions.

Type Checking of Expressions. The complete set of rules for type checking expressions in Persimmon is shown in Figure 1.11. Any type-level information required for type checking is retrieved from a complete static linkage L_S for the appropriate family path a. We type check expressions with respect to a typing context Γ and a family path context K.[7] The context K keeps track of the nesting depth of the current expression within the program.

[7]The syntax for K is shown in Figure 1.9. For singleton contexts we use the shorthand syntax $[sp]$, equivalent to $sp :: []$.

Some expressions, such as function calls and `cases` calls, are type checked by retrieving their type signatures directly from the linkage (rules `T-FamFun` and `T-Cases`). Since well-formed family definitions parse into well-formed linkages, and linkage concatenation preserves well-formedness, the retrieved signature reflects the true type of the expression.

An instance of a record type, $a.R(\{(f_i = e_i)*\})$, is well-typed if the linkage L_S for path a contains a definition for type R, and the inputs e_i are well-typed with respect to this definition (rule `T-Constr`). Any field f_j in the definition of R that does not have an input e_i within the instance expression must have a stored default value, or the instance will not type check. ADT instances are checked similarly, while also ensuring that the constructor C used to create the instance is a valid constructor for the type (rule `T-ADT`). ADTs do not take default field values, so a well-typed input e_k must be provided for every field f_k in constructor C. A pattern match expression will type check if the type of the scrutinee e is some path type $a'.R$, which has an ADT definition in the complete static linkage for path a'. The `cases` call $a.c$ within the match, and the match context $\{(f_{arg} = e_{arg})*\}$ must also be well-typed (rule `T-Match`).

$\boxed{\mathrm{K} \vdash T \ <: \ T'}$

$$\frac{\mathrm{K} \vdash \texttt{WF}(a) \qquad a \rightsquigarrow L_S \qquad R \mapsto \{(f_i : T_i)*\} \ \in \ L_S.\mathrm{TYPES} \qquad \mathrm{K} \vdash \{(f_i : T_i)*\} \ <: \ T'}{\mathrm{K} \vdash a.R \ <: \ T'} \ (\textsc{Sub-Fam})$$

$$\frac{\forall j, \ \exists T, \ \mathrm{K} \vdash T \ <: \ T_j \quad \wedge \quad (f_j : T) \ \in \ (f_i : T_i)*}{\mathrm{K} \vdash \{(f_i : T_i)*\} \ <: \ \{(f_j : T_j)*\}} \ (\textsc{Sub-Rec})$$

$$\frac{}{\mathrm{K} \vdash T \ <: \ T} \ (\textsc{Sub-Refl}) \qquad \frac{\mathrm{K} \vdash T_2 \ <: \ T_1 \qquad \mathrm{K} \vdash T_1' \ <: \ T_2'}{\mathrm{K} \vdash T_1 \to T_1' \ <: \ T_2 \to T_2'} \ (\textsc{Sub-Fun})$$

Figure 1.12: Subtyping relation.

Subtyping. Persimmon supports reflexivity of subtyping, subtyping of arrow types, depth and width subtyping of record types, and subtyping of path types (for conversion between a path type

and the corresponding record type). The full rules are shown in Figure 1.12. An extended type from the derived family is not a subtype of the corresponding type from the base family, due to undesired interactions between relative path types and inheritance [Igarashi et al., 2005]. Consider a base family `prog.X`, a derived family `prog.Y`, and a type T that appears in both families. Type `prog.Y.T` cannot be a subtype of `prog.X.T` safely, because then an instance of `prog.Y.T` could be used anywhere an instance of `prog.X.T` is expected. However, since function definitions can be overwritten in derived families, we may end up calling the overwritten version of a function in an environment where only the base version of this function was expected. If the overwritten function refers to extended constructs that were not available in the base family, soundness will be broken.

$\boxed{\mathrm{K} \vdash \texttt{WF}(T)}$

$$\frac{}{\mathrm{K} \vdash \texttt{WF}(\texttt{N})}\ (\text{WF-Num}) \qquad \frac{}{\mathrm{K} \vdash \texttt{WF}(\texttt{B})}\ (\text{WF-Bool}) \qquad \frac{\mathrm{K} \vdash \texttt{WF}(T) \qquad \mathrm{K} \vdash \texttt{WF}(T')}{\mathrm{K} \vdash \texttt{WF}(T \to T')}\ (\text{WF-Arrow})$$

$$\frac{\mathrm{K} \vdash \texttt{WF}(a) \qquad a \rightsquigarrow L_S \qquad R \mapsto \{(f_i : T_i)*\} \in L_S.\text{TYPES} \ \vee\ R \mapsto \overline{C_j\ \{(f_i : T_i)*\}} \in L_S.\text{ADTS}}{\mathrm{K} \vdash \texttt{WF}(a.R)}\ (\text{WF-Named})$$

$$\frac{\forall i,\ \mathrm{K} \vdash \texttt{WF}(T_i) \qquad \forall i\ j,\ i \neq j \implies f_i \neq f_j}{\mathrm{K} \vdash \texttt{WF}(\{(f_i : T_i)*\})}\ (\text{WF-Record})$$

Figure 1.13: Well-formedness of types.

Typing and Well-Formedness of Programs. A program p is well-typed if every family definition within p is well-formed, and the main expression e is well-typed (rule `T-Prog` in Figure 1.14). At this topmost level of nesting, the linkage context K contains one path, `prog`, which is the path to p. To prevent circular inheritance, a well-formed family definition (`WF-FamDef`) cannot have its own family path as an ancestor, and cannot inherit from a family nested within it. All nested definitions within the family must also be well-formed. Since we use the linkage context to keep track of the nesting level, all nested definitions must be checked with respect to a linkage context

$\boxed{\mathrm{K};\Gamma \vdash \mathrm{p} : T}$

$$\frac{\mathrm{p} = \mathrm{famdef}_i* \; e \qquad \forall i,\ [\texttt{prog}] \vdash \texttt{WF}(\mathrm{famdef}_i) \qquad [\texttt{prog}];[] \vdash e : T}{[];[] \vdash \mathrm{p} : T} \quad \text{(T-Prog)}$$

$\boxed{\mathrm{K} \vdash \texttt{WF}(\mathrm{def})}$

$$\frac{\begin{array}{c}\mathrm{famdef} = \textbf{Family}\ A\ (\textbf{extends}\ a.A')?\ \{\mathrm{famdef}_n*\ \ \mathrm{typdef}_q*\ \ \mathrm{adtdef}_u*\ \ \mathrm{fundef}_w*\ \ \mathrm{casesdef}_z*\} \\ \texttt{self}(sp.A) \rightsquigarrow L_S \qquad \texttt{self}(sp.A) \notin \texttt{ancestors}(L_S) \qquad \neg\texttt{nested}(a.A', \texttt{self}(sp.A)) \\ \mathrm{sp} :: \mathrm{K} \vdash \texttt{WF}(a.A') \qquad \mathrm{K}' = \texttt{self}(sp.A) :: sp :: \mathrm{K} \qquad \mathrm{K}' \vdash \texttt{EC}(L_S) \\ \forall n,\ \mathrm{K}' \vdash \texttt{WF}(\mathrm{famdef}_n) \qquad \forall q,\ \mathrm{K}' \vdash \texttt{WF}(\mathrm{typdef}_q) \qquad \forall u,\ \mathrm{K}' \vdash \texttt{WF}(\mathrm{adtdef}_u) \\ \forall w,\ \mathrm{K}' \vdash \texttt{WF}(\mathrm{fundef}_w) \qquad \forall z,\ \mathrm{K}' \vdash \texttt{WF}(\mathrm{casesdef}_z)\end{array}}{sp :: \mathrm{K} \vdash \texttt{WF}(\mathrm{famdef})} \quad \text{(WF-FamDef)}$$

$$\frac{\mathrm{K} \vdash \texttt{WF}(\{(f_i : T_i)*\})}{\mathrm{K} \vdash \texttt{WF}(\textbf{type}\ R\ =\ \{(f_i : T_i)*\})} \quad \text{(WF-TypDef)}$$

$$\frac{\mathrm{K} \vdash \texttt{WF}(\{(f_i : T_i)*\}) \qquad \forall i,\ \mathrm{K};[] \vdash v_i : T_i}{\mathrm{K} \vdash \texttt{WF}(\textbf{type}\ R\ +=\ \{(f_i : T_i = v_i)*\})} \quad \text{(WF-TypDef-Ext)}$$

$$\frac{\forall j,\ \mathrm{K} \vdash \texttt{WF}(\{(f_i : T_i)*\})}{\mathrm{K} \vdash \texttt{WF}(\textbf{type}\ R\ \ (+)?=\ \overline{C_j\ \{(f_i : T_i)*\}})} \quad \text{(WF-AdtDef)}$$

$$\frac{\mathrm{K} \vdash \texttt{WF}(T \to T') \qquad \mathrm{K};[] \vdash \lambda(x : T).e : T \to T'}{\mathrm{K} \vdash \texttt{WF}(\textbf{val}\ m\ :\ T \to T'\ =\ \lambda(x : T).e)} \quad \text{(WF-FunDef)}$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash \texttt{WF}(a) \qquad a \rightsquigarrow L_S \qquad R\ \mapsto\ \overline{C_i\ T_i}\ \in\ L_S.\texttt{ADTS} \qquad \forall j,\ \exists i,\ (C_j = C_i \wedge T_j = T_i) \\ \mathrm{K} \vdash \texttt{WF}(T \to \{(C_j : T_j \to T')*\}) \\ \mathrm{K};[] \vdash \lambda(x : T).\{(C_j = \lambda(y_j : T_j).e_j)*\} : T \to \{(C_j : T_j \to T')*\}\end{array}}{\mathrm{K} \vdash \texttt{WF}(\textbf{cases}\ c\ \langle a.R\rangle\ :\ T \to \{(C_j : T_j \to T')*\}\ \ (+)?=\ \lambda(x : T).\{(C_j = \lambda(y_j : T_j).e_j)*\})} \quad \text{(WF-CasesDef)}$$

$\boxed{\mathrm{K} \vdash \texttt{EC}(L_S)}$

$$\frac{\begin{array}{c}\forall\ (c \mapsto (\langle a.R\rangle, T \to \{(C_j : T_j \to T')*\}))\ \in\ L_S.\texttt{CASES},\ \mathrm{K} \vdash \texttt{WF}(a)\ \wedge\ a \rightsquigarrow L'_S\ \wedge\ R\ \mapsto\ \overline{C_j\ T_j}\ \in\ L'_S.\texttt{ADTS} \\ \forall\ A\ \in\ L_S.\texttt{NEST},\ \ sp = L_S.\texttt{self}\ \wedge\ \mathrm{K}' = \texttt{self}(sp.A) :: \mathrm{K}\ \wedge\ \texttt{self}(sp.A) \rightsquigarrow L''_S\ \wedge\ \mathrm{K}' \vdash \texttt{EC}(L''_S)\end{array}}{\mathrm{K} \vdash \texttt{EC}(L_S)} \quad \text{(EC-Nest)}$$

Figure 1.14: Type checking, well-formedness (WF), and exhaustivity checking (EC) of programs.

K′, which extends K with the path to the current family. We maintain the convention that the head path in the linkage context points to the immediate wrapper family of the checked definition.

Importantly, we consider exhaustivity of pattern matching a well-formedness condition. We trigger an *exhaustivity check* from `WF-FamDef` to recursively check that pattern matching is exhaustive in the current family definition, and any nested family definitions (rule `EC-Nest` in Figure 1.14). This check operates on complete static linkages, as opposed to program definitions, since we need to check the inherited `cases` constructs as well. Consider the following example:

```
1 Family A1 {
2     type T = C1 | C2
3     val f: T -> N = λ(t: T). match t with c {}
4     cases c <T>: {} -> {C1: {} -> N, C2: {} -> N} =
5       λ(_: {}). {C1 = λ(_:{}). 1, C2 = λ(_:{}). 2}
6 }
```

```
7 Family A2 extends A1 {
8     type T += C3
9
10    // no match for C3!
11    val g: T -> N = λ(t: T). (f t)
12 }
```

Since family `A2` inherits the function `f` on line 3, the inferred relative path to `T` in the input type of `f` is updated via path substitution upon inheritance, giving `f` the input type `self(prog.A2).T` inside `A2`. The input `t` on line 11 has the same type, and the application `(f t)` in the body of `g` type checks. However, the `cases` construct called by `f` has not been extended! Therefore, to ensure that all `cases` constructs in a derived family are exhaustive, we perform an exhaustivity check on all inherited and newly defined `cases` constructs.

Since exhaustivity is checked separately, the rule for well-formedness of `cases` definitions (rule `WF-CasesDef` in Figure 1.14) only requires that the constructors C_j handled by the `cases` definition appear in the definition of scrutinee type, $a.R$, with the expected input types T_j. All other definitions (such as record types, ADTs, and functions) are well-formed if the types within these definitions are well-formed, and the expressions are well-typed. We also require that all types have unique names, `cases` and functions have unique headers, and that there are no duplicate constructor names in an ADT or duplicate fields in a record type. These repetitive checks are omitted in Figure 1.14.

1.5.3 Operational Semantics

Like our type system, our operational semantics delegates the heavy lifting to linkages. In operational semantics we use dynamic linkages L_D that contain both type-level and definition-level information. Our full reduction and substitution relations are included in Figure 1.15 and Figure 1.16. Our reduction rules follow the convention of reducing subexpressions from left to right. Function calls $a.m$ and `cases` calls $a.c$ reduce directly to their definitions retrieved from the dynamic linkage L_D for family path a. The special shape of our `match` expressions means that we must reduce the scrutinee e (`R-MatchExp`) and the match context (`R-MatchCases`) before reducing the rest of the expression. Then, after the corresponding `cases` call $a.c$ is applied to the match context, the result is a record of case handlers, and we can project the required case handler for constructor C (`R-MatchFinal`).

1.5.4 Linkage Operations

Next, we discuss how *linkages* support extensibility in Persimmon. A linkage L is, essentially, a map of maps containing the information about a single family path. The static linkages L_S are used in static semantics, while the dynamic linkages L_D are used in dynamic semantics (see Figure 1.10 for a refresher on linkage syntax). We choose linkages for program representation as opposed to other options, such as an AST, for multiple reasons. Due to our use of relative path types, we must be able to easily perform *path substitution* when code is inherited (including constructs within nested families). Linkages are well-suited for this. *Linkage concatenation* – combining linkages from the base family and the derived family – is a natural fit for the modular extensibility of ADTs and pattern matching in Persimmon. Finally, our algorithmic linkage mechanism provides an easy way to look up names and signatures within any nested linkage, which helps support unrestricted,

$\boxed{\mathrm{K} \vdash e \longrightarrow e'}$

$$\frac{\mathrm{K} \vdash g \longrightarrow g'}{\mathrm{K} \vdash g\ e \longrightarrow g'\ e}\ (\text{R-App}) \qquad \frac{\mathrm{K} \vdash e \longrightarrow e'}{\mathrm{K} \vdash v\ e \longrightarrow v\ e'}\ (\text{R-LamArg}) \qquad \frac{}{\mathrm{K} \vdash (\lambda(x:T).e)\ v \longrightarrow [x := v]\ e}\ (\text{R-LamApply})$$

$$\frac{\mathrm{K} \vdash e \longrightarrow e'}{\mathrm{K} \vdash e.f \longrightarrow e'.f}\ (\text{R-Proj}) \qquad \frac{(f = v)\ \in\ (f_i = v_i)*}{\mathrm{K} \vdash \{(f_i = v_i)*\}.f \longrightarrow v}\ (\text{R-RecProj})$$

$$\frac{\mathrm{K} \vdash \texttt{WF}(a) \qquad a \rightsquigarrow L_D \qquad R \mapsto \{(f_k = v_k)*\}\ \in\ L_D.\texttt{DEFS} \qquad (f = v)\ \in\ (f_i = v_i)*\ \vee\ (f = v)\ \in\ (f_k = v_k)*}{\mathrm{K} \vdash a.R(\{(f_i = v_i)*\}).f \longrightarrow v}\ (\text{R-InstProj})$$

$$\frac{\mathrm{K} \vdash \texttt{WF}(a) \qquad a \rightsquigarrow L_D \qquad m \mapsto (T \to T', \lambda(x:T).e)\ \in\ L_D.\texttt{FUNS}}{\mathrm{K} \vdash a.m \longrightarrow \lambda(x:T).e}\ (\text{R-FamFun}) \qquad \frac{\mathrm{K} \vdash e \longrightarrow e'}{\mathrm{K} \vdash a.R(e) \longrightarrow a.R(e')}\ (\text{R-Instance})$$

$$\frac{\mathrm{K} \vdash e \longrightarrow e'}{\mathrm{K} \vdash \texttt{if}\ e\ \texttt{then}\ g\ \texttt{else}\ g' \longrightarrow \texttt{if}\ e'\ \texttt{then}\ g\ \texttt{else}\ g'}\ (\text{R-IfGuard}) \qquad \frac{\mathrm{K} \vdash e \longrightarrow e'}{\mathrm{K} \vdash a.R(C\ e) \longrightarrow a.R(C\ e')}\ (\text{R-ADT})$$

$$\frac{}{\mathrm{K} \vdash \texttt{if true then}\ g\ \texttt{else}\ g' \longrightarrow g}\ (\text{R-IfTrue}) \qquad \frac{}{\mathrm{K} \vdash \texttt{if false then}\ g\ \texttt{else}\ g' \longrightarrow g'}\ (\text{R-IfFalse})$$

$$\frac{\mathrm{K} \vdash e_j \longrightarrow e'_j}{\mathrm{K} \vdash \{f_0 = v_0, ..., f_i = v_i, f_j = e_j, ...\} \longrightarrow \{f_0 = v_0, ..., f_i = v_i, f_j = e'_j, ...\}}\ (\text{R-Rec})$$

$$\frac{\mathrm{K} \vdash e \longrightarrow e'}{\mathrm{K} \vdash \texttt{match}\ e\ \texttt{with}\ a.c\ \{(f_{arg} = e_{arg})*\} \longrightarrow \texttt{match}\ e'\ \texttt{with}\ a.c\ \{(f_{arg} = e_{arg})*\}}\ (\text{R-MatchExp})$$

$$\frac{\mathrm{K} \vdash \{(f_{arg} = e_{arg})*\} \longrightarrow \{(f_{arg} = e'_{arg})*\}}{\mathrm{K} \vdash \texttt{match}\ v\ \texttt{with}\ a.c\ \{(f_{arg} = e_{arg})*\} \longrightarrow \texttt{match}\ v\ \texttt{with}\ a.c\ \{(f_{arg} = e'_{arg})*\}}\ (\text{R-MatchCases})$$

$$\frac{}{\mathrm{K} \vdash \texttt{match}\ a'.R(C\ \{(f_k = v_k)*\})\ \texttt{with}\ a.c\ \{(f_{arg} = v_{arg})*\} \longrightarrow (a.c\ \{(f_{arg} = v_{arg})*\}).C\ \{(f_k = v_k)*\}}\ (\text{R-MatchFinal})$$

$$\frac{\mathrm{K} \vdash \texttt{WF}(a) \qquad a \rightsquigarrow L_D \qquad c \mapsto (\langle a'.R \rangle, T \to T', \lambda(x:T).e)\ \in\ L_D.\texttt{CASES}}{\mathrm{K} \vdash a.c \longrightarrow \lambda(x:T).e}\ (\text{R-Cases})$$

Figure 1.15: Reduction relation.

$\boxed{[x := s]\ e = e'}$

$$S_{Nat} : [x := s]\ n\ =\ n \qquad S_{Bool} : [x := s]\ b\ =\ b \qquad S_{Var} : [x := s]\ x\ =\ s$$

$$S_{VarNeq} : [x := s]\ y\ =\ y \qquad S_{LamNeq} : [x := s]\ \lambda(y:T).e\ =\ \lambda(y:T).([x := s]\ e)$$

$$S_{App} : [x := s]\ g\ e\ =\ ([x := s]\ g)([x := s]\ e) \qquad S_{Lam} : [x := s]\ \lambda(x:T).e\ =\ \lambda(x:T).e$$

$$S_{Rec} : [x := s]\ \{(f_i = e_i)*\}\ =\ \{(f_i = ([x := s]\ e_i))*\} \qquad S_{Proj} : [x := s]\ e.f\ =\ ([x := s]\ e).f$$

$$S_{FamFun} : [x := s]\ a.m\ =\ a.m \qquad S_{Constr} : [x := s]\ a.R(\{(f_i = e_i)*\})\ =\ a.R(\{(f_i = ([x := s]\ e_i))*\})$$

$$S_{Cases} : [x := s]\ a.c\ =\ a.c \qquad S_{ADT} : [x := s]\ a.R(C\ \{(f_i = e_i)*\})\ =\ a.R(C\ \{(f_i = ([x := s]\ e_i))*\})$$

$$S_{Match} : [x := s]\ \texttt{match}\ e\ \texttt{with}\ a.c\ \{(f_{arg} = e_{arg})*\}\ =\ \texttt{match}\ ([x := s]\ e)\ \texttt{with}\ a.c\ ([x := s]\ \{(f_{arg} = e_{arg})*\})$$

$$S_{If} : [x := s]\ \texttt{if}\ e\ \texttt{then}\ g\ \texttt{else}\ g'\ =\ \texttt{if}\ [x := s]\ e\ \texttt{then}\ [x := s]\ g\ \texttt{else}\ [x := s]\ g'$$

Figure 1.16: Variable substitution relation.

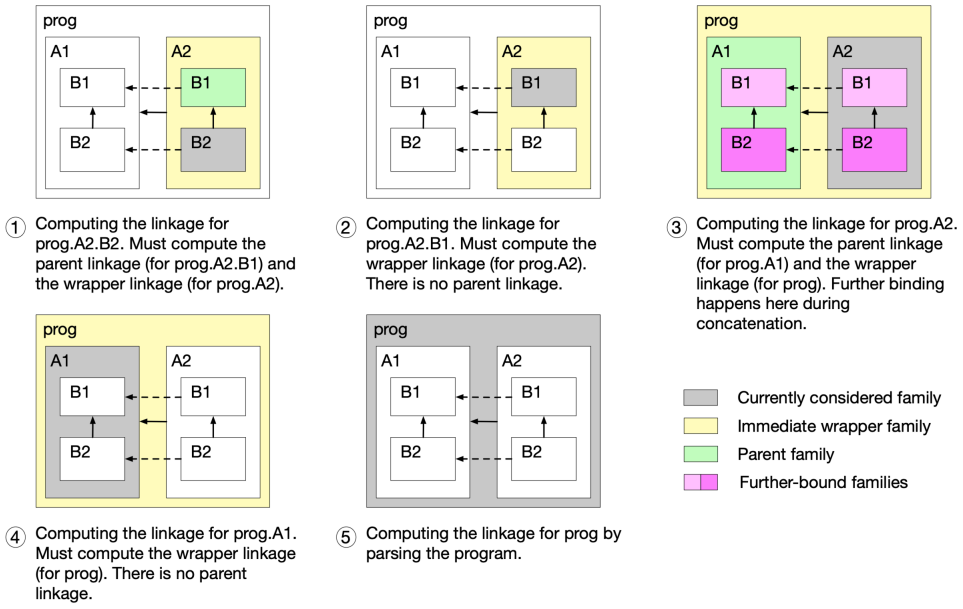

Figure 1.17: Linkage computation, intuitively. Each frame shows linkage computation for a single family (in grey), highlighting the recursive computation of the wrapper linkage (in yellow) and the parent linkage (in green).

mutually recursive references between family members in PERSIMMON.

Intuition for Linkage Computation. We compute a *complete* linkage L for family path a as shown in Figure 1.18. A complete linkage for a family path a includes all constructs inherited, extended, and newly defined at that family path, while an *incomplete* linkage only includes the constructs defined directly at that family path. Before diving into the details, let us build intuition for linkage computation with Figure 1.17. To keep succinct, we will use the phrase "a linkage for family A" to mean "a linkage for the family path to A". The frames in Figure 1.17 are diagrams of the code snippet in Figure 1.23, showing the nested family structure as well as the *extends* (solid) and *further binds* (dotted) links between families. In the code snippet, there are two top-level families, `A1` and `A2`. Family `A1` has two nested families, `B1` and `B2`, where `B2` extends `B1`. `A2` extends `A1` and further binds `B1` and `B2`. Each frame in Figure 1.17 represents the linkage computation for a single family path. For example, frame (4) represents linkage computation for

path `prog.A1` (highlighted in grey).

To compute a complete linkage for some family A, we must first recursively compute complete linkages for (i) the parent family of A, and (ii) the immediate wrapper family of A. We use the complete parent linkage to retrieve the constructs that will be inherited or extended by the derived family. We use the complete wrapper linkage to retrieve the nested, incomplete linkage for family A – which contains only the constructs appearing directly in A. Finally, we will concatenate the complete parent linkage with the incomplete linkage for A to obtain the complete linkage for A. For example, frame (1) in Figure 1.17 shows that to compute a complete linkage for the family path `prog.A2.B2` (highlighted in grey) we must first compute the complete parent linkage (at path `prog.A2.B1`, in green) and the complete wrapper linkage (at path `prog.A2`, in yellow).

Importantly, linkages are ultimately computed *from the outside in*: linkages for wrapper families are always computed before linkages for nested families. Thus, nested family linkages within a complete linkage are themselves incomplete. Consider frame (4) in Figure 1.17. Linkage computation for family path `prog.A1` does not trigger linkage computation for the nested family paths, such as `prog.A1.B2`, but instead recurses outward. This gives us a nice base case for proofs about linkage computation, since the recursive computation ends with the top-most path, `prog`. The outside-in computation order also prevents linkage computation from running into an infinite loop, such as in the case of a nested family that extends its own wrapper family.

Linkage Computation in Detail. Having established the intuition for computing linkages, we now discuss the linkage computation rules in detail (see Figure 1.18). The rule `L-Nest` governs the linkage concatenation process (represented by $+$), the process showcased by the frames in Figure 1.17. Here, we also make a distinction between *exact* linkage computation (marked $\rightsquigarrow$) and *inexact* linkage computation (marked $\rightsquigarrow_{\approx}$). *Exact* linkage computation for some path a produces a

$\boxed{a \rightsquigarrow L}$ $\boxed{a \rightsquigarrow_{\approx} L}$

$$\frac{\texttt{parse}_S(\text{p}) \;=\; L_S}{\texttt{prog} \rightsquigarrow L_S}\ \text{(L-Prog-S)} \qquad \frac{a.A \rightsquigarrow_{\approx} L}{\texttt{self}(a.A) \rightsquigarrow L}\ \text{(L-Self)} \qquad \frac{a.A \rightsquigarrow_{\approx} L \qquad L\ [a.A\ /\ \texttt{self}(a.A)] \;=\; L'}{a.A \rightsquigarrow L'}\ \text{(L-Sub)}$$

$$\frac{\texttt{parse}_D(\text{p}) \;=\; L_D}{\texttt{prog} \rightsquigarrow L_D}\ \text{(L-Prog-D)} \qquad \frac{a \rightsquigarrow L \qquad L'' = L.A \qquad L''.\texttt{super} \rightsquigarrow_{\approx} L' \qquad L' + L'' = L'''}{a.A \rightsquigarrow_{\approx} L'''}\ \text{(L-Nest)}$$

Figure 1.18: Rules for computing linkages L, parameterized by a program p.

linkage that refers to the current family using exactly path a, while *inexact* linkage computation for some path $a.A$ results in a linkage that refers to the current family by path $\texttt{self}(a.A)$. This distinction is necessary because families are allowed to extend relative or exact family paths, but linkage concatenation requires the parent linkage to refer to itself via a relative path. This requirement is necessary to perform path substitution: upon inheritance, any relative paths referring to the parent family will be substituted by relative paths referring to the derived family. If the parent linkage does not refer to itself by a relative path, this substitution cannot take place. Rule `L-Sub` serves to translate between exact and inexact linkage computation by substituting the `self`-wrapped paths with their corresponding unwrapped versions within the computed linkage L.

As for the others, rule `L-Self` computes the complete linkage L for a relative path. Finally, rules `L-Prog-S` and `L-Prog-D` compute the corresponding complete static or dynamic linkage for path `prog` by parsing the program p. Parsing also includes a process to "unfold" any wildcard cases within `cases` constructs. Each wildcard case is replaced by the explicit set of cases it implicitly covers within the given family, with the same (wildcard) handler for each case. This way, the wildcard case in a base family does not apply in a blanket fashion to any future extensions. Any derived families must provide explicit or implicit handling of all new cases in order for the match to be exhaustive.

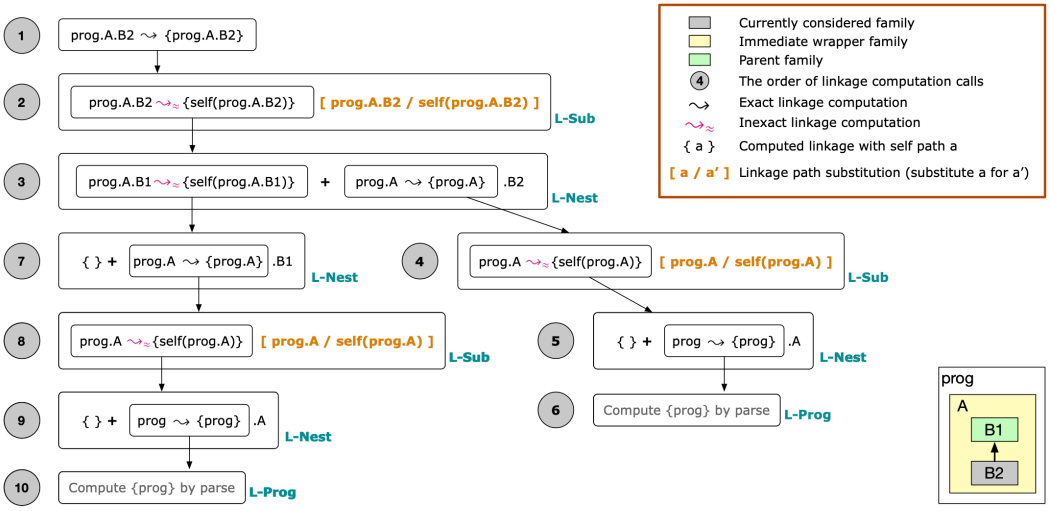

Figure 1.19: A step-by-step example of linkage computation. Here, we start at step 1 to compute the complete linkage for family `B2` nested within family `A`. Family `B2` extends family `B1` which is also nested in `A`. Each subsequent step shows the linkage computation rule applied, as well as any recursive computation calls triggered by the rule.

Linkage Computation Example. We show a step-by-step example of linkage computation in Figure 1.19. In this example, the program consists of a family `A`, which nests families `B1` and `B2`. Family `B2` extends `B1`, as shown in the inheritance diagram in the bottom right corner. To compute the exact linkage for family path `prog.A.B2` (step 1), we must first apply the `L-Sub` rule, which will compute the corresponding inexact linkage and perform path substitution (step 2). From the `L-Sub` rule, the `L-Nest` rule is called (step 3), which will compute the wrapper linkage (for path `prog.A`, steps 4-6), compute the parent linkage (for path `prog.A.B1`, steps 7-10), and then perform linkage concatenation. For each concatenation operation in the figure, we show the parent linkage on the left hand side, and the incomplete child linkage (retrieved from the complete wrapper linkage) on the right hand side. When there is no parent path, we use { } to denote an empty parent linkage. The label `L-Prog` generalizes over the two rules that perform program parsing, `L-Prog-S` and `L-Prog-D`. In our implementation, we cache the computed linkages for efficiency, which means that the exact linkage for path `prog.A` would be computed and cached in steps 4-6, and later retrieved in step 7.

Linkage Concatenation. Linkage computation relies on *linkage concatenation* to compose code from base and derived families, thereby enabling inheritance and extension. Concatenation takes the form $L_1 + L_2 = L_3$, where L_1 is the complete linkage for the parent family, L_2 is the incomplete linkage for the derived family, and L_3 is the resulting complete linkage for the derived family. L_3 includes all constructs inherited from the parent family, newly defined or extended constructs in the derived family, and constructs in the derived family that overwrite inherited constructs. We recursively propagate the concatenation operation to all nested components of the linkages: sets of nested families, types, ADTs, etc. All paths in the parent linkage that refer to the parent family are updated to refer to the derived family via *path substitution*, before concatenation. This ensures that inherited code is safe for use with the extended types in the derived family. Our rules for path substitution are available in Figure 1.20.

We show the linkage concatenation rules that are the same for both static and dynamic linkages in Figure 1.21. Any nested family linkages are recursively concatenated (rule `Cat-Nest`). Within each linkage, nested family names are mapped to their corresponding linkages (such as $A \mapsto L$). For each nested family name that exclusively appears in either the parent or the derived family, the mapping to its linkage is copied unchanged to the resulting linkage. These mappings are the symmetric difference, $\triangle$, of the two collections. However, when the same family A has a mapping in both linkages for the parent and derived families (represented by property $\mathcal{P}$ in the rule), it means that A is further bound in the derived family. We handle further binding in the same way as inheritance, via linkage concatenation. We concatenate the linkage L that A maps to in the parent linkage with the linkage L' that A maps to in the derived linkage.

