Technical Report MPI-SWS-2015-001 April 2015

Doctoral Thesis

Interactive Typed Tactic Programming in the Coq Proof Assistant

Author:
Beta Ziliani

Supervisor:
Dr. Derek Dreyer

Thesis for obtaining the title of
Doctor of Engineering Science
of the

Faculties of Natural Science and Technology

of

Saarland University

Saarbrücken
April 2015

Date of the colloquium:	24.03.2015
Dean:	Prof. Dr. Markus Bläser
Reporter:	Dr. Derek Dreyer
	Dr. Georges Gonthier
	Prof. Dr. Gert Smolka
Chairman of the Examination board:	Prof. Dr. Sebastian Hack
Scientific Assistant:	Dr. Ori Lahav

"Those who do not move, do not notice their chains."

Rosa Luxemburg

Abstract

Interactive Typed Tactic Programming in the Coq Proof Assistant

Beta Ziliani

2015

In order to allow for the verification of realistic problems, Coq provides a language for *tactic* programming, therefore enabling general-purpose scripting of automation routines. However, this language is untyped, and as a result, tactics are known to be difficult to compose, debug, and maintain. In this thesis, I develop two different approaches to typed tactic programming in the context of Coq: *Lemma Overloading* and *Mtac*. The first one utilizes the existing mechanism of overloading, already incorporated into Coq, to build typed tactics in a style that resembles that of dependently typed *logic* programming. The second one, Mtac, is a lightweight yet powerful extension to Coq that supports dependently typed *functional* tactic programming, with additional imperative features.

I motivate the different characteristics of Lemma Overloading and Mtac through a wide range of examples, mainly coming from program verification. I also show how to combine these approaches in order to obtain the best of both worlds, resulting in *extensible*, typed tactics that can be programmed interactively.

Both approaches rely heavily on the unification algorithm of Coq, which currently suffers from two main drawbacks: it incorporates heuristics not appropriate for tactic programming, and it is undocumented. In this dissertation, in addition to the aforementioned approaches to tactic programming, I build and describe a new unification algorithm better suited for tactic programming in Coq.

Kurzdarstellung

Interactive Typed Tactic Programming in the Coq Proof Assistant

Beta Ziliani

2015

Um realistische Programme zu verifizieren, bietet Coq eine Sprache zum Programmieren von *Taktiken*. Sie ermöglicht das Schreiben universeller Automatisierungsroutinen. Diese Sprache ist allerdings ungetypt. Die resultierenden Taktiken sind daher bekannt dafür, schwer zusammensetzbar, testbar und wartbar zu sein. Ich entwickle in dieser Doktorarbeit zwei Ansätze für getypte Taktiken im Kontext von Coq: Das *Überladen von Lemmata* und *Mtac*. Ersteres benutzt den in Coq vorhandenen Überladungsmechanismus um Taktiken im Stil von Dependently Typed Logic Programming zu erstellen. Letzeres, Mtac, ist eine leichtgewichtige, aber mächtige Erweiterung zu Coq, die das Erstellen von funktionalen, Dependently Typed Taktiken mit imperativen Features erlaubt.

Ich motiviere die verschiedenen Charakteristika der beiden Ansätze durch eine große Auswahl an Beispielen, die hauptsächlich aus der Programm-Verifizierung kommen. Ich demonstriere außerdem, wie man die Ansätze kombinieren kann, um das Beste aus beiden Welten zu bekommen: *erweiterbare*, getypte Taktiken die interaktiv programmiert werden können.

Beide Ansätze sind stark abhängig vom Unifizierungs-Algorithmus in Coq. Dieser leidet momentan unter zwei Nachteilen. Einerseits sind die Heuristiken nicht auf das Programmieren von Taktiken abgestimmt. Andererseits ist der Algorithmus undokumentiert. Zusätzlich zu den oben genannten Ansätzen zur Programmierung von Taktiken, entwickle und beschreibe ich in dieser Dissertation einen neuen Unifizierungs-Algorithmus, der besser für die Programmierung von Taktiken geeignet ist.

Acknowledgements

For long, scientists have been wrongly portrayed by the media as lone workers, with no interest for social connections at all. In my particular case, I'm proud to say that, during my thesis, I've had the opportunity to interact with many brilliant people from all around the globe. I can claim most of the original contributions in this manuscript as mine, however, without the input from the many people I interacted with within these five years, these contributions would have been nothing more than pure vodka: a good base for very complex and tasty drinks, but not that exciting on its own.

Chronologically, I have to start with my "Doktoronkel" Aleks Nanevski, a great master from which I learned Coq and the Ssreflect library. He was the one who got me interested in the problem of proof automation in Coq, so this thesis owes him a lot! He was a big supporter and great collaborator in the projects. Moreover, he was the one who put me in contact with Georges Gonthier. Georges, in turn, opened to me the mysteries of the unification algorithm of Coq and of canonical structures. He suggested the incorporation of the heuristic to remove dependencies in my unification algorithm. Georges was, and still is, a great source of inspiration for my work. When it comes to understanding the details of Coq, Matthieu Sozeau was a constant help in my thesis, and I greatly enjoy collaborating with him.

The work on Mtac got invaluable feedback from several people. First, I should mention Viktor Vafeiadis, who participated in engaging discussions when the project was only an idea on a board, and who encouraged me to continue developing it. Chung-Kil "Gil" Hur, as a post-doc in our group at the time, suggested the Mendler-style encoding of Mtac's fixpoint, saving the project from a possible inconsistency. Another post-doc at the time, Neel Krishnaswami, helped me develop the fragment of Mtac devoted to binders. As it turns out, my supervisor has an incredible hand at picking brilliant post-docs! I am also thankful to Arnaud Spiwack, for insisting on the use of telescopes for representing patterns, and Jesper Bengtson and Jonas Jensen, for testing an earlier version of the language. Other people who deserve gratitude are Nils Anders Danielsson and Antonis Stampoulis, for providing useful feedback about the paper.

Mtac keeps evolving thanks to the enthusiasm of Yann Règis-Gianas, Jan-Oliver Kaiser, and Thomas Refis. I'm certain the next version of the system will rock, and I'm looking forward to future collaborations with Yann and his group.

Many thanks to Enrico Tassi, who explained to me very thoroughly some details about the current unification algorithm in Coq, in particular the complicated heuristic for constant unfolding. Also, I am deeply thankful to Andreas Abel, who clarified some of the aspects involved in the process of unification.

To Rose Hoberman I owe my communication skills. I'm quite a terrible English student, but somehow she managed to make me write, if not proper English, at least something one can relate to!

I'd like to also thank Deepak Garg for the engaging discussions about logic and teaching.

Special mention to my great friend and comrade Scott Kilpatrick. We "grew up" together in the institute, and we shared many discussions on the board and in the pub. With him I learned a lot about politics, music, movies, etc. And, of course, modules! I will miss you big time, buddy!

And the best for last: to my Doktorvater Derek Dreyer, all my love. In one sentence: he was the structure I needed to build myself into a researcher. He was always very supportive and open. From day one, he listened to my ideas as if I were an established researcher. Despite him being one of the most intelligent researchers I know, I never felt inferior in his presence, and he never imposed a research agenda to me. As the busy man he is, I never felt alone when I needed him most. He taught me most of what I know about being a researcher, and he allowed me to develop ideas far off from his own research interests, a freedom I enjoyed greatly. Moreover, he encouraged me to visit some of the researchers listed above in order to learn from their experiences. At a personal level he's a warm, generous man, and a great cultural hub. He taught me all I know about whisky. Perhaps, thanks to his experience with drinks, he transformed the vodka from my production into the (hopefully) tasty different "coqtails" you're invited to enjoy in the next chapters!

Contents

List of Figures

Dedicated to Cecilia, my beloved girlfriend.

She convinced me to pursue a scientific career
and supported me in my endeavours.

Introduction

According to Wikipedia,[1]

> *"A proof is sufficient evidence or an argument for the truth of a proposition."*

This informal statement should raise the eyebrow of any hard-scientist: how do we declare that we have *sufficient evidence*, or a *valid* argument? Even in the rigorous world of mathematics, people tend to disagree on these concepts. Take for instance the Four Color Problem. For over more than a century many famous mathematicians provided proofs that turn out to be incorrect (Saaty and Kainen, 1977), until in 1976 the first "correct" proof was developed by Appel and Haken (1976). The proof breaks the problem into a monstrous number of cases, analyzing each with the help of a computer.

The proof presented by Appel and Haken, novel in the use of a computer program, raised reasonable complaints. After all, on which grounds should we trust a computer program? Computer programs are usually plagued with bugs, hard to verify, and hard to relate to the mathematics they are intended to model. In this particular case, to make things worse, the program was written in IBM 370 assembly! On top of that, the proof included an initial manual case analysis with $10,000$ cases where several minor mistakes were found.

Twenty years later, Robertson et al. (1997) made a cleaner presentation of the same proof, this time in C, and with an accompanying monograph describing the proof. Still, the proof relies on a large computer program, which computer scientists had to manually verify. This gap between the math and the program was finally closed for good in 2008, when Georges Gonthier presented a formal proof written entirely in the Coq proof assistant (Gonthier, 2008).

Proof assistants like Coq (Bertot and Castéran, 2004, The Coq Development Team, 2012), HOL (Gordon and Melham, 1993), Isabelle (Paulson, 1994), Lego (Luo and Pollack, 1992), LF (Harper et al., 1993), and NuPRL (Constable et al., 1986), to name just

[1] And to "common sense".

a few, are tools designed specifically to bring confidence in our results, removing the natural doubts that come from large, complicated proofs. Essentially, a proof assistant is a computer software developed to verify proofs encoded in a certain logic. Then, the assistant has the mindless job of verifying that the proof is a valid sequence of steps in that logic.

Of course, there is still a fair amount of trust involved in the usage of a proof assistant: one has to trust that the logic is a trustworthy encoding of mathematical reasoning, and that the proof assistant is a trustworthy encoding of that logic (and, in addition, that the compiler and the hardware did not introduce significant errors). Nevertheless, the assistant is a generic tool, and not an ad-hoc program built to prove a specific problem. In the case of an established proof assistant, like the aforementioned ones, they are continuously being developed and tested by many proof developments. Therefore, their reliability is not at stake, even when bugs are to be found every now and then. It is commonly agreed that a mature proof assistant is significantly more trustworthy than a human reviewer.[2]

To put it concisely, informal arguments are at one extreme of the "trust spectrum", while proof assistants are located at the opposite extreme. Needless to say, not every proof require such rigor; however, in modern mathematics (and accordingly, in computer science), proofs tend to grow large and complex, making proof assistants an increasingly adopted tool in the mathematician's toolbox. In this regard, Vladimir Voevodsky, a 2002 Fields Medal recipient, posits: [3]

> *"Soon enough, [mathematicians] are going to find themselves doing mathematics at the computer, with the aid of computer proof assistants. Soon, they won't consider a theorem proven until a computer has verified it."*

In the particular case of Coq, its popularity has grown quickly in the past years thanks to a large number of successful formalization efforts. For example, in the area of algebra, in addition to the already mentioned Four Color Theorem, the Feit-Thompson Theorem (Gonthier et al., 2013a) was recently formalized. The original proof of this theorem was published in two volumes, totaling an astounding 250 pages. Gonthier and his team were able to formalize it, together with all the necessary background material, in Coq. In the area of computer software verification, perhaps the most impressive example is the CompCert C compiler (Leroy, 2009). This compiler comprises several compilation and optimization phases for a realistic fragment of C, and the formalization in Coq ensures that each phase preserves the intended semantics of the original program.

[2] Although a human reviewer is still needed to check that the definitions and statements in the proof development are what the authors of the proof claim!

[3] http://blogs.scientificamerican.com/guest-blog/2013/10/01/voevodskys-mathematical-revolution/

The key for the success of such proof developments is due, at least in part, according to Gonthier et al. (2013a), to the use of the rich logic employed by Coq. Indeed, the *Calculus of Inductive Constructions* (CIC), as it is called, allows for very rich specifications and abstractions. However, despite its expressiveness, Gonthier et al. (2013a) also acknowledge that real verification efforts require automation beyond that allowed by the logic. We illustrate this point with a simple example from software verification, more precisely about pointer no-aliasing. Consider the following goal where we have to prove that pointers x_1, x_2, x_3 are pairwise distinct, *i.e.*, they do not alias:

$$x_1 \texttt{ != } x_2 \;\&\&\; x_2 \texttt{ != } x_3 \;\&\&\; x_1 \texttt{ != } x_3$$

in a context where the following hypothesis holds, stating that the pointers points to disjoint sections of a heap:

$$D : \mathsf{def}\ (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2 \bullet x_3 \mapsto v_3)$$

A heap is a finite map between locations and values; $x \mapsto v$ is the singleton heap containing x pointing at v; $h_1 \bullet h_2$ is the function that merges heaps h_1 and h_2 if they do not overlap on their domain, or returns a special value Undef if they do; $\mathsf{def}\ h$ is a function that returns true iif the heap h is defined.

In order to solve the goal we can use the following lemma stating that if two pointers point to disjoint locations *at the front* of a heap, and the heap is defined, then they do not alias:

$$\begin{aligned}\mathsf{noalias} : \forall h{:}\mathsf{heap}.\ \forall x_1 x_2{:}\mathsf{ptr}.\ \forall v_1{:}A_1.\ \forall v_2{:}A_2.\\ \mathsf{def}\ (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2 \bullet h) \to x_1 \texttt{ != } x_2\end{aligned}$$

In order to apply this lemma to the three inequalities, we have to rearrange the heap in the hypothesis D in order to bring the singleton pointers to the front of the heap, as required by the lemma. This rearrangement is possible, as the disjoint union operator is associative and commutative, but tedious and, more importantly, not robust to changes in the original heap.

Ideally, one would like to have a *tactic*, that is, a meta-program, to solve the goal automatically, either by rearranging the heap to move the pointers to the front of the heap and then apply the lemma above, or by searching for the two pointers in the heap, constructing a proof of their inequality along the way. Note that, in either case, we need to go beyond of the logic of the proof assistant, since the tactic requires inspecting the syntax of the heap: both $(\cdot \mapsto \cdot)$ and $(\cdot \bullet \cdot)$ are functions, not constructors of an inductive type. Modern proof assistants enable this type of meta-reasoning by incorporating a *tactic language*.

Current tactic languages, however, have serious limitations. To illustrate this point, we will present and motivate different properties desirable for a tactic language, and evaluate the existing languages with respect to these properties.

1. **Maintainability:** Tactics should be easy to maintain. During the development of a proof, it is usually the case that we need to introduce changes to our definitions and lemmas. As a consequence, some of the tactics developed may break. We would like to have immediate feedback, like a typechecker, indicating the specific places where the tactic needs to be updated.