The concatenation rules for record types (`Cat-Types`) and ADTs (`Cat-ADTs`) in Figure 1.21 follow a similar pattern. Types cannot be overwritten in Persimmon, as this would compromise the safety of inherited code in derived families. Thus, types and ADT definitions with the same

$\boxed{[a_2/a_1]\ a = a'}$

$$PS_{ReflPath} : [a_2/a_1]\ a_1 \ = \ a_2 \qquad PS_{ProgPath} : [a_2/a_1]\ \texttt{prog} \ = \ \texttt{prog}$$

$$PS_{RelPath} : [a_2/a_1]\ \texttt{self}(a.A) \ = \ \texttt{self}([a_2/a_1]\ a.A) \qquad PS_{AbsPath} : [a_2/a_1]\ a.A \ = \ ([a_2/a_1]\ a).A$$

$\boxed{[a_2/a_1]\ T = T'}$

$$PS_{NatType} : [a_2/a_1]\ N \ = \ N \qquad PS_{PathType} : [a_2/a_1]\ a.R \ = \ ([a_2/a_1]\ a).R$$

$$PS_{BoolType} : [a_2/a_1]\ B \ = \ B \qquad PS_{FunType} : [a_2/a_1]\ T \to T' \ = \ ([a_2/a_1]\ T) \to ([a_2/a_1]\ T')$$

$$PS_{RecType} : [a_2/a_1]\ \{(f_i : T_i)*\} \ = \ \{(f_i : ([a_2/a_1]\ T_i))*\}$$

$\boxed{[a_2/a_1]\ e = e'}$

$$PS_{Nat} : [a_2/a_1]\ n \ = \ n \qquad PS_{Bool} : [a_2/a_1]\ b \ = \ b \qquad PS_{Var} : [a_2/a_1]\ x \ = \ x$$

$$PS_{Lam} : [a_2/a_1]\ \lambda(x : T).e \ = \ \lambda(x : ([a_2/a_1]\ T)).([a_2/a_1]\ e)$$

$$PS_{App} : [a_2/a_1]\ g\ e \ = \ ([a_2/a_1]\ g)([a_2/a_1]\ e) \qquad PS_{FamFun} : [a_2/a_1]\ a.m \ = \ ([a_2/a_1]\ a).m$$

$$PS_{Proj} : [a_2/a_1]\ e.f \ = \ ([a_2/a_1]\ e).f \qquad PS_{Cases} : [a_2/a_1]\ a.c \ = \ ([a_2/a_1]\ a).c$$

$$PS_{Rec} : [a_2/a_1]\ \{(f_i = e_i)*\} \ = \ \{(f_i = ([a_2/a_1]\ e_i))*\}$$

$$PS_{Constr} : [a_2/a_1]\ a.R(\{(f_i = e_i)*\}) \ = \ ([a_2/a_1]\ a.R)(\{(f_i = ([a_2/a_1]\ e_i))*\})$$

$$PS_{ADT} : [a_2/a_1]\ a.R(C\ \{(f_i = e_i)*\}) \ = \ ([a_2/a_1]\ a.R)(C\ \{(f_i = ([a_2/a_1]\ e_i))*\})$$

$$PS_{Match} : [a_2/a_1]\ \texttt{match}\ e\ \texttt{with}\ a.c\ \{(f_i = e_i)*\} \ = \ \texttt{match}\ ([a_2/a_1]\ e)\ \texttt{with}\ ([a_2/a_1]\ a.c)\ ([a_2/a_1]\ \{(f_i = e_i)*\})$$

$$PS_{If} : [a_2/a_1]\ \texttt{if}\ e\ \texttt{then}\ g\ \texttt{else}\ g' \ = \ \texttt{if}\ ([a_2/a_1]\ e)\ \texttt{then}\ ([a_2/a_1]\ g)\ \texttt{else}\ ([a_2/a_1]\ g')$$

Figure 1.20: Path substitution relation.

name in both the base and derived families are treated as extensions in the derived family. After concatenation, the resulting sets of record types and ADTs include all definitions inherited from the parent, newly introduced by the extension, or extended in the derived family. For record types, we define the concatenation operation + in the usual way, with no duplicate fields allowed. Concatenation for ADT definitions works similarly, with the additional constraint that constructor names cannot be duplicated.

Concatenation for function and `cases` signatures is shown by `Cat-Funs-S` and `Cat- Cases-S` in Figure 1.22. The resulting set of signatures contains the symmetric difference of the signatures. Any functions with the same signature are treated as overwritten in the extension. For `cases`, we allow overwriting when the definition in the derived family has the same name, scrutinee type, and

$$\frac{\begin{array}{c} NEST\ [sp'\ /\ sp]\ +\ NEST'\ =\ NEST'' \\ TYPES\ [sp'\ /\ sp]\ +\ TYPES'\ =\ TYPES'' \\ DEFS\ [sp'\ /\ sp]\ +\ DEFS'\ =\ DEFS'' \\ ADTS\ [sp'\ /\ sp]\ +\ ADTS'\ =\ ADTS'' \\ FUNS\ [sp'\ /\ sp]\ +\ FUNS'\ =\ FUNS'' \\ CASES\ [sp'\ /\ sp]\ +\ CASES'\ =\ CASES'' \end{array}}{\left\{\begin{array}{c} \texttt{self} = sp \\ \texttt{super} = a \\ NEST \\ TYPES \\ DEFS \\ ADTS \\ FUNS \\ CASES \end{array}\right\} + \left\{\begin{array}{c} \texttt{self} = sp' \\ \texttt{super} = a' \\ NEST' \\ TYPES' \\ DEFS' \\ ADTS' \\ FUNS' \\ CASES' \end{array}\right\} = \left\{\begin{array}{c} \texttt{self} = sp' \\ \texttt{super} = a' \\ NEST'' \\ TYPES'' \\ DEFS'' \\ ADTS'' \\ FUNS'' \\ CASES'' \end{array}\right\}} \quad \text{(Cat-Top)}$$

$$\frac{\begin{array}{c} NEST'' = \{A \mapsto L \in NEST \triangle NEST'\} \cup \{A \mapsto L'' : \mathcal{P}(A, L'')\} \\ \mathcal{P}(A, L'') = A \mapsto L\ \in\ NEST\ \wedge\ A \mapsto L'\ \in\ NEST'\ \wedge\ L + L' = L'' \end{array}}{NEST + NEST' = NEST''} \quad \text{(Cat-Nest)}$$

$$\frac{\begin{array}{c} TYPES''\ =\ \{R \mapsto \{(f_k : T_k)*\} \in TYPES \triangle TYPES'\} \cup \{R \mapsto \{(f_k : T_k)*\} : \mathcal{P}(R, \{(f_k : T_k)*\})\} \\ \mathcal{P}(R, \{(f_k : T_k)*\}) = R \mapsto \{(f_i : T_i)*\}\ \in\ TYPES\ \wedge\ R \mapsto \{(f_j : T_j)*\}\ \in\ TYPES'\ \wedge \\ \{(f_i : T_i)*\} + \{(f_j : T_j)*\} = \{(f_k : T_k)*\} \end{array}}{TYPES + TYPES' = TYPES''} \quad \text{(Cat-Types)}$$

$$\frac{\begin{array}{c} ADTS''\ =\ \{R \mapsto \overline{C_k\ \{(f_n : T_n)*\}} \in ADTS \triangle ADTS'\} \cup \{R \mapsto \overline{C_k\ \{(f_n : T_n)*\}} : \mathcal{P}(R,\ \overline{C_k\ \{(f_n : T_n)*\}})\} \\ \mathcal{P}(R,\ \overline{C_k\ \{(f_n : T_n)*\}}) = R \mapsto \overline{C_i\ \{(f_j : T_j)*\}}\ \in\ ADTS\ \wedge\ R \mapsto \overline{C'_i\ \{(f'_j : T'_j)*\}}\ \in\ ADTS'\ \wedge \\ \overline{C_i\ \{(f_j : T_j)*\}} + \overline{C'_i\ \{(f'_j : T'_j)*\}} = \overline{C_k\ \{(f_n : T_n)*\}} \end{array}}{ADTS + ADTS' = ADTS''} \quad \text{(Cat-ADTs)}$$

Figure 1.21: Linkage concatenation (common rules).

$$\frac{DEFS''_S \;=\; \{R \mapsto (f_k)* \in DEFS_S \;\triangle\; DEFS'_S\} \cup \{R \mapsto (f_k)* : \mathcal{P}(R,(f_k)*)\} \quad \mathcal{P}(R,(f_k)*) = R \mapsto (f_i)* \;\in\; DEFS_S \;\wedge\; R \mapsto (f_j)* \;\in\; DEFS'_S \;\wedge\; (f_i)* \;+\; (f_j)* = (f_k)*}{DEFS_S + DEFS'_S = DEFS''_S} \text{ (Cat-Defaults-S)}$$

$$\frac{FUNS''_S \;=\; \{m \mapsto T \to T' \in FUNS_S \;\triangle\; FUNS'_S\} \cup \{m \mapsto T \to T' \in FUNS_S \bigcap FUNS'_S\}}{FUNS_S + FUNS'_S = FUNS''_S} \text{ (Cat-Funs-S)}$$

$$\frac{\begin{array}{c}CASES''_S \;=\; \{c \mapsto (\langle a.R\rangle, T \to T') \in CASES_S \;\triangle\; CASES'_S\} \;\cup \{c \mapsto (\langle a.R\rangle, T \to T') \in CASES_S \;\bigcap\; CASES'_S\} \cup \\ \{c \mapsto (\langle a.R\rangle, T \to T''') : \mathcal{P}(c, \langle a.R\rangle, T \to T''')\} \\ \mathcal{P}(c, \langle a.R\rangle, T \to T''') = c \mapsto (\langle a.R\rangle, T \to T') \;\in\; CASES_S \;\wedge\; c \mapsto (\langle a.R\rangle, T \to T'') \;\in\; CASES'_S \;\wedge\; T' + T'' = T'''\end{array}}{CASES_S + CASES'_S = CASES''_S} \text{ (Cat-Cases-S)}$$

$$\frac{\begin{array}{c}DEFS''_D \;=\; \{R \mapsto \{(f_k = v_k)*\} \in DEFS_D \;\triangle\; DEFS'_D\} \cup \{R \mapsto \{(f_k = v_k)*\} : \mathcal{P}(R, \{(f_k = v_k)*\})\} \\ \mathcal{P}(R, \{(f_k = v_k)*\}) = R \mapsto \{(f_i = v_i)*\} \;\in\; DEFS_D \;\wedge\; R \mapsto \{(f_j = v_j)*\} \;\in\; DEFS'_D \;\wedge \\ \{(f_i = v_i)*\} + \{(f_j = v_j)*\} = \{(f_k = v_k)*\}\end{array}}{DEFS_D + DEFS'_D = DEFS''_D} \text{ (Cat-Defaults-D)}$$

$$\frac{\begin{array}{c}FUNS''_D \;=\; \{m \mapsto (T \to T', \lambda(x:T).e) \in FUNS_D \;\triangle\; FUNS'_D\} \cup \\ \{m \mapsto (T \to T', \lambda(x:T).e) : \mathcal{P}(m, (T \to T', \lambda(x:T).e))\} \\ \mathcal{P}(m, (T \to T', \lambda(x:T).e)) = m \mapsto (T \to T', \lambda(x':T).e') \;\in\; FUNS_D \;\wedge\; m \mapsto (T \to T', \lambda(x:T).e) \;\in\; FUNS'_D\end{array}}{FUNS_D + FUNS'_D = FUNS''_D} \text{ (Cat-Funs-D)}$$

$$\frac{\begin{array}{c}CASES''_D \;=\; \{c \mapsto (\langle a.R\rangle, T \to T', \lambda(x:T).e) \in CASES_D \;\triangle\; CASES'_D\} \cup \\ \{c \mapsto (\langle a.R\rangle, T \to T', \lambda(x:T).e) : \mathcal{P}(c, (\langle a.R\rangle, T \to T', \lambda(x:T).e)) \vee \mathcal{P}'(c, (\langle a.R\rangle, T \to T', \lambda(x_f:T).e))\} \\ \mathcal{P}(c, (\langle a.R\rangle, T \to T', \lambda(x:T).e)) = \\ c \mapsto (\langle a.R\rangle, T \to T', \lambda(x':T).e') \;\in\; CASES_D \;\wedge\; c \mapsto (\langle a.R\rangle, T \to T', \lambda(x:T).e) \;\in\; CASES'_D \\ \mathcal{P}'(c, (\langle a.R\rangle, T \to T''', \lambda(x_f:T).e)) = \\ c \mapsto (\langle a.R\rangle, T \to T', \lambda(x':T).e') \;\in\; CASES_D \;\wedge\; c \mapsto (\langle a.R\rangle, T \to T'', \lambda(x'':T).e'') \;\in\; CASES'_D \;\wedge \\ T' + T'' = T''' \;\wedge\; \texttt{fresh}\; x_f \;\wedge\; e'[x_f/x'] + e''[x_f/x''] = e\end{array}}{CASES_D + CASES'_D = CASES''_D} \text{ (Cat-Cases-D)}$$

Figure 1.22: Linkage concatenation (specialized rules).

arrow type. When a `cases` definition is extended, its resulting output type is a concatenation of the output types from the base and derived families. Similarly, the rules for extending function and `cases` definitions are included in Figure 1.22 as `Cat-Funs-D` and `Cat-Cases-D`. We concatenate `cases` constructs by concatenating their records of case handlers, after replacing the bound variable representing the match context inside each definition with a fresh variable. We also ensure that after extension the `cases` construct does not have any duplicate case handlers.

```
Family A1 {
  Family B1 {
    type Exp = ENat {n : N}
    val f: N -> N = λ(n: N). n
    val ev: Exp -> N =
      λ(e: Exp). match e with evc {}
    cases evc <Exp>: {} -> {ENat: {n: N} -> N} =
      λ(unit: {}). {ENat = λ(x: {n: N}). x.n}
  }
  Family B2 extends .B1 {
    val f: N -> N = λ(n: N). n+1
  }
}
```

```
Family A2 extends A1 {
  Family B1 {
    val f: N -> N = λ(n: N). n+2
  }
  Family B2 extends .B1 {
    type X = {x: B}
    type Exp += EPlus {e1: Exp, e2: Exp}
    cases evc <Exp>: {} -> {EPlus: {e1: Exp, e2: Exp} -> N} +=
      λ(unit: {}).
        {EPlus = λ(x: {e1: Exp, e2: Exp}).(ev(x.e1) + ev(x.e2))}
  }
}
```

Figure 1.23: PERSIMMON code snippet exhibiting both inheritance and further binding.

Precedence of further binding. In our system, further binding takes precedence over inheritance, mirroring other related systems with nested inheritance, such as Jx [Nystrom et al., 2004]. Consider family `A2.B2` in Figure 1.23, which extends family `A2.B1`, and further binds family `A1.B2`. The function f is defined in both `A2.B1` and `A1.B2`, but the definition in `A1.B2` (further bound) takes precedence, since `A1.B2` is considered structurally more similar to `A2.B2`. Rule `Cat-Nest`, along with the linkage computation rule `L-Nest` (Figure 1.18), ensures this. When we compute the complete linkage for `A2.B2` via `L-Nest`, we concatenate the complete parent linkage L' (for `A2.B1`) with the incomplete child linkage L'' (for `A2.B2`). The latter is retrieved from the complete linkage L for the wrapper family, `A2`. Further binding of any nested families is performed by rule `Cat-Nest` when L – the linkage for the wrapper family – is computed. Thus, any further bound nested components will be on the right hand side of concatenation in `Cat-Nest`, taking precedence over the inherited components on the left hand side.

1.6 Formal Results

We prove that Persimmon is sound by proving progress and preservation for our calculus.

Theorem 1 (Progress)**.** *For any main expression e in program p, if* $[]; [] \vdash p : T$ *and* $[\texttt{prog}]; [] \vdash e : T'$*, then either e is a value or there exists some e' such that* $[\texttt{prog}] \vdash e \longrightarrow e'$.

Theorem 2 (Preservation)**.** *For any main expression e in program p, if* $[]; [] \vdash p : T$*,* $[\texttt{prog}]; [] \vdash e : T'$*, and forall e' such that* $[\texttt{prog}] \vdash e \longrightarrow e'$*, then* $[\texttt{prog}]; [] \vdash e' : T'$.

We prove these properties by induction on the typing derivation $[\texttt{prog}]; [] \vdash e : T'$. For progress, most cases follow directly from our operational semantics. For preservation, most cases are handled in a straightforward way using induction hypotheses for sub-derivations. Proof cases for rules `T-FamFun` and `T-Cases` rely on the fact that function and `cases` definitions retrieved from linkages are well-typed. We show this by proving that linkages parsed from well-typed programs are well-formed, and well-formedness is preserved by linkage concatenation. The full proofs are available in Appendix B.

1.7 Compilation to Scala

We have implemented a prototype compiler for Persimmon. The compiler consists of about 2,300 lines of Scala code. Code generation works by translating Persimmon code into Scala code. Scala is already a powerful language with advanced, statically typed code reuse and extensibility mechanisms. However, it is not powerful enough to support Persimmon's nested family polymorphism and extensible variant types out of the box. Therefore, to enable code sharing, our compiler has to parameterize code with explicit extensibility hooks, use wrapper types and trampoline procedures to make dispatching explicit, and insert run-time type casts.

```scala
import reflect.Selectable.reflectiveSelectable
object A2$B2 {
  // Types
  type X = {val x: Boolean}
  // ADTs
  sealed trait Exp
  // Defined constructors
  case class EPlus[self$$$Exp](e2: self$$$Exp, e1: self$$$Exp)
    extends Exp
  // Inherited constructors
  case class A2$B1$$Exp(inherited: A2$B1.Exp) extends Exp {
    override def toString(): String = inherited.toString()
  }
  case class A1$B2$$Exp(inherited: A1$B2.Exp) extends Exp {
    override def toString(): String = inherited.toString()
  }
  // Path interface
  trait Interface extends A2$B1.Interface
    with A1$B2.Interface {
      self$ =>
      // Self Named types
      type X
      // Self ADTs
      type Exp
      // Functions
      val ev: self$.Exp => Int
      val f: Int => Int
      // Cases
      def evc(matched: self$.Exp): Unit => Int
      // Translations
      def A2$B2$$Exp(from: A2$B2.Exp): Exp
  }
  // Path implementation
  object Family extends A2$B2.Interface { self$ =>
    // Self named types instantiation
    override type X = A2$B2.X
    // Self ADTs instantiation
    override type Exp = A2$B2.Exp
    // Function implementations
    override val ev: self$.Exp => Int = ev$Impl(A2.Family,self$)
    def ev$Impl(self$1: A2.Interface, self$: A2$B2.Interface):
      self$.Exp => Int = A1$B1.Family.ev$Impl(self$1, self$)
    override val f: Int => Int = f$Impl(A2.Family, self$)
    def f$Impl(self$1: A2.Interface, self$: A2$B2.Interface):
      Int => Int = A1$B2.Family.f$Impl(self$1, self$)
    // Cases implementations
    def evc(matched: self$.Exp): Unit => Int =
      evc$Impl(A2.Family,self$)(matched.asInstanceOf[A2$B2.Exp])
    def evc$Impl(self$1: A2.Interface, self$: A2$B2.Interface)
      (matched: A2$B2.Exp): Unit => Int =
      (unit: Unit) => matched match {
      case matched@A2$B2.EPlus(_, _) =>
        val x: A2$B2.EPlus[self$.Exp] =
          matched.asInstanceOf[A2$B2.EPlus[self$.Exp]]
        (self$.ev.asInstanceOf[self$.Exp => Int](x.e1) +
         self$.ev.asInstanceOf[self$.Exp => Int](x.e2))
      case A2$B2.A2$B1$$Exp(inherited) =>
        A2$B1.Family.evc$Impl(self$1, self$)(inherited)(unit)
      case A2$B2.A1$B2$$Exp(inherited) =>
        A1$B2.Family.evc$Impl(self$1, self$)(inherited)(unit)
    }
    // Translation function implementations
    override def A2$B2$$Exp(from: A2$B2.Exp): Exp = from
    override def A2$B1$$Exp(from: A2$B1.Exp): Exp =
      A2$B2.A2$B1$$Exp(from)
    override def A1$B1$$Exp(from: A1$B1.Exp): Exp =
      A2$B2.A2$B1$$Exp(A2$B1.Family.A1$B1$$Exp(from))
    override def A1$B2$$Exp(from: A1$B2.Exp): Exp =
      A2$B2.A1$B2$$Exp(from)
  }
}
```

Figure 1.24: The translation of PERSIMMON code in Figure 1.23 to Scala code.

An excerpt of the translated code from PERSIMMON to Scala is available in Figure 1.24. A family, however nested, is compiled into a top-level Scala "trait." Each extensible variant type is compiled into a "sealed trait," with each constructor a "case class" and with case classes for inherited constructors. The translation functions enable converting from an inherited instance through a chain of inherited constructors.

The trait `Interface` generated for each family provides a layer of abstraction for each family's constructs, so that they can be safely reused in future extensions. The singleton object `Family`, which implements the interface, then provides definitions for all of the constructs. In the singleton, helper functions ending with `$Impl` are generated for the actual right-hand side implementations. These helper functions are parameterized by a list of selfs, breaking down the path of a family from the outermost `self$1` to the innermost `self$` families. As needed, these helper functions perform explicit dispatching to the relevant extending or further binding family.

1.8 Evaluation

In this section, we revisit our design goals and show how our solution meets these goals. We also compare Persimmon to existing extensibility solutions, namely object-oriented decomposition [Zenger & Odersky, 2004] and compositional programming [Zhang et al., 2021]. Finally, we include a case study of mixin compilers, showcasing the expressive power of Persimmon that is not easily replicated by other solutions.

1.8.1 Design Goals

We aimed to achieve the following design goals with our solution, in addition to the classic goal of **type safety**:

- **Extensibility at scale**. We believe that extensibility should exist at the large scale of reusable, nested components. Persimmon supports this through nested family polymorphism. All structural and hierarchical relationships between families are preserved during inheritance. We showcase the inheritance and extension of nested components in Persimmon with our extensible compilers example in Figure 1.7.

- **Scalable extensibility**. We believe that support for extensibility should not come at the cost of parameter clutter and painstaking advance preparation by the user. Code should look similar in the presence and absence of extensions, and dependencies between components should be minimized. Persimmon supports this by treating most constructs as built-in extensibility hooks. Relative path types and path substitution keep inherited code type-safe, while keeping the names of base and derived constructs consistent. In order to highlight the user-friendly aspects of our approach as well as the expressive power of Persimmon, we encode a case study

from the work on independently extensible solutions [Zenger & Odersky, 2004] in Figure 1.25, and discuss the comparison below.

- **Mutual recursion**. We believe that the nested components of a program should support mutually recursive references to constructs in other components. PERSIMMON supports this via our algorithmic linkage mechanism. Linkages provide a way to look up names and signatures of all constructs for type checking, including inherited and extended constructs, which may not be easily available in other representations.

- **Composable extensions**. We believe that parallel extensions should be composable, promoting code reuse and minimizing linear dependencies between families. Since nested family polymorphism has the expressive power to encode mixins, PERSIMMON supports composable extensions via an encoding (as shown in the case study in Figure 1.25). Since mixins in PERSIMMON are encoded as families, they can themselves nest families to arbitrary depth. Inheritance of nested components makes mixins in PERSIMMON especially powerful, allowing us to express examples such as the mixin compilers in Section 1.8.4.

- **Idiomatic functional style**. We believe that extensible programming should feel natural to a functional programmer and be user-friendly to novices. Programmers can enjoy the familiar functional style in PERSIMMON, while novices can enjoy the convenience of built-in extensible constructs.

1.8.2 Comparison to the Independently Extensible Solutions

Zenger & Odersky [2004] propose two independently extensible solutions expressed in Scala: object-oriented decomposition and functional decomposition. With the first approach, new variants can be added easily using shallow mixin composition, but adding functionality requires deep mixin

```
1  (* Base code *)
2  Family Base {
3    type Exp = Num {v: N}
4    def eval: Exp -> N =
5      case Num(v: N) = v
6  }
7  (* Adding variants: plus and negation *)
8  Mixin BasePlus extends Base {
9    type Exp += Plus {l: Exp,r: Exp}
10   def eval: Exp -> N +=
11     case Plus(l: Exp,r: Exp) =
12         (eval l) + (eval r)
13 }
14 Mixin BaseNeg extends Base {
15   type Exp += Neg {t: Exp}
16   def eval: Exp -> N +=
17     case Neg(t: Exp) = -(eval t)
18 }
19 Family BasePlusNeg extends Base
20   with BasePlus, BaseNeg {}
```

```
21 (* Adding functionality: printing *)
22 Mixin ShowPlusNeg extends BasePlusNeg {
23   def show: Exp -> Str =
24     case Num(v: N) = "" + v
25     case Plus(l: Exp,r: Exp) =
26         (show l) + "+" + (show r)
27     case Neg(t: Exp) =
28         "-(" + (show t) + ")"
29 }
30 (* Adding functionality: doubling *)
31 Mixin DblePlusNeg extends BasePlusNeg {
32   def dble: Exp -> Exp =
33     case Num(v: N) = Exp(Num{v = v * v})
34     case Plus(l: Exp,r: Exp) =
35         Exp(Plus{l=dble(l),r=dble(r)})
36     case Neg(t: Exp) = Exp(Neg{t = dble(t)})
37 }
38 (* Mixing in both sets of added functionality *)
39 Family ShowDblePlusNeg extends BasePlusNeg
40   with ShowPlusNeg, DblePlusNeg {}
```

Figure 1.25: Code in Persimmon for the object-oriented decomposition case study by Zenger & Odersky [2004].

composition. On the other hand, the functional approach easily accommodates adding functionality, but variants must be added via deep mixin composition. Functional decomposition also requires the use of the Visitor pattern, adding extra code that is not relevant to the semantics of the extensions.

Our solution offers multiple advantages. The Persimmon code corresponding to the object-oriented decomposition example is shown in Figure 1.25. We also include the original example in Figure 1.26 for comparison. In our language, we can add both variants and functionality via shallow mixin composition (lines 19 and 39 in Figure 1.25), eliminating the need for deep mixin composition. Since Persimmon uses the same technique for extensibility in both dimensions, the user need not choose which dimension (variants or functionality) to prioritize. Another advantage of Persimmon is that ADT constructors do not need to be manually re-parameterized by the extended type when functionality is added. For example, in object-oriented decomposition, adding functionality requires all variants to be restated to resolve the abstract expression type to the appropriate (extended) concrete type. For an example of this, see `trait DblePlusNeg` on line 30 in Figure 1.26. In Persimmon, this resolution is accomplished automatically by path substitution

```
1  (* Base code *)
2  trait Base {
3      type exp <: Exp;
4      trait Exp { def eval: int }
5      class Num(v: int) extends Exp {
6          val value = v;
7          def eval = value }}
8  (* Adding variants: plus and negation *)
9  trait BasePlus extends Base {
10     class Plus(l: exp,r: exp) extends Exp {
11         val left = l; val right = r;
12         def eval = left.eval + right.eval }}
13 trait BaseNeg extends Base {
14     class Neg(t: exp) extends Exp {
15         val term = t;
16         def eval = - term.eval; }}
17 trait BasePlusNeg extends BasePlus with BaseNeg;
18 (* Adding functionality: printing *)
19 trait Show extends Base {
20     type exp <: Exp;
21     trait Exp extends super.Exp { def show: String; }
22     class Num(v: int) extends super.Num(v) with Exp {
23         def show = value.toString(); }}
24 trait ShowPlusNeg extends BasePlusNeg with Show {
25     class Plus(l: exp,r: exp) extends super.Plus(l,r) with Exp {
26             def show = left.show + "+" + right.show; }
27     class Neg(t: exp) extends super.Neg(t) with Exp {
28         def show = "-(" + term.show + ")"; }}
29 (* Adding functionality: doubling *)
30 trait DblePlusNeg extends BasePlusNeg {
31     type exp <: Exp;
32     trait Exp extends super.Exp { def dble: exp; }
33     def Num(v: int): exp;
34     def Plus(l: exp,r: exp): exp;
35     def Neg(t: exp): exp;
36     class Num(v: int) extends super.Num(v) with Exp {
37         def dble = Num(v*2) }
38     class Plus(l: exp,r: exp) extends super.Plus(l,r) with Exp {
39         def dble = Plus(left.dble, right.dble); }
40     class Neg(t: exp) extends super.Neg(t) with Exp {
41         def dble = Neg(t.dble); }}
42 (* Mixing in both sets of added functionality via deep mixin composition *)
43 trait ShowDblePlusNeg extends ShowPlusNeg with DblePlusNeg {
44     type exp <: Exp;
45     trait Exp extends super[ShowPlusNeg].Exp with super[DblePlusNeg].Exp;
46     class Num(v: int) extends super[ShowPlusNeg].Num(v)
47         with super[DblePlusNeg].Num(v) with Exp;
48     class Plus(l: exp,r: exp) extends super[ShowPlusNeg].Plus(l,r)
49         with super[DblePlusNeg].Plus(l,r) with Exp;
50     class Neg(t: exp) extends super[ShowPlusNeg].Neg(t)
51         with super[DblePlusNeg].Neg(t) with Exp; }
```

Figure 1.26: Object-oriented decomposition by Zenger & Odersky [2004].

and does not require any effort from the user (see `Mixin DblePlusNeg` on line 31 in Figure 1.25). Finally, extensibility in Persimmon does not rely on the use of programming patterns, further reducing user burden and improving code readability.

1.8.3 Comparison to Compositional Programming

```
1  (* Base functionality *)
2  Family Base {
3    type Exp = Lit {n: N} | Add {e1: Exp, e2: Exp}
4    def evalNum: Exp -> N =
5      case Lit (n: N) = n
6      case Add (e1: Exp, e2: Exp) = (evalNum e1) + (evalNum e2)
7  }
8  (* Adding functionality: printing *)
9  Mixin Print extends Base {
10   def printNum: Exp -> Str =
11     case Lit (n: N) = "" + n
12     case Add (e1: Exp, e2: Exp) = "(" + (printNum e1) + "+" + (printNum e2) + ")"
13 }
14 Family BasePrint extends Base with Print {}
15 (* Adding variants: multiplication *)
16 Family BasePrintMul extends BasePrint {
17   type Exp += Mul {e1: Exp, e2: Exp}
18   def evalNum: Exp -> N +=
19     case Mul(e1: Exp, e2: Exp) = (evalNum e1) * (evalNum e2)
20   def printNum: Exp -> Str +=
21     case Mul(e1: Exp, e2: Exp) = "(" + (printNum e1) + "*" + (printNum e2) + ")"
22 }
```

Figure 1.27: Code in Persimmon for the compositional programming case study by Zhang et al. [2021].

Zhang et al. [2021] propose compositional programming (CP): a new, highly modular programming style that is presented as an alternative to functional and object-oriented styles. This solution, while not presented as "functional," does support extensible variant types as well as nested family polymorphism via a unifying notion of first-class traits. An instance of an object that supports both extended variants and extended functionality can be created using nested composition of traits. We include an example from this work in Figure 1.28, and the Persimmon version of the same example in Figure 1.27. Persimmon differs from CP in some important ways. Persimmon treats types as members of the family, while CP allows only top-level type definitions. In Persimmon,

```
(* Base code *)
type ExpSig<Exp> = {
    Lit: Int -> Exp;
    Add: Exp -> Exp -> Exp;
};
type Eval = {eval: Int};
evalNum = trait implements ExpSig<Eval> => {
    (Lit n).eval = n;
    (Add e1 e2).eval = e1.eval + e2.eval;
};
(* Adding functionality: printing *)
type Print = {print: String};
printNum = trait implements ExpSig<Print> => {
    (Lit n).print = n.toString;
    (Add e1 e2).print = "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
(* Adding variants: multiplication *)
type MulSig<Exp> extends ExpSig<Exp> = {
    Mul: Exp -> Exp -> Exp;
}
evalMul = trait implements MulSig<Eval> inherits evalNum => {
    (Mul e1 e2).eval = e1.eval * e2.eval;
};
printMul = trait implements MulSig<Print> inherits printNum => {
    (Mul e1 e2).print = "(" ++ e1.print ++ "*" ++ e2.print ++ ")";
}
(* Constructing the expression *)
e = new evalNum ,, printNum ,, expAdd @(Eval&Print);
```

Figure 1.28: Compositional programming example by Zhang et al. [2021].

users can define types at the exact nesting level where they are needed, while avoiding excessive type parameterization and explicit type applications. Type instances in Persimmon can be constructed by simply using the name of the type; there is no need for manual composition of traits on the part of the user. Type members in Persimmon are also quite expressive, as they can refer to themselves and other type members recursively. The nested compilers example in Figure 1.7 is more difficult to model in CP, as the type members of the nested families are recursive within the family. CP requires explicit parameterization to express mutually defined data types. Finally, while CP strongly enforces the separation of interfaces and implementations, Persimmon takes the more familiar functional programming approach: both the interface and the implementation are specified within the family.

1.8.4 Case Study: Mixin Compilers

```
1 Family BaseComp { (* contents of the base compiler *) }
2 Mixin IfExt extends BaseComp { (* mixin compiler that supports if−statements *) }
3 Mixin ArithExt extends BaseComp { (* mixin compiler that supports arithmetic *) }
4 Family IfArithComp extends BaseComp with IfExt, ArithExt {}
```

Finally, we highlight the expressive power of mixins in Persimmon with a case study of mixin compilers. Mixin compilers are extensible compilers that are also composable in parallel. We build on the example in Figure 1.7, while making each compiler itself a mixin. Persimmon mixins are themselves families and can thus contain nested families, which can be inherited and extended upon mixin composition. We include a partial implementation in Figure 1.29.

In comparison, other closely related works cannot support this example quite as elegantly. For example, FPOP [Jin et al., 2023] does not support nested family polymorphism or unrestricted mutually recursive references between families, due to its application in a proof assistant. Compositional programming [Zhang et al., 2021] supports nested family polymorphism, but does not support the use of types as nested family members, making it difficult to represent types that are recursive via the family.

1.9 Related Work

Family Polymorphism and Nested Inheritance. Families can be represented in object-oriented systems using an *object-based* or a *class-based* approach. The seminal object-based solution by Ernst [2001] represents families as enclosing *family objects*, while the class attributes of the object comprise family members. With this approach, any number of object instances (and thus, families) can exist at runtime. In the class-based approach, proposed in .FJ by Igarashi et al. [2005], a family is associated with the class itself. This approach restricts the number of families at run time, but has a more straightforward implementation as a nested class system: families are top-level classes,

```
Family STLCArith extends STLCBase {
    type Ty += TInt
    type Val += Int(i: N)
    type Exp += EPlus(e1: Exp, e2: Exp)
    def eval: Exp -> Option Val +=
      case EPlus(e1, e2) = (eval e1) + (eval e2)
    (* extensions to subst, tc, print, etc *)
}
(* Compiler extension: add integers and addition to STLC and ILs *)
Mixin ArithExt extends BaseComp {
  Family STLC extends STLCArith {}
  Family IL {
    type Ty += TInt
    type Val += Int(i: N)
    type Exp += EPlus(e1: Exp, e2: Exp)
    def eval(fs: List Fun): Exp -> Option Val += ... (* new EPlus case *)
    def apply(fs: List Fun, vs: List Val): Val -> Option Val +=
      ... (* new Int case *)
    ... (* subst, tc, print, etc. *)
  }
  Family ILC extends IL {
    def unpack(fs: List Fun, □: Str, x: Str, e: Exp):
        Val -> Option Val +=
      case Int(i) = None
    ... (* subst, tc, print, etc. *)
  }
  def cps_val(k: ILK.Val): STLC.Val -> ILK.Exp += ... (* new cases *)
  def cps_exp(k: ILK.Val): STLC.Exp -> ILK.Exp += ... (* new cases *)
  def cc_val: ILK.Val -> (List ILC.Fun, ILC.Val) += ... (* new cases *)
  def cc_exp: ILK.Exp -> (List ILC.Fun, ILC.Exp) += ... (* new cases *)
}
```

Figure 1.29: A mixin compiler that adds support for integers and addition in extended Persimmon syntax.

and family members are nested classes. Persimmon is inspired by the class-based approach, with top-level families nesting other families, types, functions, and `cases` as family members. Families in Persimmon are not types, and path types cannot be subtypes of each other due to potential unsafe uses [Ernst, 2001; Igarashi et al., 2005]. This differs from a follow up work by Igarashi & Viroli [2007], where variant path types reconcile class-based family polymorphism with subtyping.

Jx, .FJ, *vc*, Tribe, and Familia are all class-based systems that support type-safe, nested family polymorphism [Nystrom et al., 2004; Zhang & Myers, 2017; Igarashi et al., 2005; Ernst et al., 2006; Clarke et al., 2007]. Jx utilizes the notion of containers and their inheritable components (including nested containers), *vc* and Tribe are based around virtual classes, while Familia unifies the genericity

mechanisms of inheritance and parametric polymorphism. These systems differ from Persimmon in that they follow the object-oriented approach and support extensible variant types via subclasses, while Persimmon supports extensible variant types through functional programming mechanisms. Persimmon guarantees that pattern matching is type safe in the presence of extended variants and nested family polymorphism by introducing `cases`, which are direct family members and are thus polymorphic to the family.

Recently, Jin et al. [2023] presented a family polymorphism design (FPOP) for extensible metatheory mechanization, including type-safe, extensible pattern matching. Unlike Persimmon, FPOP does not support nested family polymorphism, limiting its capability for modular reuse. Furthermore, Persimmon supports unrestricted mutually recursive references, while FPOP cannot support this due to its application in a proof assistant. Building on FPOP, Ebresafe et al. [2025] introduce a family polymorphic language design for the modular construction of certified compilers. A related earlier approach by Delaware et al. [2013], targets modular metatheory in Rocq via the "data types à la carte" encoding [Swierstra, 2008].

Solutions to the Expression Problem Persimmon meets some of the goals of the Expression Problem [Wadler, 1998]; namely, our calculus supports type-safe extension of data types *and* functionality over those data types. Other goals of the Expression Problem, such as modular type checking, are not met by the current calculus. Multiple works have since proposed an additional requirement that extensions should also be composable [Zenger & Odersky, 2004; Nystrom et al., 2006], which Persimmon also meets. Unlike J& [Nystrom et al., 2006], which introduces intersection types to support composable extensions, Persimmon uses a nested family encoding to support composition instead of introducing a new type. One limitation of our approach is that the encoding does impose a linear overwriting order. Zenger & Odersky [2004]; Odersky & Zenger [2005] support

composable extensions through the use of Scala traits, the Visitor pattern, and deep mixin composition. In comparison, Persimmon reduces user effort, as it does not require any setup of patterns or manual composition of mixins. Persimmon also cuts the parameter clutter by treating most constructs as built-in extensibility hooks. A recent object-oriented solution, SuperOOP, supports mixin composition and open recursion via late binding of the keywords `this` and `super` [Fan & Parreaux, 2023]. Persimmon supports open recursion via relative path types. Unlike SuperOOP, Persimmon also supports the composition of arbitrarily nested families.

Oliveira & Cook [2012] use object algebras (an abstraction related to Church encodings) and rely on simple generics, which makes the solution applicable to mainstream languages. Object algebras are powerful abstractions that can express family polymorphism. One downside is that modular composition of object algebras requires manual setup by defining a combinator. In Persimmon, extensions can be composed in parallel using our convenient mixin syntax, as in Section 1.8.4.

Related to object algebras is also the "tagless final" approach, which relies on interpreters and a skillful embedding of DSLs in the host language [Kiselyov, 2012]. Extensibility is achieved by adding syntactic forms or interpreters, and it is possible to abstract over families of interpreters [Carette et al., 2009]. The tagless final approach requires explicit parameterization of interpreter instances by the representation type, while Persimmon uses relative path types that adapt upon inheritance.

Continuing this line of work, Zhang et al. [2021] have recently proposed "compositional programming," a style for statically typed modular programming in a language design called CP, which solves the Expression Problem as well as more generally the problem of expressing dependencies in a modular way. Fan et al. [2022] later develop a type-directed operational semantics for the core calculus, streamlining the necessary proof techniques to support language features such as recursion. Compared to Persimmon, CP still requires parameterization (in particular, self-type annotations to inject dependencies). Unlike CP, types in Persimmon are family members – they

can be defined at any level of nesting, and can be recursive via the family.

Other EP solutions propose more flexible definitions of data types. For example, open data types and open functions can be scattered throughout modules, allowing the definitions to be provided at any point in the program [Löh & Hinze, 2006]. Swierstra [2008] achieves a similar goal in Haskell by encoding data types as coproducts of functors and defining extensible operations as type class instances. Polymorphic variants [Garrigue, 2000] allow constructors to exist independently of types, and support open pattern matching. In contrast, PERSIMMON keeps code safe for reuse in derived families by using relative path types to support family polymorphism. Families in PERSIMMON retain the organizational advantages of modules and support code reuse at a large scale via nested inheritance.

Extensible Variant Types and Pattern Matching Some record-based solutions rely on row polymorphism to support extensible variants [Gaster & Jones, 1996]. Gaster & Jones [1996] also propose an extension to their system, which makes pattern match cases first-class, extensible values. In a related work, [Blume et al., 2006] support extensible pattern matching and composable extensions via extensible first-class cases, capitalizing on the dual relationship between polymorphic records and sums. While these solutions support extensible variants and pattern matching, they do not support family polymorphism. In PERSIMMON, `cases` are not first-class, as their usage is restricted to application within a match expression; however, they are family members and are polymorphic to the family.

Zenger & Odersky [2001] implement extensible ADTs by providing default variants that subsume any future extensions. Their solution uses a new design pattern for extensible visitors. Pattern matching becomes extensible by delegating computation in the default case to the methods overridden in the extension. In PERSIMMON, the delegation is implicit thanks to relative path types

and path substitution. Unlike Persimmon, this solution does not support composable extensions. Recently, Zhang & Oliveira [2020] introduced a Scala-based solution using extensible generative visitors, which supports exhaustive and composable pattern matching. However, some exhaustivity checking for pattern matching must be delayed to the visitor instantiation site, whereas Persimmon ensures exhaustivity at definition.

OCaml has introduced extensible variant types as well as polymorphic variants [Garrigue, 1998]. Extensible functions can be implemented by keeping a reference to the evolving function in a polymorphic record field [Balestrieri & Mauny, 2018]. In Persimmon, functions are not extensible in the general case, but `cases` are directly extensible constructs within the family.

Extensible ML (EML), supports hierarchical, extensible data types and extensible functionality over those data types, while preserving modular type checking [Millstein et al., 2004]. Both Persimmon and EML support exhaustivity checking for pattern match expressions at definition. Syme et al. [2007] implement extensible pattern matching through the use of active patterns in $F\#$, handling both partial and total decompositions. Persimmon does not support partial patterns due to the conflict with exhaustivity checking. `match` is an extensible language which implements extensible pattern matching for Racket using macros [Tobin-Hochstadt, 2011]. JMatch [Isradisaikul & Myers, 2013] is an extension of Java that provides modal abstraction (integration of pattern matching and iteration abstractions), where patterns are not tied to constructors. Both Persimmon and JMatch ensure static exhaustivity checking, while `match` does not. Among the pattern match techniques evaluated by Emir et al. [2007], our solution is most similar to case classes in Scala. However, the shortcoming of case classes – inability to define new patterns for new variants – is addressed in Persimmon with extensible `cases`.

1.10 Conclusion

We present Persimmon, the first functional system with nested family polymorphism and extensible variant types. Nested, extensible families in Persimmon combine the benefits of modules (code modularity and reuse), the benefits of family polymorphism (type safety of inherited and extended constructs), and the benefits of composable extensions. Linkages are the engine behind extensibility in Persimmon, eliminating the need for complex type checking and operational semantics. Our explicit `cases` constructs separate match case definitions from their uses, and provide a natural mechanism for extensibility of pattern matching. Exhaustivity of pattern matching is maintained by the well-formedness checking of definitions. Since types and `cases` in Persimmon serve as built-in extensibility hooks, parameter clutter is not an issue in our language.

1.11 Extension: Modular Type Checking in Persimmon

The original design of the Persimmon type checker supported only single-file programs and contained global reasoning about the program, in particular for identifying circular inheritance (via the `ancestors` function in the well-formedness judgment). In this section, we discuss modifications to the Persimmon calculus that enable support for multi-file programs and modular type checking. The implementation supporting this extension is available at https://github.com/akravc/persimmon/tree/modular.

1.11.1 Program Fragments

In order to reason about modular type checking and multi-file programs, we introduce a notion of *program fragment* in Persimmon, inspired by related work [Cardelli, 1997]. A multi-file program $\mathscr{P}$ consists of a finite number of program fragments p_i, where each fragment represents the text of

a single file in the program. Each file contains at least one top-level family.

$$\mathscr{P} \;=\; \{p_1, p_2, p_3 ...\}$$

A *linkage representation* $\mathscr{L}$ of a program $\mathscr{P}$ consists of an incomplete, just-parsed static (L_{S_i}) or dynamic (L_{D_i}) linkage for each program fragment p_i. As shorthand, we will also call these *linkage fragments*.

$$\mathscr{L}_S \;=\; \{L_{S_1}, L_{S_2}, L_{S_3} ...\}$$

$$\mathscr{L}_D \;=\; \{L_{D_1}, L_{D_2}, L_{D_3} ...\}$$

Since during type checking we will eventually need to combine these linkage fragments via concatenation to represent dependencies, we must be able to look up which linkage (and essentially, program fragment) a certain family appears in. Thus, each linkage will now hold an additional piece of information, the list of all valid family paths in the linkage, `L.PATHS`. This will also be useful information for checking well-formedness, as we can use this list to make sure there are no duplicate family names.

$$\left\{\begin{array}{lcl}
\texttt{self} & = & a \\
\texttt{super} & = & a.A? \\
\texttt{PATHS} & = & a* \\
\texttt{NEST} & = & \{\ (A \mapsto L_S)*\ \} \\
\texttt{TYPES} & = & \{\ (R \mapsto \{(f_i : T_i)*\}\ )*\ \} \\
\texttt{DEFS} & = & \{\ (R \mapsto (f_k)*\ )*\ \} \\
\texttt{ADTS} & = & \{\ (R \mapsto \overline{C_j\ T_j})*\ \} \\
\texttt{FUNS} & = & \{\ (m \mapsto T)*\ \} \\
\texttt{CASES} & = & \{\ (c \mapsto (\langle T \rangle,\ T'))*\}
\end{array}\right\}_{L_S}
\qquad
\left\{\begin{array}{lcl}
\texttt{self} & = & a \\
\texttt{super} & = & a.A? \\
\texttt{PATHS} & = & a* \\
\texttt{NEST} & = & \{\ (A \mapsto L_D)*\ \} \\
\texttt{TYPES} & = & \{\ (R \mapsto \{(f_i : T_i)*\}\ )*\ \} \\
\texttt{DEFS} & = & \{\ (R \mapsto \{(f_k = v_k)*\}\ )*\ \} \\
\texttt{ADTS} & = & \{\ (R \mapsto \overline{C_j\ T_j})*\ \} \\
\texttt{FUNS} & = & \{\ (m \mapsto (T,\ e))*\ \} \\
\texttt{CASES} & = & \{\ (c \mapsto (\langle T \rangle,\ T',\ e))*\}
\end{array}\right\}_{L_D}$$

Figure 1.30: Linkage syntax for static linkages L_S (left) and dynamic linkages L_D (right), with the added `PATHS` attribute.

The new, modular rules for linkage computation shown in Figure 1.31 are parameterized by

the program $\mathscr{P}$ and its linkage representation $\mathscr{L}$. There are now also two additional contexts, Δ for encountered parent paths during computation (this prevents circular reasoning in a more on-demand fashion, as opposed to the global reasoning about `ancestors` as in Persimmon's original design) and a context $\mathcal{M}$ that holds the linkages for each program fragment involved in the current computation. Essentially, the global reasoning about ancestors and dependencies has been replaced by accumulating contexts. The original, non-modular linkage computation rules are available in Figure 1.18 for quick comparison.

$\boxed{\mathcal{M};\Delta \vdash a \rightsquigarrow L}$ $\boxed{\mathcal{M};\Delta \vdash a \rightsquigarrow_{\approx} L}$

$$\frac{\texttt{parse}_S(\mathscr{P}) = \mathscr{L}_S \qquad \mathcal{M} \subseteq \mathscr{L}_S \qquad L_S = \texttt{link}(\mathcal{M})}{\mathcal{M};\Delta \vdash \texttt{prog} \rightsquigarrow L_S}\ \text{(L-Prog-S)} \qquad \frac{\mathcal{M};\Delta \vdash a.A \rightsquigarrow_{\approx} L}{\mathcal{M};\Delta \vdash \texttt{self}(a.A) \rightsquigarrow L}\ \text{(L-Self)}$$

$$\frac{\texttt{parse}_D(\mathscr{P}) = \mathscr{L}_D \qquad \mathcal{M} \subseteq \mathscr{L}_D \qquad L_D = \texttt{link}(\mathcal{M})}{\mathcal{M};\Delta \vdash \texttt{prog} \rightsquigarrow L_D}\ \text{(L-Prog-D)} \qquad \frac{\mathcal{M};\Delta \vdash a.A \rightsquigarrow_{\approx} L \qquad L\ [a.A\ /\ \texttt{self}(a.A)] = L'}{\mathcal{M};\Delta \vdash a.A \rightsquigarrow L'}\ \text{(L-Sub)}$$

$$\frac{\mathcal{M};\Delta \vdash a \rightsquigarrow L \qquad L'' = L.A \qquad L' + L'' = L''' \qquad L''.\texttt{super} \notin \Delta \qquad \mathcal{M}';\Delta' \vdash L''.\texttt{super} \rightsquigarrow_{\approx} L' \qquad \mathcal{M}' = \texttt{getFragment}(\mathscr{L}, L''.\texttt{super}) :: \mathcal{M} \qquad \Delta' = L''.\texttt{super} :: \Delta}{\mathcal{M};\Delta \vdash a.A \rightsquigarrow_{\approx} L'''}\ \text{(L-Nest)}$$

Figure 1.31: The new, modular rules for computing linkages L, parameterized by a program $\mathscr{P}$ and its linkage representation $\mathscr{L}$.

The rules L-Self and L-Sub which simply perform path substitution within the computed linkage remain unchanged apart from the addition of the two new contexts, as is to be expected. The L-Nest rule now additionally ensures that there is no circular inheritance by checking that the parent path of the current family (`L''.super`) does not appear in the parent path context Δ, and by computing the linkage for the parent (`L''.super`) with respect to an extended context Δ' that contains the path `L''.super`. This ensures that a family cannot extend itself or any of the other parents in the chain of inheritance. When computing the parent linkage, the context $\mathcal{M}$ of linkage fragments (essentially, the context of linkage dependencies of the currently computed linkage) is also extended with the linkage fragment that contains the parent family. We retrieve this fragment

by a lookup function `getFragment` which searches the linkage representation $\mathscr{L}$ of the program for the appropriate linkage fragment and returns it. Due to this need to look up linkage fragments before one of the base case rules is triggered, we must now ensure that parsing $\texttt{parse}(\mathscr{P}) = \mathscr{L}$ happens before linkage computation, and not just at the base cases. We assume access to this representation in the formal rules. In the implementation, the parsing step is performed (if needed) before applying the rules.

When computation reaches the base case (L-Prog-S or L-Prog-D), the context $\mathcal{M}$ will contain all the linkage fragments that represent the dependencies of the linkage being computed. This context will be a subset of the linkage representation $\mathscr{L}$ of program $\mathscr{P}$ – it may or may not contain all the linkage fragments. The `prog` linkage that is computed at the base case is a *linking* ($\texttt{link}(\mathcal{M})$) of the linkage fragments that represent only the relevant program fragments for this computation. Since each linkage in $\mathcal{M}$ is a `prog` linkage, we can seamlessly combine them via linkage concatenation into a composite `prog` linkage that contains all information about the inspected family, and any of its dependencies across program fragments.

1.11.2 Contexts.

Here, we would like to summarize the meaning and the uses of all four contexts that appear in the theory. The contexts $\mathcal{M}$ and Δ are used during *modular* linkage computation. The context Δ keeps track of all parent paths in the inheritance chain to avoid circular inheritance in a modular fashion. The context $\mathcal{M}$ contains the linkage fragments that are relevant to a given computation based on its dependencies. The context Γ that appears in type checking is the standard typing context for variables appearing in expressions. Finally, the context K that appears in well-formedness judgments (and is propagated to type checking) is the context that we use to check that a self-referencing path is in the scope of the nesting level where it appears. Essentially, code inside a

family cannot reference a self-referencing path that is only defined within a further nested family.

1.11.3 Insights

Since Persimmon essentially uses linkages as on-demand lookup tables for type checking, adding handling of multi-file programs and modular type checking to the calculus reduces to making linkage computation modular. Due to this separation of concerns supported by linkages, the rest of the judgments require minimal changes, particularly invoking linkage computation with new additional contexts. When the program is contained in a single fragment, the modular rules are designed to be equivalent to the single-file rules. The full modular semantics figures are provided in Appendix A, Section A.2.

2

Dafny Sketcher: Neurosymbolic Approaches for Proof Synthesis

2.1 Introduction

The powerful code generation capabilities of frontier large language models (LLMs) have inspired a wave of neurosymbolic tools for automated proof synthesis [Poesia et al., 2024; Brandfonbrener et al., 2024; Yang et al., 2023; First et al., 2023; Thompson et al., 2025; Jiang et al., 2023; Hu et al., 2025]. The potential of LLMs within such tools is usually realized in two complementary dimensions: the generation of a proof candidate, and the iterative repair of a partial proof with

errors. Naturally, the importance of a quality proof candidate, or an on-track partial proof, is a consistent theme across these systems. On the surface, most of the recent work in the community supports the assumption that partial proofs are helpful for LLM-aided proof synthesis.

We discover a counterexample we dub *the scaffolding paradox*: providing an LLM with an on-track proof skeleton (a top-level case breakdown of the proof into subcases) does not meaningfully improve Dafny proof generation. This paradox holds even when the proof skeleton has been directly extracted from a reference solution. This is a surprising discovery in the overwhelming landscape of generate-and-repair tools which have shown partial proofs useful [Jiang et al., 2023; Hu et al., 2025].

To explore this paradox, we use *Dafny Sketcher*, a neurosymbolic framework for Dafny proof synthesis with built-in symbolic routines for constructing and manipulating partial proofs.[1] We perform multiple experiments with the help of Dafny Sketcher, and compare the success of an LLM repair loop over a symbolic proof skeleton versus an empty lemma body. Reliably, the empty seed for the lemma body performs on par with the proof skeleton seed when combined with a simple repair loop. We also explore some approaches to mitigating the paradox, such as focused case repair and a process supervision prompting technique. Finally, we discuss our findings with respect to the broader literature landscape.

2.2 Dafny Sketcher

In general, sketchers are neurosymbolic routines that fill in or repair partial programs with partial programs. They operate in specific contexts (e.g., a lemma body or a specific case analysis branch within a lemma), proposing program fragments (sketches) to fill those gaps. Dafny Sketcher proposes a collection of sketchers for Dafny verification, and also serves as a layer over Dafny for

[1] Dafny Sketcher is available at https://github.com/namin/dafny-sketcher

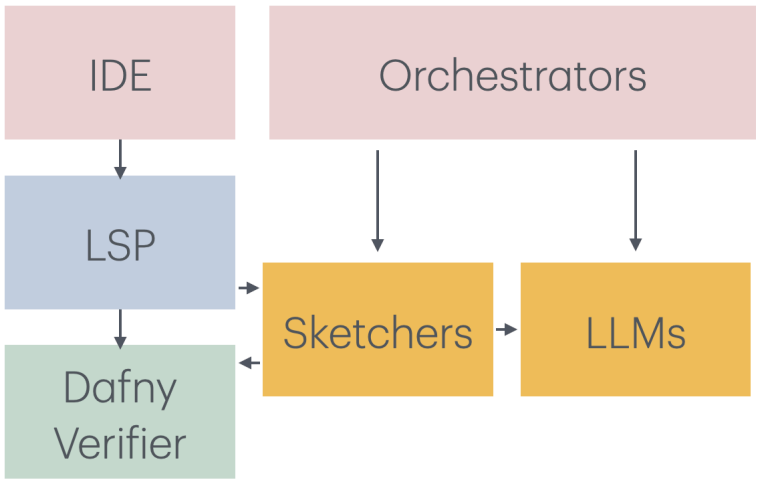

Figure 2.1: Overview of Dafny Sketcher. Orchestrators (user, LLM-agent, or automated loop) invoke sketchers to extract proof structure from the Dafny AST and produce a sketch (for example, a inductive case breakdown). The sketch along with verifier feedback can be fed into an LLM for repair, or shown to a user in the IDE for interactive development.

manipulating proofs, simplifying operations like the insertion of lemma bodies, and the extraction of structural information about lemmas directly from the Dafny AST. This section introduces the two main components of Dafny Sketcher: sketchers, which guide proof structure and direction, and orchestration methods, which coordinate the search for viable proof candidates using sketcher feedback. We show an overview of Dafny Sketcher in Figure 2.1. We also show an interactive session in Section 2.3.

2.2.1 Inductive Sketcher

The basic inductive sketcher identifies a suitable inductive variable (for structural induction) or a function (for rule-based induction) and generates the corresponding inductive proof sketch, consisting of the case-by-case analysis and the recursive calls to the lemmas. The recursive calls are filled heuristically based on the original structure or function. The inductive target can also be specified directly using the {: `induction_on`} lemma attribute.

For auxiliary variables in the lemmas, a few options are possible. We could find the invariants

that hold for these variables, or we can just eschew the problem by filling in with constants (this is the current strategy used by Dafny Sketcher). These heuristics might not always work, but they are a starting point.

Besides the basic inductive sketcher, we use the inductive search sketcher by default: it performs a search over possible inductive variables and functions (for rule induction) and keeps the sketch with the fewest errors remaining.

2.2.2 Counterexample Sketcher

The counterexample sketcher retrieves counterexamples from the Dafny verifier to steer the proof search. Built into Dafny is a command-line argument `--extract-counterexample` that produces potential counterexamples to a given lemma that does not verify. The counterexamples generated by this option are not themselves verified by Dafny, which is why the sketcher performs an extra pass to filter only those counterexamples that provably verify. These verified counterexamples are especially useful in guiding the LLM's edits. Rather than attempting to prove a lemma that is unprovable, the LLM can use the counterexample as evidence to redirect its effort, for example, by choosing to revisit a function definition instead.

2.2.3 Orchestration

Automated Orchestration

For our experiments, we use a simple orchestrator that tries an empty proof sketch or a proof skeleton to fill a proof, then iterates through a number of repairs with an LLM, providing the previous attempt and its errors in the prompt.

LLM-based Orchestration

We also provide a command-line tool and MCP to the sketchers, and let a system like Claude Code or Cursor orchestrate the use of sketchers. LLM-based orchestration enables adaptive use of sketchers, reduces manual intervention, and can be practically integrated with LLM tools.

User Orchestration

Dafny Sketcher routines have been made readily available through a VSCode extension, and the user can compose them freely with other edits.

2.3 Using Sketchers Interactively

We show an interactive session of using the Dafny Sketcher routines. This section can be replicated with an LLM-based automated run that allows backtracking edits based on counterexamples. While the initial vision for Dafny Sketcher was the use of sketcher routines interactively within the editor, applying the routines within an automated LLM repair loop revealed the scaffolding paradox.

We start with a "buggy" program, where the `optimal` predicate is part of the spec, but the `optimize` function can be changed. The lemma is stated as an axiom so the whole file verifies.

```
datatype Expr =
 | Const(value: int)
 | Var(name: string)
 | Add(left: Expr, right: Expr)

predicate {: spec} optimal(e: Expr)
{
 match e
 case Add(Const(0), _) ⇒ false
 case Add(_, Const(0)) ⇒ false
 case Add(e1, e2) ⇒ optimal(e1) ∧ optimal(e2)
 case _ ⇒ true
}

function optimize(e: Expr): Expr
{
```

```
 match e
 case Add(Const(0), e2) ⇒ optimize(e2)
 case Add(e1, Const(0)) ⇒ optimize(e1)
 case Add(e1, e2) ⇒ Add(optimize(e1), optimize(e2))
 case _ ⇒ e
}

lemma { : axiom} optimizeOptimal(e : Expr)
ensures optimal(optimize(e))
```

If we try to prove the lemma with an empty body, we get an error that the postcondition could not be proved.

```
lemma optimizeOptimal(e : Expr)
ensures optimal(optimize(e))
{
// error : a postcondition could not
//   be proved on this return path
}
```

We can use the inductive search sketcher, which will fill the following sketch:

```
lemma optimizeOptimal(e : Expr)
ensures optimal(optimize(e))
{
 // Inductive proof using rule induction
 // following function : optimize
 match e {
   case Add(Const(0), e2) ⇒ {
     optimizeOptimal(e2);
   }
   case Add(e1, Const(0)) ⇒ {
     optimizeOptimal(e1);
   }
   case Add(e1, e2) ⇒ {
     optimizeOptimal(e1);
     optimizeOptimal(e2);
     // error : a postcondition could not
     //   be proved on this return path
   }
   case _ ⇒ {
   }
 }
}
```

We notice the error is localized in the general `Add(e1, e2)` case. We can call the counterexample sketcher and we get the verified counterexample, where the parameter `e` is given the following

condition.

```
Expr.Add(
  Expr.Add(Expr.Const(0), Expr.Const(13)),
  Expr.Add(Expr.Const(0), Expr.Const(0))) = e
```

We rewrite the `optimize` function:

```
function optimize(e: Expr): Expr
{
 match e
 case Add(e1, e2) ⇒
  match (optimize(e1), optimize(e2)) {
  case (Const(0), oe2) ⇒ oe2
  case (oe1, Const(0)) ⇒ oe1
  case (oe1, oe2) ⇒ Add(oe1, oe2)
 }
 case _ ⇒ e
}
```

Now the empty proof still fails, but the inductive search sketcher gives a sketch that completes the proof:

```
lemma optimizeOptimal(e: Expr)
ensures optimal(optimize(e))
{
 // Structural induction on e
 match e {
   case Const(value) ⇒ {
   }
   case Var(name) ⇒ {
   }
   case Add(left, right) ⇒ {
     optimizeOptimal(left);
     optimizeOptimal(right);
   }
 }
}
```

2.4 Experiments

We ran an experiment to evaluate whether providing an on-track proof skeleton to an LLM repair loop improved LLM-based Dafny lemma repair. Each lemma containing a reference proof with a top-level branching structure, such as `match` cases or `if`-statements, was processed using an

Proof skeleton

```
lemma sum_plus(s: seq<int>, i: nat)
  requires i < |s|
  ensures sum(s, i) + s[i] = sum(s, i+1)
{
  if i = 0 {

  } else {

  }
}
```

Reference solution

```
lemma sum_plus(s: seq<int>, i: nat)
  requires i < |s|
  ensures sum(s, i) + s[i] = sum(s, i+1)
{
  if i = 0 {
    // sum(s, 0) = 0 and sum(s, 1) = s[0]; trivial
  } else {
    // IH on the tail: sum(s[1..], i−1) + s[1..][i−1] = sum(s[1..], i)
    // and s[1..][i−1] = s[i], so this closes the goal
    sum_plus(s[1..], i − 1);
  }
}
```

Figure 2.2: Example initialization for the repair loop: a control-flow proof skeleton extracted from a reference lemma, and the corresponding reference solution.

identical LLM-assisted repair loop under two conditions:

1. lemma initialized with an empty body

2. lemma initialized with a control flow skeleton extracted from the reference solution (see Figure 2.2). The skeleton preserved the branching structure, but not the contents of each branch.

After seeding the lemma body, 3 iterations of LLM repair were performed in each condition. On each iteration, the repair loop prompted the model with the current program and Dafny error diagnostics, parsed the generated candidate body from LLM output, and used the Dafny verifier

Initialization modes.

Mode A (Empty): initialize lemma body to {}.

Mode B (Skeleton): initialize lemma body to top-level control-flow skeleton extracted from the known-correct proof (preserve branch headers, erase branch internals).

Mode C (Process): before repair, prompt the LLM to explain step-by-step why the skeleton's case analysis is the right approach, what each branch's proof obligation is, and what technique would discharge it. Then proceed with the same repair loop as Mode B.

Per-lemma procedure.

1. Extract lemma solution from the reference file.
2. Build skeleton by recovering top-level `match/case` or `if/else` branches, keep headers only. Skip lemma if no such structure is recognized.
3. Early trivial check: if the empty body already verifies, record both modes as solved at `-1`.
4. For each mode independently, run a bounded repair loop (budget: 3 iterations):
   (a) Insert current candidate body into the full program.
   (b) Query Dafny errors for the target lemma.
   (c) Prompt the LLM with current program and structured error list from the verifier.
   (d) Parse returned body from the output.
   (e) Reinsert body and re-verify, stop on first verification success.
5. Record outcome: `-1` (solved without LLM), `0..2` (solved at iteration), or `None` (unsolved within budget).

Figure 2.3: Experiment algorithm for isolating the effect of proof skeleton priors.

to check for errors after insertion of the repaired body. The first iteration with zero errors was counted as a successful outcome. The exact procedure is shown in Figure 2.3.

For our benchmark across all experiments, we used a subset of DafnyBench [Loughridge et al., 2024]. For this subset, we filtered out files that contained only empty-body lemmas (no body to generate), and files that only contained axioms. We also removed any lemmas with intentionally wrong specifications (for example, exercises showing why a given specification is not strong enough). We also completed the lemmas that did not verify with reference solutions.

The benchmark results report per-lemma success/failure statistics as well as the number of

iterations to reach success for both conditions (table 2.1). Out of 604 total lemmas, 114 lemmas had a usable top-level branching structure for an on-track proof skeleton. For Claude Sonnet 4.6 and Opus 4.6, providing an on-track skeleton as opposed to an empty lemma body produced only a marginal gain (Sonnet: 87.7% solved over skeleton vs. 86.0% solved over empty, +1.7 pp; Opus: 92.1% solved over skeleton vs. 88.6% solved over empty, +3.5 pp). When both conditions resulted in a successful repair, the speed (i.e., the number of LLM iterations required to reach the solution) was almost identical, as 85 out of 94 lemmas used the same number of iterations for repair in both conditions. These results illustrate the scaffolding paradox: providing even structurally correct scaffolding does not seem to translate into meaningful verification improvement, neither in the success rate nor in speed.

We also wanted to test out the paradox with a model that was designed for efficiency (Gemini 2.5 Flash Lite) more so than complex reasoning. Gemini shows a slightly larger improvement with skeleton seed over empty seed, a 8.8 percentage point increase (10 lemmas). This suggests that the effect of symbolic scaffolding may also be model-dependent, rather than uniform. However, it is also important to consider the success of the model at generating proof code overall. When using the Gemini model, only 56.1% of proofs were solved when compared to Opus (92.1%). Any small improvement from scaffolding may not translate to competitive overall performance.