2. **Composability:** It should be easy to compose tactics and lemmas. This point is better explained with an example. Consider the following part of the goal above, now showing the implicit coercion from a boolean to a proposition:

   $$x_1 \texttt{ != } x_2 = \mathsf{true}$$

   (`!=` is a boolean inequality operator, and any boolean b is automatically coerced to an equality $b = \mathsf{true}$ when a proposition is required.) In this goal one must provide a *positive* proof of a pointer *inequality*. An equivalent goal requires a *negative* proof of their *equality*:

   $$x_1 \texttt{ == } x_2 = \mathsf{false}$$

   In a tactic language without composability, this trivial change of the goal can be quite problematic, since the original tactic, let's call it auto_noalias, was originally conceived to solve pointer inequalities, and not equalities. The problem gets exacerbated if we suppose that the pointer (in-)equalities we want to resolve are embedded in a larger context, *e.g.*,

   $$G : \mathsf{if}\ (x_2 \texttt{ == } x_3)\ \&\&\ (x_1 \texttt{ != } x_2)\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2$$

   In this situation, we cannot apply the tactic, directly to reduce $(x_2 \texttt{ == } x_3)$ and $(x_1 \texttt{ != } x_2)$ to false and true, respectively, since those (in-)equalities are not the top-level goal. Coq's rewrite primitive is designed precisely for this situation—it enables one to reduce all (in-)equalities within G that match the conclusion of a particular lemma—but it is not applicable to tactics (like auto_noalias).

   Thus, with the auto_noalias tactic, we are left with essentially two options: (i) use it to prove a bespoke lemma about one specific inequality (say, $x_1 \texttt{ != } x_2$), perform a rewrite using that lemma, and repeat for other (in-)equalities of interest, or (ii) implement another custom tactic that crawls over the goal G searching for any and all (in-)equalities that auto_noalias might resolve. The former option sacrifices

the benefits of automation, while the latter option redundantly duplicates the functionality of rewrite.

Instead, we would like the language for proof automation to give tactics a similar status as standard lemmas, therefore enabling the composition of the rewrite tactic and the auto_noalias tactic, additionally allowing tactics to be composed with standard lemmas. For instance, in order to rewrite pointer equalities, we would like to be able to compose the tactic auto_noalias with the following standard lemma, which allows us to transform a problem of the form x_1 != x_2 into one of the form x_1 == x_2 = false:

$$\mathsf{negbTE} : \forall b{:}\mathsf{bool}.\ !b \to b = \mathsf{false}$$

3. **Interactivity:** In order to ease the development of tactics, the tactic developer should be able to construct *proof terms* appearing in the tactic *interactively*. A proof term is, basically, a term in CIC witnessing a proof. Proof assistants like Coq provide tools to build proof terms in an step-wise manner, where the proof developer interacts with the proof assistant to help construct the proof, either backwardly changing the goal to match some hypothesis, and/or forwardly changing the hypothesis to match the goal. These tools should be available for constructing tactics as well.

4. **Simplicity:** The tactic language should have a clear and simple semantics. It should be easy to guess what the output of a tactic should be based on a quick scrutiny of the code.

5. **Formality:** The semantics of the tactic language should be formally described, and it should be proven sound.

6. **Efficiency:** Since speed is a critical factor in an interactive proof assistant, the tactic language should provide the means to build efficient tactics.

7. **Extensibility:** The functionality of a tactic should be easily extensible. In the example above the heap from the hypothesis was constructed using functions $\mapsto$ and $\bullet$. If we want to consider a new function for heap subtraction, then it should be easy to extended the tactic to consider the new case.

Current tactic languages support only a subset of these properties. Coq incorporates two tactic languages: OCaml, the language on which Coq is built, and Ltac (Delahaye, 2000). The former, in the specific case of tactic programming, is a very low-level language, as the developer is forced to use the internal representation of terms—for the interested reader, Coq is built using a locally nameless approach (Gordon, 1994). OCaml tactics

are efficient and extensible, but they are hard to maintain, compose and reason about. And, of course, they cannot be built interactively.

On the other hand, Ltac is a purely functional high-level language, which allows for meta-programming without exposing the internal representation of terms. However, it also fails to satisfy almost all of the properties mentioned above, with the sole exception of extensibility. The primary drawback of Ltac, partially shared with OCaml, is that tactics lack the precise typing of the theorem prover's base logic (they are essentially untyped). To begin with, the lack of precise typing can make tactics much more difficult to maintain than, say, lemmas, as changes in basic definitions do not necessarily raise type errors in the code of the tactics affected by the changes. Rather, type checking is performed on the goals obtained during tactic execution, resulting in potentially obscure error messages and unpredictable proof states in the case of failure. Moreover, the behavior of a tactic cannot be specified, nor can it be verified against a specification.

Also, due to their lack of precise typing, tactics suffer a second-class status, in the sense that they may not be used as flexibly as lemmas. Another important restriction of Ltac is that tactics cannot be built interactively. And this is not just a technical restriction, but a fundamental one, as without a type the proof assistant is unable to figure out the missing goal required to be solved interactively.

Concerning performance, Ltac tactics are restricted to a pure functional model of computation, making it impossible to use hash tables or other fast imperative data structures to build efficient tactics.

As pointed out in Malecha et al. (2014), Ltac semantics are extremely complicated, making tactics hard to understand and debug. The recent efforts in pin-pointing Ltac's semantics (Jedynak, 2013, Jedynak et al., 2013) fall short for two reasons: firstly, they miss the ability of Ltac to handle proof terms (constr: for the knowledgeable), and secondly, and more importantly, they do not describe the unification algorithm that Ltac uses. Unification is a critical component of meta-programming and, as a matter of fact, Ltac uses an unsound unification algorithm. Since Ltac cannot trust the answer of this algorithm, it checks the result, and backtracks if a problem is found, making the behavior of tactics unpredictable.

As we mentioned above, Ltac does allow for extensible tactics. More precisely, *some OCaml tactics* (*e.g.*, auto and autorewrite) are extensible, and they can be accessed in Ltac. These tactics have access to *hint databases*, where a selection of previously proved lemmas are stored. Then, these tactics employ the hints in a given database to (try to) solve the goal at hand. At any future point in time, the proof developer is entitled to extend the database, increasing the solving power of the tactic. However, as Uncle

Ben[4] said: "With great power comes great responsibility", in this case meaning that the increased solving power may not be responsibly handled by the tactic engine. Since the length of proof search paths is potentially augmented with each new hint, previously solvable goals may become unsolvable in the time allocated by the tactic engine.

Despite all these problems, however, Ltac is still a powerful automation tool. Chlipala (2011b) shows how, once the beast is tamed, it can reduce significantly the number of lines of code required to verify low-level procedures. In Chlipala (2013), the same author teaches the art of beast-taming. That said, the issues mentioned here are well known by the Coq community, and partially documented in Malecha et al. (2014). In fact, in a recent survey[5] the Coq community voted the creation of a better tactic language as the fourth most important request for a future version of Coq.

Many of the problems with Ltac mentioned above are a direct consequence of it lacking a proper type system. Several researchers in the past few years have worked on the problem of creating *typed* tactic languages, giving birth to proof assistants like Delphin (Poswolsky and Schürmann, 2009), Beluga (Pientka, 2008), and more recently VeriML (Stampoulis, 2012). The tactic languages baked into these assistants employ rich, formally described type systems to ensure tactics comply with a given specification. Therefore, by having a specification, tactics are easy to maintain and compose. However, to the best of our knowledge, none of these typed tactic languages allow for interactive tactic programming nor extensible tactics. With respect to performance, VeriML is the only one that supports effectful computation and imperative data-structures, in a style resembling that of ML. But more importantly, the main downside of these languages is that they are not easy to incorporate into well-established proof assistants like Coq, mainly due to the advanced type-theoretic machinery on which they rely—*e.g.*, Contextual Modal Type Theory (Nanevski et al., 2008c) for Beluga and VeriML.

Thesis Statement

In this thesis, I study a novel approach to typed tactic languages in the context of proof assistants based on the Calculus of Inductive Constructions (CIC), in particular—but not restricted to—Coq. In this approach, unlike all previous tactic languages, the line dividing proof terms and tactics is blurred in order to encourage reuse of the mechanisms readily available in proof assistants. As a result, I obtain all the aforementioned properties in a tactic language for Coq. More succinctly:

[4] Character from Spider-Man.
[5] http://www.irisa.fr/celtique/pichardie/cuw2014/braibant.pdf

> **My thesis is that interactive typechecked tactic programming in Coq is possible in two different programming styles: (1) logic programming (by exploiting the existing overloading mechanism) and (2) functional programming extended with imperative features (by creating a new language). Both styles can be formally described, and their combination leads to a novel way of creating extensible tactics.**

Structure of the Thesis: In the remainder of this thesis I start by introducing the basic concepts of Coq and its underlying logic (Chapter 1). Then, in Chapter 2, I introduce *Lemma Overloading*, a technique developed in collaboration with G. Gonthier, A. Nanevski and D. Dreyer (Gonthier et al., 2013b). The key idea in this chapter is to leverage the overloading mechanism à la Haskell's type classes, readily available in Coq, in order to perform typed tactic programming. The programming style afforded by overloading resembles that of (dependently) typed logic programming.

In contrast, in Chapter 3, I present a novel *functional* tactic language, Mtac, which was developed in collaboration with D. Dreyer, N. Krishnaswami, A. Nanevski and V. Vafeiadis (Ziliani et al.). Mtac, in combination with Lemma Overloading, validates the claims made in my thesis statement.

As it will become evident in the following chapters, both programming idioms rely heavily on the unification algorithm. This complex algorithm is in charge of finding a solution to a problem of the form $t \approx u$, where t and u are *open* terms, that is, terms with *meta-variables* (holes) in them. The algorithm should find, if possible, a substitution Σ substituting meta-variables for (open) terms, such that the application of this substitution to t and u yields two *convertible* terms t' and u'.

Unfortunately, the current unification algorithm of Coq incorporates some heuristics not appropriate for proof automation, and that makes the algorithm unpredictable. Indeed, seemingly equivalent unification problems may have completely different outcomes. Furthermore, there is no good source of information to understand how it works. For these reasons, in Chapter 4 I introduce a new unification algorithm, built in collaboration with M. Sozeau, with simpler, clearer, and well-documented semantics.

The work related to this dissertation is discussed in Chapter 5. Chapter 6 presents the conclusions and directions for future work.

Chapter 1

Basics

This chapter presents the Coq proof assistant from both an informal and a formal point of view. First, in §1.1, we provide a short, and necessarily incomplete, Coq tutorial. Then, in §1.2, we provide the core technical aspects of the logic: the Calculus of Inductive Constructions (CIC). To conclude, in §1.3, we provide the basics of *canonical structures*, Coq's overloading mechanism, which we used extensively in Chapter 2.

The knowledgeable reader may skim over this chapter, but is encouraged to pay special attention to the notation introduced, as well as the advanced features described in §1.2.1.1 and §1.3.

1.1 Coq, an Interactive Proof Assistant

This section provides a quick introduction to the main concepts behind the *interactive proof assistant* called **Coq**. It is not intended to serve as a full tutorial—Coq's web page is full of better introductory materials than this chapter[1]—but it gives the casual reader a big picture of the system, hopefully enough to comprehend the main concepts behind this thesis.

We start by giving an analogy, borrowed from Xavier Leroy. He described Coq as a game: The proof developer (proof dev for short) starts by providing a theorem she wants to solve. Then, Coq asks her for a proof. She responds by providing a *tactic* to somehow simplify the goal. This tactic can be to introduce a hypothesis, to use induction on a given variable, to rewrite a part of the goal with some given equality, to apply a previously proven lemma, etc. Coq, after performing the requested change in the goal, responds with the new goal. The game continues, perhaps branching into

[1] http://coq.inria.fr

```
01 Lemma addn0 : ∀ n : nat. n + 0 = n.
02 Proof.
03 (* ============================ *)
04 (* forall n :  nat, n + 0 = n *)
05 elim ⇒ [ | n' IH ].
06 (* 2 subgoals, subgoal 1 *)
07 (* ============================ *)
08 (* 0 + 0 = 0 *)
09 (* subgoal 2 is:  *)
10 (* S n' + 0 = S n' *)
11 - by [].
12 (* n' :  nat *)
13 (* IH : n' + 0 = n' *)
14 (* ============================ *)
15 (* S n' + 0 = S n' *)
16 - simpl.
17 (* n' :  nat *)
18 (* IH : n' + 0 = n' *)
19 (* ============================ *)
20 (* S (n' + 0) = S n' *)
21   rewrite IH.
22 (* n' :  nat *)
23 (* IH : n' + 0 = n' *)
24 (* ============================ *)
25 (* S n' = S n' *)
26   by [].
27 (* No more subgoals.  *)
28 Qed.
29 (* addn0 is defined *)
```

FIGURE 1.1: Example of interaction with the Coq proof assistant.

different subgoals (subcases in a pen and paper proof, like when using induction), until every subgoal is solved. Coq then communicates its defeat and the proof dev gives the last estocade by typing **Qed**. If at any moment the step provided by the proof dev is invalid, Coq immediately complains. This back-and-forth interaction between Coq and the proof dev is what is meant by the term *interactive* in an interactive proof assistant, in sharp contrast with proof assistants like Twelf (Pfenning and Schürmann, 1999), which compiles a proof in a batch fashion very much like a programming language compiler.

Figure 1.1 shows a very simple example of this "game". In it, we prove a lemma stating that $n + 0$ is equal to n, for every natural number n. Note that the proof is very short in itself—only four lines long—but we have interleaved comments, enclosed with (* *), showing the proof state (*i.e.*, Coq's response) after a command or a tactic is executed. Commands in Coq are written in capitalized case, as in **Proof** or **Qed**, while tactics are

written in lower case, as in elim. Commands modify and query the global environment, while tactics modify a current proof state.

Notational Convention 1. Throughout this thesis, for Coq programs we will mostly use the original Coq syntax. However, in a few exceptional cases, we will take the liberty of making the syntax more "math friendly". For instance, we will write functions and products as $\lambda x.\ t$ and $\forall x.\ u$, respectively, instead of **fun** $x \Rightarrow t$ and **forall** x, u.

Coming back to Figure 1.1, after stating the lemma in the first line, Coq responds with the proof obligation displayed in comments in lines 3 and 4. This proof obligation can be read as "under no assumptions, you need to prove that for all n, ..." If there were any assumptions, as we are going to see next, they will be displayed above the double line. The command **Proof** in line 2 is just a no-op, but it is a Coq convention that every proof should start with it.

In line 5 the first tactic is provided. It is the elim tactic from Ssreflect, which performs induction on the first variable appearing in the goal (in this case, n). Throughout this thesis we will use the idiom for tactics afforded by the library Ssreflect (Gonthier et al., 2008), which is better crafted than Coq's own tactics. The elim tactic takes an *intro pattern* (what comes after the $\Rightarrow$), which is a list of lists of names. The outermost list should have one element per subgoal, separated by |. Each element of this list is a list of names separated by a space. In this case, the induction on natural numbers generates two subgoals, representing the base case, when $n = 0$, and the inductive case, when $n = n' + 1$ for some n'. The definition of natural numbers, together with the addition function and its notation, is standard[2] and can be found in Figure 1.2.