2.4.1 Mitigating Strategy: Process Supervision

We investigate whether prompting the LLM to reason about *why* the skeleton is correct before attempting repair can mitigate the paradox. Inspired by the finding that process supervision outperforms outcome supervision [Lightman et al., 2023], we hypothesize that a correct skeleton is an "outcome" but not the "process" that generates it: the LLM needs the thinking trace, not just the structure. In Mode C (Process), we prompt the model to explain step-by-step why the

Table 2.1: Benchmark outcomes for empty-body vs. skeleton initialization, using Claude Sonnet/Opus 4.6, Gemini 2.5 Flash Lite, and Qwen3(-Coder-Next), for LLM repair. The Qwen3 count is on a smaller easier dataset.

Metric	Count			
	Sonnet	**Opus**	**Gemini**	**Qwen3**
Considered for repair	114	114	114	12
Empty solved	98 (86.0%)	101 (88.6%)	54 (47.4%)	4 (33.3%)
Skeleton solved	100 (87.7%)	105 (92.1%)	64 (56.1%)	5 (41.7%)
Solved diff (skeleton - empty)	1.7 pp	3.5 pp	8.8 pp	8.4 pp
Both solved	94	101	40	3
Only empty solved	4	0	14	1
Only skeleton solved	6	4	24	2
Neither solved	10	9	36	6
Among solved by both:	(94)	(101)	(40)	(3)
Skeleton faster	5	7	4	1
Empty faster	4	4	3	2
Same number of iterations	85	90	33	0

Table 2.2: Effect of process supervision on the scaffolding paradox benchmark. Process mode prompts the LLM to reason about the skeleton before repair.

Metric	**Sonnet**	**Opus**
Considered for repair	114	114
Empty solved	98 (86.0%)	101 (88.6%)
Skeleton solved	100 (87.7%)	105 (92.1%)
Process solved	105 (92.1%)	112 (98.2%)
Process solved (max 3 calls incl. process)	105	108

skeleton's case analysis is the right approach before proceeding with the same repair loop.

Results (table 2.2) show that process supervision yields a meaningful improvement over both the empty and skeleton conditions. Opus improves from 105 solved with the skeleton alone to 112 with process supervision (98.2%, a 6-point gain over the skeleton's 92.1%). Sonnet similarly improves from 100 to 105 (92.1%, up from 87.7%). These gains persist even when controlling for the extra LLM call consumed by the reasoning step: at a budget of 3 total calls including the process call, Opus still solves 108 lemmas (vs. 105 with skeleton) and Sonnet solves 105 (vs. 100 with skeleton).

Table 2.3: Case-repair benchmark outcomes for empty-body vs. skeleton initialization, including subsets restricted to the top k ($k \in \{60, 30, 20\}$) longest reference proofs that could not also be solved by an empty body.

Metric	**All considered**	**Top 60**	**Top 30**	**Top 20**
Considered for repair	121	60	30	20
Empty solved	65 (53.7%)	22 (36.7%)	8 (26.7%)	2 (10.0%)
Skeleton solved	76 (62.8%)	31 (51.7%)	15 (50.0%)	9 (45.0%)
Solved diff (skeleton - empty)	9.1 pp	15.0 pp	23.3 pp	35.0 pp
Both solved	56	18	6	2
Only empty solved	9	4	2	0
Only skeleton solved	20	13	9	7
Neither solved	36	25	13	11
Among solved by both:	(56)	(18)	(6)	(2)
Skeleton faster	10	5	2	2
Empty faster	14	6	2	0
Same number of iterations	32	7	2	0

This suggests that the scaffolding paradox can be partially reversed. A symbolic skeleton provided as an opaque artifact does not align with the model's reasoning, but when the model first simulates the reasoning process that would produce the skeleton, articulating why each case split is necessary and what each branch must establish, the structure becomes actionable. The skeleton transitions from an external constraint to an internalized proof plan. This is consistent with the process supervision literature, which finds that models trained or prompted with intermediate reasoning steps outperform those given only final outcomes.

2.4.2 Mitigating Strategy: Case Repair

We also investigate whether a stage repair pipeline focusing on case-level edits may mitigate the scaffolding paradox. The idea is that by focusing on case-level edits, the model may be able to capitalize on the case breakdown in the skeleton more successfully.

In this pipeline, after the lemma body is seeded with an empty body (Mode A) or a proof

Table 2.4: Gemini simple repair loop solved counts and rates on top-k subsets.

Metric	All considered	Top 60	Top 30	Top 20
Considered for repair	114	60	30	20
Empty solved	54 (47.4%)	18 (30.0%)	5 (16.7%)	3 (15.0%)
Skeleton solved	64 (56.1%)	24 (40.0%)	11 (36.7%)	8 (40.0%)
Solved diff (skeleton - empty)	8.8 pp	10.0 pp	20.0 pp	25.0 pp

skeleton body (Mode B), one iteration of whole-proof repair follows. This ensures that Mode A is not at a disadvantage due to the absence of cases, since it seeds an empty body. After this initial repair, if the proof does not verify, case-level repair follows (2 iterations total), focusing on the first failing top-level case. We consider "cases" to mean if-statements, or match cases, as both structures represent a case breakdown in the proof. A case repair is accepted if it reduces the number of errors within the case. If unsuccessful, case-level repair falls back to statement-level fixes within that branch (2 iterations). Finally, if the targeted repair was still unsuccessful, we perform 2 additional iterations of whole-proof repair. Throughout the process, candidates that fail a syntax check are not considered. This pipeline also supports processing different lemmas with duplicate names, resulting in a slight increase of total proofs considered for repair.

The results of the case repair experiment, run with the Gemini 2.5 Flash Lite model, are shown in table 2.3. On the full benchmark, the skeleton seed shows a slight advantage, resulting in 62.8% of proofs solved, a 9.1 percentage point increase over the empty seed. We then restricted our analysis to top-k subsets ($k \in \{60, 30, 20\}$) of lemmas with the longest reference solutions, for which the empty body was not also a valid solution. In these focused subsets, the skeleton seed consistently results in a higher percentage of solved proofs over the empty seed, and the percentage point difference for solved proofs increases as reference solutions get longer.

We also consider that the length of the reference solution may affect the utility of the skeleton seed independently of the case-repair approach. To examine this effect, we analyze the original

whole-repair loop experiment (described in Figure 2.3) on the same top-k subsets (results shown in table 2.4). This analysis shows a similar trend of a growing gap between the solved rates, suggesting the benefit of the skeleton seed for longer proofs regardless of the repair pipeline. However, the case repair pipeline also appears to have an effect, increasing the solved rate in almost all cases (with one exception, the "empty solved" rate for Top 20 longest proofs, where the difference in count is just 1 proof). The percentage point gap between solved rates also grows wider in the case-repair pipeline, suggesting that the utility of the skeleton seed increases when combined with case repair.

Overall, the results suggest that an on-track skeleton may be a more helpful seed when the proof is longer and perhaps more difficult to generate in one go, and when combined with a mechanism for case repair to take full advantage of the skeleton case breakdown.

2.5 Related Work

Automated proof synthesis has been gaining traction as frontier LLM models grow in their capabilities. With that, a wide range of hybrid approaches has emerged, combining LLM assistance with symbolic methods for tasks such as prompting, proof candidate or annotation generation, and iterative proof repair. We examine the most closely related works with respect to two dimensions that our scaffolding paradox touches on: how a proof sketch (or candidate) is obtained (symbolically or via an LLM), and how any remaining errors are repaired.

2.5.1 LLM Generation with LLM (or Hybrid) Repair

Draft, Sketch, and Prove (DSP) [Jiang et al., 2023] introduces a workflow for Isabelle proof generation in which an LLM generates a formal proof sketch with holes from a natural-language informal proof, and then fills the holes using commercially available automated provers. Hybrid-

Prover [Hu et al., 2025] follows a similar approach by sampling multiple whole-proof candidates from an LLM, abstracting away fine-grained details to create proof sketches, and then using a tactic generation model for iterative refinement. Baldur [First et al., 2023] implements a classic LLM-assisted generate-then-repair loop for Isabelle. After a whole-proof candidate is synthesized by an LLM, this candidate, along with the errors from the proof assistant, is fed into a fine-tuned LLM repair pipeline. Importantly, these tools show that an *LLM-generated* proof sketch or whole-proof candidate can help guide LLM-assisted repair.

Since Dafny is a verification language, verification annotations may be embedded throughout the program, as opposed to just inside lemmas. DafnyPro [Banerjee et al., 2026] targets the generation of verification annotations for Dafny programs: an LLM proposes annotations, and a repair pipeline using both symbolic and LLM-assisted mechanisms refines the solution. This work suggests that *LLM-generated* verification annotations in Dafny can be useful in conjunction with a hybrid repair approach.

2.5.2 LLM Generation with Symbolic Repair

Several systems show that symbolic repair approaches can also be helpful in guiding proof search, when composed with LLM-generated proof candidates or annotations. PALM [Lu et al., 2024] targets Rocq proof synthesis and equips an LLM whole-proof generator with a set of symbolic repair mechanisms. The helper set is designed to target common errors that LLMs make in generated proofs. This pipeline is also supported by a symbolic backtracking mechanism (powered by CoqHammer), which alone accounts for a 5x increase in repaired proofs when compared to repair rules alone. DafnyAnnotator [Poesia et al., 2024] also takes a symbolic repair approach by placing LLM-synthesized annotation candidates at every syntactically valid location in the program, and using Dafny verifier feedback to select the first annotation and location that allows Dafny to make

progress. Together, these systems show that symbolic strategies for repair can be powerful when the proof code itself was synthesized by an LLM.

2.5.3 Other LLM-Based Proof Synthesis Approaches

Some approaches use retrieval-augmented generation (RAG) [Lewis et al., 2020] to improve proof synthesis by providing relevant context to the model, such as library lemmas or related proofs. LeanDojo's ReProver model [Yang et al., 2023] combines a retriever, which selects relevant premises from Lean's math library, with a tactic generator: a fine-tuned LLM that proposes proof tactics given the proof state and retrieved premises. Rango [Thompson et al., 2025] similarly identifies relevant premises and similar proofs from the current Rocq project and uses them at every step of the proof synthesis, together with a tactic generator and composer. Misu et al. [2024] show that structured Chain of Thought (CoT) prompts decomposing the problem step by step, combined with RAG-generated few shot examples, can improve synthesis of Dafny programs.

On the other hand, Clover [Sun et al., 2024] checks the consistency of LLM-generated code by generating (1) the code, (2) the docstring, and (3) the formal annotations from a natural language problem description, and cross-checking pairwise.

2.6 Discussion

Our experiments support what we call the *scaffolding paradox*: providing an on-track proof skeleton, even when it is derived from a reference solution, does not meaningfully help an LLM construct a Dafny proof. This finding is a valuable datapoint in the landscape of existing literature, which on the surface appears to encourage partial proofs as a strategy to improve LLM proof synthesis.

At the same time, two mitigation experiments show that the process by which the scaffolding structure is consumed matters. Process supervision, where the LLM is prompted to re-derive the

reasoning behind the provided skeleton, sees an improvement in outcomes over both empty seed and skeleton seed approaches, even under a matched call budget. A pipeline using case repair also shows some improvement, especially within subset analyses of proofs with the longest reference solutions. Regardless of the repair pipeline, the skeleton seed shows a slight advantage when using a weaker reasoning model to solve longer proofs.

Altogether, these results suggest that a pre-determined, on-track proof skeleton is not by itself sufficient guidance for improving LLM-assisted proof synthesis. The scaffolding must be paired with repair approaches that aid the LLM in using the partial solution successfully. In our experiments, this corresponds to combining the proof skeleton seed with a separate process reasoning step, or with focused edits within failing cases. This perspective helps situate our results with respect to existing systems like DSP and HybridProver, which show that LLM-generated sketches are beneficial to proof synthesis [Jiang et al., 2023; Hu et al., 2025]. Within these existing systems, the pipelines tend to align the generated proof sketch and the repair process with the model's own plan for synthesizing the proof. The model is not guided to refine an external proof plan, which is what a proof skeleton becomes in our setting.

Limitations. Our results are restricted by the current experimental scope. While our experiments evaluate LLM proof synthesis on a filtered subset of DafnyBench including only proofs with a top-level case breakdown, different benchmarks or longer reference proof sizes could expose a different pattern. Results are also model- and version- dependent, as we consider only Claude Sonnet and Opus 4.6, Gemini 2.5 Flash Lite, and Qwen3(-Coder-Next). Additionally, our definition of scaffolding is intentionally restricted to top-level control-flow proof skeletons, which does not exclude the possibility that proof synthesis might benefit from more in-depth scaffolding.

2.7 Data Availability

Dafny Sketcher is available at https://github.com/namin/dafny-sketcher. Our experiment logs and the instructions to reproduce our experiments are available at https://github.com/namin/dafny-sketcher/tree/main/vfp/logs_paradox.

2.8 Conclusion

We present evidence for the scaffolding paradox: on-track proof skeletons derived from reference solutions do not meaningfully improve LLM-assisted synthesis of Dafny proofs. We use Dafny Sketcher, our neurosymbolic framework for sketching and manipulating Dafny proofs, to demonstrate this effect, while also exploring two mitigating strategies: process supervision and case repair. Our findings suggest that symbolic scaffolding could benefit from mechanisms that help the model to reason about the imposed proof structure, as opposed to taking it at face value.

3

The Human Aspects of Proof Synthesis

3.1 Introduction

Interactive theorem provers, such as Rocq, Lean, and Agda [de Moura et al., 2015; Norell, 2009; The Rocq Development Team, 2024], support users in constructing formal models and proofs with a high degree of assurance. These systems provide expressive specification languages and powerful tactic frameworks to aid in proof construction, however, the users must still manually orchestrate the proof steps while managing complex proof states. As a result, completing proof tasks successfully requires the users to develop a familiarity with each tool's interface, as well as a deep understanding of each tool's native proof tactics and strategies. This steep usability curve is one of the reasons

why formal verification remains a niche activity reserved primarily for experts in academia and industry.

Many tools for proof automation already exist to curb user effort required for formal verification. These include built-in tactics (Rocq's `auto` and `lia`, Lean's `simp`), hammers for discharging proof goals to external solvers (CoqHammer for Rocq [Czajka & Kaliszyk, 2018], SledgeHammer for Isabelle [Meng & Paulson, 2008]), and ML-based tactic predictors (Tactician [Blaauwbroek et al., 2020], Diva [First & Brun, 2022], Proofster [Agrawal et al., 2023]). More recently, proof synthesis tools powered by large language models (LLMs) have joined the landscape (for example, Rango [Thompson et al., 2025]). As the commercially available models grow more powerful, general purpose LLM-based assistants (such as Claude [Anthropic, 2024] and ChatGPT [OpenAI, 2023]) are also gaining popularity for verification tasks.

These tools aim to reduce verification effort by discharging simpler goals, predicting tactics for next proof steps, generating candidate proof scripts, or explaining proof obligations. However, the tools differ significantly in their interfaces (affecting the users' ability to interact with the tools) and in the reliability of the assistance they provide. For example, LLM-based assistants such as Claude support user interaction via a natural language interface, supporting a variety of user queries, from informal explanations, to proposed code snippets, to conversational follow-ups. They are also easily accessible through the web. While these interfaces can feel intuitive and accessible (especially to novices), the reliability of their outputs may be lacking. The generated proof code may be incorrect syntactically or semantically, rely on non-existent tactics or lemmas, or be misaligned with the proof state at various points in the script.

On the other hand, the more specialized proof synthesis tools, such as Tactitian and Proofster, operate over the formal structure of the proof and the actual proof state at each transition point. The interfaces of such tools are typically more rigid, allowing only a specific subset of interactions.

For example, once Proofster is invoked in-editor, it performs a linear search using tactic prediction to complete the proof from beginning to end, and then presents its feedback to the user as a proof search tree of explored paths. This rigidity makes user interaction less fluid, but the resulting proof artifacts are guaranteed to be well-formed and correct by construction. If Proofster does find a proof, it is always a valid solution to the proof task at hand.

These contrasting characteristics of helper tool interfaces raise important questions about how users experience and evaluate different tools while interacting with them. Does a natural language interface lower the barrier to entry, even if the tool's assistance may be unreliable? Do users prefer more structured, reliable outputs, even at the cost of a more rigid interaction model? Do users prioritize the understanding of a proof task, or the verified solution even without understanding? How do these tools shape users' strategies, trust, and overall satisfaction during proof development?

To investigate these questions, we conducted a user study in which participants solved proof tasks in Rocq under three conditions: (i) without any external helper tools, (ii) with access to Claude as a helper tool, and (iii) with access to both Claude and Proofster.[1] Participants were asked to complete proof tasks of comparable difficulty and to provide feedback on their experience with each configuration. By comparing user behavior and perceptions across these conditions, we aim to better understand how different helper tool interfaces influence proof development, and what features the users might benefit from in future proof assistance systems. In particular, we examine how users balance convenience and correctness, how they interpret and recover from incorrect (or partially correct) suggestions, and how the presence of multiple helper tools affects their workflow. Ultimately, this work seeks to inform the design of next-generation proof automation tools that integrate the latest technologies in a way that meaningfully supports users in interactive proof tasks.

[1]The version of Proofster used for this study is available at https://github.com/akravc/VERSE-Toolchain/tree/coq-synthesis-vscode.

3.2 Related Work

It is well known that proof engineering is rife with usability challenges. Ringer et al. [2019] provide a comprehensive survey of these challenges, highlighting steep learning curves, proof brittleness under refactoring, and a lack of supportive tool design that reflects the practical workflows of verification engineers. It is not surprising that great effort has been put into building tools to help automate proof engineering: from automating more granular steps within a proof via tactics and hammers [Czajka & Kaliszyk, 2018; Meng & Paulson, 2008], to automated proof repair in response to definitional changes [Ringer et al., 2021], to tactic and whole-proof generation backed by machine learning [Blaauwbroek et al., 2020; First & Brun, 2022; Agrawal et al., 2023] and LLMs [Hu et al., 2025; First et al., 2023; Thompson et al., 2025]. The existing body of work on proof automation has largely focused on benchmark success rates for proof generation, while the question of *user interaction* with these tools has remained relatively unexplored. Our study aims to address this gap by assessing factors such as user interaction, perceived tool utility, and tool trust across two different interfaces of proof synthesis tools.

LLM and ML-Based Proof Synthesis. Existing automated proof synthesis tools can be grouped by their approach to searching the proof space. One line of work focuses on tactic prediction given the current proof state, guiding the search via supervised or reinforcement learning [Huang et al., 2018; Yang & Deng, 2019; Polu & Sutskever, 2020; Sanchez-Stern et al., 2020; Yang et al., 2023; Blaauwbroek et al., 2024; Sanchez-Stern et al., 2025; Wang et al., 2025; Hubert et al., 2025]. A related line emphasizes retrieval-augmented generation (RAG [Lewis et al., 2020]) grounding generation in relevant context, such as related library lemmas or proofs [Yang et al., 2023; Thompson et al., 2025; Misu et al., 2024]. Some systems further combine retrieval with stateful agentic search [Thakur et al., 2024]. Another approach capitalizes on whole-proof generation and repair

via LLMs, in which the LLM proposes partial or complete proofs, and then iteratively repairs them based on verifier feedback [First et al., 2023; Lu et al., 2024; Kasibatla et al., 2026]. A complementary approach employs structured proof generation, modeling the intermediate reasoning steps via subgoal decomposition [Ren et al., 2025]. Some systems take advantage of neurosymbolic pipelines, where the flexibility of LLMs is combined with the power of automated theorem provers (ATPs) to discharge proof obligations [Jiang et al., 2022, 2023; Hu et al., 2025]. Finally, some related tools embed LLM-based assistance directly into proof editors. CoqPilot [Kozyrev et al., 2024] is a VSCode extension that fills in `admit` holes in Rocq proofs by querying an LLM, and iteratively repairing the candidates using compiler feedback. Lean Copilot [Song et al., 2024] makes LLM-based tactic suggestion and goal completion available to users within the the Lean proof assistant.

User Studies of LLM-Assisted Programming. Other related works explore how programmers interact with LLM-based coding assistants (not specifically in a proof setting). Vaithilingam et al. [2022] explored user interactions and perception of Copilot [GitHub, Inc., 2026], a tool for LLM-assisted programming. The results indicated that users had a favorable perception of Copilot, even though it did not reduce task completion time or increase task success rate. One point of frustration among users was the difficulty or assessing the correctness of the generated code. Similarly, Mozannar et al. [2024] analyzed the hidden costs of common user interactions with Copilot, and found that programmers spend a large fraction of time on double-checking and editing Copilot suggestions. In contrast, our study participants were able to rely on the Rocq verifier to assess the correctness of outputs, and expressed greater difficulty in debugging proof strategies than the code itself.

Barke et al. [2023] found that the way users interact with Copilot depends on whether the user knows how to proceed with the task: when they are sure of the next step, they use it to speed

up their progress, but when they are unsure, they use the tool to explore strategies. Liang et al. [2024] surveyed 410 developers using AI programming assistants to explore the motivation behind developer use of such tools, and found their abilities to autocomplete, finish programming tasks quickly, and recall syntax among the top motivators. Ross et al. [2023] studied conversational LLM-based programming assistants and found that users valued their multi-turn capabilities. These insights were mirrored by our participants as well.

A recent qualitative study explored user perceptions of "vibe coding," a paradigm that sees users collaborate with artificial intelligence (AI) on coding tasks using predominantly natural language [Pimenova et al., 2025]. One of the findings was that users with high trust in AI are more likely to delegate the work to AI. Some of the user-reported vibe coding pain points, such as debugging difficulties and the burden of reviewing LLM-generated code, were also expressed by our participants. More broadly, Sarkar et al. [2022] claim that LLM-assisted programming is a new, distinct programming style with its own challenges.

User-Oriented Work in Theorem Proving Environments. Some prior work in interactive theorem proving explored the interface design of provers and assisting tools. Komendantskaya et al. [2012] investigated ML-based assistance in Proof General, highlighting the importance of the proof process, and not just the final result, early on. Beyond the presentation and granularity of proof hints, they also discussed the importance of backward interfacing: user ability to interpret generated results. This point is also reflected in our study, where participants expressed displeasure with the interpretability of tool feedback for both tools (overwhelming walls of text, detailed search tree), although they ultimately preferred LLM feedback.

Similarly, Ringer et al. [2020] instrumented Rocq's REPL to collect fine-grained data the proof development process, finding that proof development is highly iterative, and highlighting particu-

larly repetitive types of refactoring. Rather than replaying recorded sessions, participants in our study were observed proving tasks in Rocq directly, capturing user interactions with the interfaces as well as their reasoning in real time.

Some tools aim specifically to address user comfort when navigating provers. CoqPIE set out to support common proof-development workflows for proof engineers, such as lemma extraction, and proof replay [Roe & Smith, 2016]. jsCoq and ProofWeb target quick onboarding and accessibility by providing a web-based Rocq interface [Arias et al., 2017; Hendriks et al., 2010]. Alectryon [Pit-Claudel, 2020] re-renders Rocq proofs with their intermediate goal states inlined to improve readability of proofs.

3.3 Methods

In order to compare user interactions with different proof automation tools via their interfaces, we performed a within-subjects user study with 7 Rocq users, ranging in experience from beginners to proficient in Rocq. The participants were asked to complete Rocq proof tasks of similar difficulty using the VSCode IDE, under three conditions. In the control condition, the participants did not have access to any external helper tools. In the first experimental condition (LLM-only), the participants had access to the web interface of Claude, running Sonnet 4.5, for assistance during the proof task. In the second experimental condition (LLM + Proofster), the participants could access both Claude and Proofster (in-editor) for assistance during the proof task. Each participant experienced all three conditions, and the order of conditions was counterbalanced. The 3 tasks for each participant (one per condition) were chosen from a pool of 6 tasks of similar difficulty, and the choice of the tasks was also counterbalanced. The time commitment for each participant was about 70 minutes total.

3.3.1 Research Questions

We aimed to answer the following research questions with this study:

RQ1. How do users balance interface comfort (e.g., ease of interaction, cognitive load) against the perceived trustworthiness of each tool's outputs?

RQ2. To what extent does using the tools in combination provide additional benefits over the use of each tool alone?

RQ3. How does user trust in either interface change (if at all) after task completion or non-completion?

RQ4. How effectively do users perceive their intent is captured and executed across the two interfaces?

RQ5. Do users report feeling misled or deceived by either interface, and under what circumstances?

RQ6. To what extent does feedback from each tool contribute to users' understanding of the task and proof construction?

3.3.2 Participants

Mechanized proofs in Rocq are not part of the standard computer science curriculum, and are usually taught in advanced undergraduate and graduate courses. The prior knowledge required to complete the tasks was quite specialized, and we were not able to successfully recruit any participants by advertising to the general student body (Harvard University mailing list). Due to the specialized nature of the study, we opted to reach out to the instructors of ongoing university courses teaching Rocq, requesting permission to recruit their students through the course mailing

Table 3.1: Participant demographics and roles.

Participant	Age range	Gender	Current role
P1	25–34	Male	Master's student
P2	18–24	Male	PhD student
P3	18–24	Male	Undergraduate student
P4	25–34	Male	PhD student
P5	18–24	Male	Master's student
P6	18–24	Male	Undergraduate student
P7	18–24	Male	Undergraduate student

Table 3.2: Participant familiarity and experience levels for LLM, proof automation, and Rocq.

Participant	LLM familiarity	Proof automation familiarity	Rocq experience level
P1	■■■■■■■	■■■■■□□	■■■■□□□
P2	■■■■■■□	■■■■■■■	■■■■■■□
P3	■■■■■■□	■■■■□□□	■■□□□□□
P4	■■■□□□□	■■■■■□□	■■■■■□□
P5	■■■■■■■	■■■■□□□	■■■■□□□
P6	■■■■■□□	■■■□□□□	■■□□□□□
P7	■■■■■■□	■■■■■□□	■■□□□□□

lists. We recruited a total of 7 participants, two of which were recruited through the Zulip channel for Rocq users, and five of which were recruited from the following university courses:

- CIS5000: Software Foundations (University of Pennsylvania, Fall 2025)
- EECS 755: Software Modeling and Analysis (The University of Kansas, Fall 2025)
- CMSC631: Program Analysis and Understanding (University of Maryland, Fall 2025)
- CS 4160: Formal Verification (Columbia University, Spring 2026)

The participant pool was entirely within the 18–34 age range, with 5/7 participants in the 18–24 range, and the remaining 2 participants in the 25–34 range (Table 3.1). The sample included

undergraduate (3/7), master's (2/7), and PhD (2/7) students, and all participants identified as male.

Participants generally reported high familiarity with LLMs, while familiarity with proof automation tools[2] and Rocq was more mixed (Table 3.2). In particular, the majority of participants reported beginner-to-intermediate Rocq experience levels, with only a few reporting more advanced experience.

The students did not receive any special course credit, and were reimbursed with a $20 Amazon gift card for their participation. All participants completed the study via a remote Zoom call.

3.3.3 Procedure

```
1   From Coq Require Import Lists.List.
2   Import ListNotations.
3
4   (* This function counts the number of times x appears in list l. *)
5   Fixpoint count_occurrences (x : nat) (l : list nat) : nat :=
6     match l with
7     | nil => 0
8     | cons h t => if Nat.eqb x h then S (count_occurrences x t) else
      count_occurrences x t
9     end.
10
11  (* This function removes all occurrences of x from l. *)
12  Fixpoint remove_all (x : nat) (l : list nat) : list nat :=
13    match l with
14    | nil => nil
15    | cons h t => if Nat.eqb x h then remove_all x t else cons h (remove_all x t)
16    end.
17
18  (* Lemma: relationship between count and remove. *)
19  Lemma count_remove_relationship :
20    forall x y l,
21      count_occurrences x (remove_all y l) =
22      if Nat.eqb x y then 0 else count_occurrences x l.
23  Proof.
24  Admitted.
```

Figure 3.1: An example of a verification task that the participants were asked to complete.

[2]Such as built-in proof tactics, plug-ins, proof synthesis tools, and LLM-based tools.

We conducted the following procedure for each participant. First, the participants were given a brief overview of the study structure (tasks, surveys, interview). We then asked the participants to examine and sign our consent form (including consent for video and audio recording), and to fill out a pre-study form with questions about their demographics and background. Each participant experienced all three conditions of the study: the control condition with no helper tools, and two tool-assisted experimental conditions: LLM-only, and LLM + Proofster. In each of the three conditions, participants were given 10 minutes to attempt to prove a theorem in Rocq using the VSCode IDE, and encouraged to think aloud while working. Figure 3.1 shows an example of a verification task that participants experienced during the study. In the tool-assisted conditions, prior to starting the task, participants were shown short tutorials with examples explaining how to use the interfaces alone and in combination. After each task, participants completed a post-task survey, with questions regarding their tool use during the task, as well as a 6-question NASA Task Load Index survey. Finally, after all tasks were completed, we concluded with a brief 10-minute interview about the participants' experience across tasks.

3.3.4 Tool Interfaces

Participants interacted with two tool interfaces during the study: the Claude web interface (both tool-assisted conditions) and the in-editor Proofster interface (LLM + Proofster condition). Below, we describe both interfaces and how they can be used for interactive theorem proving.

Claude Web Interface. The Claude web interface is pictured in Figure 3.2. The interface is essentially a chat log between the participant and the LLM, where the participant can type prompts to the LLM using natural language, and the LLM responds in natural language. The participant can also include Rocq code directly into the prompts, and optionally give directions to the LLM

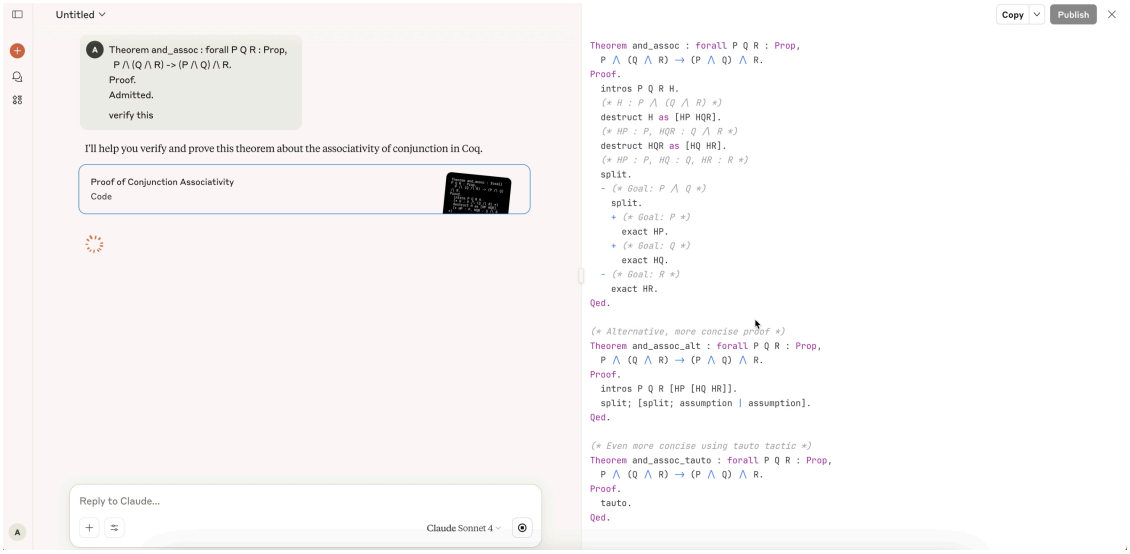

Figure 3.2: Claude web interface used in both tool-assisted conditions (tutorial screenshot).

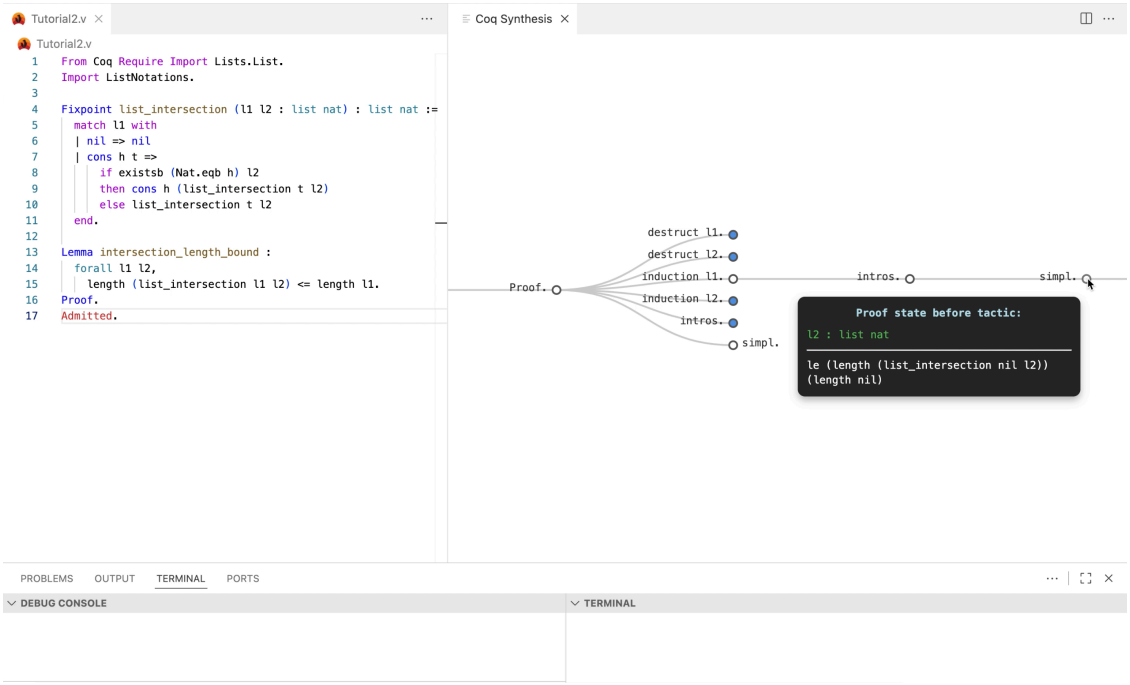

Figure 3.3: Proofster interface used in the LLM + Proofster condition (tutorial screenshot).

via natural language, such as "verify the proof," "explain the task," "suggest improvements," or "identify errors." Once Claude generates a response, any relevant parts can be copied back into VSCode or into the prompt box to continue the conversation. For example, if Claude-generated Rocq code does not verify, the participant can follow up by prompting the LLM with the displayed error.

Proofster Interface. Proofster [Agrawal et al., 2023] is a VSCode extension for automated Rocq proof synthesis. Under the hood, it takes a user-provided Rocq theorem and passes it to a machine learning powered backend, Proverbot9001 [Sanchez-Stern et al., 2020], that performs a search of possible proof tactics. The backend uses a predictive model, trained on a corpus of human-written proofs, to predict the likely next tactic at each step, then calls Rocq to evaluate those tactics and update the proof state.

The Proofster interface is shown in Figure 3.3. Participants can run Proofster directly within the VSCode IDE by using a keyboard shortcut. When the shortcut is triggered, Proofster attempts to synthesize a proof automatically, starting from the last tactic in the proof body (or from the beginning, if the body is empty). If Proofster is able to synthesize a complete solution, it is pasted directly into the editor. If a solution was not found within the time limit, Proofster reports an incomplete status.

After each run, Proofster displays a tree visualization showing the tactics that were explored during the search. For example, in Figure 3.3, after the initial `Proof` keyword starting the proof, Proofster search explored 6 options for the next tactic.

The tree is interactive: participants can expand or contract nodes by clicking on them, hover over nodes to preview the proof state before the tactic at that node is executed, and right-click a node to copy the proof script up to that point. For example, in Figure 3.3, the node `induction`

`l1.` has been expanded, and the user is hovering over the node `simpl.` to display the proof state.

Proofster and Claude can be used in complementary ways. A partially correct response from Claude can be copied into VSCode, and Proofster can then be run to complete the partial proof. Alternatively, a promising direction from the Proofster proof search can be copied into the Claude prompt box. These workflows were shown to the participants in the interface tutorials.

3.4 Results

3.4.1 Task Completion and Interaction Patterns

We carefully managed the difficulty level of the tasks in our task pool so that they were not easily solvable by a human in the allotted 10-minute time period, and also not immediately solvable by either tool (Claude or Proofster). Our aim was to observe the participants interacting with each task and the tools for as long as possible. It was expected that most participants would not complete the tasks in the given time, however, three participants did solve the task in the combined condition (LLM + Proofster). Completion outcomes for all conditions are shown in Table 3.3.

Table 3.3: Task completion outcomes by participant for the no-tools, LLM-only, and LLM + Proofster conditions.

Participant	No Tools	LLM-only	LLM + Proofster
P1	×	×	×
P2	×	×	✓
P3	×	×	✓
P4	×	×	✓
P5	×	×	×
P6	×	×	×
P7	×	×	×

We analyzed the screen recordings to get a sense of how the participants interacted with the tools in each condition. In the LLM-only condition, participants fell into a common pattern of a copy-paste loop: pasting Rocq code into Claude, copying Claude's output into VSCode, encountering an

error in the proof, and then feeding that error back to Claude. In some cases, this cycle repeated many times without any progress. For example, participant P5 repeated the same broken loop 5 times, while getting nearly identical errors each time.[3] Participant P7, after a few broken loops, declared the process *"a waste of time"*, but still employed the copy-paste loop one more time later on. Two participants (P2, P4) eventually stopped using Claude altogether and attempted to finish the task on their own.

On the other hand, the LLM + Proofster condition showed more varied and exploratory user interactions. 6/7 participants ran Proofster immediately, and spent time interacting with the resulting proof search tree, hovering over nodes to see proof state, expanding nodes, and copying partial proof scripts. The participants naturally leaned toward using the tools in tandem, for example, running Proofster and turning to Claude for explanation while Proofster computes. Some participants carried over information from one tool to the other, asking Claude about tactics that Proofster mentioned, or running Proofster from a partially correct Claude proof. Two out of the three participants who completed the task in the combined condition (P3, P4) displayed approaches that integrated both tools in coherent workflows. P3 kept replacing failing tactics from Claude's attempts with `Admitted.` and re-running Proofster. P4 strategically used Proofster to prove helper lemmas, although ultimately completed the task using Claude.[4] On the other hand, P2 completed the task without using the LLM at all, by interacting with the Proofster search tree to start the proof, and then re-running Proofster to complete the proof.

Most commonly, LLM-generated Rocq code tended to encounter errors at the point of: applying or rewriting hypotheses or constructors (9 instances), failing tactics (6 instances of `reflexivity`,

[3]For this task, the statement of the proof was not provable as-is and required a specification change. Although Claude did point that out, this feedback was not sufficient to break the loop.

[4]The task was designed in such a way that Claude could not immediately solve it. Reasoning about helper lemmas may have helped Claude arrive at the correct solution.

`lia`, and `assumption`), and unrecognized lemma names (3 instances).

3.4.2 Using the Tools in Combination

During the interview, when asked which of the three conditions felt most helpful, six out of seven participants preferred the combined (LLM + Proofster) condition, while one participant (P5) preferred the LLM-only condition. Four participants (P2, P4, P6, P7) stated that they preferred to have multiple tools or perspectives on hand. Three participants (P1, P3, and P6) described specific workflows involving both tools that they considered helpful. Using the tools in combination helped participants learn new proof tactics, debug more effectively, or run the LLM in the background for high-level explanation while using Proofster to obtain a *"good structure of the proof"* (P6).

> *"The LLM was not even thinking about unfolding in the third case. But unfold is an available option that Proofster showed me." (P1)*

> *"In the future, it would be great if you could have an LLM that could use Proofster by itself." (P3)*

The participant who preferred the LLM-only condition (P5) stated that it was highly reflective of their regular workflow when writing Rocq proofs, and mentioned that Proofster might be more useful to those who *"start off by trying to work it [the proof] on their own"* before attempting to automate it.

When asked how their experience using the combination of LLM and Proofster was different from just using the LLM, five participants gave feedback beyond the simple preference for more tools on hand. P5 and P7 saw the utility of using Proofster to discharge simpler proofs or annoying cases. P1 and P6 thought that Proofster could provide assistance with complex proofs or proofs that the LLM was struggling with. Finally, P3 specifically mentioned the interaction and feedback between the tools: running Proofster in the background, and then giving its feedback to the LLM.

> *"For complex proofs, I prefer to use both together, since LLM is very committed to its solution." (P1)*

Table 3.4: Perceived workflow improvement and progress by condition (7-point Likert scale).

Survey Item	**Mean** (M)
LLM-only condition	
The LLM improved my verification workflow compared to working without assistance.	4.14
The LLM helped me make meaningful progress on the task.	4.86
LLM + Proofster condition	
The combination of LLM and Proofster improved my verification workflow compared to working without assistance.	5.43
The combination of LLM and Proofster helped me make meaningful progress on the task.	5.43
Interacting with both the LLM and Proofster, as opposed to either tool alone, made the experience more effective.	4.71
It was clear when I should consult the LLM versus Proofster.	3.86

> *"If it's a simple enough proof, it [Proofster] would just automatically finish it on its own." (P5)*

Participant responses to post-task survey questions (shown in Table 3.4) also provide some insight. Both the LLM alone and the combination of the tools helped participants make meaningful progress on verification tasks, with the combined condition scoring higher on average. There was a larger difference in scores for the question comparing the tool-assisted conditions to working without assistance, with the combined condition scoring 1.29 points higher on average than the LLM-only condition. The question about whether it was clear when each tool should be consulted had the lowest average score, mapping between "Somewhat Disagree" and "Neither Agree nor Disagree" on the scale.

3.4.3 Perceived Cognitive Burden

When asked whether they were overwhelmed or distracted by any of the tools, three participants mentioned being overwhelmed by the LLM (P1, P2, P7). Participants reported that they were overwhelmed by the size of LLM text output (P2, P7), particularly convincing output (P1), and by

the LLM showing reasoning that was useless or completely wrong (P2). One additional participant mentioned being overwhelmed by Proofster's proof tree UI specifically (P3).

> *"I think I was more overwhelmed by the LLM mostly because it's very convincing." (P1)*

> *"I don't want to see your [LLM] reasoning. I definitely don't want to see your wrong reasoning, because that makes me not want to use the tool." (P2)*

> *"I think the LLM just overwhelmed me, just in terms of, like, how much text output there was. I feel like it is not good for when I just need something." (P7)*

Two participants (P2 and P7) also mentioned that Proofster's feedback was hard to understand when describing their experience using the combination of LLM and Proofster:

> *"I didn't really understand the interface to make effective use of it." (P2)*

> *"It also takes a lot of time to parse what exactly Proofster is doing." (P7)*

Additionally, participants rated the perceived cognitive overhead associated with the tools (shown in Table 3.5). The mean scores indicate that in the combined condition, the cognitive overhead associated with the LLM dropped. Adding Proofster as an available tool may have helped to structure or guide the proving process, offloading part of the cognitive burden associated with the LLM, or transferring some of that burden to Proofster. Overall, the scores suggest that using the LLM alone was more mentally taxing than using either tool in the combined condition.

After each task, we also measured task load using the NASA Task Load Index (NASA-TLX [Hart & Staveland, 1988]), with subscale questions shown in Table 3.6. The sum of the average participant scores for all questions was 23.71 in the control condition, 22.14 in the LLM-only condition, and 17.86 in the combined (LLM + Proofster) condition, suggesting that the availability of tools (and especially both tools at once) reduced overall task load for participants.

Table 3.5: Perceived cognitive overhead by condition (7-point Likert scale).

Survey Item	**Mean** (M)
LLM-only condition	
Using the LLM added cognitive overhead or made the task harder.	3.86
LLM + Proofster condition	
Using the LLM added cognitive overhead or made the task harder.	2.71
Using Proofster added cognitive overhead or made the task harder.	3.29

Table 3.6: NASA Task Load Index (NASA-TLX) subscale questions administered after each task condition, with responses recorded on a 7-point Likert scale.

Subscale	**Question**
Mental Demand	How mentally demanding was the task?
Physical Demand	How physically demanding was the task?
Temporal Demand	How hurried or rushed was the pace of the task?
Performance	How successful were you in accomplishing what you were asked to do?
Effort	How hard did you have to work to accomplish your level of performance?
Frustration	How insecure, discouraged, irritated, stressed, and annoyed were you?

3.4.4 Capturing Intent Across Interfaces

In the short interview portion, we asked the participants whether it was easier to communicate their intent to the LLM or Proofster. Overwhelmingly, six out of seven participants stated that it was easier to communicate their intent to the LLM. Four of those participants (P1, P2, P3, and P5) credited the ability to use natural language to communicate their intent as the main reason why it was easier.

> *"I was able to use words with it." (P1)*

> *"You could just type down whatever, and it would understand." (P5)*

Additionally, two participants (P2 and P7) stated that including additional detail was easier via the LLM interface. P2 described using analogies, or stating proof strategies outright to the LLM, while P7 highlighted the fact that they could specify as much detail as they saw fit via the LLM interface.

Table 3.7: Perceived usability and expressiveness by interface (combined condition; 7-point Likert scale).

Survey Item	**Mean** (M)
Proofster interface	
The interface for interacting with Proofster was easy to use and navigate.	4.00
I knew how to make requests to Proofster effectively.	4.14
LLM interface	
The interface for interacting with the LLM was easy to use and navigate.	5.71
I knew how to ask questions or make requests to the LLM effectively.	6.43

One participant (P4) specifically pointed out being able to have a conversation with the LLM in the same way as one would with a human, implying that the communication of intent was tied to the received feedback.

The final participant (P6), who stated that it was easier to communicate their intent to Proofster, explained that with Proofster they were able to clearly follow the path that the tool was taking to find the solution (through the displayed proof search tree).

We also collected responses to the following questions on perceived usability through the post-task survey for the combined condition (shown in Table 3.7). The scores suggest that, on average, the LLM interface was easier to use and navigate and made it easier for participants to convey requests in the combined condition. These results are reflective of the participant feedback collected through the interview.

3.4.5 Tool Feedback Shaping Understanding

During the short interview, when asked which tool gave clearer or more actionable feedback, six out of seven participants favored LLM feedback. Four participants attributed this to the fact that the LLM is more conversational and allows the user to interact with it (P1, P2, P4, and P5). More specifically, P1 pointed out that the LLM can sketch out or verify an idea that it is given, P4 mentioned having a human-like conversation with the LLM, and P5 appreciated the ability to ask

follow up questions to get an explanation. P2 had a nuanced take on the interactive aspect, since LLM feedback can sometimes be confidently wrong, and it's really further interaction with this wrong feedback that helps figure out a solution.

> *"I was able to tell it [LLM] my idea and have it verified, and then have it show me a sketch of what my idea would look like." (P1)*

> *"The LLM gives more actionable feedback, but it's usually because the LLM is giving the wrong feedback. [...] So while I would say that, yes, I got better feedback from the LLM, it's not a compliment to the LLM." (P2)*

> *"You can have a sort of conversation with it in the same language that you use to express yourself to another human." (P4)*

> *"You could ask it to explain why the issue might be there, or you could even ask it to explain what needs to be done." (P5)*

Proofster feedback was not favored overall, with participants stating it just does its own thing without direct user interaction (P1), and provides limited incomplete output, such as the shallow search tree (P2, P7). One participant (P6) did prefer Proofster feedback, since it showed the branches of the search tree and they were able to see if they were going in the right direction.

The participants were also asked how the feedback from each tool shaped their understanding of the task and their approach to constructing the proof. One participant (P4) indicated that neither tool's feedback helped shape their understanding or approach because they did not interact with the feedback at all. Among the participants that did interact with tool feedback, three found that the LLM's feedback was useful for a conceptual understanding of the proof or breaking down proof strategies, and for learning which tactics were available (P1, P5, P7). One participant (P4) highlighted the LLM's ability to transform a goal they did not want to bother solving into a goal they knew how to solve easily.

> *"LLM seemed more descriptive of how to do it, and breaking it down for me more." (P1)*

> *"The LLM, I feel like it explained it better for me to understand what's actually going on, than the Proofster." (P5)*

Table 3.8: Perceived usefulness and interpretability of each tool's feedback in the combined condition (7-point Likert scale).

Survey Item	**Mean** (M)
LLM feedback	
The LLM's responses were useful and relevant to the task I was working on.	5.14
I found the LLM helpful in diagnosing or resolving proof errors.	4.29
The LLM's feedback made it easier to understand what changes were needed in my proof.	3.86
I understood why the LLM made the suggestions it did.	4.14
Proofster feedback	
Proofster's feedback was useful and relevant to the task I was working on.	2.86
I found Proofster helpful in diagnosing or resolving proof errors.	3.00
Proofster's feedback made it easier to understand what changes were needed in my proof.	3.43
I understood why Proofster made the suggestions it did.	2.86

On the other hand, Proofster was perceived as more useful for discharging cases without thinking too much (P1, P7). The displayed proof search tree got mixed reviews. Specifically, P2 mentioned that Proofster's feedback validated their intuition for the proof approach, while P5 and P7 thought that visually seeing the options that Proofster explored did not help much.

> *"With Proofster, it was reassurance that my intuitions were correct about how to solve the problem." (P2)*

Additionally, participants expressed that there were caveats to the usefulness of the feedback, such as the LLM feedback being overwhelming for easier proofs and more helpful for harder proofs (P3) and feedback from either tool not being useful for diagnosing proof errors (P7).

Responses to post-task survey questions on feedback quality and interpretability are summarized in Table 3.8. Overall, the participants rated LLM feedback higher with respect to usefulness and interpretability, with usefulness showing the largest difference in mean scores (LLM: M = 5.14, Proofster: M = 2.86). The question about understanding what changes were needed in the proof based on tool feedback had the lowest difference in mean scores, also reflecting P7's sentiment above.

3.4.6 Recall of Task and Approach

After each task, the participants were asked to recall the statement of the proof and their approach to proving it. The results suggest that the participants' ability to recall what they were working on may have been impacted by the availability of helper tools (Table 3.9).

Table 3.9: Participants' ability to recall the statement of the proof task in each condition, with × indicating instances when the participant was not sure or could not recall the statement.

Participant	No Tools	LLM	LLM + Proofster
P1	✓	✓	×
P2	✓	×	✓
P3	×	✓	✓
P4	✓	✓	×
P5	×	×	×
P6	✓	✓	×
P7	✓	✓	✓

Across conditions, recall performance was mixed, but the combined LLM + Proofster condition was associated with the weakest recall overall. For each of the other conditions (no supportive tools and LLM-only), 5/7 participants were able to recall the proof statement, as opposed to 3/7 for LLM + Proofster (Table 3.9). According to the quotes shown in Table 3.10, the participants largely indicated partial or vague memory of the theorem statement rather than a complete absence of task understanding.

When describing their approach to solving the task at hand, multiple participants expressed that they outsourced the reasoning part to the tools themselves, and acted in a more supervisory capacity when managing the tools. In the LLM-only condition, P5 started off *"by giving the entire thing to Claude,"* and P6 *"was doing whatever Claude was saying to do."* In the combined condition (LLM + Proofster), P1 and P3 described using both tools side by side or in the background, while P5 focused on iterating with Claude after Proofster did not produce a solution straight away. P4

Table 3.10: Participant recall quotes grouped by study condition.

Condition	Representative quotes
No Tools	*"In my own words, I was still trying to figure it out." (P3)*
	"My mind's blank for some reason right now." (P5)
LLM	*"Can I? Oh my gosh, I just did it. Did I just blank? [...] No, it's completely lost on me. I'm sorry." (P2)*
	"I think it was something to do with deletion and how that works." (P5)
LLM + Proofster	*"I don't remember the exact theorem, but that's what generally the theorem was expecting." (P1)*
	"When you delete something in a record, its size decreases. I'm not sure about anything more than that." (P4)
	"It was something. I don't remember the entire lemma, honestly." (P5)
	"The task seemed to be something relating to, like, typing." (P6)

attempted to use both tools in tandem by synthesizing helper lemmas via the LLM, and solving them using Proofster. Ultimately, none of the helper lemmas were necessary for the task at hand (the solution did not require them), but this interaction may have helped P4 arrive at the solution.

> *"The way I approached this was to not think about it at all. [...] I'm just going to use the proof [...] automating systems." (P1)*

> *"My approach was to try running it again and again with Claude until Claude figures out where the issue is and shows me the correct solution." (P5)*

3.4.7 User Trust

During the short interview, participants were asked how much they trusted the information and suggestions provided by the tools. Six out of seven participants expressed distrust in the LLM, citing low confidence in the LLM's ability to handle complex proofs (P1), to handle proofs in general (P2, P7), or referencing their experience with a particular proof task (P6). P3 expressed a general distrust of Claude, while P4 did not trust any tools before trying them out.

> *"I trusted the tools slightly more than [...] usual for simpler proofs, but I highly doubted them for complex proofs." (P1)*

> *"I know how LLMs work, and I definitely do not trust their output for an interactive, you know, theorem proving setting. [...] They will routinely take you down a wrong path, and they will leave you there." (P2)*

> *"For actual proofs I didn't trust the LLM at all." (P7)*

On the other hand, four participants reported trusting Proofster, referencing its ability to build on existing proof progress (P6), its guarantee of correct solutions (P7), and their ability to use the tree output to look things up or backtrack as needed (P2, P3). At the same time, P7 expressed that while they trusted the solutions produced by Proofster, they did not trust that Proofster would find a solution.

> *"I can back it [Proofster] up to a point where I still agree that the goal is provable and then I can continue manually from there." (P2)*

> *"I trust in the fact that everything [Proofster generated] was correct like 100%" (P7)*

When asked if there were specific interactions or tasks that increased or decreased their trust, participants' responses showed a clear pattern: using Proofster to effectively solve proof goals increased their trust in Proofster (P2, P3), while failing to solve proof goals with either tool decreased their trust in that tool (P1, P7). P7 added that after a failing result, combing through Proofster's output restored their trust in the tool. Some participants also mentioned specific LLM behaviors that decreased their trust, like rewriting the whole proof (P6) or pushing them down a wrong proof trajectory (P1). Finally, P5 expressed that just because LLMs are not guaranteed to return a correct solution, it doesn't mean we should stop trusting them.

> *"If it's not giving you the correct answer, it doesn't mean that you should not trust it anymore. It's just that these are learning models, so they're still learning." (P5)*

Before beginning the study, participants submitted responses to the following pre-study survey questions about confidence, trust, and critical assessment of tools, summarized in Table 3.11. The mean scores suggest that participants were generally more confident in using outputs from proof automation tools than LLM outputs, and reported critically assessing LLM outputs more frequently.

Table 3.11: Self-reported confidence, trust, and critical assessment for LLMs and proof automation tools (7-point Likert scale).

Survey Item	**Mean** (M)
Large language models	
How confident are you in using the outputs from large language models?	4.86
How much do you trust large language models to provide correct results?	3.86
How likely are you to critically assess the results produced by a large language model?	6.00
Have you ever had a negative experience with a large language model that led you to question its credibility?	4.57
Proof automation tools	
How confident are you in using the outputs from proof automation tools?	5.57
How much do you trust proof automation tools to provide correct results?	4.86
How likely are you to critically assess the results produced by a proof automation tool?	3.43
Have you ever had a negative experience with a proof automation tool that led you to question its credibility?	2.29

Table 3.12: Trust in tool feedback and suggestions by condition (7-point Likert scale).

Survey Item	**Mean** (M)
LLM-only condition	
I trusted the LLM's feedback and suggestions.	3.86
LLM + Proofster condition	
I trusted the LLM's feedback and suggestions.	4.43
I trusted Proofster's feedback and suggestions.	3.29

After each task, we also measured participant trust in each tool's feedback and suggestions (Table 3.12). When trust was anchored to a specific task, participants on average trusted LLM feedback more than Proofster feedback. They also trusted LLM feedback more when the tools were paired together, than when using the LLM by itself.

3.4.8 Perceived Deception

During the short interview portion, participants were asked to report whether they felt misled or deceived by either interface, and under what circumstances. Three out of seven participants reported being misled by the LLM interface (P1, P6, and P7). Two out of the three participants

attributed this to a bad proof strategy or approach taken by the LLM, with P1 crediting their own misleading prompt, and P6 mentioning that the LLM was not willing to change its approach when given feedback.

> *"I think me saying my idea helped it misguide me, even if it did know how to do it right." (P1)*

> *"[The LLM] has, like, a fixation on being right." (P6)*

P7 reported that the LLM interface was misleading because the LLM was unable to give *"plausible back justification for why they did something"* and mentioned LLM hallucination. Most of the participants (P2, P3, P4, P5) did not report being misled or deceived by either interface. While P4 did not report being misled by the LLM during this particular study, they did report having been misled by LLMs in the past. No participants reported being deceived or misled by Proofster, although one participant (P7) mentioned that the information given by Proofster was unclear.

3.5 Discussion

Our study explored how participants engaged with two different tool interfaces that supported them in Rocq verification tasks: a large language model (Claude) and a search-based proof automation tool (Proofster). The participants interacted with the tools in two experimental conditions (LLM-only, and LLM + Proofster), and also experienced a control condition working on a verification task without helper tools. Our results touch on themes around interface comfort, clarity and usefulness of tool feedback, user trust in tool outputs, and conveying user intent via tool interfaces. Below we discuss our results with respect to the research questions posed earlier.

RQ1: How do users balance interface comfort against the perceived trustworthiness of each tool's outputs? Participants overwhelmingly preferred the LLM on usability dimensions: they found the LLM interface easier to use and navigate ($M = 5.71$ for the LLM vs. $M = 4.00$

for Proofster in the combined condition), and easier to comunicate with effectively ($M = 6.43$ vs. $M = 4.14$, see Table 3.7).[5] Participants especially appreciated the natural language support of the LLM interface, which enabled them to express more nuanced intent: query explanations, ask follow up questions, or ask for proof strategies rather than solutions.

On the other hand, this usability preference for the LLM interface was accompanied by a stark distrust in the LLM's outputs. When asked how much they trusted the information and suggestions provided by the tools during task completion, six out of seven participants expressed distrust in the LLM. Similarly, the results of our pre-study survey confirmed that participants were generally more confident in using outputs from proof automation tools ($M = 5.57$) than LLM outputs ($M = 4.86$), and were less trusting of LLMs to provide correct results ($M = 3.86$) as opposed to proof automation tools ($M = 4.86$). Participants expressed their awareness of common LLM failure points in proof engineering settings, for example, buggy proof scripts and confident exploration of wrong proof strategies.

Another interesting point is that during the study participants reported trusting Proofster's feedback and suggestions less ($M = 3.29$ for LLM + Proofster) than the LLM's feedback and suggestions ($M = 3.86$ for LLM-only, $M = 4.43$ for LLM + Proofster). This was surprising, given that the pre-study survey showed higher confidence and trust for proof automation tools as opposed to LLMs. This indicates that there may be other factors influencing user trust beyond the simple categorization of the helper tool. Familiarity may be one such aspect, as multiple users expressed that they were not able to use the Proofster interface effectively as it was their first time using it, or their first time seeing a tree-based proving UI at all. Indeed, in the pre-study survey, participants expressed to be more familiar with LLMs (5/7 use LLMs "often", 1/7 "always," and 1/7 "occasionally") than proof automation tools (4/7 use them "sometimes", 1/7 "often," 1/7

[5] All scores are on a 7-point Likert scale.

"always," and 1/7 "occasionally").

This disconnect between tool usability and trustworthiness is an important consideration for tool design. Users may prefer tools that are easier to navigate, even if they trust their outputs less. Users may also trust helper tools less simply because they are less navigable. One implication of this disconnect is that seeking interface comfort may lead users to overly rely on incorrect outputs. One example of this was the copy-paste loop pattern we observed in the LLM-only condition.

RQ2: To what extent does using the tools in combination provide additional benefits over the use of each tool alone? Our results suggest that using the tools in combination may improve user experience, support more effective verification workloads, and lessen cognitive load.

Overall, six out of seven participants thought that the combined condition was most helpful. Using the tools together yielded the highest task completion rate (3/7, versus 0/7 in the other two conditions). Compared to working without assistance, participants reported that the combination of LLM + Proofster improved their verification workflows more so than the LLM alone ($M = 5.43$ versus $M = 4.14$ in the LLM-only condition). While the LLM alone already helped users make meaningful progress ($M = 4.86$), using the tools in combination was even more helpful ($M = 5.43$). This complementary effect of the tools may stem from their different strengths: flexibility for the LLM, and certainty for Proofster. Participants reported that the LLM was useful for high-level explanation and problem decomposition, as well as for asking follow up questions. On the other hand, Proofster was more valued for its ability to display the search process (allowing backtracking) and for discharging simpler goals.

Participants also reported that the LLM added less cognitive overhead when Proofster was also available to use ($M = 2.71$ versus $M = 3.86$ in the LLM-only condition). NASA-TLX scores were consistent with participant interviews: the average combined task load score for all 6 questions

dropped from 22.14 in the LLM-only condition to 17.86 in the combined condition. These results suggest that the presence of Proofster reduced some of the mental load associated with LLM use, perhaps by providing a more symbolic anchoring point and reducing the uncertainty that the participants reported of LLM outputs.

The lowest-scoring survey item in the combined condition asked whether it was clear when to consult each tool ($M = 3.86$), suggesting that while both tools may have been beneficial to participants, it was not always obvious how to use them in tandem. This highlights an opportunity for tool design: offering a larger variety of helper tools may be more effective when combined with scaffolding to coordinate the use of each tool.

RQ3: How does user trust in either interface change (if at all) after task completion or non-completion? Participant trust in each tool seemed to depend on whether the tool appeared to make progress on the task at hand. Discharging proof goals with Proofster or combing through the tree output to follow its reasoning increased trust in Proofster (P2, P3, P7), while failures with either tool decreased trust (P1, P7). After experiencing an error in LLM-generated code, we observed two users stop using Claude altogether for the rest of the task (P2, P4). Participant P4 also expressed not trusting either tool at all until trying them out. These findings imply that user trust in tools is contingent on each tool's performance and evolves in response to the tool's behavior during interaction. This interpretation is consistent with the dynamic model of trust proposed by Lee & See [2004].

Interestingly, according to the post-task survey for the combined condition, the trust ratings for both tools were lower on average for those participants who completed the task:

Item	Mean (M) Completed	Mean (M) Not completed
I trusted Proofster's feedback and suggestions.	2.67	3.75
I trusted the LLM's feedback and suggestions.	3.33	5.25

This is a somewhat surprising result. However, the lower trust scores for participants who completed the task may reflect a better calibration of trust with respect to tool capabilities. By engaging deeply with the task and the tools, these participants may have calibrated appropriate trust in the tools, which may not necessarily align with high trust [Lee & See, 2004]. On the other hand, participants who reported higher trust may have been overly reliant on the tools, leading to task non-completion.

RQ4: How effectively do users perceive their intent is captured and executed across the two interfaces? Six out of seven participants found it easier to communicate their intent to the LLM, primarily citing the LLM's natural language capabilities. Participants felt that interaction through natural language supported a variety of solving strategies, from stating proof goals directly, to using analogies, to specifying any range of detail. The conversational nature of the LLM interface also supported iterative interaction between the user and the interface, where the latter interactions could build on the prior ones, including the addition of user feedback such as error messages or a change in strategy.

On the other hand, Proofster's ability to capture intent was limited to completing an existing proof script. P2 described the intent conveyed to Proofster as straightforward: *"I want you to solve the proof, [...] there's one button for that."* P1 mentioned not being able to interact with Proofster's solving process directly. Conveying intent to Proofster through a partial proof required participants to essentially express themselves in Rocq, which created a higher barrier for commu-

nication. The one participant (P6) who preferred Proofster for communicating intent, did so due to the transparency of the search tree. This response blurs the line between communicating intent and receiving feedback: the feedback from Proofster helped this participant express their intent as they were able to make sure that the tool was following the proof path they intended.

These findings suggest that the two interfaces provide different avenues for users to communicate their intent. The LLM supports iterative sketching and refinement of proof strategy, while Proofster supports one-shot proof completion and inspection of proof structure. A more integrated interface could help unify these communication styles, perhaps, *"an LLM that could use Proofster by itself"* as P3 envisioned.

RQ5: Do users report feeling misled or deceived by either interface, and under what circumstances? While most participants did not report feeling misled or deceived by either interface, three participants reported feeling misled by the LLM, citing the LLM's *"fixation on being right"* (P6), the participant's own misdirection (P1), and the lack of plausible justification (P7). The common thread in these reports is the LLM's tendency to stick with the existing train of thought, even if it is not correct.

Notably, no participants reported feeling misled by Proofster's interface. This difference in perceived deception for each interface may reflect the ways both tools convey failure feedback. When Proofster fails to synthesize a proof, it fails transparently by reporting the failure and displaying the search tree. On the other hand, the LLM may not communicate a failure at all, and leave it to be discovered later by the user (for example, after pasting the buggy Rocq code into their editor). The generative nature of LLMs is also conducive to a convincing text argument, regardless of whether a failure is present. This difference in failure transparency may be one of the reasons why participants report feeling misled by the LLM, but not Proofster. In regards to tool design,

surfacing failures early (or as P2 expressed, *"fail fast"*), could help mitigate the misleading effects of some interfaces.

RQ6: To what extent does feedback from each tool contribute to users' understanding of the task and proof construction? Six out of seven participants thought that the LLM gave clearer and more actionable feedback. In the combined condition, participants rated LLM feedback higher than Proofster's in both usefulness ($M = 5.14$ for LLM, $M = 2.86$ for Proofster) and interpretability ($M = 4.14$ for LLM, $M = 2.86$ for Proofster)[6]. LLM feedback was described by participants as useful for high-level understanding of the proof and strategy, and for learning which tactics were available. Participants also found the conversational nature of the LLM useful, enabling them to ask follow-up questions or seek clarification.

Proofster feedback was not favored overall, but still appreciated by some for its assurance on solved goals, and its ability to validate participants' intuitions through exploring the search tree. The search tree was described as informative and useful by some (P6), but as difficult to interpret by others (P2, P7). The informative value of the tree seemed to vary by participant, potentially indicating that individual familiarity with tree-based UIs and similar proof automation tools may have been a factor.

Neither tool was regarded as particularly helpful in making it clear what changes were needed in the proof (LLM: $M = 3.86$, Proofster: $M = 3.43$). The small difference in average scores reflects a shared limitation of the tools: their feedback, which for both tools can be considered quite detailed, does not effectively communicate concrete guidance on how to debug or finish the proof. This suggests a gap between the perceived usefulness of feedback, and translating this feedback into tangible proof construction steps.

[6]It's worth noting that while interpretability was rated higher for the LLM, that score still corresponds to about a "neutral" rating on the 7-point Likert scale.