For the base case, no new hypotheses are added to the context, so no new names are given in the first list of the intro pattern. For the inductive case, two hypotheses are added: the number n' and the inductive hypothesis stating that $n' + 0 = n'$. Coq's answer is displayed in lines 6–10. In line 6, it communicates that we have to prove two subgoals, which are identified with numbers 1 and 2, respectively, and tell us that we are currently proving goal #1. Then, in lines 7–10, it shows the two subgoals. It does not show the context of subgoal #2.

The first subgoal is trivial, we need to prove that $0 + 0 = 0$ under no assumptions, and this holds by computation. We instruct Coq to dismiss this goal as trivial with Ssreflect's *tactical* by in line 11. (A tactical si simply a tactic that tas another tactic as argument.) The by tactical uses the tactic given as argument to prove the goal and to check that

[2]It is interesting to note that the Coq language is minimal: it does not even include natural numbers natively; they are instead defined in the standard library. Ssreflect provides a slightly different notation for numbers, but we stick to the standard one for presentation purposes.

```
Inductive nat : Set := O : nat | S : nat → nat

Definition addn :=
 fix plus (n m : nat) {struct n} : nat :=
   match n with
   | 0 ⇒ m
   | S p ⇒ S (plus p m)
   end.

Notation "a + b" := (addn a b).
```

FIGURE 1.2: Natural numbers in Coq.

the goal was indeed solved by it. In this case we do not need any tactic, since the goal is trivial, so we provide [] as argument.

After the first subgoal is solved, Coq outputs the remaining subgoal (show in comments in lines 12–15). We note that the hypotheses for the second subgoal appear now in the context, with the names provided in the intro pattern from line 5. This subgoal requires us to prove that $\mathsf{S}\ n' + 0 = \mathsf{S}\ n'$. By performing some steps of computation on the left hand side, we can change the goal to $\mathsf{S}\ (n' + 0) = \mathsf{S}\ n'$. This is done with the simpl tactic in line 16 (Ssreflect also allows a more general way to perform the same, writing rewrite /=). The output of the simpl tactic is in the comments on lines 17–20.

At this point we can use the inductive hypothesis to rewrite the goal into its final form. This is done in line 21. The new goal is now trivial, as can be seen in lines 22–25, so we finish the proof again using the by tactical. Note that the last three steps can be accomplished in just one line, thanks to the Ssreflect's powerful rewrite tactic. The one-liner equivalent is

$$\mathsf{by\ rewrite}\ /= IH$$

The proof is completed and Coq communicates that there are no subgoals left. *Quod Erat Demostrandum* (**Qed**), it is then demonstrated. Lastly, Coq announces that the lemma is now part of the global environment, *i.e.*, the global knowledge we can make use of in any other proof.

1.1.1 Proof Terms

Coq does not store the proof script in its *environment* (the database of knowledge available to the user). Instead, it stores the *proof term* that the script helped generate.

A proof term is a λ-term that, following the Curry-Howard isomorphism (*e.g.*, Sørensen and Urzyczyn, 2006), represents a valid proof of the theorem stated in its type. For instance, this is the proof term, followed by its type, generated for the lemma above:

$\lambda\ n : \mathsf{nat}.$
$\mathsf{nat\_ind}\ (\lambda\ n_0 : \mathsf{nat}.\ n_0 + 0 = n_0)\ \mathsf{eq\_refl}$
$\quad (\lambda\ (n_0 : \mathsf{nat})\ (IH : n_0 + 0 = n_0).$
$\quad\ \mathsf{eq\_ind\_r}\ (\lambda\ n_1 : \mathsf{nat}.\ S\ n_1 = S\ n_0)\ \mathsf{eq\_refl}\ IH)\ n$
$\quad\quad : \forall\ n : \mathsf{nat}.\ n + 0 = n$

It is possible in Coq to write proof terms directly, without using tactics. Similarly, in the context of proof automation, it is possible to automate the generation of proof *scripts* or proof *terms*. In this dissertation we focus on the latter problem. For this reason, it is important to know Coq's λ-calculus, the Calculus of Inductive Constructions, which will be covered in the next section.

1.2 The Calculus of Inductive Constructions

One of the aspects that make Coq so special is its complex type system. It is so complex, in fact, that there is actually no distinction between the syntactical classes of terms and types! We have seen this already: the lemma `addn0` from last section has *type* $\forall n : \mathsf{nat}.\ n + 0 = 0$, that is, a type including a call to the addition function. As a consequence, a term in this language may represent either a proof, a function, a theorem, or a function type. It is the type system that is in charge of determining a term's true nature, making sure that, for instance, a function $\lambda x : T.\ t$ *has* a type but *is not in itself* a type.

More precisely, the base language of Coq, the *Calculus of Inductive Constructions* (CIC) (The Coq Development Team, 2012, chap. 4), is a dependently typed λ-calculus extended with inductive types. It also includes co-inductive types, but their formulation is not important for this thesis, so it will be omitted.

In order to understand the technical contributions of this thesis, it is important to grasp an intuition on how terms in CIC are constructed, evaluated, and typechecked. Therefore, in the following sections we define the syntax, semantics, and statics of CIC. The reader well versed in Coq will still find this section useful, as we introduce the notation used throughout the thesis.

1.2.1 Syntax

The terms (and types) of the language are defined as

$$
\begin{aligned}
t, u, T, U, \rho \mathrel{\widehat{=}} \; & x \mid c \mid i && x \in \mathcal{V}, c \in \mathcal{C}, i \in \mathcal{I} \\
& \mid k \mid s \mid ?x[\sigma] && k \in \mathcal{K}, s \in \mathcal{S}, ?u \in \mathcal{M} \\
& \mid \forall x : T.\ U \mid \lambda x : T.\ t \mid t\ u \mid \mathbf{let}\ x := t : T\ \mathbf{in}\ u \\
& \mid \mathbf{match}_{\rho}\ t\ \mathbf{with}\ k_1\ \overline{x_1} \Rightarrow t_1 \mid \ldots \mid k_n\ \overline{x_n} \Rightarrow t_n\ \mathbf{end} \\
& \mid \mathbf{fix}_j\ \{x_1/n_1 : T_1 := t_1; \ldots; x_m/n_m : T_m := t_m\}
\end{aligned}
$$

$$
\sigma \mathrel{\widehat{=}} \overline{t}
$$

where $\mathcal{V}$ is a disjoint sets of variables, $\mathcal{M}$ of meta-variables (also called unification variables), $\mathcal{C}$ of constants, $\mathcal{I}$ of inductive types, $\mathcal{K}$ of inductive constructors, and $\mathcal{S}$ is an infinite set of sorts defined as $\{\mathsf{Prop}, \mathsf{Type}(i) \mid i \in \mathsf{N}\}$.

Notational Convention 2. Throughout this work we will use the following conventions:

- x, y, z denote variables.
- f, g denote variables with function type.
- t, u denote terms.
- A, B, C denote type variables.
- T, U, τ denote types.
- s denotes a sort variable.
- ρ denotes a type predicate, that is, a function from any type to Prop or Type.
- c, i, and k denote unknown constants, inductive type constructors and constructors, respectively. i and k will also be used as indexes, in which case their meaning will be clear from the context.
- Constants, inductive type constructors and constructors are noted in sans-serif font, as in addn0, nat, and S.
- A list of elements is denoted with a bar above a representative element. For instance, $\overline{x}$ is a list of variables, $\overline{t}$ is a list of terms, and so on. Sometimes we require the list to have exactly n elements, in which case we annotate the list

with the subscript n, as in $\overline{x_n}$. To refer to the j-th element in the list we use the subscript notation without the bar: x_j is the j-th element of the list $\overline{x}$.

Coming back to the syntax, terms include variables x, constants c, inductive *type* constructors i, inductive *value* constructors k, and sorts s. Terms may contain a *hole*, representing a missing piece of the term (or proof). Holes are represented with *meta-variables*, a variable prepended with a question mark, as in $?x$ or $?f$ (the latter representing an unknown function). For reasons that will become apparent soon, meta-variables are applied to a *suspended substitution* $[\sigma]$, which is nothing more than a list of terms.

In order to destruct an element of an inductive type, CIC provides regular pattern **match**ing and mutually recursive **fix**points. Their notation is slightly different from, but easily related to, the actual notation from Coq. **match** is annotated with the return predicate ρ, meaning that the type of the whole **match** expression may depend on the element being pattern matched (**as**...**in**... in standard Coq notation). In the **fix** expression, $x/n : T := t$ means that T is a type starting with at least n product types, and the n-th variable is the decreasing one in t (**struct** in Coq notation). The subscript j of **fix** selects the j-th function as the main entry point of the mutually recursive fixpoints.

In order to typecheck and reduce terms, Coq uses several contexts, each handling different types of knowledge:

1. The local context Γ, including bound variables and **let**-bound expressions;
2. the meta-context Σ, containing meta-variable declarations and definitions; and
3. the global environment E, containing the global knowledge; that is, axioms, theorems, and inductive definitions.

Formally, they are defined as follows:

$$\begin{aligned}
\Gamma, \Psi &\widehat{=} \ \cdot \mid x : T, \Gamma \mid x := t : T, \Gamma \\
\Sigma &\widehat{=} \ \cdot \mid ?x : T[\Psi], \Sigma \mid ?x := t : T[\Psi], \Sigma \\
E &\widehat{=} \ \cdot \mid c : T, E \mid c := t : T, E \mid I, E
\end{aligned}$$

$$I \widehat{=} \ \forall \Gamma. \ \{ \ \overline{i : \forall \overline{y : T_h}. \ s := \{k_1 : U_1; \ldots; k_n : U_n\}} \ \}$$

The local context is standard, and requires no further explanation. Meta-variables have *contextual types*, meaning that the type T of a meta-variable must have all of its free variables bound within the local context Ψ. In this work we borrow the notation $T[\Psi]$

from Contextual Modal Type Theory (Nanevski et al., 2008c). Note that this differs from Pientka and Pfenning (2003), which denotes the same contextual type as $\Psi \vdash T$. A meta-variable can be instantiated with a term t, noted $?x := t : T[\Psi]$. In this case, t should also contain only free variables occurring in Ψ.

The global environment associates a constant c with a type and, optionally, a definition. In the first case, c is an axiom, while in the second c is a theorem proved by term t. Additionally, this environment may also contain (mutually recursive) inductive types.

A set of mutually recursive inductive types I is prepended with a list of parameters Γ. Every inductive type i defined in the set has sort type, with parameters $\overline{y : T_h}$. It has a possibly empty list of constructors $k_1, \ldots, k_n$. For every j, each type U_j of constructor k_j has shape $\forall \overline{z : U'}.\ i\ t_1\ \ldots\ t_h$.

Inductive definitions are restricted to avoid circularity, meaning that every type constructor i can only appear in a strictly positive position in the type of every constructor. For the purpose of this work, understanding this restriction is not crucial, and we refer the interested reader to The Coq Development Team (2012, chap. 4).

1.2.1.1 Meta-Variables and Contextual Types

Meta-variables play a central role in the different processes involving the interactive construction of terms. Understanding the details is crucial for understanding the most advanced parts of this work.

At a high-level, meta-variables are holes in a term, which are expected to be filled out at a later point in time. For instance, when a lemma is applied to solve some goal, Coq internally creates fresh meta-variables for all the formal parameters of the lemma, and proceeds to unify the goal with the conclusion of the lemma. During unification, the meta-variables are instantiated so that both terms (the goal and the conclusion of the lemma) become *convertible* (equal modulo reduction rules, as we will see in Section 1.2.2).

As a simple example, consider the application of the lemma `addn0`, from the previous chapter, in the goal

$$\frac{x : \mathsf{nat}}{x + 0 = x}$$

where, as customary, above the line is the context—in this case containing only variable x—and below is the actual goal we want to prove.

The lemma addn0 has one parameter, n. Coq creates a fresh meta-variable, say $?y$, which has type nat. (For the moment, let's ignore the contextual type and suspended substitution.) Then it proceeds to unify the type of addn0 $?y$ with the goal, that is,

$$?y + 0 = ?y \approx x + 0 = x$$

The unification process instantiates $?y$ with x, therefore obtaining the proof term addn0 x, which has the exact type of the goal.

In this simple example, contextual types played no role but, as we are going to see in the next example, they prevent illegal instantiations of meta-variables. For instance, such illegal instantiations could potentially happen if the same meta-variable occurs at different locations in a term, with different variables in the scope of each occurrence. Suppose, for example, that we define a function f as follows:

$$f := \lambda w : \mathsf{nat}.\ (\_ : \mathsf{nat})$$

where the *implicit* value _ is an indication to Coq's elaboration mechanism to "fill in this hole with a meta-variable". The accessory typing annotation provides the expected type for the meta-variable. When this function is elaborated, it will become $\lambda w : \mathsf{nat}.\ ?v[w]$ for some fresh meta-variable $?v : \mathsf{nat}[w : \mathsf{nat}]$. The contextual type of $?v$ specifies that $?v$ may only be instantiated by a term with at most a single free variable w of type nat, and the *suspended substitution* $[w]$ specifies how to transform such a term into one that is well-typed under the current context. (The substitution is the identity at first, because the current context and the context under which $?v$ was created are both $w : \mathsf{nat}$.)

Now suppose that we define

$$g := \lambda x\ y : \mathsf{nat}.\ f\ x \qquad h := \lambda z : \mathsf{nat}.\ f\ z$$

and then at some point we attempt to solve the following unification problem:

$$g \approx \lambda x\ y : \mathsf{nat}.\ x \tag{1.1}$$

Should this unification succeed, and if so, what should $?v$ be instantiated with? First, to solve the unification goal, Coq will attempt to unify $f\ x \approx x$, and then, after β-reducing $f\ x$, to unify $?v[x] \approx x$. This is where the contextual type of $?v$ comes into play. If we did not have the contextual type (and suspended substitution) for $?v$, it would seem that the only solution for $?v$ is x, but that solution would not make any sense at the point where $?v$ appears in h, since x is not in scope there. Given the contextual information,

$$
\begin{array}{rcll}
(\lambda x : T.\ t)\ u & \rightsquigarrow_{\beta} & t\{u/x\} & \\
\textsf{let}\ x := u : T\ \textsf{in}\ t & \rightsquigarrow_{\zeta} & t\{u/x\} & \\
x & \rightsquigarrow_{\delta\Gamma} & t & \text{if } (x := t : T) \in \Gamma \\
?x[\sigma] & \rightsquigarrow_{\delta\Sigma} & t\{\sigma/\widehat{\Psi}\} & \text{if } ?x := t : T[\Psi] \in \Sigma \\
c & \rightsquigarrow_{\delta E} & t & \text{if } (c := t : T) \in E \\
\textsf{match}_T\ k_j\ \overline{t}\ \textsf{with}\ \overline{k\ \overline{x} \Rightarrow u}\ \textsf{end} & \rightsquigarrow_{\iota} & u_j\{\overline{t/x_j}\} & \\
\textsf{fix}_j\ \{F\}\ \overline{a} & \rightsquigarrow_{\iota} & t_j\{\overline{\textsf{fix}_m\ \{F\}/x_m}\}\ \overline{a} & F = \overline{x/n : T := t} \\
t & \rightsquigarrow_{\eta} & \lambda y : T.\ (t\ y) & E;\Sigma;\Gamma \vdash t : \forall x : T.\ U
\end{array}
$$

FIGURE 1.3: Reduction rules in CIC.

however, Coq will correctly realize that $?v$ should be instantiated with w, not x. Under that instantiation, g will normalize to $\lambda x\ y : \mathsf{nat}.\ x$, and h will normalize to $\lambda z : \mathsf{nat}.\ z$.