At the same time, participant recall of the proof statement they were working on was lowest in the combined condition (3/7, as opposed to 5/7 in both other conditions). When participants have multiple helper tools available to them, they may outsource some of the reasoning to the tools, and therefore undercut their own understanding of the problem. This is consistent with the participants' own descriptions of how they approached the tasks, describing a more supervisory role over the tools in the tool-assisted conditions, as opposed to describing their own reasoning as in the control condition. Any design of tool-assisted workflows for formal verification should take into account not only task completion, but also task understanding.

In fact, participant interactions with helper tools did indicate that they sought an understanding of the problem beyond just the solution. Overall, participants tended to interact with tool feedback (LLM explanations, Proofster search tree) before copying and pasting code into their editor. Some participants directly requested that the LLM not give away the solution (P1, P4). P7 and P1 prompted the LLM with questions about specific tactics, and P4 prompted for helper lemma statements. These behaviors reflect participant desire to understand the task at hand, and not just to solve it. This is an important consideration to keep in mind when designing proof automating tools, as most existing tools on the market focus on correctness rather than understanding.

Limitations. When interpreting the results of our study, some limitations apply and should be considered. With only 7 participants, the size of our participant pool is not large enough to detect any statistically significant differences between conditions. The results should be interpreted qualitatively, and as more exploratory in nature. Since the tool-assisted conditions have some tool use overlap (both LLM-only and LLM + Proofster allow access to Claude), there may be some order effects at hand, where exposure to the LLM in a prior condition may have affected participant use of the LLM in a later condition. The 10-minute time constraint for each proof task may not be

reflective of how participants would naturally use these tools in their own development workflows. In naturalistic settings, users may spend more time getting familiar with tool interfaces, and develop more involved strategies for using the tools in tandem. Finally, trust, cognitive load, and tool usability were assessed through self-reported survey and interview questions, which are inherently subjective.

3.6 Conclusion

In conclusion, this study assessed how users interact with helper tool interfaces while solving verification tasks in Rocq. Our findings show that users heavily favored the LLM's natural language interface even though they highly distrusted its outputs, and at times felt misled by it. Proofster's interface provided complementary strengths, such as its transparent proof search process and the ability to discharge simpler goals, leading to reports of reduced cognitive load in the combined condition. Our study highlights some opportunities for the future design of proof automation tools: clear communication of failure to avoid deception, individual tool use guidance within integrated systems, and expressing feedback to the user in a way that translates to clear, actionable next steps. Finally, tool design should support users in engaging with and understanding the problem at hand, and not just in finding a solution.

4

Future Work

One common thread across the different facets of this work is the idea of structure or scaffolding that serves to support safer code evolution. In language design, structure appears in the form of language constructs and abstractions that empower safe composition. In proof automation tools, it takes the shape of partial proofs that guide proof synthesis, as well as the strategies that exploit these partial solutions. In user interfaces, structure appears in the actions that are available to the user through the interface and the feedback given by the tool. Below, we examine some future research directions that continue along this common thread.

4.1 The Meta-Extensibility Framework

Persimmon is just one of the many existing approaches to built-in extensibility in programming languages. One future direction that could inform the broader landscape of extensible programming is the idea of a *meta-extensibility framework*: a model that captures the essence of extensibility within systems, focused around the principles of extension (e.g., inheritance and overwriting) more so than any specific vehicle enabling it (e.g., relative path types). By generalizing the notion of extensible constructs and what it means to extend them, we can arrive at a unified way to express extensibility across languages and proof assistants.

One way to generalize the extensible constructs we saw in Persimmon is by grouping language constructs into *containers* and *code blocks*. Families in Persimmon act as containers, and they can hold nested containers or code blocks (such as type definitions or function bodies). Then, extensibility can be reduced to the safe addition of new containers (such as nested or derived families) and code blocks (such as new pattern match cases). A small set of general rules for adding containers and code blocks soundly and compositionally would then govern extensibility.

Such an abstraction would not be tied to any specific features of Persimmon. Containers might represent families, modules, or classes in different languages, and the set of code blocks would depend on each language's syntax. A meta-extensibility framework of this kind would allow language designs to instantiate the framework with their own extensible constructs and operations defining their composition. Any safety guarantees would hold of the framework itself, meaning that each instance of an extensible language built using this framework would enjoy those guarantees by construction.

Proof Sketch Structure. In our exploration of the scaffolding paradox, we focused on proof skeletons that were restricted to a top-level sketch. Future work could explore how the depth of a sketch (from top-level skeletons to almost-complete proofs with holes) or different types of scaffolding within a sketch (e.g., inclusion or exclusion of inductive calls within inductive cases) affect the results. Is there a point on the spectrum from reference solution to top-level sketch at which the sketch loses just enough information to cease being helpful? Is there a point at which adding more structure into the sketch leads the model to get stuck in local exploration, and fail to recover a global solution? Evaluating a variety of sketch depths and constructions would shine light on what makes a "good" partial proof for seeding LLM-aided proof synthesis.

Case-Focused Repair for Longer Proofs. We could also evaluate whether case-focused repair is a worthwhile investment, particularly for synthesizing longer proofs. Since each proof may break down into multiple inductive subcases, and each case may need multiple LLM calls to repair it, it makes sense that case-focused repair requires a larger call budget. This increased cost may not be justified for shorter proofs – if fewer whole-proof repair calls can yield a solution – but may be necessary for longer proofs. In our exploration of case repair, all of our benchmark reference solutions were under 100 lines of Dafny code. Future work could consider benchmarks with longer proofs, however, Dafny's inference engine is quite powerful, and Dafny proofs are often shorter than the proofs of similar properties in other languages (such as Rocq). This makes Dafny a great target for LLM-assisted proof synthesis, but also makes longer reference solutions hard to find. An evaluation based around proof length may require the construction of a new benchmark with longer proofs.

Orchestration Strategies. There is also the matter of trying out a more sophisticated orchestrator than a simple bounded repair loop. In particular, an orchestrator capable of planning strategic moves, such as digging in deeper on a case repair or backtracking, could improve the success rates. One difficulty in implementing a strategy-based orchestrator would be designing the heuristics to assess how close the current proof state is to the goal, and what the next move should be. As LLMs models grow more capable, they could become such assessors or orchestrators themselves.

Robustness to Novel Verification Approaches. Since commercially available LLMs are trained on public data, their solve rates for proof synthesis tasks may be affected by whether a certain proof strategy has appeared in their training data. Seeding lemma bodies with proof sketches derived from publicly available reference solutions (such as open-source benchmarks) may boost solve rates as the model may have been exposed to the particular solution during training. On the other hand, novel verification strategies for a given lemma (e.g., a proof by induction on a different variable) may result in lower solve rates even when the alternative approach is also valid. Future work may explore model robustness with respect to varying proof strategies, for example by comparing solve rates for lemma bodies seeded with a range of valid proof sketches that differ in structure and verification approach. Evaluating performance across a range of verification approaches would help differentiate between the model's ability to reason about the given proof direction and its ability to recall solutions based on training data.

4.3 User-Centric Interfaces for Proof Synthesis Tools

Adaptive Interfaces. During our user study, participants expressed being uncomfortable with large amounts of potentially unreliable feedback (e.g., Claude's wall of text). Participants also expressed that it was difficult to make quick moves or parse actionable items given the feedback.

One path forward is to consider proof automation interfaces that adapt the displayed feedback based on user cues, such as experience level and perceived task difficulty. For example, an expert might get a shorter, denser description of the solution, while a novice might get a more in-depth explanation of the approach with step-by-step hints.

Tool Selection in Multi-Tool Systems. Another user-reported challenge in our study was the uncertainty about when each tool should be used. Tool selection could be integrated into multi-tool proof automation frameworks, with the orchestrator of the tools either discharging tasks to appropriate tools automatically (and handling any failures internally), or surfacing tool suggestions to the user based on the task at hand. It is important to note that when orchestration help is offered to the user, it must be grounded in the strengths of each tool, as users perceive them. Such an assessment of tool strengths could be informed by further comparative studies of tool interfaces.

Encouraging Task Comprehension. One other concern surfaced by our study is the potential relationship between the use of proof synthesis tools and task comprehension. In our study, participants had the lowest recall of the proof task and approach in the combined tool condition. At the same time, participants sought to understand the problem and the candidate solutions, interacting with the tool output and asking follow-up queries. Future work could explore tool design approaches that actively foster user understanding of the problem and the synthesis feedback. For example, before synthesizing a candidate solution, a tool might present a choice of proof strategies to the user, along with some pointers about the benefits and downsides of each. The user could pick or alter a strategy before the solution work begins. For tools that present partial solutions to the user (such as Proofster), the large search space of partial proofs could be summarized in a digestible way. For example, a partial solution could be presented to the user not as a list of tactics, but in terms of subgoals that were opened and solved.

5

Conclusion

This dissertation touches on three ways to support safer code evolution: programming languages that support evolving code through built-in features, neurosymbolic frameworks that can enable the co-evolution of code and mechanized proofs, and user interfaces for proof automation. It first presents Persimmon, a functional language design that supports extensible variant types and pattern matching via nested family polymorphism. Then, it explores a neurosymbolic approach to automated Dafny proof synthesis using Dafny Sketcher, and highlights a scaffolding paradox that arises when seeding proof synthesis with proof skeletons from reference solutions. Finally, it investigates how users interact with supportive tool interfaces for automated proof synthesis, and offers considerations for future tool design. Altogether, this work aims to inspire a tool- and

language-based perspective for safer code evolution.

Beyond the contributions of the individual projects, the three facets of our work highlight a few big-picture lessons on the relationship between safety guarantees, the structure that supports these guarantees, and the humans who interface with these pieces. We expand on these lessons below.

Safety Needs Structure. Safety guarantees are too complex to be implemented as surface features. Safety is a *system property* that must be structurally supported within languages and tools. Well-chosen unifying abstractions can serve as structural support for safety. In Persimmon, the family polymorphic safety guarantees rely on our choice of *linkages* as the unifying abstraction and the internal representation for families. In contrast to traditional class tables, which are essentially flat maps of class names to class definitions, linkages preserve the inheritance and nesting structure of the program within the representation. Thus, the linkage representation naturally supports reasoning about type safe extensibility of nested components. Similarly, neurosymbolic tools for proof automation rely on partial proofs as scaffolding for proof synthesis. Partial proofs serve as supporting structure that an LLM or symbolic tools can fill in to finish the proof. The right abstractions can support the expression and enforcement of safety properties, whether by a type system or by a verifier checking the output of a neurosymbolic tool.

Choosing Beneficial Structure. Simply knowing that structure is helpful is not enough. It is also important to choose structural support that fits the problem and helps manage complexity. Nested extensibility in Persimmon needed structure that a traditional class table could not provide. The scaffolding paradox experiments with Dafny Sketcher showed that proof skeletons retrieved from reference solutions did not meaningfully help Dafny proof synthesis, even though the structure provided was on track for proof completion. In the user study on proof automation interfaces, the informative value of the proof search tree displayed by Proofster was favored for its failure

transparency (it was clear that a solution was not found), but not for its clarity (not immediately clear which branch showed the most promise). The fitting abstractions and structures must strike a balance: managing existing complexity, but not compounding it. The work of selecting a fitting abstraction is thus an important component of tool and language design.

Shifting the Cost. Since safety is a property and not a surface feature, the *complexity* of supporting safety guarantees cannot be eliminated, it can only be shifted between the components of the system. Persimmon shifts complexity from type checking to linkage computation, relying on linkage concatenation to "handle" extensibility on demand. Automated proof synthesis tools shift complexity away from constructing the proofs to orchestrating and managing the tools themselves. In our work with Dafny Sketcher, we experienced the difficulty of orchestrating LLMs with symbolic sketcher tools firsthand via the scaffolding paradox. The LLM models' lack of transparency came up in both the orchestration of sketchers (hard to pinpoint why certain orchestration strategies work better than others), and the user study (our participants complained that the LLM failed silently). This is a lesson worth learning: when shifting complexity around, it's important to know where the costs will land after all, and how they will be experienced by the users.

The Human Burden. Of course, we must reflect on the underlying motivation of this work: the burdens of code evolution should not fall to the programmers. As programming language researchers, we love to build languages and tools, with good intentions. However, this dissertation makes it clear that engineering is only part of the job. The other, more difficult, component is the interaction between the tools and the users, and how it is facilitated. If helper tools are built to aid the lives of programmers, then we cannot overlook this human aspect. Our user study suggests how programmers may experience shifting complexity through tool feedback. Our participants gravitated to the comfortable natural language interface of the LLM, even though they indicated

low trust in its outputs. Proofster's interface guaranteed correctness-by-construction, but restricted the users' ability to express their intent. Participants reported low confidence on when to use each tool, and wished for more effective multi-tool workflows. Our work shows that even when the safety features within the tools are well-intended, their effect in practice is shaped by human engagement. Simply making the tools available is not a guarantee of reduced burden.

I'd like to end this section with a final takeaway. While the burdens of safer code evolution should not fall to the programmers, this does not mean removing them from the equation. The programmers must remain a central point in this conversation to ensure that they reap the benefits of the tooling created on their behalf.

A

Persimmon Semantics

A.1 Core Semantics

$$\frac{\begin{array}{c}\exists\, a,\ L_S.\texttt{self} = a \qquad L_S.\texttt{self} \notin \texttt{ancestors}(L_S) \\ (\exists\, a'.A,\ L_S.\texttt{super} = a'.A \ \wedge\ \neg\texttt{nested}(a'.A, L_S.\texttt{self})) \ \vee\ L_S.\texttt{super} = \texttt{null} \\ \mathrm{K} = sp :: \mathrm{K}' \qquad \forall\, A' \mapsto L'_S \in L_S.\texttt{NEST},\ \texttt{self}(sp).A' :: \mathrm{K} \vdash \texttt{WF}(L'_S) \\ \forall\, R \mapsto \{(f_i : T_i)*\} \in L_S.\texttt{TYPES},\ \mathrm{K} \vdash \texttt{WF}(\{(f_i : T_i)*\}) \\ \forall\, R \mapsto (f_k)* \in L_S.\texttt{DEFS},\ \exists\, R \mapsto \{(f_i : T_i)*\} \in L_S.\texttt{TYPES} \wedge \forall k,\ f_k \in (f_i)* \\ \forall\, R \mapsto \overline{C_j\ \{(f_i : T_i)*\}} \in L_S.\texttt{ADTS},\ \forall j,\ \mathrm{K} \vdash \texttt{WF}(\{(f_i : T_i)*\}) \\ \forall\, m \mapsto (T \to T') \in L_S.\texttt{FUNS},\ \mathrm{K} \vdash \texttt{WF}(T \to T') \\ \forall\, (c \mapsto (\langle a'.R\rangle, T \to \{(C_j : T_j \to T')*\})) \in L_S.\texttt{CASES}, \\ \mathrm{K} \vdash \texttt{WF}(a'') \ \wedge\ a'' \rightsquigarrow L'_S \ \wedge\ R \mapsto \overline{C_i\ T_i} \in L'_S.\texttt{ADTS} \ \wedge \\ \forall j, \exists i, (C_j = C_i \wedge T_j = T_i) \ \wedge\ \mathrm{K} \vdash \texttt{WF}(T \to \{(C_j : T_j \to T')*\})\end{array}}{\mathrm{K} \vdash \texttt{WF}(L_S)}\ \text{(WF-Linkage-S)}$$

Figure A.1: Linkage well-formedness for static linkages L_S.

$$\frac{\begin{array}{c}\exists\ a,\ L_D.\texttt{self}\ =\ a \qquad L_D.\texttt{self} \notin \texttt{ancestors}(L_D) \\ (\exists\ a'.A,\ L_D.\texttt{super} = a'.A\ \wedge\ \neg\texttt{nested}(a'.A, L_D.\texttt{self}))\ \vee\ L_D.\texttt{super}\ =\ \texttt{null} \\ \mathrm{K} = sp :: \mathrm{K}' \qquad \forall\ A' \mapsto L'_D\ \in\ L_D.\texttt{NEST},\ \texttt{self}(sp).A' :: \mathrm{K} \vdash \texttt{WF}(L'_D) \\ \forall\ R \mapsto \{(f_i : T_i)*\}\ \in\ L_D.\texttt{TYPES},\ \mathrm{K} \vdash \texttt{WF}(\{(f_i : T_i)*\}) \\ \forall\ R \mapsto \{(f_k = v_k)*\}\ \in\ L_D.\texttt{DEFS},\ \exists\ R \mapsto \{(f_i : T_i)*\}\ \in\ L_D.\texttt{TYPES}\ \wedge\ \forall k,\ \exists i, f_k = f_i\ \wedge\ \mathrm{K}; [] \vdash v_i : T_i \\ \forall\ R \mapsto \overline{C_j\ \{(f_i : T_i)*\}}\ \in\ L_D.\texttt{ADTS},\ \forall j,\ \mathrm{K} \vdash \texttt{WF}(\{(f_i : T_i)*\}) \\ \forall\ m \mapsto (T \to T',\ \lambda(x : T).e)\ \in\ L_D.\texttt{FUNS},\ \mathrm{K} \vdash \texttt{WF}(T \to T')\ \wedge\ \mathrm{K}; [] \vdash \lambda(x : T).e : T \to T' \\ \forall\ (c \mapsto (\langle a''.R \rangle, T \to \{(C_j : T_j \to T')*\}, \lambda(x : T).e))\ \in\ L_D.\texttt{CASES}, \\ \mathrm{K} \vdash \texttt{WF}(a'')\ \wedge\ a'' \rightsquigarrow L'_D\ \wedge\ R\ \mapsto\ \overline{C_i\ T_i}\ \in\ L'_D.\texttt{ADTS}\ \wedge\ \forall j, \exists i, (C_j = C_i \wedge T_j = T_i)\ \wedge \\ \mathrm{K} \vdash \texttt{WF}(T \to \{(C_j : T_j \to T')*\})\ \wedge\ \mathrm{K}; [] \vdash \lambda(x : T).e : T \to \{(C_j : T_j \to T')*\}\end{array}}{\mathrm{K} \vdash \texttt{WF}(L_D)}\ \text{(WF-Linkage-D)}$$

Figure A.2: Linkage well-formedness for dynamic linkages L_D.

$$\frac{\begin{array}{c}\mathrm{p}\ =\ \text{famdef}*\ e \qquad L_S = \{\texttt{self} = \texttt{prog},\ \texttt{super} = \texttt{null},\ \texttt{NEST} = \{A \mapsto L'_S : \mathcal{P}(A, L'_S)\}\} \\ \mathcal{P}(A, L'_S) = \textbf{Family}\ A\ (\textbf{extends}\ a.A')?\ \ \{...\}\ \in\ \text{famdef}*\ \wedge\ L'_S = \texttt{parse}_S(\texttt{prog}, \textbf{Family}\ A\ (\textbf{extends}\ a.A')?\ \ \{...\})\end{array}}{\texttt{parse}_S(\mathrm{p})\ =\ L_S}\ \text{(Parse-Prog-S)}$$

$$\frac{\begin{array}{c}L_S = \{\texttt{self} = \texttt{self}(sp.A),\ \texttt{super} = a.A',\ \texttt{NEST},\ \texttt{TYPES},\ \texttt{DEFS},\ \texttt{ADTS},\ \texttt{FUNS},\ \texttt{CASES}\} \\ \texttt{NEST} = \{A' \mapsto L'_S : \mathcal{P}(A', L'_S)\} \\ \mathcal{P}(A', L'_S) = \textbf{Family}\ A'\ (\textbf{extends}\ a'.A'')?\ \ \{...\}\ \in\ \text{famdef}*\ \wedge\ L'_S = \texttt{parse}_S(\texttt{self}(sp.A), \textbf{Family}\ A'\ (\textbf{extends}\ a'.A'')?\ \ \{...\}) \\ \texttt{TYPES} = \{R \mapsto \{(f_i : T_i)*\} : \textbf{type}\ R = \{(f_i : T_i)*\}\ \in\ \text{typedef}*\}\ \cup \\ \qquad \{R \mapsto \{(f_i : T_i)*\} : \textbf{type}\ R\ +\!= \{(f_i : T_i = v_i)*\}\ \in\ \text{typedef}*\} \\ \texttt{DEFS} = \{R \mapsto (f_i)* : \textbf{type}\ R\ +\!= \{(f_i : T_i = v_i)*\}\ \in\ \text{typedef}*\} \\ \texttt{ADTS} = \{R \mapsto \overline{C_j\ \{(f_i : T_i)*\}} : \textbf{type}\ R\ (+)?= \overline{C_j\ \{(f_i : T_i)*\}}\ \in\ \text{adtdef}*\} \\ \texttt{FUNS} = \{m \mapsto (T \to T') : \textbf{val}\ m : T \to T' = \lambda(x : T).e\ \in\ \text{fundef}*\} \\ \texttt{CASES} = \{c \mapsto (\langle a''.R \rangle, T \to T') : \textbf{cases}\ c\ \langle a''.R \rangle : T \to T'\ (+)?= \lambda(x : T).e\ \in\ \text{casesdef}*\}\end{array}}{\texttt{parse}_S(sp,\ \textbf{Family}\ A\ (\textbf{extends}\ a.A')?\ \ \{\text{famdef}*\ \ \text{typdef}*\ \ \text{adtdef}*\ \ \text{fundef}*\ \ \text{casesdef}*\}) = L_S}\ \text{(Parse-Fam-S)}$$

$$\frac{\begin{array}{c}\mathrm{p}\ =\ \text{famdef}*\ e \qquad L_D = \{\texttt{self} = \texttt{prog},\ \texttt{super} = \texttt{null},\ \texttt{NEST} = \{A \mapsto L'_D : \mathcal{P}(A, L'_D)\}\} \\ \mathcal{P}(A, L'_D) = \textbf{Family}\ A\ (\textbf{extends}\ a.A')?\ \ \{...\}\ \in\ \text{famdef}*\ \wedge\ L'_D = \texttt{parse}_D(\texttt{prog}, \textbf{Family}\ A\ (\textbf{extends}\ a.A')?\ \ \{...\})\end{array}}{\texttt{parse}_D(\mathrm{p})\ =\ L_D}\ \text{(Parse-Prog-D)}$$

$$\frac{\begin{array}{c}L_D = \{\texttt{self} = \texttt{self}(sp.A),\ \texttt{super} = a.A',\ \texttt{NEST},\ \texttt{TYPES},\ \texttt{DEFS},\ \texttt{ADTS},\ \texttt{FUNS},\ \texttt{CASES}\} \\ \texttt{NEST} = \{A' \mapsto L'_D : \mathcal{P}(A', L'_D)\} \\ \mathcal{P}(A', L'_D) = \textbf{Family}\ A'\ (\textbf{extends}\ a'.A'')?\ \ \{...\}\ \in\ \text{famdef}*\ \wedge\ L'_D = \texttt{parse}_D(\texttt{self}(sp.A), \textbf{Family}\ A'\ (\textbf{extends}\ a'.A'')?\ \ \{...\}) \\ \texttt{TYPES} = \{R \mapsto \{(f_i : T_i)*\} : \textbf{type}\ R = \{(f_i : T_i)*\}\ \in\ \text{typedef}*\}\ \cup \\ \qquad \{R \mapsto \{(f_i : T_i)*\} : \textbf{type}\ R\ +\!= \{(f_i : T_i = v_i)*\}\ \in\ \text{typedef}*\} \\ \texttt{DEFS} = \{R \mapsto \{(f_i = v_i)*\} : \textbf{type}\ R\ +\!= \{(f_i : T_i = v_i)*\}\ \in\ \text{typedef}*\} \\ \texttt{ADTS} = \{R \mapsto \overline{C_j\ \{(f_i : T_i)*\}} : \textbf{type}\ R\ (+)?= \overline{C_j\ \{(f_i : T_i)*\}}\ \in\ \text{adtdef}*\} \\ \texttt{FUNS} = \{m \mapsto (T \to T', \lambda(x : T).e) : \textbf{val}\ m : T \to T' = \lambda(x : T).e\ \in\ \text{fundef}*\} \\ \texttt{CASES} = \{c \mapsto (\langle a''.R \rangle, T \to T', \lambda(x : T).e) : \textbf{cases}\ c\ \langle a''.R \rangle : T \to T'\ (+)?= \lambda(x : T).e\ \in\ \text{casesdef}*\}\end{array}}{\texttt{parse}_D(sp,\ \textbf{Family}\ A\ (\textbf{extends}\ a.A')?\ \ \{\text{famdef}*\ \ \text{typdef}*\ \ \text{adtdef}*\ \ \text{fundef}*\ \ \text{casesdef}*\}) = L_D}\ \text{(Parse-Fam-D)}$$

Figure A.3: Parsing of programs into static linkages L_S and dynamic linkages L_D.

$$\frac{\begin{array}{c}(L_S.\texttt{super} = \texttt{null}) \vee \\ (sp' = \texttt{relativize}(L_S.\texttt{super}) \ \wedge \ sp \neq sp' \ \wedge \ \mathrm{K} = \texttt{allPaths}(L_S) \ \wedge \\ \mathrm{K} \vdash \texttt{WF}(sp') \ \wedge \ sp' \rightsquigarrow L'_S \ \wedge \ sp \notin \texttt{ancestors}(sp'))\end{array}}{sp \notin \texttt{ancestors}(L_S)}$$

$$\frac{\texttt{relativize}(a.A) \notin \texttt{ancestors}(L_S)}{a.A \notin \texttt{ancestors}(L_S)}$$

$$\frac{\texttt{pathPrefix}(sp, \texttt{relativize}(a))}{\texttt{nested}(a, sp)} \qquad \frac{\texttt{pathPrefix}(\texttt{relativize}(a'.A), \texttt{relativize}(a))}{\texttt{nested}(a, a'.A)}$$

$$\frac{}{\texttt{pathPrefix}(\texttt{prog}, sp')} \qquad \frac{\texttt{pathPrefix}(\texttt{self}(sp.A), sp')}{\texttt{pathPrefix}(\texttt{self}(sp.A), \texttt{self}(sp'.A'))}$$

$$\texttt{relativize}(sp) = sp \qquad \texttt{relativize}(a.A) = \texttt{self}(\texttt{relativize}(a).A)$$

Figure A.4: Definitions for `ancestors`, `nested`, and their helpers.

A.2 Extension: Modular Semantics

$\boxed{\mathrm{K} \vdash T \ <: \ T'}$

$$\frac{\mathrm{K} \vdash \texttt{WF}(a) \qquad \mathcal{M} = \texttt{getFragment}(\mathscr{L}, a) \qquad \mathcal{M}; [a] \vdash a \rightsquigarrow L_S \qquad R \mapsto \{(f_i : T_i)*\} \ \in \ L_S.\text{TYPES} \qquad \mathrm{K} \vdash \{(f_i : T_i)*\} \ <: \ T'}{\mathrm{K} \vdash a.R \ <: \ T'} \ (\textsc{Sub-Fam})$$

$$\frac{\forall j, \ \exists T, \ \mathrm{K} \vdash T \ <: \ T_j \ \wedge \ (f_j : T) \ \in \ (f_i : T_i)*}{\mathrm{K} \vdash \{(f_i : T_i)*\} \ <: \ \{(f_j : T_j)*\}} \ (\textsc{Sub-Rec})$$

$$\frac{}{\mathrm{K} \vdash T \ <: \ T} \ (\textsc{Sub-Refl}) \qquad \frac{\mathrm{K} \vdash T_2 \ <: \ T_1 \qquad \mathrm{K} \vdash T'_1 \ <: \ T'_2}{\mathrm{K} \vdash T_1 \to T'_1 \ <: \ T_2 \to T'_2} \ (\textsc{Sub-Fun})$$

Figure A.5: Modular subtyping relation.

$\boxed{\mathrm{K};\Gamma \vdash e : T}$

$$\frac{}{\mathrm{K};\Gamma \vdash n : \mathtt{N}}\ \text{(T-Num)} \qquad \frac{}{\mathrm{K};\Gamma \vdash b : \mathtt{B}}\ \text{(T-Bool)} \qquad \frac{x : T \in \Gamma}{\mathrm{K};\Gamma \vdash x : T}\ \text{(T-Var)}$$

$$\frac{\mathrm{K} \vdash \mathtt{WF}(T) \qquad \mathrm{K};(x : T, \Gamma) \vdash e : T'}{\mathrm{K};\Gamma \vdash \lambda(x : T).e : T \to T'}\ \text{(T-Lam)} \qquad \frac{\mathrm{K};\Gamma \vdash e : T \qquad \mathrm{K};\Gamma \vdash g : T \to T'}{\mathrm{K};\Gamma \vdash g\ e : T'}\ \text{(T-App)}$$

$$\frac{\forall i,\ \mathrm{K};\Gamma \vdash e_i : T_i}{\mathrm{K};\Gamma \vdash \{(f_i = e_i)*\} : \{(f_i : T_i)*\}}\ \text{(T-Rec)} \qquad \frac{\mathrm{K};\Gamma \vdash e : \{(f_i : T_i)*\} \qquad f : T \in (f_i : T_i)*}{\mathrm{K};\Gamma \vdash e.f : T}\ \text{(T-Proj)}$$

$$\frac{\mathrm{K};\Gamma \vdash e : T' \qquad \mathrm{K} \vdash T' <: T}{\mathrm{K};\Gamma \vdash e : T}\ \text{(T-Subs)} \qquad \frac{\mathrm{K};\Gamma \vdash e : \mathtt{B} \qquad \mathrm{K};\Gamma \vdash g : T \qquad \mathrm{K};\Gamma \vdash g' : T}{\mathrm{K};\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ g\ \mathtt{else}\ g' : T}\ \text{(T-If)}$$

$$\frac{\mathrm{K} \vdash \mathtt{WF}(a) \qquad \mathcal{M} = \mathtt{getFragment}(\mathscr{L}, a) \qquad \mathcal{M};[a] \vdash a \rightsquigarrow L_S \qquad m \mapsto (T \to T') \in L_S.\mathrm{FUNS}}{\mathrm{K};\Gamma \vdash a.m : T \to T'}\ \text{(T-FamFun)}$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash \mathtt{WF}(a) \qquad \mathcal{M} = \mathtt{getFragment}(\mathscr{L}, a) \qquad \mathcal{M};[a] \vdash a \rightsquigarrow L_S \qquad R \mapsto \{(f_j : T_j)*\} \in L_S.\mathrm{TYPES} \\ R \mapsto (f_k)* \in L_S.\mathrm{DEFS} \qquad \forall i,\ \exists j,\ f_i = f_j \wedge \mathrm{K};\Gamma \vdash e_i : T_j \qquad \forall j,\ f_j \in (f_i)* \vee f_j \in (f_k)*\end{array}}{\mathrm{K};\Gamma \vdash a.R(\{(f_i = e_i)*\}) : a.R}\ \text{(T-Constr)}$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash \mathtt{WF}(a) \qquad \mathcal{M} = \mathtt{getFragment}(\mathscr{L}, a) \qquad \mathcal{M};[a] \vdash a \rightsquigarrow L_S \\ R \mapsto \overline{C_j\ \{(f_i : T_i)*\}} \in L_S.\mathrm{ADTS} \qquad C\ \{(f_k : T_k)*\} \in \overline{C_j\ \{(f_i : T_i)*\}} \qquad \forall k,\ \mathrm{K};\Gamma \vdash e_k : T_k\end{array}}{\mathrm{K};\Gamma \vdash a.R(C\ \{(f_k = e_k)*\}) : a.R}\ \text{(T-ADT)}$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash \mathtt{WF}(a) \qquad \mathcal{M} = \mathtt{getFragment}(\mathscr{L}, a) \qquad \mathcal{M};[a] \vdash a \rightsquigarrow L_S \\ c \mapsto (\langle a'.R \rangle, \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}) \in L_S.\mathrm{CASES}\end{array}}{\mathrm{K};\Gamma \vdash a.c : \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}}\ \text{(T-Cases)}$$

$$\frac{\begin{array}{c}\mathrm{K};\Gamma \vdash e : a'.R \qquad \mathcal{M} = \mathtt{getFragment}(\mathscr{L}, a') \qquad \mathcal{M};[a'] \vdash a' \rightsquigarrow L_S \qquad R \mapsto \overline{C_j\ \{(f_i : T_i)*\}} \in L_S.\mathrm{ADTS} \\ \mathrm{K};\Gamma \vdash a.c : \{(f_{arg} : T_{arg})*\} \to \{(C_j : \{(f_i : T_i)*\} \to T)*\} \qquad \mathrm{K};\Gamma \vdash \{(f_{arg} = e_{arg})*\} : \{(f_{arg} : T_{arg})*\}\end{array}}{\mathrm{K};\Gamma \vdash \mathtt{match}\ e\ \mathtt{with}\ a.c\ \{(f_{arg} = e_{arg})*\} : T}\ \text{(T-Match)}$$

Figure A.6: Modular type checking of expressions.