The suspended substitution and the contextual type are the tools that the unification algorithm uses to know how to instantiate the meta-variable. Explaining the complex process of unification in a language like Coq will be the main topic of Chapter 4, but we can for the moment provide an intuitive hint: when Coq faces a problem of the form

$$?u[y_1, \ldots, y_n] \approx e$$

where the $y_1, \ldots, y_n$ are all distinct variables, then the *most general* solution to the problem is to *invert* the substitution and apply it on the right-hand side of the equation (Miller, 1991b), in other words instantiating $?u$ with $e\{x_1/y_1, \ldots, x_n/y_n\}$, where $x_1, \ldots, x_n$ are the variables in the local context of $?u$ (and assuming the free variables of e are in $\{y_1, \ldots, y_n\}$).

In the example above, at the point where Coq tries to unify $?u[x] \approx x$, the solution (through inversion) is to instantiate $?u$ with $x\{w/x\}$, that is, w.

1.2.2 Semantics

Reduction of CIC terms is performed through a set of rules listed in Figure 1.3. Besides the standard β rule, CIC provides six more rules to destruct the different term constructions: the ζ rule, which expands let-definitions, three δ rules, which expand definitions from each of the contexts, and two ι rules, which evaluate pattern **match**ings and **fix**points.

Most of the rules are self explanatory, with the sole exception of the $\delta\Sigma$ rule. It takes a meta-variable $?x$, applied to suspended substitution σ, and replaces it by its definition t, replacing each variable from its local context Ψ by the corresponding term from substitution σ. For this we use the multi-substitution of terms, mapping the variables coming from the domain of Ψ with terms in σ. To obtain the domain of Ψ, we use the type-eraser function $\widehat{\cdot}$, defined as:

$$\widehat{x_1 : T_1, \ldots, x_n : T_n} \triangleq x_1, \ldots, x_n$$

The *unfolding* rules ($\delta\Gamma$, $\delta\Sigma$, δE), of course, depend on the contexts. As customary, we will always consider the environment E implicit. We will also omit Γ and Σ when there is no room for ambiguity.

Another important rule is η-expansion, which takes a term t with functional type $\forall x : T.\ U$ and expands it into $\lambda y : T.\ (t\ y)$ (for y a fresh variable). Note that this rule, unlike the rest of the rules in Figure 1.3, requires knowledge of the type of t.

The set of rules described so far allows us to define a concept widely used in this thesis:

Definition 1.1 (Convertibility)**.** Two terms t_1 and t_2 are *convertible*, noted $\Sigma; \Gamma \vdash t_1 \equiv t_2$, if they reduce to the same normal form using the $\beta, \delta, \eta, \iota, \zeta$-rules. As with the reduction rules, we will often omit the contexts, and write $t_1 \equiv t_2$ instead. In some cases we will be explicit about the rules employed in the normalization of the terms, noting for instance $t_1 \equiv_{\beta\delta} t_2$ for the β, δ-*conversion* of t_1 and t_2.

1.2.3 Statics

We provide the typing rules for reference, although a deep understanding of the rules will not be necessary. The typing rules are similar to those found in The Coq Development Team (2012), with the addition of rules for meta-variables similar to those found in Pientka and Pfenning (2003).

Figure 1.4 shows the rules for checking well-formedness of the contexts. The environment cannot depend on local variables or on meta-variables, so the judgment is simply

$$\vdash E$$

In turn, the meta-variables in the meta-context may refer only to constants and inductive definitions from the environment:

$$E \vdash \Sigma$$

$$\frac{}{\vdash \cdot}\ \text{E-Empty} \qquad \frac{E;\emptyset;\emptyset \vdash t : T \qquad c \notin E}{\vdash E, c := t : T}\ \text{E-Def}$$

$$\frac{E;\emptyset;\emptyset \vdash T : s \qquad s \in \mathcal{S} \qquad c \notin E}{\vdash E, c : T}\ \text{E-Ass}$$

$$\frac{U_k^j = \forall \Gamma_j.\ i_j\ \overline{t} \qquad \overset{\forall j,k \qquad E;\emptyset;\Gamma_0 \vdash T_j : s}{E;\emptyset;\Gamma_0, \overline{i : T}, \Gamma_j \vdash i_j\ \overline{t} : s} \qquad \overline{i} \text{ satisfy the positivity condition}}{\vdash E,\ \forall \Gamma_0.\ \{\ \overline{i : T := \{k_1 : U_1; \ldots; k_m : U_m\}}\ \}}\ \text{E-Ind}$$

$$\frac{\vdash E}{E \vdash \cdot}\ \text{M-Empty} \qquad \frac{E;\Sigma;\Psi \vdash T : s \qquad s \in \mathcal{S} \qquad ?x \notin \Sigma}{E \vdash \Sigma, ?x : T[\Psi]}\ \text{M-Unk}$$

$$\frac{E;\Sigma;\Psi \vdash t : T \qquad ?x \notin \Sigma}{E \vdash \Sigma, ?x := t : T[\Psi]}\ \text{M-Def} \qquad \frac{E \vdash \Sigma}{E;\Sigma \vdash \cdot}\ \text{C-Empty}$$

$$\frac{E;\Sigma;\Gamma \vdash T : s \qquad s \in \mathcal{S} \qquad x \notin \Gamma}{E;\Sigma \vdash \Gamma, x : T}\ \text{C-Abs} \qquad \frac{E;\Sigma;\Gamma \vdash t : T \qquad x \notin \Gamma}{E;\Sigma \vdash \Gamma, x := t : T}\ \text{C-Let}$$

FIGURE 1.4: Well-formed judgment for contexts.

Finally, the local context may depend on both the environment and the meta-context:

$$E;\Sigma \vdash \Gamma$$

Figure 1.5 shows the rules for the typing judgment for terms, which has the standard form:

$$E;\Sigma;\Gamma \vdash t : T$$

1.3 Canonical Structures

Canonical structures is a powerful overloading mechanism, heavily used in large formalization efforts like the Feit-Thompson Theorem (Gonthier et al., 2013a), in order to make notations and proofs tractable. We will see in Chapter 2 that canonical structures can be cleverly used also for proof automation. This section is an introduction to canonical structures, starting with a canonical—in the sense of common—example and concluding with an overview of its semantics. In Chapter 4 we will provide a detailed description of its semantics.

We have to note that, in the literature and everyday use of Coq, the word "structure" is used interchangeably (and confusingly) to mean both dependent records *and* the types

$$\frac{E;\Sigma \vdash \Gamma}{E;\Sigma;\Gamma \vdash \mathsf{Prop} : \mathsf{Type}(1)}\ \textsc{Ax1} \qquad \frac{E;\Sigma \vdash \Gamma \qquad i < j}{E;\Sigma;\Gamma \vdash \mathsf{Type}(i) : \mathsf{Type}(j)}\ \textsc{Ax2}$$

$$\frac{\vdash E \qquad \forall \Gamma_0.\ \{\ldots; i : T := \ldots; \ldots\} \in E}{E;\Sigma;\Gamma \vdash i : \forall \Gamma_0.\ T}\ \textsc{Ax3}$$

$$\frac{\vdash E \qquad \forall \Gamma_0.\ \{\ldots; i : T := \{\ldots; k : U; \ldots\}; \ldots\} \in E}{E;\Sigma;\Gamma \vdash k : \forall \Gamma_0.\ U}\ \textsc{Ax4}$$

$$\frac{E;\Sigma \vdash \Gamma \qquad (x : T) \in \Gamma \text{ or } (x := t : T) \in \Gamma}{E;\Sigma;\Gamma \vdash x : T}\ \textsc{Var}$$

$$\frac{E;\Sigma \vdash \Gamma \qquad (c : T) \in E \text{ or } (c := t : T) \in E}{E;\Sigma;\Gamma \vdash c : T}\ \textsc{Const}$$

$$\frac{E;\Sigma;\Gamma \vdash T : s \qquad s \in \mathcal{S} \qquad E;\Sigma;\Gamma, x : T \vdash U : \mathsf{Prop}}{E;\Sigma;\Gamma \vdash \forall x : T.\ U : \mathsf{Prop}}\ \textsc{Prod1}$$

$$\frac{E;\Sigma;\Gamma \vdash T : \mathsf{Type}(i) \qquad i \leq k \qquad E;\Sigma;\Gamma, x : T \vdash U : \mathsf{Type}(j) \qquad j \leq k}{E;\Sigma;\Gamma \vdash \forall x : T.\ U : \mathsf{Type}(k)}\ \textsc{Prod2}$$

$$\frac{E;\Sigma;\Gamma, x : T \vdash t : U \qquad E;\Sigma;\Gamma \vdash \forall x : T.\ U : s}{E;\Sigma;\Gamma \vdash \lambda x : T.\ t : \forall x : T.\ U}\ \textsc{Lam}$$

$$\frac{E;\Sigma;\Gamma \vdash u : U \qquad E;\Sigma;\Gamma \vdash t : \forall x : U.\ T}{E;\Sigma;\Gamma \vdash t\ u : T\{u/x\}}\ \textsc{App}$$

$$\frac{E;\Sigma;\Gamma \vdash u : U \qquad E;\Sigma;\Gamma, x := u : U \vdash t : T}{E;\Sigma;\Gamma \vdash \mathbf{let}\ x := u : U\ \mathbf{in}\ t : T\{u/x\}}\ \textsc{Let}$$

$$\frac{\begin{array}{c} E;\Sigma \vdash \Gamma \\ (?x : T[\Psi]) \in \Sigma\ \vee\ (?x := t : T[\Psi]) \in \Sigma \qquad E;\Sigma;\Gamma \vdash \sigma : \Psi \end{array}}{E;\Sigma;\Gamma \vdash ?x[\sigma] : T\{\sigma/\widehat{\Psi}\}}\ \textsc{Metavar}$$

$$\frac{\begin{array}{c} E;\Sigma;\Gamma \vdash t : i\ \overline{t'} \qquad \forall \Gamma_0.\ \{\ldots; i : T := \{\overline{k : \forall \Gamma'.\ U}\}; \ldots\} \in E \\ \forall j \qquad E;\Sigma;\Gamma, \Gamma_0, \Gamma'_j \vdash u_j : \rho\ \overline{t'}\ (k_j\ \widehat{\Gamma'_j}) \end{array}}{E;\Sigma;\Gamma \vdash \mathbf{match}_\rho\ t\ \mathbf{with}\ \overline{k\ \overline{x} \Rightarrow u}\ \mathbf{end} : \rho\ \overline{t'}\ t}\ \textsc{Case}$$

$$\frac{\begin{array}{c} \forall l \qquad E;\Sigma;\Gamma \vdash T_l : s_l \qquad E;\Sigma;\Gamma, \overline{f : T} \vdash t_l : A_l \\ t_l \text{ satisfy the guarded condition} \end{array}}{E;\Sigma;\Gamma \vdash \mathbf{fix}_j\ \{\overline{f/n : T := t}\} : T_j}\ \textsc{Fix}$$

$$\frac{E;\Sigma;\Gamma \vdash t : T \qquad T \equiv U}{E;\Sigma;\Gamma \vdash t : U}\ \textsc{Check-Conv}$$

FIGURE 1.5: Typing judgment for CIC with meta-variables.

$$\frac{}{E;\Sigma;\Gamma \vdash \cdot : \cdot}\ \textsc{Id} \qquad \frac{E;\Sigma;\Gamma \vdash \sigma : \Psi \qquad E;\Sigma;\Gamma \vdash t : T\{\sigma/\widehat{\Psi}\}}{E;\Sigma;\Gamma \vdash (\sigma, t) : (\Psi, x : T)}\ \textsc{Assumption}$$

$$\frac{E;\Sigma;\Gamma \vdash \sigma : \Psi \qquad E;\Sigma;\Gamma \vdash t : T\{\sigma/\widehat{\Psi}\} \qquad t =_{\beta\zeta\delta} u\{\sigma/\widehat{\Psi}\}}{E;\Sigma;\Gamma \vdash (\sigma, t) : (\Psi, x := u : T)}\ \textsc{Definition}$$

FIGURE 1.6: Typing judgment for substitutions.

they inhabit. To disambiguate, in this thesis we use *structure* for the type, *instance* for the value, and *canonical instance* for a canonical value of a certain type. We will use the term *canonical structures* only when referring generally to the use of all of these mechanisms in tandem.

The following definition is a simplified example of a structure (*i.e.*, type) taken from the standard Ssreflect library (Gonthier and Mahboubi, 2010):

$$\begin{aligned}\textbf{Structure}\ \mathsf{eqType} := \mathsf{EqType}\ \{\ & \mathsf{sort} : \mathsf{Type};\\ & \mathsf{equal} : \mathsf{sort} \to \mathsf{sort} \to \mathsf{bool};\\ & \_ : \forall x\ y : \mathsf{sort}.\ \mathsf{equal}\ x\ y \leftrightarrow x = y\ \}\end{aligned}$$

The definition makes eqType a record type, with EqType as its constructor, taking three arguments: a type sort, a boolean binary operation equal on sort, and a proof that equal decides the equality on sort. For example, one possible eqType *instance* for the type bool, may be

$$\mathsf{eqType\_bool} := \mathsf{EqType}\ \mathsf{bool}\ \mathsf{eq\_bool}\ \mathsf{pf\_bool}$$

where $\mathsf{eq\_bool}\ x\ y := (x\ \&\&\ y)\ ||\ (!x\ \&\&\ !y)$, and pf_bool is a proof, omitted here, that $\forall x\ y : \mathsf{bool}.\ \mathsf{eq\_bool}\ x\ y \leftrightarrow x = y$. (Note that, in Coq, although it may seem as though EqType is declared as taking a single record argument with three components, applications of EqType pass the three arguments in curried style.)

The labels for the record fields serve as projections out of the record, so the definition of eqType also introduces the constants:

$$\begin{aligned}\mathsf{sort} &: \mathsf{eqType} \to \mathsf{Type}\\ \mathsf{equal} &: \forall A{:}\mathsf{eqType}.\ \mathsf{sort}\ A \to \mathsf{sort}\ A \to \mathsf{bool}\end{aligned}$$

We do not care to project out the proof component of the record, so we declare it anonymous by naming it with an underscore.

Notational Convention 3. We will usually omit the argument A of equal, and write $\mathsf{equal}\ t\ u$ instead of $\mathsf{equal}\ T\ t\ u$, as T can be inferred from the types of t and u. We

use the same convention for other functions as well, and make implicit such arguments that can be inferred from the types of other arguments. This is a standard notational convention in Coq.

It is also very useful to define *generic* instances. For example, consider the eqType instance for the pair type $A \times B$, where A and B are themselves instances of eqType:

$$\begin{aligned}&\mathsf{eqType\_pair}\ (A\ B : \mathsf{eqType}) :=\\ &\quad \mathsf{EqType}\ (\mathsf{sort}\ A \times \mathsf{sort}\ B)\ (\mathsf{eq\_pair}\ A\ B)\ (\mathsf{pf\_pair}\ A\ B)\end{aligned}$$

where

$$\begin{aligned}&\mathsf{eq\_pair}\ (A\ B : \mathsf{eqType})\ (u\ v : \mathsf{sort}\ A \times \mathsf{sort}\ B) :=\\ &\quad \mathsf{equal}\ (\pi_1\ u)\ (\pi_1\ v)\ \&\&\ \mathsf{equal}\ (\pi_2\ u)\ (\pi_2\ v)\end{aligned}$$

and pf_pair is a proof, omitted just like pf_bool above, that $\forall A\ B : \mathsf{eqType}.\, \forall x\ y : (\mathsf{sort}\ A \times \mathsf{sort}\ B).\, \mathsf{eq\_pair}\ x\ y \leftrightarrow x = y$.

Declaring both eqType_bool and eqType_pair now as *canonical instances*—using Coq's **Canonical** keyword—will have the following effect: whenever the type checker is asked to type a term like equal (b_1, b_2) (c_1, c_2), where b_1, b_2, c_1 and c_2 are of type bool, it will generate a unification problem matching the expected and inferred type of the first non-implicit argument of equal, that is,

$$\mathsf{sort}\ ?e \approx \mathsf{bool} \times \mathsf{bool}$$

for some meta-variable $?e$, generated implicitly at the application of equal.

Notational Convention 4. In some sections or chapter we will not bother with contextual types for meta-variables, as they play no role in the examples therein. In these cases we will overload the notation $?x$ to refer to both the meta-variable and the application of the meta-variable to an implicit suspended substitution.

In the equation above, Coq will try to solve this problem using the canonical instance eqType_pair, resulting in two new unification subproblems, for fresh meta-variables $?A$ and $?B$:

$$\mathsf{sort}\ ?A \approx \mathsf{bool} \qquad \mathsf{sort}\ ?B \approx \mathsf{bool}$$

Next, it will choose $?A :=$ eqType_bool and $?B :=$ eqType_bool, with the final result that equal (b_1, b_2) (c_1, c_2) reduces implicitly to eq_bool b_1 c_1 && eq_bool b_2 c_2, as one would expect.

In this manner, canonical instances can be used for *overloading*, similar to the way type classes are used in Haskell (Hall et al., 1996, Wadler and Blott, 1989).[3] We can declare a number of canonical eqType instances, for various primitive types, as well as generic instances for type constructors (like the pair example above). Then we can uniformly write equal t u, and the typechecker will compute the canonical implementation of equality at the types of t and u by solving for equal's implicit argument A.

1.3.1 Formalities

In Coq, a *structure* is not a primitive element of the language; it is encoded as a particular inductive type: it has only one constructor, and it generates one projector for each argument of the constructor. The general syntax is

$$\textbf{Structure}\ i\ \Gamma_0 : s := k\ \{p_1 : U_1; \ldots; p_n : U_n\}$$

where Γ_0 is a list of *arguments* of the type. Each p_j is a *projector* name. This language construct generates an inductive type

$$\{\ i : \forall \Gamma_0.\ s := \{\ k : \forall \Gamma_0.\ \forall \overline{p : U}.\ i\ \widehat{\Gamma_0}\ \}\ \}$$

and for each projector *name* p_j it generates a projector *function*:

$$\lambda \Gamma_0.\ \lambda z : i\ \widehat{\Gamma_0}.\ \textbf{match}\ s\ \textbf{with}\ k\ x_1\ \ldots\ x_j\ \ldots\ x_n \Rightarrow x_j\ \textbf{end}\ :\ \forall \Gamma_0.\ \forall z : i\ \widehat{\Gamma_0}.\ U_j$$

An instance ι of the structure is created with the constructor k:

$$\iota := \forall \Gamma_1.\ k\ t_1\ \ldots\ t_{m+n}$$

where m is the number of arguments of the structure. Terms t_1 to t_m correspond to the arguments of the structure, and t_{m+1} to t_{m+n} to each of the p_j.

As we saw in the previous section, the important aspect of structures is that their instances can be deemed "canonical". A canonical instance instructs the unification algorithm to instantiate a structure meta-variable with the instance, if certain conditions hold. More precisely, a canonical instance populates the canonical instance database Δ_{db} with triples (p_j, h_j, ι), where h_j is the head constant appearing in value t_{m+j}. (As a matter of fact, h_j can also be an implication ($\to$), a sort, or a variable.) Then, whenever

[3] It is worth noting that Coq also provides a built-in *type class* mechanism, but this feature is independent of canonical structures. We discuss Coq type classes more in Section 5.

the unification algorithm has to solve a problem of the form

$$p_j\ ?s \approx h_j\ \overline{u} \tag{1.2}$$

it instantiates $?s$ with ι.

For instance, in the unification equation generated by the equal application above, the projector p_j is sort, the head constant h_j is $(\cdot \times \cdot)$, and the head constant arguments $\overline{u}$ are bool and bool.

If the head symbol of the field p_j in instance ι is a *variable*, then ι is chosen for unification with $?s$, irrespectively of h in equation (1.2). In this case, we refer to ι as a *default* canonical instance for p_j.

We emphasize that: (1) to control the number of (p_j, h_j)-pairs that the typechecker has to remember, we will frequently anonymize the projections if they are not important for the application, as in the case of the proof component in eqType above; (2) there can only be one specified canonical instance for any given p_j and h_j. In particular, overlapping canonical instances for the same p_j and h_j are not permitted. This limitation, however, can be easily circumvented, as we will see shortly in §2.1.

Chapter 2

Lemma Overloading

In this chapter we explore the first of the two tactic programming approaches that we explore in this thesis. We show how to use canonical structures to enable a sort of dependently typed logic meta-programming in Coq that we call "Lemma Overloading". As we will see in Section 2.1, this style of programming requires *overlapping instances*, which are not allowed in Coq. Despite this limitation, we are able to encode overlapping instance using a very simple *design pattern* that we introduce in Section 2.2. With this pattern we are able to encode the first *overloaded lemma* in Section 2.3. We devote Section 2.4 to provide an interesting example involving heap cancellation that uses heavily this design pattern. Since canonical structures lies at the heart of unification, we are able to exploit unification and write a rather powerful pattern involving higher-order unification, which allows us to greatly improve the automation of the hoare-style rules used in HTT (Section 2.5). However, composing procedures can be sometimes challenging with Lemma Overloading. We present in Section 2.6 a problematic example, and a design pattern to solve this type of examples. We close the chapter with an analysis of Lemma Overloading (Section 2.7).

2.1 "Logic" Programming

Although the eqType example from Section 1.3 is typical of how canonical structures are used in much existing Coq code, it is not actually representative of the style of canonical structure programming that we explore in this thesis. Our idiomatic style is closer in flavor to logic programming and relies on the fact that, unlike in Haskell, the construction of canonical instances in Coq can be guided not only by the structure of types (such as the sort projection of eqType) but by the structure of *terms* as well.

To make matters concrete, let's consider a simple automation task, one which we will employ gainfully in Section 2.3 when we present our first "overloaded lemma". We will first present a naïve approach to solving the task, which *almost* works; the manner in which it fails will motivate our first "design pattern" (Section 2.2).

The task is as follows: search for a pointer x in the domain of a heap h. If the search is successful, that is, if h is of the form

$$\cdots \bullet (\cdots \bullet x \mapsto v \bullet \cdots) \bullet \cdots ,$$

then return a proof that $x \in \mathsf{dom}\ h$. To solve this task using canonical structures, we will first define a structure find:

$$\begin{aligned} \textbf{Structure}\ \mathsf{find}\ x := \mathsf{Find}\ \{\ & \mathsf{heap\_of} : \mathsf{heap}; \\ & \_ : \mathsf{spec}\ x\ \mathsf{heap\_of}\ \} \end{aligned}$$

where spec is defined as

$$\mathsf{spec}\ x\ h := \mathsf{def}\ h \to x \in \mathsf{dom}\ h$$

The first thing to note here is that the structure find is *parameterized* by the pointer x (causing the constructor Find to be implicitly parameterized by x as well). This is a common idiom in canonical structure programming—and we will see that structure parameters can be used for various different purposes—but here x may be viewed simply as an "input" to the automation task. The second thing to note here is that the structure has no type component, only a $\mathsf{heap\_of}$ projection, together with a proof that $x \in \mathsf{dom\ heap\_of}$ (under the assumption that $\mathsf{heap\_of}$ is well-defined).

The search task will commence when some input heap h gets unified with $\mathsf{heap\_of}\ ?f$ for an unknown $?f : \mathsf{find}\ x$, at which point Coq's unification algorithm will recursively deconstruct h in order to search for a canonical implementation of $?f$ such that $\mathsf{heap\_of}\ ?f = h$. If that search is successful, the last field of $?f$ will be a proof of $\mathsf{spec}\ x\ h$, which we can apply to a proof of $\mathsf{def}\ h$ to obtain a proof of $x \in \mathsf{dom}\ h$, as desired. (By way of analogy, this is similar to what we previously did for $\mathsf{eqType\_pair}$. The construction of a canonical equality operator at a given type T will commence when T is unified with $\mathsf{sort}\ ?e$ for an unknown $?e : \mathsf{eqType}$, and the unification algorithm will proceed to solve for $?e$ by recursively deconstructing T and composing the relevant canonical instances.)

The structure find provides a formal specification of what a successful completion of the search task will produce, but now we need to actually implement the search. We do that by defining several canonical instances of find corresponding to the different cases of the

recursive search, and relying on Coq's unification algorithm to perform the recursion:

$$\textbf{Canonical}\ \mathsf{found\_struct}\ A\ x\ (v : A) :=$$
$$\quad \mathsf{Find}\ x\ (x \mapsto v)\ (\mathsf{found\_pf}\ A\ x\ v)$$

$$\textbf{Canonical}\ \mathsf{left\_struct}\ x\ h\ (f : \mathsf{find}\ x)\ :=$$
$$\quad \mathsf{Find}\ x\ ((\mathsf{heap\_of}\ f) \bullet h)\ (\mathsf{left\_pf}\ x\ h\ f)$$

$$\textbf{Canonical}\ \mathsf{right\_struct}\ x\ h\ (f : \mathsf{find}\ x)\ :=$$
$$\quad \mathsf{Find}\ x\ (h \bullet (\mathsf{heap\_of}\ f))\ (\mathsf{right\_pf}\ x\ h\ f)$$

Note that the first argument to the constructor Find in these instances is the parameter x of the find structure.

The first instance, found_struct, corresponds to the case where the heap_of projection is a singleton heap whose domain contains precisely the x we're searching for. (If the heap is $y \mapsto v$ for $y \neq x$, then unification fails.) The second and third instances, left_struct and right_struct, handle the cases where the heap_of projection is of the form $h_1 \bullet h_2$, and x is in the domain of h_1 or h_2, respectively. Note that the recursive nature of the search is implicit in the fact that the latter two instances are parameterized by instances $f : \mathsf{find}\ x$ whose heap_of projections are unified with the subheaps h_1 or h_2 of the original heap_of projection.

Notational Convention 5. In the declarations above, found_pf, left_pf and right_pf are proofs, witnessing that spec relates x and the appropriate heap expression. We omit the proofs here, but they are available in our source files (Ziliani, 2014). From now on, we omit writing such explicit proofs in instances, and simply replace them with "...", as in: $\mathsf{Find}\ x\ ((\mathsf{heap\_of}\ f) \bullet h)\ \ldots$

Unfortunately, this set of canonical instances does not quite work. The trouble is that left_struct and right_struct are overlapping instances since both match against the same head symbol (namely, $\bullet$), and overlapping instances are not permitted in Coq. Moreover, even if overlapping instances were permitted, we would still need some way to tell Coq that it should try one instance first and then, if that fails, to backtrack and try another. Consequently, we need some way to deterministically specify the order in which overlapping instances are to be considered. For this, we introduce our first design pattern.

2.2 Tagging: A Technique for Ordering Canonical Instances

Our approach to ordering canonical instances is, in programming terms, remarkably simple. However, understanding why it actually works is quite tricky because its success relies critically on an aspect of Coq's unification algorithm that (a) is not well known, and (b) diverges significantly from how unification works in, say, Haskell. We will thus first illustrate the pattern concretely in terms of our find example, and then explain afterwards how it solves the problem.

The Pattern: First, we define a "tagged" version of the type of thing we're recursively analyzing—in this case, the heap type:

$$\textbf{Structure}\ \mathsf{tagged\_heap} := \mathsf{Tag}\ \{\mathsf{untag} : \mathsf{heap}\}$$

This structure declaration also introduces two functions witnessing the isomorphism between heap and tagged_heap:

$$\begin{array}{lcl} \mathsf{Tag} & : & \mathsf{heap} \to \mathsf{tagged\_heap} \\ \mathsf{untag} & : & \mathsf{tagged\_heap} \to \mathsf{heap} \end{array}$$

Then, we modify the find structure to carry a tagged_heap instead of a plain heap, *i.e.*, we declare

$$\begin{array}{l} \mathsf{spec}\ x\ (h : \mathsf{tagged\_heap}) := \\ \quad \mathsf{def}\ (\mathsf{untag}\ h) \to x \in \mathsf{dom}\ (\mathsf{untag}\ h) \end{array}$$

$$\begin{array}{l} \textbf{Structure}\ \mathsf{find}\ x := \mathsf{Find}\ \{\mathsf{heap\_of} : \mathsf{tagged\_heap}; \\ \qquad\qquad\qquad\qquad \_ : \mathsf{spec}\ x\ \mathsf{heap\_of}\} \end{array}$$

Next, we define a sequence of *synonyms* for Tag, one for each canonical instance of find. Importantly, we define the tag synonyms in the *reverse* order in which we want the canonical instances to be considered during unification, and we make the *last* tag synonym in the sequence be *the* canonical instance of the tagged_heap structure itself. (The order does not matter much in this particular example, but it does in other examples in the paper.)

$$\begin{array}{l} \mathsf{right\_tag}\ h := \mathsf{Tag}\ h \\ \mathsf{left\_tag}\ h := \mathsf{right\_tag}\ h \\ \textbf{Canonical}\ \mathsf{found\_tag}\ h := \mathsf{left\_tag}\ h \end{array}$$

Notice that found_tag is a *default instance* for the untag projector matching any heap h.

Finally, we modify each canonical instance so that its heap_of projection is wrapped with the corresponding tag synonym.