$\boxed{\mathrm{K} \vdash \texttt{WF}(a)}$

$$\frac{\mathrm{K} \vdash \texttt{WF}(a) \qquad \mathcal{M} = \texttt{getFragment}(\mathscr{L}, a.A) \qquad \mathcal{M}; [a] \vdash a \rightsquigarrow L_S \qquad A \in L_S.\texttt{NEST}}{\mathrm{K} \vdash \texttt{WF}(a.A)} \text{ (WF-Path-Abs)}$$

$$\frac{sp \in K}{\mathrm{K} \vdash \texttt{WF}(sp)} \text{ (WF-Path-Self)}$$

$\boxed{\mathrm{K} \vdash \texttt{WF}(T)}$

$$\frac{}{\mathrm{K} \vdash \texttt{WF}(\texttt{N})} \text{ (WF-Num)} \qquad \frac{}{\mathrm{K} \vdash \texttt{WF}(\texttt{B})} \text{ (WF-Bool)} \qquad \frac{\mathrm{K} \vdash \texttt{WF}(T) \qquad \mathrm{K} \vdash \texttt{WF}(T')}{\mathrm{K} \vdash \texttt{WF}(T \to T')} \text{ (WF-Arrow)}$$

$$\frac{\mathrm{K} \vdash \texttt{WF}(a) \qquad \mathcal{M} = \texttt{getFragment}(\mathscr{L}, a) \qquad \mathcal{M}; [a] \vdash a \rightsquigarrow L_S \qquad R \mapsto \{(f_i : T_i)*\} \in L_S.\text{TYPES} \ \vee \ R \mapsto \overline{C_j \ \{(f_i : T_i)*\}} \in L_S.\text{ADTS}}{\mathrm{K} \vdash \texttt{WF}(a.R)} \text{ (WF-Named)}$$

$$\frac{\forall i, \ \mathrm{K} \vdash \texttt{WF}(T_i) \qquad \forall i \ j, \ i \neq j \implies f_i \neq f_j}{\mathrm{K} \vdash \texttt{WF}(\{(f_i : T_i)*\})} \text{ (WF-Record)}$$

Figure A.7: Modular well-formedness of family paths a (top) and types T (bottom).

$\boxed{\mathrm{K};\Gamma \vdash p : T}$

$$\frac{p = \mathrm{famdef}_i{*}\ e \qquad \mathrm{K} = [\texttt{prog}] \qquad \forall i,\ \mathrm{K} \vdash \texttt{WF}(\mathrm{famdef}_i) \qquad \mathrm{K}, [] \vdash e : T}{[];[] \vdash p : T} \text{ (T-Prog)}$$

$\boxed{\mathrm{K} \vdash \texttt{WF}(\mathrm{def})}$

$$\frac{\begin{array}{c}\mathrm{famdef} = \textbf{Family}\ A\ (\textbf{extends}\ a.A')?\ \{\mathrm{famdef}_n{*}\ \ \mathrm{typdef}_q{*}\ \ \mathrm{adtdef}_u{*}\ \ \mathrm{fundef}_w{*}\ \ \mathrm{casesdef}_z{*}\} \\ \mathcal{M} = \texttt{getFragment}(\mathscr{L}, \texttt{self}(sp.A)) \qquad \Delta = [\texttt{self}(sp.A)] \qquad \mathrm{K}' = \texttt{self}(sp.A) :: sp :: \mathrm{K} \\ \mathcal{M}; \Delta \vdash \texttt{self}(sp.A) \rightsquigarrow L_S \qquad \neg\texttt{nested}(a.A', \texttt{self}(sp.A)) \qquad \mathrm{sp} :: \mathrm{K} \vdash \texttt{WF}(a.A') \qquad \mathrm{K}' \vdash \texttt{EC}(L_S) \\ \forall n,\ \mathrm{K}' \vdash \texttt{WF}(\mathrm{famdef}_n) \qquad \forall q,\ \mathrm{K}' \vdash \texttt{WF}(\mathrm{typdef}_q) \qquad \forall u,\ \mathrm{K}' \vdash \texttt{WF}(\mathrm{adtdef}_u) \\ \forall w,\ \mathrm{K}' \vdash \texttt{WF}(\mathrm{fundef}_w) \qquad \forall z,\ \mathrm{K}' \vdash \texttt{WF}(\mathrm{casesdef}_z)\end{array}}{sp :: \mathrm{K} \vdash \texttt{WF}(\mathrm{famdef})} \text{ (WF-FamDef)}$$

$$\frac{\mathrm{K} \vdash \texttt{WF}(\{(f_i : T_i){*}\})}{\mathrm{K} \vdash \texttt{WF}(\textbf{type}\ R\ =\ \{(f_i : T_i){*}\})} \text{ (WF-TypDef)} \qquad \frac{\mathrm{K} \vdash \texttt{WF}(\{(f_i : T_i){*}\}) \qquad \forall i,\ \mathrm{K};[] \vdash v_i : T_i}{\mathrm{K} \vdash \texttt{WF}(\textbf{type}\ R\ +=\ \{(f_i : T_i = v_i){*}\})} \text{ (WF-TypDef-Ext)}$$

$$\frac{\forall j,\ \mathrm{K} \vdash \texttt{WF}(\{(f_i : T_i){*}\})}{\mathrm{K} \vdash \texttt{WF}(\textbf{type}\ R\ (+)?=\ \overline{C_j\ \{(f_i : T_i){*}\}})} \text{ (WF-AdtDef)} \qquad \frac{\mathrm{K} \vdash \texttt{WF}(T \to T') \qquad \mathrm{K};[] \vdash \lambda(x : T).e : T \to T'}{\mathrm{K} \vdash \texttt{WF}(\textbf{val}\ m\ :\ T \to T'\ =\ \lambda(x : T).e)} \text{ (WF-FunDef)}$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash \texttt{WF}(a) \qquad \mathcal{M} = \texttt{getFragment}(\mathscr{L}, a) \qquad \mathcal{M};[a] \vdash a \rightsquigarrow L_S \\ R\ \mapsto\ \overline{C_i\ T_i}\ \in\ L_S.\texttt{ADTS} \qquad \forall j,\ \exists i,\ (C_j = C_i \wedge T_j = T_i) \qquad \mathrm{K} \vdash \texttt{WF}(T \to \{(C_j : T_j \to T'){*}\}) \\ \mathrm{K};[] \vdash \lambda(x : T).\{(C_j = \lambda(y_j : T_j).e_j){*}\} : T \to \{(C_j : T_j \to T'){*}\}\end{array}}{\mathrm{K} \vdash \texttt{WF}(\textbf{cases}\ c\ \langle a.R\rangle\ :\ T \to \{(C_j : T_j \to T'){*}\}\ \ (+)?=\ \ \lambda(x : T).\{(C_j = \lambda(y_j : T_j).e_j){*}\})} \text{ (WF-CasesDef)}$$

$\boxed{\mathrm{K} \vdash \texttt{EC}(L_S)}$

$$\frac{\begin{array}{c}\forall\ (c \mapsto (\langle a.R\rangle, T \to \{(C_j : T_j \to T'){*}\}))\ \in\ L_S.\texttt{CASES}, \\ \mathrm{K} \vdash \texttt{WF}(a)\ \wedge\ \mathcal{M} = \texttt{getFragment}(\mathscr{L}, a)\ \wedge\ \mathcal{M};[a] \vdash a \rightsquigarrow L'_S\ \wedge\ R\ \mapsto\ \overline{C_j\ T_j}\ \in\ L'_S.\texttt{ADTS} \\ \forall\ A\ \in\ L_S.\texttt{NEST},\ \ sp = L_S.\texttt{self}\ \wedge\ \mathrm{K}' = \texttt{self}(sp.A) :: \mathrm{K}\ \wedge\ \mathcal{M} = \texttt{getFragment}(\mathscr{L}, \texttt{self}(sp.A))\ \wedge \\ \mathcal{M};[\texttt{self}(sp.A)] \vdash \texttt{self}(sp.A) \rightsquigarrow L''_S\ \wedge\ \mathrm{K}' \vdash \texttt{EC}(L''_S)\end{array}}{\mathrm{K} \vdash \texttt{EC}(L_S)} \text{ (EC-Nest)}$$

Figure A.8: Modular type checking of program fragments p (top), well-formedness of definitions (WF, middle), and exhaustivity checking (EC, bottom)

B

Persimmon Proofs

Additional proofs not included in this appendix are available at https://dl.acm.org/doi/10.1145/3649836#supplementary-materials.

B.1 Proof of Progress

Theorem 3 (Progress)**.** *For any main expression e in program p,*

$$([]; [] \vdash p : T_p \ \wedge \ [\texttt{prog}]; [] \vdash e : T') \implies (\mathit{value}(e) \ \vee \ \exists e', [\texttt{prog}] \vdash e \longrightarrow e').$$

Proof. Proof by induction on the typing derivation $[\texttt{prog}]; [] \vdash e : T'$.

The following are already included in the premise where necessary:

- premises from induction on the typing derivation (named H1, H2, etc)
- induction hypotheses (named IND1, IND2, etc)

The unnamed premises come from the statement of the proof.

Case 1: T-Num. The expression is a value.

Case 2: T-Bool. The expression is a value.

Case 3: T-Var.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash x : T'$

H1: $x : T' \in []$

Contradiction in H1

Case 4: T-Lam. The expression is a value.

Case 5: T-App.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash g\ e : T'$

H1: $[\texttt{prog}]; [] \vdash e : T$

H2: $[\texttt{prog}]; [] \vdash g : T \to T'$

IND1: $\text{value}(e) \vee \exists e', [\texttt{prog}] \vdash e \longrightarrow e'$

IND2: $\text{value}(g) \vee \exists g', [\texttt{prog}] \vdash g \longrightarrow g'$

Subcase 5.1: $\exists g', [\texttt{prog}] \vdash g \longrightarrow g'$.

Take $e_1 = g'\ e$. By R-App,

$$[\texttt{prog}] \vdash g\ e \longrightarrow g'\ e.$$

Thus, $\exists e_1, [\texttt{prog}] \vdash g\ e \longrightarrow e_1$.

Subcase 5.2: value(g) and $\exists e', [\texttt{prog}] \vdash e \longrightarrow e'$.

Take $e_1 = g\ e'$. By R-LamArg,

$$[\texttt{prog}] \vdash g\ e \longrightarrow g\ e'.$$

Thus, $\exists e_1, [\texttt{prog}] \vdash g\ e \longrightarrow e_1$.

Subcase 5.3: value(g) and value(e).

Since $[\texttt{prog}]; [] \vdash g : T \to T'$ (H2) and value(g), by theorem Canonical-Fun there exist x, S, and e_0 such that $g = \lambda(x : S).e_0$ and $[\texttt{prog}] \vdash T <: S$. Write $e = v$ (because value(e)). Take

$e_1 = [x := v]\ e_0$. By R-LamApply,

$$[\texttt{prog}] \vdash \lambda(x : S).e_0\ v \longrightarrow [x := v]\ e_0.$$

Thus, $\exists e_1, [\texttt{prog}] \vdash g\ e \longrightarrow e_1$.

Case 6: T-Rec.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash \{(f_i = e_i)*\} : \{(f_i : T_i)*\}$

$\forall i,$

Hi: $[\texttt{prog}]; [] \vdash e_i : T_i$

$\forall i,$

INDi: $\text{value}(e_i) \vee \exists e_i', [\texttt{prog}] \vdash e_i \longrightarrow e_i'$

By IND_i, each e_i is either a value or it can reduce. If all e_i's are values v_i, then the record $\{(f_i = v_i)*\}$ is a value. If for some i, e_i is the first non-value, then the record can reduce.

Consider e_i in order from left to right. All expressions in the record at indices before e_i are values.

Subcase 6.1: For the first non-value field, $\exists e_i', [\texttt{prog}] \vdash e_i \longrightarrow e_i'$.

Take

$$e_1 = \{f_0 = v_0, \ldots, f_i = e_i', \ldots\}.$$

By R-Rec,

$$[\texttt{prog}] \vdash \{f_0 = v_0, \ldots, f_i = e_i, \ldots\} \longrightarrow \{f_0 = v_0, \ldots, f_i = e_i', \ldots\}.$$

Thus, $\exists e_1, [\texttt{prog}] \vdash \{(f_i = e_i)*\} \longrightarrow e_1$.

Subcase 6.2: All fields are values.

Then $\{(f_i = v_i)*\}$ is a value, so the goal holds.

Case 7: T-FamFun.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash a.m : T \to T'$

H1: $[\texttt{prog}] \vdash \text{WF}(a)$

H2: $a \rightsquigarrow L_S$

H3: $m \mapsto (T \to T') \in L_S.\text{FUNS}$

Since the program is well-typed, all the family definitions in the program are well-formed. Since $m \mapsto (T \to T') \in L_S.\text{FUNS}$ (H3), there must exist a definition $\lambda(x : T).e_0$ for m in the program by WF-FunDef. Thus, exists $\lambda(x : T).e_0$,

$$[\texttt{prog}] \vdash a.m \longrightarrow \lambda(x : T).e_0$$

by R-FamFun.

Case 8: T-Cases.

$[]; [] \vdash p : T_p$

$[] \vdash a.c : \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}$

H1: $[\texttt{prog}] \vdash \mathrm{WF}(a)$

H2: $a \rightsquigarrow L_S$

H3: $c \mapsto (\langle a'.R \rangle, \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}) \in L_S.\mathrm{CASES}$

Since the program is well-typed, all the family definitions in the program are well-formed. Since $c \mapsto (\langle a'.R \rangle, \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}) \in L_S.\mathrm{CASES}$ (H3), there must exist a definition $\lambda(x : T).e_0$ for c in the program by WF-CasesDef. Thus, exists $\lambda(x : T).e_0$,

$$[\texttt{prog}] \vdash a.c \longrightarrow \lambda(x : T).e_0$$

by R-Cases.

Case 9: T-Constr.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash a.R(\{(f_i = e_i)*\}) : a.R$

$\forall i,$

INDi: $\mathrm{value}(e_i) \vee \exists e'_i, [\texttt{prog}] \vdash e_i \longrightarrow e'_i$

Subcase 9.1: There is a first field e_i such that $\exists e'_i, [\texttt{prog}] \vdash e_i \longrightarrow e'_i$.

Take

$$e_r = \{f_0 = v_0, \ldots, f_{i-1} = v_{i-1}, f_i = e'_i, \ldots\}.$$

By R-Rec,

$$[\texttt{prog}] \vdash \{f_0 = v_0, \ldots, f_{i-1} = v_{i-1}, f_i = e_i, \ldots\} \longrightarrow e_r.$$

Take $e_1 = a.R(e_r)$. By R-Instance,

$$[\texttt{prog}] \vdash a.R(\{f_0 = v_0, \ldots, f_{i-1} = v_{i-1}, f_i = e_i, \ldots\}) \longrightarrow a.R(e_r).$$

Thus $\exists e_1, [\texttt{prog}] \vdash a.R(\{(f_i = e_i)*\}) \longrightarrow e_1$.

Subcase 9.2: All e_i are values.

Then $a.R(\{(f_i = v_i)*\})$ is a value, so the goal holds.

Case 10: T-ADT.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash a.R(C\{(f_k = e_k)*\}) : a.R$

$\forall k,$

INDk: $\text{value}(e_k) \vee \exists e'_k, [\texttt{prog}] \vdash e_k \longrightarrow e'_k$

Subcase 10.1: There is a first field e_k such that $\exists e'_k, [\texttt{prog}] \vdash e_k \longrightarrow e'_k$.

Take

$$e_r = \{f_0 = v_0, \ldots, f_{k-1} = v_{k-1}, f_k = e'_k, \ldots\}.$$

By R-Rec,

$$[\texttt{prog}] \vdash \{f_0 = v_0, \ldots, f_{k-1} = v_{k-1}, f_k = e_k, \ldots\} \longrightarrow e_r.$$

Take $e_1 = a.R(C\ e_r)$. By R-ADT,

$$[\texttt{prog}] \vdash a.R(C\ \{f_0 = v_0, \ldots, f_{k-1} = v_{k-1}, f_k = e_k, \ldots\}) \longrightarrow a.R(C\ e_r).$$

Thus, $\exists e_1, [\texttt{prog}] \vdash a.R(C\ \{(f_k = e_k)*\}) \longrightarrow e_1$.

Subcase 10.2: All e_k are values.

Then $a.R(C\ \{(f_k = v_k)*\})$ is a value, so the goal holds.

Case 11: T-Subs.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash e : T'$

IND1: $\text{value}(e) \vee \exists e', [\texttt{prog}] \vdash e \longrightarrow e'$

By IND1, $\text{value}(e) \vee \exists e', [\texttt{prog}] \vdash e \longrightarrow e'$

Case 12: T-Proj.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash e.f : T'$

H1: $[\texttt{prog}]; [] \vdash e : \{(f_i : T_i)*\}$

H2: $(f : T) \in (f_i : T_i)*$

IND: value$(e) \vee \exists e', [\texttt{prog}] \vdash e \longrightarrow e'$

Subcase 12.1: $\exists e', [\texttt{prog}] \vdash e \longrightarrow e'$.

Take $e_1 = e'.f$. By R-Proj,

$$[\texttt{prog}] \vdash e.f \longrightarrow e'.f.$$

Thus, $\exists e_1, [\texttt{prog}] \vdash e.f \longrightarrow e_1$.

Subcase 12.2: value(e).

By theorem Canonical-Rec (using H1), there are two forms for e.

Subcase 12.2.1: $e = \{(f_j = v_j)*\}$.

From H2, the field f appears in the required record type, so there is a field $(f = v)$ in $(f_j = v_j)*$.

By R-RecProj,

$$[\texttt{prog}] \vdash \{(f_j = v_j)*\}.f \longrightarrow v.$$

Thus, $\exists v, [\texttt{prog}] \vdash e.f \longrightarrow v$.

Subcase 12.2.2: $e = a.R(\{(f_j = v_j)*\})$.

By theorem Canonical-Rec, we also have $[\texttt{prog}] \vdash a.R <: \{(f_i : T_i)*\}$. By inversion of typing for

$a.R(\{(f_j = v_j)*\})$ (rule T-CONSTR), there exist declared fields $(f_k : T_k)*$ and defaults $(f_d)*$ such that:

$$R \mapsto \{(f_k : T_k)*\} \in L_S.\text{TYPES} \quad \text{and}$$

$$R \mapsto (f_d)* \in L_S.\text{DEFS} \quad \text{and}$$

$$\forall k,\ f_k \in (f_j)* \vee f_k \in (f_d)*.$$

Since $[\texttt{prog}] \vdash a.R <: \{(f_i : T_i)*\}$ and $R \mapsto \{(f_k : T_k)*\} \in L_S.\text{TYPES}$, we get

$$[\texttt{prog}] \vdash \{(f_k : T_k)*\} <: \{(f_i : T_i)*\}$$

by SUB-FAM. Since $[\texttt{prog}] \vdash \{(f_k : T_k)*\} <: \{(f_i : T_i)*\}$, all required fields f_i appear among $(f_k)*$ by SUB-REC; in particular, $f \in (f_k)*$. Thus, either:

Subcase 12.2.2.a: $f \in (f_j)*$.

Then for some v, $(f = v)$ is in $(f_j = v_j)*$, and by R-INSTPROJ,

$$[\texttt{prog}] \vdash a.R(\{(f_j = v_j)*\}) \longrightarrow v.$$

Subcase 12.2.2.b: $f \notin (f_j)*$.

Then $f \in (f_d)*$, so the default value v for f is selected from the dynamic linkage; by R-INSTPROJ,

$$[\texttt{prog}] \vdash a.R(\{(f_j = v_j)*\}) \longrightarrow v.$$

In both subcases, $\exists v, [\texttt{prog}] \vdash e.f \longrightarrow v$.

Case 13: T-Match.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash \text{match } e \text{ with } a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\} : T'$

H1: $a' \rightsquigarrow L_S$

H2: $[\texttt{prog}]; [] \vdash e : a'.R$

H3: $R \mapsto \overline{C_j\{(f_i : T_i)*\}} \in L_S.\text{ADTS}$

H4: $[\texttt{prog}]; [] \vdash a.c : \{(f_{\text{arg}} : T_{\text{arg}})*\} \to \{(C_j : \{(f_i : T_i)*\} \to T')*\}$

H5: $[\texttt{prog}]; [] \vdash \{(f_{\text{arg}} = e_{\text{arg}})*\} : \{(f_{\text{arg}} : T_{\text{arg}})*\}$

IND2: $\text{value}(e) \vee \exists e', [\texttt{prog}] \vdash e \longrightarrow e'$

IND5: $\text{value}(\{(f_{\text{arg}} = e_{\text{arg}})*\}) \vee \exists e'', [\texttt{prog}] \vdash \{(f_{\text{arg}} = e_{\text{arg}})*\} \longrightarrow e''$

Subcase 13.1: $\exists e', [\texttt{prog}] \vdash e \longrightarrow e'$.

Take

$e_1 = \text{match } e' \text{ with } a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\}.$

By R-MatchExp,

$[\texttt{prog}] \vdash \text{match } e \text{ with } a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\} \longrightarrow e_1.$

Thus, $\exists e_1, [\texttt{prog}] \vdash \text{match } e \text{ with } a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\} \longrightarrow e_1$.

Subcase 13.2: $\text{value}(e)$ and $\exists e'', [\texttt{prog}] \vdash \{(f_{\text{arg}} = e_{\text{arg}})*\} \longrightarrow e''$.

Write $e = v$ (since $\text{value}(e)$) and take

$e_1 = \text{match } v \text{ with } a.c\ e''.$

By R-MatchCases,

$$[\texttt{prog}] \vdash \text{match } v \text{ with } a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\} \longrightarrow e_1.$$

Thus, $\exists e_1, [\texttt{prog}] \vdash \text{match } e \text{ with } a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\} \longrightarrow e_1$.

Subcase 13.3: value(e) and $\{(f_{\text{arg}} = v_{\text{arg}})*\}$ is a value.

From H2 and theorem Canonical-Fam, e is either a named-record value or an ADT value. Since H3 gives an ADT definition for R in L_S.ADTS, e has the form $a'.R(C\ \{(f_k = v_k)*\})$. Take

$$e_1 = (a.c\ \{(f_{\text{arg}} = v_{\text{arg}})*\}).C\ \{(f_k = v_k)*\}.$$

By R-MatchFinal,

$$[\texttt{prog}] \vdash \text{match } a'.R(C\ \{(f_k = v_k)*\}) \text{ with } a.c\ \{(f_{\text{arg}} = v_{\text{arg}})*\} \longrightarrow e_1.$$

Thus, $\exists e_1, [\texttt{prog}] \vdash \text{match } e \text{ with } a.c\ \{(f_{\text{arg}} = v_{\text{arg}})*\} \longrightarrow e_1$.

Case 14: T-If.

$$[]; [] \vdash p : T_p$$

$$[\texttt{prog}]; [] \vdash \text{if } e \text{ then } g \text{ else } g' : T'$$

IND1: value(e) $\vee\ \exists e', [\texttt{prog}] \vdash e \longrightarrow e'$

Subcase 14.1: $\exists e', [\texttt{prog}] \vdash e \longrightarrow e'$.

Take $e_1 = \text{if } e' \text{ then } g \text{ else } g'$. By R-IfGuard,

$$[\texttt{prog}] \vdash \text{if } e \text{ then } g \text{ else } g' \longrightarrow \text{if } e' \text{ then } g \text{ else } g'.$$

Thus, $\exists e_1, [\texttt{prog}] \vdash \text{if } e \text{ then } g \text{ else } g' \longrightarrow e_1$.

Subcase 14.2: value(e).

By inversion on the derivation $[\texttt{prog}]; [] \vdash \text{if } e \text{ then } g \text{ else } g' : T'$ (rule T-IF), we have $[\texttt{prog}]; [] \vdash e : \text{B}$. By canonical forms for booleans, $e = \text{true}$ or $e = \text{false}$.

Subcase 14.2.a: $e = \text{true}$.

Take $e_1 = g$. By R-IfTrue,

$$[\texttt{prog}] \vdash \text{if true then } g \text{ else } g' \longrightarrow g.$$

Thus, $\exists e_1, [\texttt{prog}] \vdash \text{if } e \text{ then } g \text{ else } g' \longrightarrow e_1$.

Subcase 14.2.b: $e = \text{false}$.

Take $e_1 = g'$. By R-IfFalse,

$$[\texttt{prog}] \vdash \text{if false then } g \text{ else } g' \longrightarrow g'.$$

Thus, $\exists e_1, [\texttt{prog}] \vdash \text{if } e \text{ then } g \text{ else } g' \longrightarrow e_1$.

□

B.2 Proof of Preservation

Theorem 4 (Preservation)**.** *For any main expression e in program p, and any e',*

$$([]; [] \vdash p : T_p \ \wedge \ [\texttt{prog}]; [] \vdash e : T' \ \wedge \ [\texttt{prog}] \vdash e \longrightarrow e') \implies [\texttt{prog}]; [] \vdash e' : T'.$$

Proof. Proof by induction on the typing derivation $[\texttt{prog}]; [] \vdash e : T'$.

The following are already included in the premises where necessary:

- premises from inversion on the typing derivation (named H1, H2, etc)

The unnamed premises come from the theorem statement.

Case 1: T-Num.

$$[]; [] \vdash p : T_p$$

$$[\texttt{prog}]; [] \vdash n : \mathrm{N}$$

$$[\texttt{prog}] \vdash n \longrightarrow e'$$

Contradiction, n is a value and cannot reduce.

Case 2: T-Bool.

$$[]; [] \vdash p : T_p$$

$$[\texttt{prog}]; [] \vdash b : \mathrm{B}$$

$$[\texttt{prog}] \vdash b \longrightarrow e'$$

Contradiction, b is a value and cannot reduce.

Case 3: T-Var.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash x : T'$

$[\texttt{prog}] \vdash x \longrightarrow e'$

H1: $x : T' \in []$

Contradiction in H1.

Case 4: T-Lam.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash \lambda(x : T).e_0 : T \to T'$

$[\texttt{prog}] \vdash \lambda(x : T).e_0 \longrightarrow e'$

Contradiction, λ-abstractions are values and cannot reduce.

Case 5: T-App.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash g\ e : T'$

$[\texttt{prog}] \vdash g\ e \longrightarrow e''$

H1: $[\texttt{prog}]; [] \vdash e : T$

H2: $[\texttt{prog}]; [] \vdash g : T \to T'$

Since $g\ e$ reduces, there are three reduction forms.

Subcase 5.1: $[\texttt{prog}] \vdash g \longrightarrow g'$ and R-App applies.

Then

$$[\texttt{prog}] \vdash g\ e \longrightarrow g'\ e.$$

By induction on H2, $[\texttt{prog}]; [] \vdash g' : T \to T'$. From H1 and T-App, we derive

$$[\texttt{prog}]; [] \vdash g'\ e : T'.$$

Subcase 5.2: $g = v$ is a value, $[\texttt{prog}] \vdash e \longrightarrow e'$, and R-LamArg applies.

Then

$$[\texttt{prog}] \vdash v\ e \longrightarrow v\ e'.$$

By induction on H1, $[\texttt{prog}]; [] \vdash e' : T$. Using H2 (with $g = v$) and T-App, we derive

$$[\texttt{prog}]; [] \vdash v\ e' : T'.$$

Subcase 5.3: value(g) and value(e).

By theorem Canonical-Fun and H2, there exist x, S, and e_0 such that $g = \lambda(x : S).e_0$ and $[\texttt{prog}] \vdash T <: S$. Write $e = v$ since value(e). By R-LamApply,

$$[\texttt{prog}] \vdash (\lambda(x : S).e_0)\ v \longrightarrow [x := v]\ e_0.$$

It remains to show $[\texttt{prog}]; [] \vdash [x := v]\ e_0 : T'$. From inversion on H2 (T-Lam), we have:

$$[\texttt{prog}]; (x : S, []) \vdash e_0 : T'.$$

From H1 we have $[\texttt{prog}]; [] \vdash v : T$, and with $[\texttt{prog}] \vdash T <: S$ we get $[\texttt{prog}]; [] \vdash v : S$ by T-Subs. By theorem Subst-Preserves-Typing, we have:

$$[\texttt{prog}]; [] \vdash [x := v]\ e_0 : T'.$$

Case 6: T-Rec.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash \{(f_i = e_i)*\} : \{(f_i : T_i)*\}$

$[\texttt{prog}] \vdash \{(f_i = e_i)*\} \longrightarrow e'$

$\forall i$, Hi: $[\texttt{prog}]; [] \vdash e_i : T_i$

Since the record reduces, not all fields are already values. So there exists a first reducible field e_i, and by R-Rec we have:

$$[\texttt{prog}] \vdash \{f_0 = v_0, \ldots, f_i = e_i, \ldots\} \longrightarrow \{f_0 = v_0, \ldots, f_i = e_i', \ldots\}.$$

By induction on Hi, $[\mathsf{prog}]; [] \vdash e_i' : T_i$. All other fields keep their original typing from the premises.

Thus, by T-Rec,

$$[\mathsf{prog}]; [] \vdash \{f_0 = v_0, \ldots, f_i = e_i', \ldots\} : \{(f_i : T_i)*\}.$$

Case 7: T-Proj.

$[]; [] \vdash p : T_p$

$[\mathsf{prog}]; [] \vdash e.f : T'$

$[\mathsf{prog}] \vdash e.f \longrightarrow e'$

H1: $[\mathsf{prog}]; [] \vdash e : \{(f_i : T_i)*\}$

H2: $(f : T') \in (f_i : T_i)*$

The expression $e.f$ can reduce. The following cases are possible:

Subcase 7.1: $[\mathsf{prog}] \vdash e \longrightarrow e''$ and R-Proj applies.

Then

$$[\mathsf{prog}] \vdash e.f \longrightarrow e''.f.$$

By induction on H1, $[\mathsf{prog}]; [] \vdash e'' : \{(f_i : T_i)*\}$. By H2 and T-Proj,

$$[\mathsf{prog}]; [] \vdash e''.f : T'.$$

Subcase 7.2: e is a value of a record type. By theorem Canonical-Rec there are two forms.

Subcase 7.2.1: $e = \{(f_j = v_j)*\}$.

Then R-RecProj yields

$$[\texttt{prog}] \vdash \{(f_j = v_j)*\}.f \longrightarrow v'$$

for some $(f = v') \in (f_j = v_j)*$. From H1 and Canonical-Rec (record-value form), we know that $[\texttt{prog}]; [] \vdash \{(f_j = v_j)*\} : \{(f_i : T_i)*\}$ and each field value v_j in the record has the corresponding declared field type. Since H2 gives $(f : T') \in (f_i : T_i)*$ and field names are unique, the projected field $(f = v')$ must be the entry whose declared type is T'. Therefore

$$[\texttt{prog}]; [] \vdash v' : T'.$$

Subcase 7.2.2: $e = a.R(\{(f_j = v_j)*\})$ and $[\texttt{prog}] \vdash a.R <: \{(f_i : T_i)*\}$.

Then R-InstProj gives two possibilities:

Subcase 7.2.2.a: projection from explicit fields.

There exists v' such that

$$[\texttt{prog}] \vdash a.R(\{(f_j = v_j)*\}).f \longrightarrow v'$$

and $(f = v') \in (f_j = v_j)*$. As above, matching field names and typing information imply $[\texttt{prog}]; [] \vdash v' : T'$.

Subcase 7.2.2.b: projection from defaults in linkage.

There exists a defaults definition $R \mapsto \{(f_k = v_k)*\}$ and $(f = v') \in (f_k = v_k)*$. Because the program is well-typed, linkage definitions are well-typed; therefore the selected default has the declared field type:

$$[\texttt{prog}]; [] \vdash v' : T'.$$

Case 8: T-FamFun.

$$[]; [] \vdash p : T_p$$

$$[\texttt{prog}]; [] \vdash a.m : T \to T'$$

$$[\texttt{prog}] \vdash a.m \longrightarrow e'$$

The reduction must be R-FamFun: there exists $\lambda(x : T).e_0$ such that

$$[\texttt{prog}] \vdash a.m \longrightarrow \lambda(x : T).e_0,$$

with $[\texttt{prog}] \vdash \text{WF}(a)$, $a \rightsquigarrow L_D$, and $m \mapsto (T \to T', \lambda(x : T).e_0) \in L_D.\text{FUNS}$. Since the program is well-typed, all nested family definitions are well-formed, and thus the definition of m in the linkage must be well-typed. Thus,

$$[\texttt{prog}]; [] \vdash \lambda(x : T).e_0 : T \to T'.$$

Case 9: T-Cases.

$$[]; [] \vdash p : T_p$$

$$[\texttt{prog}]; [] \vdash a.c : \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}$$

$$[\texttt{prog}] \vdash a.c \longrightarrow e'$$

The reduction must be R-Cases: there exists $\lambda(x : T).e_0$ such that

$$[\texttt{prog}] \vdash a.c \longrightarrow \lambda(x : T).e_0,$$

with $[\texttt{prog}] \vdash \text{WF}(a)$, $a \rightsquigarrow L_D$, and $c \mapsto (\langle a'.R \rangle, T \to T', \lambda(x : T).e_0) \in L_D$.CASES. Rule T-CASES retrieves the same type from the corresponding static linkage (L_S) entry, so $\{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}$ is the corresponding $T \to T'$. Since the program is well-typed, this definition is well-typed; therefore:

$$[\texttt{prog}]; [] \vdash \lambda(x : T).e_0 : T \to T'$$

and equivalently:

$$[\texttt{prog}]; [] \vdash \lambda(x : \{(f_i : T_i)*\}).e_0 : \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}$$

Case 10: T-Constr.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash a.R(\{(f_i = e_i)*\}) : a.R$

$[\texttt{prog}] \vdash a.R(\{(f_i = e_i)*\}) \longrightarrow e'$

H0: $[\texttt{prog}] \vdash \text{WF}(a)$

H1: $a \rightsquigarrow L_S$

H2: $R \mapsto \{(f_j : T_j)*\} \in L_S$.TYPES

H3: $R \mapsto (f_k)* \in L_S$.DEFS

$\forall i$, Hi: $\exists j,\ f_i = f_j \wedge [\texttt{prog}]; [] \vdash e_i : T_j$

$\forall j$, Hj: $f_j \in (f_i)* \vee f_j \in (f_k)*$

Since the constructor expression reduces, the outer step is R-INSTANCE:

$$[\texttt{prog}] \vdash a.R(\{(f_i = e_i)*\}) \longrightarrow a.R(e_r)$$

for some e_r with $[\texttt{prog}] \vdash \{(f_i = e_i)*\} \longrightarrow e_r$. The only record rule is R-Rec, so $e_r = \{(f_i = e_i')*\}$, where exactly one first reducible field changes. Let it be index ix, with $[\texttt{prog}] \vdash e_{ix} \longrightarrow e_{ix}'$.