Canonical found_struct $A\ x\ (v : A) :=$
  Find x (found_tag $(x \mapsto v)$) ...

Canonical left_struct $x\ h\ (f : \mathsf{find}\ x)\ :=$
  Find x (left_tag ((untag (heap_of f)) $\bullet\ h$)) ...

Canonical right_struct $x\ h\ (f : \mathsf{find}\ x)\ :=$
  Find x (right_tag $(h \bullet$ (untag (heap_of f)))) ...

The Explanation: The key to the tagging pattern is that, by employing different tags for each of the canonical instance declarations, we are able to syntactically differentiate the head constants of the heap_of projections, thereby circumventing the need for overlapping instances. But the reader is probably wondering: (1) how can semantically equivalent tag synonyms differentiate anything? and (2) what's the deal with defining them in the reverse order?

The answer to (1) is that Coq does *not* unfold *all* definitions automatically during the unification process—it only unfolds the definition of a term like found_tag h automatically if that term is unified with something else and the unification fails (see the next paragraph). This stands in contrast to Haskell type inference, which implicitly expands all (type) synonyms right away. Thus, even though found_tag, left_tag, and right_tag are all semantically equivalent to Tag, the unification algorithm can distinguish between them, rendering the three canonical instances of find non-overlapping.

The answer to (2) is as follows. By making the last tag synonym found_tag the sole canonical instance of tagged_heap, we guarantee that unification always pattern-matches against the found_struct case of the search algorithm first before any other. To see this, observe that the execution of the search for x in h will get triggered when a unification problem arises of the form

$$\mathsf{untag}\ (\mathsf{heap\_of}\ ?f) \approx h,$$

for some unknown $?f$: find x. Since found_tag is a default canonical instance, the problem will be reduced to unifying

$$\mathsf{heap\_of}\ ?f \approx \mathsf{found\_tag}\ h$$

As found_struct is the only canonical instance of find whose heap_of projection has found_tag as its head constant, Coq will first attempt to unify $?f$ with some instantiation of found_struct. If h is a singleton heap containing x, then the unification will

succeed. Otherwise, Coq will backtrack and try unfolding the definition of found_tag h instead, resulting in the new unification problem

$$\mathsf{heap\_of}\ ?f \approx \mathsf{left\_tag}\ h,$$

which will in turn cause Coq to try unifying $?f$ with some instantiation of left_struct. If that fails again, left_tag h will be unfolded to right_tag h and Coq will try right_struct. If in the end that fails as well, then it means that the search has failed to find x in h, and Coq will correctly flag the original unification problem as unsolvable.

2.3 A Simple Overloaded Lemma

Let us now attempt our first example of lemma overloading, which makes immediate use of the find structure that we developed in the previous section. First, here is the *non*-overloaded version:

$$\begin{aligned}\mathsf{indom} \ : \ & \forall A{:}\mathsf{Type}.\, \forall x{:}\mathsf{ptr}.\, \forall v{:}A.\, \forall h{:}\mathsf{heap}. \\ & \mathsf{def}\ (x \mapsto v \bullet h) \to x \in \mathsf{dom}\ (x \mapsto v \bullet h)\end{aligned}$$

The indom lemma is somewhat simpler than noalias from the introduction, but the problems in applying them are the same—neither lemma is applicable unless its heap expressions are of a special syntactic form, with the relevant pointer(s) at the top of the heap.

To lift this restriction, we will rephrase the lemma into the following form:

$$\begin{aligned}\mathsf{indomR} \ : \ & \forall x{:}\mathsf{ptr}.\, \forall f{:}\mathsf{find}\ x. \\ & \mathsf{def}\ (\mathsf{untag}\ (\mathsf{heap\_of}\ f)) \to \\ & \quad x \in \mathsf{dom}\ (\mathsf{untag}\ (\mathsf{heap\_of}\ f))\end{aligned}$$

The lemma is now parameterized over an instance f of structure find x, which we know—just from the definition of find alone—contains within it a heap $h = \mathsf{untag}\ (\mathsf{heap\_of}\ f)$, together with a proof of $\mathsf{def}\ h \to x \in \mathsf{dom}\ h$. Based on this, it should come as no surprise that the proof of indomR is trivial (it's a half-line long in Ssreflect). In fact, the lemma is really just the projection function corresponding to the unnamed spec component from the find structure, much as the overloaded equal function from Section 1.3 is a projection function from the eqType structure.

2.3.1 Applying the Lemma: The High-Level Picture

To demonstrate the automated nature of indomR on a concrete example, we will explain how Coq type inference proceeds when indomR is applied to prove the goal

$$z \in \mathsf{dom}\ h$$

in a context where $x\ y\ z : \mathsf{ptr}$, $u\ v\ w : A$ for some type A, $h : \mathsf{heap} := x \mapsto u \bullet y \mapsto v \bullet z \mapsto w$, and $D : \mathsf{def}\ h$. For the moment, we will omit certain details for the sake of clarity; in the next subsection, we give a much more detailed explanation.

To begin with, let us first explain the steps involved in the application of a lemma to some goal, and how this produces the equation needed to solve the instance f of the structure.

When a lemma (*e.g.*, indomR) is applied to a goal, the following process takes place:

1. The lemma's formal parameters are turned into meta-variables $?x$ and $?f : \mathsf{find}\ ?x$, which will be subsequently constrained by the unification process.
2. The lemma's conclusion $?x \in \mathsf{dom}\ (\mathsf{untag}\ (\mathsf{heap\_of}\ ?f))$ is unified with the goal.

Given this last step, the system tries to unify

$$?x \in \mathsf{dom}\ (\mathsf{untag}\ (\mathsf{heap\_of}\ ?f)) \approx z \in \mathsf{dom}\ h$$

solving subproblems from left to right, that is, first getting $?x = z$, and then

$$\mathsf{untag}\ (\mathsf{heap\_of}\ ?f) \approx h$$

By canonicity of found_tag, it then tries to solve

$$\mathsf{heap\_of}\ ?f \approx \mathsf{found\_tag}\ h$$

Expanding the heap variable, and since $\bullet$ is left-associative, this is equivalent to

$$\mathsf{heap\_of}\ ?f \approx \mathsf{found\_tag}\ ((x \mapsto u \bullet y \mapsto v) \bullet z \mapsto w)$$

At this point, guided by the instances we defined in the previous section, the search for a canonical instantiation of $?f$ begins. Coq will first try to instantiate $?f$ with found_struct, but this attempt will fail when it tries to unify the entire heap with a singleton heap. Then, Coq will try instantiating $?f$ instead with left_struct, which leads it to create a

fresh variable $?f_2$: $\mathsf{find}\ z$ and recursively solve the equation $\mathsf{untag}\ (\mathsf{heap\_of}\ ?f_2) \approx x \mapsto u \bullet y \mapsto v$. This attempt will ultimately fail again, because z is not in $x \mapsto u \bullet y \mapsto v$, the left subheap of the original h. Finally, Coq will backtrack and try to instantiate $?f$ with $\mathsf{right\_struct}$, which will lead it to create a fresh variable $?f_3$: $\mathsf{find}\ z$ and recursively solve the equation $\mathsf{untag}\ (\mathsf{heap\_of}\ ?f_3) \approx z \mapsto w$. This final attempt *will* indeed succeed by instantiating $?f_3$ with $\mathsf{found\_struct}$, since the right subheap of h is precisely the singleton heap we are looking for.

Putting the pieces together, the unification algorithm instantiates $?f$ with

$$?f = \mathsf{right\_struct}\ z\ (x \mapsto u \bullet y \mapsto v)\ (\mathsf{found\_struct}\ z\ w)$$

Effectively, the $\mathsf{heap\_of}$ component of $?f$ contains the (tagged) heap h that was input to the search, and the proof component contains the output proof that z is in the domain of h.

2.3.2 The Gory Details

To understand better how it works, we will now spell out the "trace" of how Coq's unification algorithm implements proof search in the example above, with a particular emphasis on how it treats resolution of canonical instances. This knowledge is not critical for understanding most of the examples in the chapter—indeed, the whole point of our "design patterns" is to avoid the need for one to think about unification at this level of gory detail. But it will nonetheless be useful in understanding *why* the design patterns work, as well as how to control Coq's unification algorithm in more complex examples where the design patterns do not immediately apply.

Let us look again at the initial equation that started the search:

$$\mathsf{untag}\ (\mathsf{heap\_of}\ ?f) \approx h$$

As mentioned before, the canonicity of $\mathsf{found\_tag}$ reduces this to solving

$$\mathsf{heap\_of}\ ?f \approx \mathsf{found\_tag}\ h$$

Unification tries to instantiate $?f$ with $\mathsf{found\_struct}$, but for that it must unify the entire heap h with $z \mapsto ?v$, which fails. Before giving up, the system realizes it can unfold the definitions of h and of $\mathsf{found\_tag}$, yielding

$$\mathsf{heap\_of}\ ?f \approx \mathsf{left\_tag}\ ((x \mapsto u \bullet y \mapsto v) \bullet z \mapsto w) \tag{2.1}$$

With this unfolding, $?f$ has become eligible for instantiation by left_struct, since left_struct is the canonical instance of the find structure with head constant left_tag. To figure out if/whether this instantiation is possible, Coq will engage in the following procedure, which we will describe here quite carefully—and in as general a manner as possible—since it is important for understanding subsequent, more complex examples.

To instantiate $?f$ with left_struct, Coq first "opens" the right-hand side of the definition of left_struct by generating fresh unification variables for each of left_struct's formal parameters: $?y$: ptr, $?h$: heap, and $?f_2$: find $?y$. It will eventually unify $?f$ with left_struct $?y$ $?h$ $?f_2$, but *before* doing that, it must figure out how to marry together two sources of (hopefully compatible) information about $?f$: the unification goal (*i.e.*, Equation (2.1)) and the definition of left_struct. Each of these provides information about:

1. The type of the structure we are solving for (in this case, $?f$). From the initial unification steps described in Section 2.3.1, we already know that $?f$ must have type find z, while the type of left_struct $?y$ $?h$ $?f_2$ is find $?y$. This leads to the unification

$$?y \approx z$$

2. The (full) value of the projection in question (in this case, heap_of $?f$). From Equation (2.1), we know that heap_of $?f$ must equal left_tag $((x \mapsto u \bullet y \mapsto v) \bullet z \mapsto w)$, while from the definition of left_struct, we know that heap_of (left_struct $?y$ $?h$ $?f_2$) equals left_tag ((untag (heap_of $?f_2$)) $\bullet$ $?h$). This leads to the unification

$$\text{left\_tag}\ ((\text{untag}\ (\text{heap\_of}\ ?f_2)) \bullet ?h) \approx \text{left\_tag}\ ((x \mapsto u \bullet y \mapsto v) \bullet z \mapsto w),$$

   which in turn induces the following unification equations:

$$\begin{aligned} \text{untag}\ (\text{heap\_of}\ ?f_2) &\approx x \mapsto u \bullet y \mapsto v \\ ?h &\approx z \mapsto w \end{aligned}$$

Putting it all together, the attempt to instantiate $?f$ with left_struct generates the following unification problems, which Coq processes in order:

$$\begin{aligned} ?y &\approx z \\ \mathsf{untag}\ (\mathsf{heap\_of}\ ?f_2) &\approx x \mapsto u \bullet y \mapsto v \\ ?h &\approx z \mapsto w \\ ?f &\approx \mathsf{left\_struct}\ ?y\ ?h\ ?f_2 \end{aligned}$$

Note here that the order matters! For instance, the first equation will be resolved immediately, thus concretizing the type of $?f_2$ to find z. It is important that this happens *before* solving the second equation, so that when we attempt to solve the second equation we know what pointer (z) we are searching for in the heap $x \mapsto u \bullet y \mapsto v$. (Otherwise, we would be searching for the unification variable $?y$, which would produce senseless results.)

Attempting to solve the second equation, Coq again applies found_tag and found_struct and fails. Then, it unfolds found_tag to get left_tag and the following equation:

$$\mathsf{heap\_of}\ ?f_2 \approx \mathsf{left\_tag}\ (x \mapsto u \bullet y \mapsto v) \tag{2.2}$$

It attempts to instantiate $?f_2$ with left_struct, by the same procedure as described above, obtaining the following equations:

$$\begin{aligned} ?y' &\approx z \\ \mathsf{untag}\ (\mathsf{heap\_of}\ ?f_3) &\approx x \mapsto u \qquad\qquad \text{where } ?f_3 : \mathsf{find}\ ?y' \end{aligned}$$

After solving the first one, the attempt to solve the second one will fail. Specifically, Coq will first try to use found_struct as before; however, this will not work because, although the heap in question ($x \mapsto u$) is a singleton heap, the pointer in its domain is not z. Unfolding to left_struct and right_struct will not help because those solutions only apply to heaps with $\bullet$ as the top-level constructor.

Rolling back to Equation (2.2), Coq unfolds left_tag to right_tag and tries to instantiate $?f_2$ with right_struct. As before, it fails because z is not y.

Rolling back further to Equation (2.1), Coq unfolds left_tag to right_tag, resulting in

$$\mathsf{heap\_of}\ ?f \approx \mathsf{right\_tag}\ ((x \mapsto u \bullet y \mapsto v) \bullet z \mapsto w).$$

It then chooses right_struct and applies the same procedure as before to obtain the following equations.

$$\begin{aligned} ?x' &\approx z \\ ?h' &\approx x \mapsto u \bullet y \mapsto v \\ \mathsf{untag}\ (\mathsf{heap\_of}\ ?f_2') &\approx z \mapsto w \qquad \text{where } ?f_2' : \mathsf{find}\ ?x' \\ ?f &\approx \mathsf{right\_struct}\ ?x'\ ?h'\ ?f_2' \end{aligned}$$

The first two equations unify immediately, after which the third one is solved by applying found_tag and choosing found_struct for $?f_2'$. After that, the third equation also unifies right away, producing the final result

$$?f = \mathsf{right\_struct}\ z\ (x \mapsto u \bullet y \mapsto v)\ (\mathsf{found\_struct}\ z\ w)$$

It is a fair question to ask why, in case of failure of an equation like 2.1, Coq unfolds the right definition (the tag) instead of the heap_of projector. After all, both are defined constants, the latter being the definition of a pattern matching on the instance of the structure. As a quick answer, the unification algorithm performs a heuristic that makes projectors on unknown instances to not be unfolded. The full details can be found in Chapter 4.

2.4 Reflection: Turning Semantics into Syntax

As canonical structures are closely coupled with the type checker, it is possible to fruitfully combine the logic-programming idiom afforded by canonical structures together with ordinary functional programming in Coq. In this section, we illustrate the combination by developing a thoroughly worked example of an overloaded lemma for performing cancellation on heap equations using a technique known as *proof by reflection* (*e.g.*, Grégoire and Mahboubi (2005)), which exploits computation at the level of types to drastically reduce the size of proof terms, and therefore the time spent typechecking them.