By induction on the typing premise for e_{ix}, the reduced field preserves its type. All other fields are unchanged. Thus,

$$\forall i,\ \exists j,\ f_i = f_j \wedge [\texttt{prog}]; [] \vdash e_i' : T_j.$$

Combining this with H0–H3 and Hj forall j, rule T-Constr yields

$$[\texttt{prog}]; [] \vdash a.R(\{(f_i = e_i')*\}) : a.R.$$

Thus the reduced term has type $a.R$.

Case 11: T-ADT.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash a.R(C\ \{(f_k = e_k)*\}) : a.R$

$[\texttt{prog}] \vdash a.R(C\ \{(f_k = e_k)*\}) \longrightarrow e'$

H0: $[\texttt{prog}] \vdash \mathrm{WF}(a)$

H1: $a \rightsquigarrow L_S$

H2: $R \mapsto \overline{C_j\{(f_i : T_i)*\}} \in L_S.\mathrm{ADTS}$

H3: $C\{(f_k : T_k)*\} \in \overline{C_j\{(f_i : T_i)*\}}$

$\forall k$, Hk: $[\texttt{prog}]; [] \vdash e_k : T_k$

Since the ADT value reduces, the outer step is R-ADT:

$$[\texttt{prog}] \vdash a.R(C\ \{(f_k = e_k)*\}) \longrightarrow a.R(C\ e_r)$$

for some e_r with $[\texttt{prog}] \vdash \{(f_k = e_k)*\} \longrightarrow e_r$. By R-Rec, $e_r = \{(f_k = e'_k)*\}$ with one first reducible field. Let that field be $e_{kx} \longrightarrow e'_{kx}$.

By induction on the typing derivation for e_{kx}, the reduced field keeps its type; other fields are unchanged. Therefore

$$\forall k,\ [\texttt{prog}]; [] \vdash e'_k : T_k.$$

Using H0–H3 and rule T-ADT, we derive

$$[\texttt{prog}]; [] \vdash a.R(C\ \{(f_k = e'_k)*\}) : a.R.$$

Thus, type is preserved.

Case 12: T-Match.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash \text{match } e \text{ with } a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\} : T'$

$[\texttt{prog}] \vdash \text{match } e \text{ with } a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\} \longrightarrow e''$

H0: $[\texttt{prog}] \vdash \text{WF}(a')$

H1: $a' \rightsquigarrow L_S$

H2: $[\texttt{prog}]; [] \vdash e : a'.R$

H3: $R \mapsto \overline{C_j\{(f_i : T_i)*\}} \in L_S.\text{ADTS}$

H4: $[\texttt{prog}]; [] \vdash a.c : \{(f_{\text{arg}} : T_{\text{arg}})*\} \to \{(C_j : \{(f_i : T_i)*\} \to T')*\}$

H5: $[\texttt{prog}]; [] \vdash \{(f_{\text{arg}} = e_{\text{arg}})*\} : \{(f_{\text{arg}} : T_{\text{arg}})*\}$

The match expression can reduce, and there are three possible cases.

Subcase 12.1: R-MatchExp.

We have $[\texttt{prog}] \vdash e \longrightarrow e'$ and

$$[\texttt{prog}] \vdash \text{match } e \text{ with } a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\} \longrightarrow \text{match } e' \text{ with } a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\}.$$

By induction on H2, $[\texttt{prog}]; [] \vdash e' : a'.R$. Using H0, H1, H3, H4, H5 and T-Match, we get

$$[\texttt{prog}]; [] \vdash \text{match } e' \text{ with } a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\}\ : T'.$$

Subcase 12.2: R-MatchCases.

We have $e = v$ and

$$[\texttt{prog}] \vdash \{(f_{\text{arg}} = e_{\text{arg}})*\} \longrightarrow \{(f_{\text{arg}} = e'_{\text{arg}})*\},$$

such that

$$[\texttt{prog}] \vdash \text{match } v \text{ with } a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\} \longrightarrow \text{match } v \text{ with } a.c\ \{(f_{\text{arg}} = e'_{\text{arg}})*\}.$$

From H2 (with $e = v$), $[\texttt{prog}]; [] \vdash v : a'.R$. By induction on H5, the argument record keeps type $\{(f_{\text{arg}} : T_{\text{arg}})*\}$. Then T-Match gives

$$[\texttt{prog}]; [] \vdash \text{match } v \text{ with } a.c\ \{(f_{\text{arg}} = e'_{\text{arg}})*\}\ : T'.$$

Subcase 12.3: R-MatchFinal.

We have

$$e = a'.R(C\ \{(f_k = v_k)*\}), \quad \{(f_{\text{arg}} = e_{\text{arg}})*\} = \{(f_{\text{arg}} = v_{\text{arg}})*\},$$

and

$[\texttt{prog}] \vdash \text{match } a'.R(C\ \{(f_k = v_k)*\}) \text{ with } a.c\ \{(f_{\text{arg}} = v_{\text{arg}})*\} \longrightarrow (a.c\ \{(f_{\text{arg}} = v_{\text{arg}})*\}).C\ \{(f_k = v_k)*\}$.

We must show

$$[\texttt{prog}]; [] \vdash (a.c\ \{(f_{\text{arg}} = v_{\text{arg}})*\}).C\ \{(f_k = v_k)*\} : T'.$$

From H4 and H5, by T-App:

$$[\texttt{prog}]; [] \vdash a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\}\ : \{(C_j : \{(f_i : T_i)*\} \to T')*\}.$$

From H2 with $e = a'.R(C\ \{(f_k = v_k)*\})$, inversion of T-ADT gives $C\{(f_k : T_k)*\} \in \overline{C_j\{(f_i : T_i)*\}}$ and $\forall k, [\texttt{prog}]; [] \vdash v_k : T_k$, which is equivalent to $[\texttt{prog}]; [] \vdash \{(f_k = v_k)*\} : \{(f_k : T_k)*\}$.

Since we have $C\{(f_k : T_k)*\} \in \overline{C_j\{(f_i : T_i)*\}}$, and since by convention the j and i indices in the definition of R and in the type of $a.c$ range over the same constructors and fields, we have: $(C : \{(f_k : T_k)*\} \to T') \in \{(C_j : \{(f_i : T_i)*\} \to T')*\}$. Since we have

$$[\texttt{prog}]; [] \vdash a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\}\ :\ \{(C_j : \{(f_i : T_i)*\} \to T')*\}$$

and

$$(C : \{(f_k : T_k)*\} \to T') \in \{(C_j : \{(f_i : T_i)*\} \to T')*\},$$

we derive by T-Proj

$$[\texttt{prog}]; [] \vdash (a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\}).C : \{(f_k : T_k)*\} \to T'.$$

Combining with $[\texttt{prog}]; [] \vdash \{(f_k = v_k)*\} : \{(f_k : T_k)*\}$ and T-App, we conclude

$$[\texttt{prog}]; [] \vdash (a.c\ \{(f_{\text{arg}} = e_{\text{arg}})*\}).C\ \{(f_k = v_k)*\} : T'.$$

Case 13: T-Subs.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash e : T'$

$[\texttt{prog}] \vdash e \longrightarrow e'$

H1: $[\texttt{prog}]; [] \vdash e : T$

H2: $[\texttt{prog}] \vdash T <: T'$

By induction on H1, $[\texttt{prog}]; [] \vdash e' : T$. With H2 and T-Subs, we derive

$[\texttt{prog}]; [] \vdash e' : T'$.

Case 14: T-If.

$[]; [] \vdash p : T_p$

$[\texttt{prog}]; [] \vdash \text{if } e \text{ then } g \text{ else } g' : T'$

$[\texttt{prog}] \vdash \text{if } e \text{ then } g \text{ else } g' \longrightarrow e''$

H1: $[\texttt{prog}]; [] \vdash e : \text{B}$

H2: $[\texttt{prog}]; [] \vdash g : T'$

H3: $[\texttt{prog}]; [] \vdash g' : T'$

There are three reduction forms.

Subcase 14.1: R-IfGuard.

We have $[\texttt{prog}] \vdash e \longrightarrow e'$ and

$$[\texttt{prog}] \vdash \text{if } e \text{ then } g \text{ else } g' \longrightarrow \text{if } e' \text{ then } g \text{ else } g'.$$

By induction on H1, $[\texttt{prog}]; [] \vdash e' : \mathrm{B}$. Using H2 and H3, T-If yields

$$[\texttt{prog}]; [] \vdash \text{if } e' \text{ then } g \text{ else } g' : T'.$$

Subcase 14.2: R-IfTrue.

If

$$[\texttt{prog}] \vdash \text{if true then } g \text{ else } g' \longrightarrow g,$$

then by H2:

$$[\texttt{prog}]; [] \vdash g : T'.$$

Subcase 14.3: R-IfFalse.

If

$$[\texttt{prog}] \vdash \text{if false then } g \text{ else } g' \longrightarrow g',$$

then by H3:

$$[\texttt{prog}]; [] \vdash g' : T'.$$

□

B.3 Transitivity of Subtyping

Theorem 5 (Sub-Trans)**.** *For all* K, T_1, T_2, T_3*:*

$$\mathrm{K} \vdash T_1 <: T_2 \implies \mathrm{K} \vdash T_2 <: T_3 \implies \mathrm{K} \vdash T_1 <: T_3.$$

Proof. By induction on T_2.

Case 1: $T_2 = N$**.** Trivial by reflexivity (Sub-Refl) and premise 2 ($\mathrm{K} \vdash T_2 <: T_3$).

Case 2: $T_2 = B$**.** Trivial by reflexivity (Sub-Refl) and premise 2 ($\mathrm{K} \vdash T_2 <: T_3$).

Case 3: $T_2 = a.R$**.** We have:

$$(1) \quad \mathrm{K} \vdash T_1 <: a.R$$

$$(2) \quad \mathrm{K} \vdash a.R <: T_3$$

By inversion of the derivation of premise (1), the last rule is Sub-Refl. Thus, $T_1 = a.R$ and we already know $\mathrm{K} \vdash a.R <: T_3$ by premise (2).

Case 4: $T_2 = T_{21} \to T_{22}$**.** We have:

$$(1) \quad \mathrm{K} \vdash T_1 <: T_{21} \to T_{22}$$

$$(2) \quad \mathrm{K} \vdash T_{21} \to T_{22} <: T_3$$

By inversion of the derivation of premise (1), we have the following subcases: Sub-Refl, Sub-Fun.

Subcase 4.1: In premise (1), the last rule is Sub-Refl.

Then $T_1 = T_{21} \to T_{22}$, and premise (2) finishes the proof.

Subcase 4.2: In premise (1), the last rule is Sub-Fun.

Then the following is also true:

$$T_1 = T_{11} \to T_{12}$$

$$\mathrm{K} \vdash T_{21} <: T_{11}$$

$$\mathrm{K} \vdash T_{12} <: T_{22}$$

By inversion of the derivation of premise (2), we have the following subcases: Sub-Refl, Sub-Fun.

If the last rule is Sub-Refl, the solution is trivial by premise (1).

If the last rule is Sub-Fun, we also know:

$$T_3 = T_{31} \to T_{32}$$

$$\mathrm{K} \vdash T_{31} <: T_{21}$$

$$\mathrm{K} \vdash T_{22} <: T_{32}$$

We need to show $\mathrm{K} \vdash T_{11} \to T_{12} <: T_{31} \to T_{32}$, or equivalently:

$\mathrm{K} \vdash T_{31} <: T_{11}$ and $\mathrm{K} \vdash T_{12} <: T_{32}$.

By the induction hypothesis:

- Since $\mathrm{K} \vdash T_{31} <: T_{21}$ and $\mathrm{K} \vdash T_{21} <: T_{11}$, we have $\mathrm{K} \vdash T_{31} <: T_{11}$.
- Since $\mathrm{K} \vdash T_{12} <: T_{22}$ and $\mathrm{K} \vdash T_{22} <: T_{32}$, we have $\mathrm{K} \vdash T_{12} <: T_{32}$.

No other subcases.

Case 5: $T_2 = \{(f_j : T_j)*\}$**.** We have:

$$(1) \quad \mathrm{K} \vdash T_1 \ <: \ \{(f_j : T_j)*\}$$

$$(2) \quad \mathrm{K} \vdash \{(f_j : T_j)*\} \ <: \ T_3$$

By inversion of the derivation of premise (1), we have the following subcases: Sub-Refl, Sub-Rec, Sub-Fam.

Subcase 5.1: In premise (1), the last rule is Sub-Refl.

Then $T_1 = \{(f_j : T_j)*\}$, and proven by premise (2).

Subcase 5.2: In premise (1), the last rule is Sub-Rec.

Then we know:

$$T_1 = \{(f_i : T_i)*\}$$

$$\forall j, \quad \exists T, \quad \mathrm{K} \vdash T \ <: \ T_j \quad \wedge \quad (f_j : T) \ \in \ (f_i : T_i)*$$

By inversion of the derivation of premise (2), we have the following subcases: Sub-Refl, Sub-Rec.

If the last rule is Sub-Refl, the solution is trivial by premise (1).

If the last rule is Sub-Rec, we also know:

$$T_3 = \{(f_k : T_k)*\}$$

$$\forall k, \quad \exists T', \quad \mathrm{K} \vdash T' \ <: \ T_k \quad \wedge \quad (f_k : T') \ \in \ (f_j : T_j)*$$

We need to show $\mathrm{K} \vdash T_1 \ <: \ T_3$, or equivalently:

$$\forall k, \quad \exists T_s, \quad \mathrm{K} \vdash T_s \ <: \ T_k \ \ \wedge \ \ (f_k : T_s) \ \in \ (f_i : T_i)*$$

Fix an arbitrary k. By premise (2), choose T' such that

$$\mathrm{K} \vdash T' \ <: \ T_k \quad \text{and} \quad (f_k : T') \ \in \ (f_j : T_j)\,* \,.$$

Using premise (1) for the field label f_k, choose T_s such that

$$\mathrm{K} \vdash T_s \ <: \ T' \quad \text{and} \quad (f_k : T_s) \ \in \ (f_i : T_i)\,* \,.$$

By the induction hypothesis on $\mathrm{K} \vdash T_s \ <: \ T'$ and $\mathrm{K} \vdash T' \ <: \ T_k$, we obtain $\mathrm{K} \vdash T_s \ <: \ T_k$. Since k was arbitrary, the required statement follows.

Subcase 5.3: In premise (1), the last rule is Sub-Fam.

Then we have $T_1 = a.R$, and:

$$\mathrm{K} \vdash \texttt{WF}(a)$$

$$a \rightsquigarrow L_S$$

$$R \mapsto \{(f_i : T_i)*\} \ \in \ L_S.\mathrm{TYPES}$$

$$\mathrm{K} \vdash \{(f_i : T_i)*\} \ <: \ \{(f_j : T_j)*\}$$

To conclude $\mathrm{K} \vdash a.R \ <: \ T_3$, by Sub-Fam it suffices to show $\mathrm{K} \vdash \{(f_i : T_i)*\} \ <: \ T_3$. This follows from Subcase 5.2 using $\mathrm{K} \vdash \{(f_i : T_i)*\} \ <: \ \{(f_j : T_j)*\}$ (above) and premise (2). Hence $\mathrm{K} \vdash a.R \ <: \ T_3$ by Sub-Fam.

□

References

Agrawal, A., First, E., Kaufman, Z., Reichel, T., Zhang, S., Zhou, T., Sanchez-Stern, A., Ringer, T., & Brun, Y. (2023). Proofster: Automated formal verification. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (pp. 26–30).: IEEE.

Anthropic (2024). Claude: Ai assistant.

Arias, E. J. G., Pin, B., & Jouvelot, P. (2017). jscoq: Towards hybrid theorem proving interfaces. *arXiv preprint arXiv:1701.07125*.

Balestrieri, F. & Mauny, M. (2018). Generic programming in ocaml. *arXiv preprint arXiv:1812.11665*.

Banerjee, D., Bouissou, O., & Zetzsche, S. (2026). DafnyPro: LLM-assisted automated verification for Dafny programs. In *Proceedings of the Workshop on Auto-Active Program Verification (Dafny)*, Dafny 2026, co-located with POPL 2026 Rennes, France. Available at https://arxiv.org/abs/2601.05385.

Barke, S., James, M. B., & Polikarpova, N. (2023). Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1), 85–111.

Blaauwbroek, L., Olšák, M., Rute, J., Massolo, F. I. S., Piepenbrock, J., & Pestun, V. (2024). Graph2tac: online representation learning of formal math concepts. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24: JMLR.org.

Blaauwbroek, L., Urban, J., & Geuvers, H. (2020). The tactician: A seamless, interactive tactic learner and prover for coq. In *International Conference on Intelligent Computer Mathematics* (pp. 271–277).: Springer.

Blume, M., Acar, U. A., & Chae, W. (2006). Extensible programming with first-class cases. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming - ICFP '06* (pp. 239). New York, New York, USA: ACM Press.

Brandfonbrener, D., Henniger, S., Raja, S., Prasad, T., Loughridge, C., Cassano, F., Hu, S. R., Yang, J., Byrd, W. E., Zinkov, R., & Amin, N. (2024). VerMCTS: Synthesizing multi-step programs using a verifier, a large language model, and tree search.

Cardelli, L. (1997). Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (pp. 266–277).

Carette, J., Kiselyov, O., & Shan, C.-c. (2009). Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5), 509–543.

Chakravarty, M. M. T., Keller, G., Jones, S. P., & Marlow, S. (2005). Associated types with class. In *ACM Symp. on Principles of Programming Languages (POPL)*.

Clarke, D., Drossopoulou, S., Noble, J., & Wrigstad, T. (2007). Tribe: A simple virtual class calculus. In *Proceedings of the 6th international conference on Aspect-oriented software development - AOSD '07* (pp. 121). New York, New York, USA: ACM Press.

Czajka, Ł. & Kaliszyk, C. (2018). Hammer for coq: Automation for dependent type theory. *Journal of automated reasoning*, 61(1), 423–453.

de Moura, L., Kong, S., Avigad, J., van Doorn, F., & von Raumer, J. (2015). The lean theorem prover (system description). In A. P. Felty & A. Middeldorp (Eds.), *Automated Deduction - CADE-25*, volume 9195 of *Lecture notes in computer science* (pp. 378–388). Cham: Springer International Publishing.

Delaware, B., d. S. Oliveira, B. C., & Schrijvers, T. (2013). Meta-theory à la carte. *ACM SIGPLAN Notices*, 48(1), 207–218.

Ebresafe, O., Zhao, I., Jin, E., Bright, A., Jian, C., & Zhang, Y. (2025). Certified compilers à la carte. *Proceedings of the ACM on Programming Languages*, 9(PLDI), 372–395.

Emir, B., Odersky, M., & Williams, J. (2007). Matching objects with patterns. In *European Conference on Object-Oriented Programming* (pp. 273–298).: Springer.

Ernst, E. (2001). Family polymorphism. In J. L. Knudsen, G. Goos, J. Hartmanis, & J. van Leeuwen (Eds.), *ECOOP 2001 —Object-Oriented Programming*, volume 2072 of *Lecture notes in computer science* (pp. 303–326). Berlin, Heidelberg: Springer Berlin Heidelberg.

Ernst, E. (2003). Higher-order hierarchies. In *European Conference on Object-Oriented Programming* (pp. 303–328).: Springer.

Ernst, E., Ostermann, K., & Cook, W. R. (2006). A virtual class calculus. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL'06* (pp. 270–282). New York, New York, USA: ACM Press.

Fan, A., Huang, X., Xu, H., Sun, Y., & Oliveira, B. C. d. S. (2022). Direct foundations for compositional programming. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*: Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

Fan, A. & Parreaux, L. (2023). super-charging object-oriented programming through precise typing of open recursion. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*: Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

First, E. & Brun, Y. (2022). Diversity-driven automated formal verification. In *Proceedings of the 44th International Conference on Software Engineering* (pp. 749–761).

First, E., Rabe, M. N., Ringer, T., & Brun, Y. (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* (pp. 1229–1241).

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

Garrigue, J. (1998). Programming with polymorphic variants. In *ML workshop*, volume 13: Baltimore.

Garrigue, J. (2000). Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, volume 13.

Gaster, B. R. & Jones, M. P. (1996). *A polymorphic type system for extensible records and variants*. Technical report, Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham.

GitHub, Inc. (2026). Github copilot. https://github.com/features/copilot. Accessed: 2026-04-04.

Hart, S. G. & Staveland, L. E. (1988). Development of nasa-tlx (task load index): Results of empirical and theoretical research. In *Advances in psychology*, volume 52 (pp. 139–183). Elsevier.

Hendriks, M., Kaliszyk, C., Raamsdonk, F. v., & Wiedijk, F. (2010). Teaching logic using a state-of-the-art proof assistant.

Hu, J., Zhang, J., Zhao, Y., & Ringer, T. (2025). Hybridprover: Augmenting theorem proving with llm-driven proof synthesis and refinement.

Huang, D., Dhariwal, P., Song, D., & Sutskever, I. (2018). GamePad: A learning environment for theorem proving.

Hubert, T., Mehta, R., Sartran, L., Horváth, M. Z., Žužić, G., Wieser, E., Huang, A., Schrittwieser, J., Schroecker, Y., Masoom, H., et al. (2025). Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, (pp. 1–3).

Igarashi, A., Saito, C., & Viroli, M. (2005). Lightweight family polymorphism. In K. Yi, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, & G. Weikum (Eds.), *Programming languages and systems*, volume 3780 of *Lecture notes in computer science* (pp. 161–177). Berlin, Heidelberg: Springer Berlin Heidelberg.

Igarashi, A. & Viroli, M. (2007). Variant path types for scalable extensibility. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications* (pp. 113–132).

Isradisaikul, C. & Myers, A. C. (2013). Reconciling exhaustive pattern matching with objects. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13 (pp. 343–354). New York, NY, USA: Association for Computing Machinery.

Jiang, A. Q., Li, W., Tworkowski, S., Czechowski, K., Odrzygóźdź, T., Miłoś, P., Wu, Y., & Jamnik, M. (2022). Thor: Wielding hammers to integrate language models and automated theorem provers. *Advances in Neural Information Processing Systems*, 35, 8360–8373.

Jiang, A. Q., Welleck, S., Zhou, J. P., Lacroix, T., Liu, J., Li, W., Jamnik, M., Lample, G., & Wu, Y. (2023). Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*.

Jin, E., Amin, N., & Zhang, Y. (2023). Extensible metatheory mechanization via family polymorphism. *Proceedings of the ACM on Programming Languages*, 7(PLDI).

Kasibatla, S. R., Agarwal, A., Brun, Y., Lerner, S., Ringer, T., & First, E. (2026). Cobblestone: A divide-and-conquer approach for automating formal verification. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering*, ICSE '26. Available at https://arxiv.org/abs/2410.19940.

Kiselyov, O. (2012). Typed tagless final interpreters. In *Generic and Indexed Programming* (pp. 130–174). Springer.

Komendantskaya, E., Heras, J., & Grov, G. (2012). Machine learning in proof general: Interfacing interfaces. *arXiv preprint arXiv:1212.3618*.

Kozyrev, A., Solovev, G., Khramov, N., & Podkopaev, A. (2024). Coqpilot, a plugin for llm-based generation of proofs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24 (pp. 2382–2385). New York, NY, USA: Association for Computing Machinery.

Kravchuk-Kirilyuk, A., Feng, G., Iskander, J., Zhang, Y., & Amin, N. (2024). Persimmon: Nested family polymorphism with extensible variant types. *Proc. ACM Program. Lang.*, 8(OOPSLA1).

Kravchuk-Kirilyuk, A., Holloway, T., First, E., Brun, Y., & Glassman, E. (2026a). Understanding user interactions with llms and specialized proof synthesis tools in interactive theorem proving. Manuscript in preparation.

Kravchuk-Kirilyuk, A., Nguyen, D., Karty, R., Henniger, S., Poesia, G., & Amin, N. (2026b). The scaffolding paradox: Evidence from ai-assisted dafny proof synthesis. Under submission to SAIV 2026 Workshop.

Lee, J. D. & See, K. A. (2004). Trust in automation: Designing for appropriate reliance. *Human factors*, 46(1), 50–80.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33: Curran Associates, Inc.

Liang, J. T., Yang, C., & Myers, B. A. (2024). A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24 New York, NY, USA: Association for Computing Machinery.

Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., & Alignment, K. C. (2023). Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

Löh, A. & Hinze, R. (2006). Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming - PPDP '06* (pp. 133). New York, New York, USA: ACM Press.

Loughridge, C., Sun, Q., Ahrenbach, S., Cassano, F., Sun, C., Sheng, Y., Mudide, A., Misu, M. R. H., Amin, N., & Tegmark, M. (2024). Dafnybench: A benchmark for formal software verification.

Lu, M., Delaware, B., & Zhang, T. (2024). Proof automation with large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24 (pp. 1509–1520). New York, NY, USA: Association for Computing Machinery.

Madsen, O. L., Møller-Pedersen, B., & Nygaard, K. (1993). *Object Oriented Programming in the BETA Programming Language*. Addison-Wesley.

Matsakis, N. D. & Klock, F. S. (2014). The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14 (pp. 103–104).: ACM.

Meng, J. & Paulson, L. C. (2008). Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1), 35–60.

Millstein, T., Bleckner, C., & Chambers, C. (2004). Modular typechecking for hierarchically extensible datatypes and functions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(5), 836–889.

Misu, M. R. H., Lopes, C. V., Ma, I., & Noble, J. (2024). Towards ai-assisted synthesis of verified dafny methods. *Proc. ACM Softw. Eng.*, 1(FSE).

Mozannar, H., Bansal, G., Fourney, A., & Horvitz, E. (2024). Reading between the lines: Modeling user behavior and costs in ai-assisted programming. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI '24 New York, NY, USA: Association for Computing Machinery.

Norell, U. (2009). Dependently typed programming in agda. In P. Koopman, R. Plasmeijer, & D. Swierstra (Eds.), *Advanced Functional Programming*, volume 5832 of *Lecture notes in computer science* (pp. 230–266). Berlin, Heidelberg: Springer Berlin Heidelberg.

Nystrom, N., Chong, S., & Myers, A. C. (2004). Scalable extensibility via nested inheritance. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications - OOPSLA '04* (pp.99). New York, New York, USA: ACM Press.

Nystrom, N., Qi, X., & Myers, A. C. (2006). J& nested intersection for scalable software composition. *ACM SIGPLAN Notices*, 41(10), 21–36.

Odersky, M. & Zenger, M. (2005). Scalable component abstractions. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '05* (pp.41). New York, New York, USA: ACM Press.

Oliveira, B. C. d. S. & Cook, W. R. (2012). Extensibility for the masses. In J. Noble, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, & G. Weikum (Eds.), *ECOOP 2012 – Object-Oriented Programming*, volume 7313 of *Lecture notes in computer science* (pp. 2–27). Berlin, Heidelberg: Springer Berlin Heidelberg.

OpenAI (2023). *GPT-4 Technical Report*. Technical report, OpenAI.

Peyton Jones, S. (2009). Classes, Jim, but not as we know them—type classes in Haskell: What, why, and whither. In *European Conf. on Object-Oriented Programming*.

Pimenova, V., Fakhoury, S., Bird, C., Storey, M.-A., & Endres, M. (2025). Good vibrations? a qualitative study of co-creation, communication, flow, and trust in vibe coding. *arXiv preprint arXiv:2509.12491*.

Pit-Claudel, C. (2020). Untangling mechanized proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering* (pp. 155–174).

Poesia, G., Loughridge, C., & Amin, N. (2024). dafny-annotator: AI-assisted verification of dafny programs.

Polu, S. & Sutskever, I. (2020). Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*.

Ren, Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., et al. (2025). Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*.

Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., & Tatlock, Z. (2019). QED at large: A survey of engineering of formally verified software. *Foundations and Trends®in Programming Languages*, 5(2-3), 102–281.

Ringer, T., Porter, R., Yazdani, N., Leo, J., & Grossman, D. (2021). Proof repair across type equivalences. *PLDI*.

Ringer, T., Sanchez-Stern, A., Grossman, D., & Lerner, S. (2020). Replica: Repl instrumentation for coq analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020 (pp. 99–113). New York, NY, USA: Association for Computing Machinery.

Roe, K. & Smith, S. (2016). CoqPIE: an IDE aimed at improving proof development productivity. In J. C. Blanchette & S. Merz (Eds.), *Interactive Theorem Proving*, volume 9807 of *Lecture notes in computer science* (pp. 491–499). Cham: Springer International Publishing.

Ross, S. I., Martinez, F., Houde, S., Muller, M., & Weisz, J. D. (2023). The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*, IUI '23 (pp. 491–514). New York, NY, USA: Association for Computing Machinery.

Sanchez-Stern, A., Alhessi, Y., Saul, L., & Lerner, S. (2020). Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (pp. 1–10).

Sanchez-Stern, A., Varghese, A., Kaufman, Z., Zhang, D., Ringer, T., & Brun, Y. (2025). QED-Cartographer: Automating formal verification using reward-free reinforcement learning. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*, ICSE '25.

Sarkar, A., Gordon, A. D., Negreanu, C., Poelitz, C., Ragavan, S. S., & Zorn, B. (2022). What is it like to program with artificial intelligence? In *Proceedings of the 33rd Annual Workshop of the Psychology of Programming Interest Group (PPIG)*.

Song, P., Yang, K., & Anandkumar, A. (2024). Lean Copilot: Large language models as copilots for theorem proving in Lean. *arXiv preprint arXiv:2404.12534*.

Sun, C., Sheng, Y., Padon, O., & Barrett, C. (2024). Clover: Closed-loop verifiable code generation. In *Proceedings of the Workshop on Auto-Active Program Verification (Dafny)*, Dafny 2024, co-located with POPL 2024. Available at https://arxiv.org/abs/2310.17807.

Swierstra, W. (2008). Data types à la carte. *Journal of Functional Programming*, 18(4), 423–436.

Syme, D., Neverov, G., & Margetson, J. (2007). Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming* (pp. 29–40).

Taft, S. T., Duff, R. A., Brukardt, R. L., Ploedereder, E., Leroy, P., & Schonberg, E. (2014). *Ada 2012 reference manual. Language and standard libraries: international standard ISO/IEC 8652/2012 (E)*, volume 8339. Springer.

Thakur, A., Tsoukalas, G., Wen, Y., Xin, J., & Chaudhuri, S. (2024). An in-context learning agent for formal theorem-proving. In *First Conference on Language Modeling (COLM)*.

The Rocq Development Team (2024). The rocq proof assistant.

Thompson, K., Saavedra, N., Carrott, P., Fisher, K., Sanchez-Stern, A., Brun, Y., Ferreira, J. a. F., Lerner, S., & First, E. (2025). Rango: Adaptive retrieval-augmented proving for automated software verification. ICSE '25 (pp. 347–359).: IEEE Press.

Thorup, K. K. (1997). Genericity in Java with virtual types. In *European Conf. on Object-Oriented Programming*.

Tobin-Hochstadt, S. (2011). Extensible pattern matching in an extensible language. *arXiv preprint arXiv:1106.2578*.

Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. Experience: Evaluating the usability of Code Generation Tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*: ACM.

Wadler, P. (1998). The expression problem. Discussion on Java-Genericity mailing list. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

Wang, H., Unsal, M., Lin, X., Baksys, M., Liu, J., Santos, M. D., Sung, F., Vinyes, M., Ying, Z., Zhu, Z., Lu, J., de Saxcé, H., Bailey, B., Song, C., Xiao, C., Zhang, D., Zhang, E., Pu, F., Zhu, H., Liu, J., Bayer, J., Michel, J., Yu, L., Dreyfus-Schmidt, L., Tunstall, L., Pagani, L., Machado, M., Bourigault, P., Wang, R., Polu, S., Barroyer, T., Li, W.-D., Niu, Y., Fleureau, Y., Hu, Y., Yu, Z., Wang, Z., Yang, Z., Liu, Z., & Li, J. (2025). Kimina-prover preview: Towards large formal reasoning models with reinforcement learning.

Yang, K. & Deng, J. (2019). Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning* (pp. 6984–6994).: PMLR.

Yang, K., Swope, A. M., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., & Anandkumar, A. (2023). LeanDojo: Theorem proving with retrieval-augmented language models. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Zenger, M. & Odersky, M. (2001). Extensible algebraic datatypes with defaults. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming* (pp. 241–252).

Zenger, M. & Odersky, M. (2004). Independently extensible solutions to the expression problem.

Zhang, W. & Oliveira, B. C. d. S. (2020). Castor: Programming with extensible generative visitors. *Science of Computer Programming*, 193, 102449.

Zhang, W., Sun, Y., & Oliveira, B. C. D. S. (2021). Compositional programming. *ACM Transactions on Programming Languages and Systems*, 43(3), 1–61.

Zhang, Y. & Myers, A. C. (2017). Familia: unifying interfaces, type classes, and family polymorphism. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), 1–31.