Mathematically, cancellation merely involves removing common terms from disjoint unions on the two sides of a heap equation. For example, if we are given an equation

$$x \mapsto v_1 \bullet (h_3 \bullet h_4) = h_4 \bullet x \mapsto v_2$$

and we know that the subheaps are disjoint (*i.e.*, the unions are defined), then we can extract the implied equations

$$v_1 = v_2 \wedge h_3 = \mathsf{empty}$$

We will implement the lemma in two stages, as commonly done in proof by reflection. The first stage is an extra-logical program (in our case, a canonical structure program), which *reflects* the equation, that is, turns the equation on heap expressions into an abstract syntax tree (or abstract syntax *list*, as it will turn out). Then the second stage is a functional program, which cancels common terms from the syntax tree. Notice that the functional program from the second stage cannot work directly on the heap equation for two reasons: (1) it needs to compare heap and pointer variable names, and (2) it needs to pattern match on function names, since in HTT heaps are really partial maps from locations to values, and $\mapsto$ and $\bullet$ are merely convenient functions for constructing them. As neither of these is possible within Coq's base logic, the equation has to be reflected into syntax in the first stage. The main challenge then is in implementing reflection, so that the various occurrences of one and the same heap variable or pointer variable in the equation are ascribed the same syntactic representation.

2.4.1 Cancellation

Since the second stage is simpler, we explain it first. For the purposes of presentation, we begin by restricting our pointers to only store values of some predetermined type T (although in Section 2.4.3 we will generalize it to any type, as in our actual implementation). The data type that we use for syntactic representation of heap expressions is the following:

$$\begin{aligned} &\mathsf{elem} \;:=\; \mathsf{Var\ of\ nat} \;\mid\; \mathsf{Pts\ of\ nat}\ \&\ T \\ &\mathsf{term} \;:=\; \mathsf{seq\ elem} \end{aligned}$$

An *element* of type elem identifies a heap component as being either a heap variable or a points-to clause $x \mapsto v$. In the first case, the component is represented as Var n, where n is an index identifying the *heap* variable in some environment (to be explained below). In the second case, the component is represented as Pts m v, where m is an index identifying the *pointer* variable in an environment. We do not perform any reflection on v, as it is not necessary for the cancellation algorithm. A heap expression is then represented via term as a list (seq) of elements. We could have represented the original heap expression more faithfully as a tree, but since $\bullet$ is commutative and associative, lists suffice for our purposes.

We will require two kinds of environments, which we package into the type of *contexts*:

$$\mathsf{ctx} := \mathsf{seq\ heap} \times \mathsf{seq\ ptr}$$

The first component of a context is a list of heaps. In a term reflecting a heap expression, the element Var n stands for the n-th element of this list (starting from 0-th). Similarly, the second component is a list of pointers, and in the element Pts m v, the number m stands for the m-th pointer in the list.

Because we will need to verify that our syntactic manipulation preserves the semantics of heap operations, we need a function that *interprets* syntax back into semantics. Assuming lookup functions hlook and plook which search for an index in a context of a heap or pointer, respectively, the interpretation function crawls over the syntactic term, replacing each number index with its value from the context (and returning an undefined heap, if the index is out of bounds). The function is implemented as follows:

```
interp (i : ctx) (t : term) : heap :=
  match t with
    Var n :: t' ⇒ if hlook i n is Some h then h • interp i t'
                  else Undef
  | Pts m v :: t' ⇒
      if plook i m is Some x then x ↦ v • interp i t'
      else Undef
  | nil ⇒ empty
  end
```

For example, if the context i is $([h_3, h_4], [x])$, then

$$\begin{aligned} \mathsf{interp}\ i\ [\mathsf{Pts}\ 0\ v_1, \mathsf{Var}\ 0, \mathsf{Var}\ 1] &= x \mapsto v_1 \bullet (h_3 \bullet (h_4 \bullet \mathsf{empty})) \\ \mathsf{interp}\ i\ [\mathsf{Var}\ 1, \mathsf{Pts}\ 0\ v_2] &= h_4 \bullet (x \mapsto v_2 \bullet \mathsf{empty}) \end{aligned}$$

Given this definition of term, we can now encode the cancellation algorithm as a predicate (*i.e.*, a function into Prop) in Coq (Figure 2.1). The predicate essentially constructs a conjunction of the residual equations obtained as a consequence of cancellation. Referring to Figure 2.1, the algorithm works as follows. It looks at the head element of the left term t_1, and tries to find it in the right term t_2 (keying on the index of the element). If the element is found, it is removed from both sides, before recursing over the rest of t_1. When removing a Pts element keyed on a pointer x, the values v and v' stored into x in t_1 and t_2 must be equal. Thus, the proposition computed by cancel should contain an equation between these values as a conjunct. If the element is not found in t_2, it

```
cancel (i : ctx) (t1 t2 r : term) : Prop :=
  match t1 with
    [] ⇒ interp i r = interp i t2
  | Pts m v :: t1' ⇒
      if premove m t2 is Some (v', t2') then
        cancel i t1' t2' r ∧ v = v'
      else cancel i t1' t2 (Pts m v :: r)
  | Var n :: t1' ⇒
      if hremove n t2 is Some t2' then cancel i t1' t2' r
      else cancel i t1' t2 (Var n :: r)
  end
```

FIGURE 2.1: Heap cancellation algorithm.

is shuffled to the accumulator r, before recursing. When the term t_1 is exhausted, *i.e.*, it becomes the empty list, then the accumulator stores the elements from t_1 that were not cancelled by anything in t_2. The equation between the interpretations of r and t_2 is a semantic consequence of the original equation, so cancel immediately returns it (our actual implementation performs some additional optimization before returning). The helper function premove m t_2 searches for the occurrences of the pointer index m in the term t_2, and if found, returns the value stored into the pointer, as well as the term t_2' obtained after removing m from t_2. Similarly, hremove n t_2 searches for Var n in t_2 and if found, returns t_2' obtained from t_2 after removal of n.

Soundness of cancel is established by the following lemma which shows that the facts computed by cancel do indeed follow from the input equation between heaps, when cancel is started with the empty accumulator.

```
cancel_sound : ∀i : ctx. ∀t1 t2 : term.
  def (interp i t1) → interp i t1 = interp i t2 →
  cancel i t1 t2 []
```

The proof of cancel_sound is rather involved and interesting in its own right, but we omit it here not to distract the attention of the reader from our main topic. (The interested reader can find it in our source files (Ziliani, 2014).) We could have proved the converse direction as well, to obtain a completeness result, but this was not necessary for our purposes.

The related work on proofs by reflection usually implements the cancellation phase in a manner similar to that above (see for example the work of Grégoire and Mahboubi (2005)). Where we differ from the related work is in the implementation of the reflection phase. This phase is usually implemented by a tactic, but here we show that it can be implemented with canonical structures instead.

2.4.2 Reflection via Canonical Structures

Intuitively, the reflection algorithm traverses a heap expression, and produces the corresponding syntactic term. In our overloaded lemma, presented further below, we will invoke this algorithm twice, to reflect both sides of the equation. To facilitate cancellation, we need to ensure that identical variables on the two sides of the equation, call them E_1 and E_2, are represented by identical syntactic elements. Therefore, reflection of E_1 has to produce a context of newly encountered elements and their syntactic equivalents, which is then fed as an input to the reflection of E_2. If reflection of E_2 encounters an expression which is already in the context, the expression is reflected with the syntactic element provided by the context.

Notational Convention 6. Hereafter, projections out of an instance are considered *implicit coercions*, and we will typically omit them from our syntax. For example, in Figure 2.2 (described below), the canonical instance union_struct says union_tag $(f_1 \bullet f_2)$ instead of union_tag ((untag (heap_of f_1)) $\bullet$ (untag (heap_of f_2))), which is significantly more verbose. This is a standard technique in Coq.

The reflection algorithm is encoded using the structure ast from Figure 2.2. The inputs to each traversal are the initial context i of ast, and the initial heap in the heap_of projection. The output is the (potentially extended) context j and the syntactic term t that reflects the initial heap. One invariant of the structure is precisely that the term t, when interpreted under the output heap j, reflects the input heap:

$$\mathsf{interp}\ j\ t = \mathsf{heap\_of}$$

There are two additional invariants needed to carry out the proofs:

$$\mathsf{subctx}\ i\ j \quad \text{and} \quad \mathsf{valid}\ j\ t$$

The first one states that the output context j is an extension of the input context i, while the second one ensures that the syntactic term t has no indices out of the bounds of the output context j. (We omit the definition of subctx and valid, but they can be found in the source files (Ziliani, 2014).)

There are several cases to consider during a traversal, as shown by the canonical instances in Figure 2.2. We first check if the input heap is a union, as can be seen from the ordering of tag synonyms (which is reversed, as demanded by the tagging pattern). In this case, the canonical instance is union_struct. The instance specifies that we recurse over both subheaps, by unifying the left subheap with f_1 and the right subheap with f_2.

```
var_tag h := Tag h
pts_tag h := var_tag h
empty_tag h := pts_tag h
Canonical union_tag h := empty_tag h

Structure ast (i j : ctx) (t : term) :=
  Ast { heap_of : tagged_heap;
        _ : interp j t = heap_of ∧ subctx i j ∧ valid j t }

Canonical union_struct (i j k : ctx) (t1 t2 : term)
                       (f1 : ast i j t1)(f2 : ast j k t2) :=
  Ast i k (append t1 t2) (union_tag (f1 • f2)) ...

Canonical empty_struct (i : ctx) :=
  Ast i i [] (empty_tag empty) ...

Canonical pts_struct (hs : seq heap) (xs1 xs2 : seq ptr)
                     (m : nat) (v : A) (f : xfind xs1 xs2 m) :=
  Ast (hs, xs1) (hs, xs2) [Pts m v] (pts_tag (f ↦ v)) ...

Canonical var_struct (hs1 hs2 : seq heap) (xs : seq ptr)
                     (n : nat) (f : xfind hs1 hs2 n) :=
  Ast (hs1, xs) (hs2, xs) [Var n] (var_tag f) ...
```

FIGURE 2.2: Structure ast for reflecting a heap.

The types of f_1 and f_2 show that the two recursive calls work as follows. First the call to f_1 starts with the input context i and computes the output context j and term t_1. Then the call to f_2 proceeds with input context j, and computes outputs k and t_2. The output context of the whole union is k, and the output reflected term is the list-concatenation of t_1 and t_2.

When reflecting the empty heap, the instance is empty_struct. In this case, the input context i is simply returned as output, and the reflected term is the empty list.

When reflecting a singleton heap $x \mapsto v$, the corresponding instance is ptr_struct. In this case, we first have to check if x is a pointer that already appears in the pointer part xs_1 of the input context. If so, we should obtain the index m at which x appears in xs_1. This is the number representing x, and the returned reflected elem is Pts m v. On the other hand, if x does not appear in xs_1, we need to add it. We compute a new context xs_2 which appends x at the end of xs_1, and this is the output pointer context for ptr_struct. The number m representing x in xs_2 now equals the size of xs_1, and returned reflected elem is again Pts m v. Similar considerations apply in the case where we are reflecting a heap variable h. The instance is then var_struct and we search in the heap portion of the context hs_1, producing a new heap portion hs_2.

```
Structure xtagged A := XTag {xuntag : A}

extend_tag  A (x : A) := XTag  x
recurse_tag A (x : A) := extend_tag x
Canonical found_tag A (x : A) := recurse_tag  x

Structure xfind A (s r : seq A) (i : nat) :=
  XFind { elem_of : xtagged A;
          _ : index r i = elem_of ∧ prefix s r }

Canonical found_struct A (x : A) (s : seq A) :=
  XFind (x :: s) (x :: s) 0 (found_tag x) ...

Canonical recurse_struct (i : nat) (y : A) (s r : seq A)
                         (f : xfind s r i) :=
  XFind (y :: s) (y :: r) (i + 1) (recurse_tag f) ...

Canonical extend_struct A (x : A) :=
  XFind [] [x] 0 (extend_tag x) ...
```

FIGURE 2.3: Structure xfind for searching for an element in a list, and appending the element at the end of the list if not found.

In both cases, the task of searching (and possibly extending) the context is performed by the polymorphic structure xfind (Figure 2.3), which recurses over the context lists in search of an element, relying on unification to make syntactic comparisons between expressions. The inputs to the structure are the parameter s, which is the sequence to search in, and the field elem_of, which is the (tagged) element to search for. The output sequence r equals s if elem_of is in s, or extends s with elem_of otherwise. The output parameter i is the position at which the elem_of is found *in* r.

If the searched element x appears at the head of the list, the selected instance is found_struct and the index $i = 0$. Otherwise, we recurse using recurse_struct. Ultimately, if s is empty, the returned r is the singleton $[x]$, via the instance extend_struct.

It may be interesting to notice here that while xfind is in principle similar to find from Section 2.2, it is keyed on the element being searched for, rather than on the list (or in the case of find, the heap) in which the search is being performed. This exemplifies that there are many ways in which canonical structures of similar functionality can be organized. In particular, which term one keys on (*i.e.*, which term one unifies with the projection from the structure) may in general depend on when a certain computation needs to be triggered. If we reorganized xfind to match find in this respect, then the structure ast would have to be reorganized too. Specifically, ast would have to recursively invoke xfind by unifying it against the contexts xs_1 and hs_1 in the instances pts_struct

and var_struct, respectively. As we will argue in Section 3.3.1, such unification can lead to incorrect results, if done directly, but we will be able to perform it *indirectly*, using a new design pattern.

We are now ready to present the overloaded lemma cancelR.

$$\begin{aligned}\mathsf{cancelR} : \forall j\ k : \mathsf{ctx}.\, \forall t_1\ t_2 : \mathsf{term}.\\ \quad \forall f_1 : \mathsf{ast\ nil}\ j\ t_1.\, \forall f_2 : \mathsf{ast}\ j\ k\ t_2.\\ \mathsf{def}\ (\mathsf{untag}\ (\mathsf{heap\_of}\ f_1)) \to\\ \mathsf{untag}\ (\mathsf{heap\_of}\ f_1) = \mathsf{untag}\ (\mathsf{heap\_of}\ f_2) \to\\ \mathsf{cancel}\ k\ t_1\ t_2\ []\end{aligned}$$

At first sight it may look strange that we are not using the notation convention 6 presented in this very same section. It is true that we can omit the projections and write def f_1 as the first hypothesis, but for the second one we have to be verbose. The reason is simple: if we write instead $f_1 = f_2$, Coq will consider this an equality on asts and not expand the implicit coercions, as needed.

Assuming we have a hypothesis

$$H : \overbrace{x \mapsto v_1 \bullet (h_3 \bullet h_4)}^{h_1} = \overbrace{h_4 \bullet x \mapsto v_2}^{h_2}$$

and a hypothesis D : def h_1, we can *forwardly* apply cancelR to H using D, *i.e.*,

$$\mathsf{move:}\ (\mathsf{cancelR}\ D\ H)$$

The move tactic of Ssreflect allows us to move a hypothesis from the goal to the context, or vice versa. In this case, we are taking the term cancelR D H—whose type, we expect, is the result of canceling the heaps—and get this type as a premise in the goal. In order to obtain the type of the term, Coq will fire the following unification problems:

1. def (untag (heap_of $?f_1$)) $\approx$ def h_1
2. untag (heap_of $?f_1$)) $\approx h_1$
3. untag (heap_of $?f_2$)) $\approx h_2$

Because $?f_1$ and $?f_2$ are variable instances of the structure ast, Coq will construct canonical values for them, thus reflecting the heaps into terms t_1 and t_2, respectively. The reflection of h_1 will start with the empty context, while the reflection of h_2 will start with the output context of f_1, which in this case is $([h_3, h_4], [x])$.

Finally, the lemma will perform cancel on t_1 and t_2 to produce $v_1 = v_2 \wedge h_3 = \mathsf{empty} \wedge \mathsf{empty} = \mathsf{empty}$. The trailing $\mathsf{empty} = \mathsf{empty}$ can ultimately be removed with a few simple optimizations of cancel, which we have omitted to simplify the presentation.

To Reflect or Not to Reflect: We used the power of canonical structures to inspect the structure of terms, in this case heaps, and then create abstract syntax trees representing those terms. Thus, a natural question arises: cannot we implement the cancellation algorithm entirely with canonical structures, just as it was implemented originally in one big tactic? The answer is yes, and the interested reader can find the code in the source files (Ziliani, 2014). There, we have encoded a similar algorithm to the one shown in Figure 2.1 with a structure parameterized over its inputs: the heaps on the left and on the right of the equation, the heap with "the rest"—*i.e.*, what could not be cancelled—and two invariants stating that (a) the right heap is defined and (b) the union of the left and rest heaps is equal to the right heap. As output, it returns a proposition and a proof of this proposition. Each instance of this structure will correspond to one step of the algorithm. As an immediate consequence, this direct encoding of the cancellation algorithm avoids the long proof of soundness, since each step (or instance) includes the local proof for that step only, and without having to go through the fuss of the interpretation of abstract syntax.

However, this approach has at least one big disadvantage: by comparison to the automated proof by reflection, it is slow. Indeed, this should come as no surprise, as it is well known that the method of proof by reflection is fast (*e.g.*, Grégoire and Mahboubi (2005)), and this is why we pursued it in the first place.

2.4.3 Dealing with Heterogeneous Heaps

So far we have represented singleton heaps as $x \mapsto v$, assuming that all values in the heaps are of the same type. However, as promised above, we would like to relax this assumption. Allowing for variation in the type T of v, a more faithful representation would be $x \mapsto_T v$. One easy way of supporting this without modifying our cancellation algorithm at all is to view the values in the heap as being elements of type dynamic, *i.e.*, as dependent pairs (or structures) packaging together a type and an element of the type:

$$\textbf{Structure}\ \mathsf{dynamic} := \mathsf{dyn}\ \{\ \mathsf{typ} : \mathsf{Type}; \mathsf{val} : \mathsf{typ}\ \}$$

In other words, we can simply model $x \mapsto_T v$ as $x \mapsto \mathsf{dyn}\ T\ v$. As a result, when we apply the cancelR lemma to the singleton equality $x \mapsto_{T_1} v_1 = x \mapsto_{T_2} v_2$, we would

```
Structure tagged_prop := Tag_prop { puntag : Prop }
default_tag p := Tag_prop p
dyneq_tag p := dyneq_tag p
Canonical and_tag p := dyneq_tag p

Structure simplifier (p : Prop) :=
  Simpl { prop_of : tagged_prop;
          _ : p ↔ prop_of }

Canonical and_struct (p1 p2 : Prop)
                     (f1 : simplifier p1) (f2 : simplifier p2) :=
  Simpl (p1 ∧ p2) (and_tag (f1 ∧ f2)) ...

Canonical dyneq_struct (A : Type) (v1 v2 : A) :=
  Simpl (v1 = v2) (dyneq_tag (dyn A v1 = dyn A v2)) ...

Canonical default_struct (p : Prop) :=
  Simpl p (default_tag p) ...
```

FIGURE 2.4: Algorithm for post-processing the output of cancelR.

obtain:

$$\mathsf{dyn}\ T_1\ v_1 = \mathsf{dyn}\ T_2\ v_2 \tag{2.3}$$

But if the types T_1 and T_2 are equal, we would like to also automatically obtain the equality on the underlying values:

$$v_1 = v_2 \tag{2.4}$$

(Note that this cannot be done within the pure logic of Coq since equality on types is not decidable.) A key benefit of obtaining the direct equality on v_1 and v_2, rather than on $\mathsf{dyn}\ T_1\ v_1$ and $\mathsf{dyn}\ T_2\ v_2$, is that such an equality can then be fed into the standard rewrite tactic in order to rewrite occurrences of v_1 in a goal to v_2.

This effect could perhaps be achieved by a major rewriting of cancelR, but that would require a major effort, just to come up with a solution that is not at all modular. Instead, we will show now how we can use a simple overloaded lemma to *post-process* the output of the cancellation algorithm, reducing equalities between dynamic packages to equalities on their underlying value components wherever possible.

Figure 2.4 presents the algorithm to simplify propositions using canonical instances. Basically, it inspects a proposition, traversing through a series of $\wedge$s until it finds an equality of the form $\mathsf{dyn}\ A\ v_1 = \mathsf{dyn}\ A\ v_2$—*i.e.*, where the values on both sides have the same type A—in which case it returns $v_1 = v_2$. For any other case, it just returns the same proposition. We only consider the connective $\wedge$ since the output of cancelR

contains only this connective. If necessary, we could easily extend the algorithm to consider other connectives.

The algorithm is encoded in a structure called simplifier with three canonical instances, one for each of the aforementioned cases. The proof component of this structure lets us prove the following overloaded lemma. (Following notational convention 6, we omit the projectors, so g below should be read as "the proposition in g".)

$$\mathsf{simplify} : \forall p : \mathsf{Prop}.\, \forall g : \mathsf{simplifier}\ p.$$
$$g \to p$$

By making p and g implicit arguments, we can write simplify P to obtain the simplified version of P. For example, say we have to prove some goal with hypotheses

$$D : \mathsf{def}\ (x \mapsto_T v_1 \bullet h_1)$$
$$H : x \mapsto_T v_1 \bullet h_1 = h_2 \bullet x \mapsto_T v_2$$

We can forwardly apply the composition of simplify and cancelR D to H

$$\mathsf{move} :\ (\mathsf{simplify}\ (\mathsf{cancelR}\ D\ H))$$

to get the hypothesis exactly as we wanted:

$$v_1 = v_2 \wedge h_1 = h_2$$

2.5 Solving for Functional Instances

Previous sections described examples that search for a pointer in a heap expression or for an element in a list. The pattern we show in this section requires a more complicated functionality, which we describe in the context of our higher-order implementation of separation logic (Reynolds, 2002) in Coq. Interestingly, this *search-and-replace* pattern can also be described as higher-order, as it crucially relies on the typechecker's ability to manipulate first-class functions and solve unification problems involving functions.

To set the stage, the formalization of separation logic that we use centers on the predicate

$$\mathsf{verify} : \forall A.\, \mathsf{prog}\ A \to \mathsf{heap} \to (A \to \mathsf{heap} \to \mathsf{Prop}) \to \mathsf{Prop}.$$

The exact definition of verify is not important for our purposes here, but suffice it to say that it encodes a form of Hoare-style triples. Given a program e : prog A returning

values of type A, an input heap $i : \mathsf{heap}$, and a postcondition $q : A \to \mathsf{heap} \to \mathsf{Prop}$ over A-values and heaps, the predicate

$$\mathsf{verify}\ e\ i\ q$$

holds if executing e in heap i is memory-safe, and either diverges or terminates with a value w and heap m, such that $q\ w\ m$ holds.

Programs can perform the basic heap operations: reading and writing a heap location, allocation, and deallocation. In this section, we focus on the writing primitive; given $x : \mathsf{ptr}$ and $v : A$, the program $\mathsf{write}\ x\ v : \mathsf{prog}\ \mathsf{unit}$ stores v into x and terminates. We also require the operation for sequential composition, which takes the form of monadic bind:

$$\mathsf{bind} : \forall A\ B.\, \mathsf{prog}\ A \to (A \to \mathsf{prog}\ B) \to \mathsf{prog}\ B$$

We next consider the following provable lemma, which serves as a Floyd-style rule for symbolic evaluation of write.

$$\begin{aligned}\mathsf{bnd\_write} : \forall A\ B\ C.\, \forall x : \mathsf{ptr}.\, \forall v : A.\, \forall w : C.\, \forall e : \mathsf{unit} \to \mathsf{prog}\ B.\\ \forall h : \mathsf{heap}.\, \forall q : B \to \mathsf{heap} \to \mathsf{Prop}.\\ \mathsf{verify}\ (e\ ())\ (x \mapsto v \bullet h)\ q \to\\ \mathsf{verify}\ (\mathsf{bind}\ (\mathsf{write}\ x\ v)\ e)\ (x \mapsto w \bullet h)\ q\end{aligned}$$

To verify $\mathsf{write}\ x\ v$ in a heap $x \mapsto w \bullet h$, it suffices to change the contents of x to v, and proceed to verify the continuation e.

In practice, bnd_write suffers from the same problem as indom and noalias, as each application requires the pointer x to be brought to the top of the heap. We would like to devise an automated version bnd_writeR, but, unlike indomR, application of bnd_writeR will not merely check if a pointer x is in the heap. It will remember the heap h from the goal and reproduce it in the premise, only with the contents of x in h changed from w to v.

For example, applying bnd_writeR to the goal

$$\begin{aligned}G_1 : \mathsf{verify}\ (\mathsf{bind}\ (\mathsf{write}\ x_2\ 4)\ e)\\ (i_1 \bullet (x_1 \mapsto 1 \bullet x_2 \mapsto 2) \bullet (i_2 \bullet x_3 \mapsto 3))\\ q\end{aligned}$$

should return a subgoal which changes x_2 in place, as in:

$$G_2 : \mathsf{verify}\ (e\ ())\ (i_1 \bullet (x_1 \mapsto 1 \bullet x_2 \mapsto 4) \bullet (i_2 \bullet x_3 \mapsto 3))\ q$$

```
Structure tagged_heap := Tag {untag : heap}

right_tag (h : heap) := Tag h
left_tag h := right_tag h
Canonical found_tag h := left_tag h

Structure partition (k r : heap) :=
  Partition { heap_of : tagged_heap;
              _ : heap_of = k • r }

Canonical found_struct k :=
  Partition k empty (found_tag k) ...

Canonical left_struct h r (f : ∀k. partition k r) k :=
  Partition k (r • h) (left_tag (f k • h)) ...

Canonical right_struct h r (f : ∀k. partition k r) k :=
  Partition k (h • r) (right_tag (h • f k)) ...
```

FIGURE 2.5: Structure partition for partitioning a heap into the part matching k and "the rest" (r).

2.5.1 The "Search-and-Replace" Pattern

Here is where functions come in. The bnd_writeR lemma should attempt to infer a function f which represents a heap with a "hole", so that filling the hole with $x \mapsto w$ (*i.e.*, computing $f\,(x \mapsto w)$) results in the heap from the goal. Then replacing w with v is computed as $f\,(x \mapsto v)$.

For example, in G_1 we want to "fill the hole" with $x_2 \mapsto 2$, while in G_2, we want to fill it with $x_2 \mapsto 4$. Hence, in this case, the inferred function f should be:

$$\lambda k.\ i_1 \bullet (x_1 \mapsto 1 \bullet k) \bullet (i_2 \bullet x_3 \mapsto 3)$$

To infer f using canonical structures, we generalize it from a function mapping heaps to heaps to a function mapping a heap k to a *structure*, partition $k\ r$ (defined in Figure 2.5), with a heap projection heap_of that is equal to $k \bullet r$. This heap_of projection will be used to trigger the search for the subheap that should be replaced with a hole. (The role of the additional heap parameter r will be explained later, but intuitively one can think of r as representing the rest of the heap, *i.e.*, the "frame" surrounding the hole.)

Because the range of f depends on the input k, f must have a *dependent function type*, and the bnd_writeR lemma looks as follows.

$$\begin{aligned}\mathsf{bnd\_writeR} : \forall A\ B\ C.\, \forall x : \mathsf{ptr}.\, \forall v : A.\, \forall w : C.\, \forall e : \mathsf{unit} \to \mathsf{prog}\ B.\\ \forall q : B \to \mathsf{heap} \to \mathsf{Prop}.\\ \forall r : \mathsf{heap}.\, \forall f : (\forall k : \mathsf{heap}.\, \mathsf{partition}\ k\ r).\\ \mathsf{verify}\ (e\ ())\ (f\ (x \mapsto v))\ q \to\\ \mathsf{verify}\ (\mathsf{bind}\ (\mathsf{write}\ x\ v)\ e)\ (f\ (x \mapsto w))\ q\end{aligned}$$

As before, following notational convention 6, we have omitted the projections and written $f\ (x \mapsto w)$ instead of $\mathsf{untag}\ (\mathsf{heap\_of}\ (f\ (x \mapsto w)))$, and similarly in the case of $x \mapsto v$.

When the bnd_writeR lemma is applied to a goal of the form

$$\mathsf{verify}\ (\mathsf{bind}\ (\mathsf{write}\ x\ v)\ e)\ h\ q$$

the type checker creates unification variables for each of the parameters of bind_writeR, and proceeds to unify the conclusion of the lemma with the goal, getting the equation

$$\mathsf{untag}\ (\mathsf{heap\_of}\ (?f\ (x \mapsto ?w))) \approx h$$

where $?f$ has type $\forall k : \mathsf{heap}.\, \mathsf{partition}\ k\ ?r$. (Note that, because Coq's unification follows a strict left-to-right order, x is not a unification variable but the actual location being written to in the goal.)

This unification goal will prompt the search (using the instances in Figure 2.5) for a canonical solution for $?f$ with the property that the heap component of $?f\ (x \mapsto ?w)$ *syntactically* equals h, matching exactly the order and the parenthesization of the summands in h. We have three instance selectors: one for the case where we found the heap we are looking for, and two to recurse over each side of the $\bullet$.

The reader may wonder why all the instances of partition take the k parameter *last*, thus forcing the f parameter in the latter two instances to be itself abstracted over k as well. The reason is best illustrated by following a partial trace of Coq's unification.

Suppose $h = h_0 \bullet x \mapsto z$. After trying and failing to unify $?f\ (x \mapsto ?w)$ with h using the instances found_struct and left_struct, it will proceed to try to use the instance right_struct. More precisely, as described previously in Section 2.3.2, when solving the equation

$$\mathsf{heap\_of}\ (?f\ (x \mapsto ?w)) \approx \mathsf{right\_tag}\ (h_0 \bullet x \mapsto z) \tag{2.5}$$

Coq will:

1. Build an instance
$$\mathsf{right\_struct}\ ?h'\ ?r'\ ?f'\ ?k' \tag{2.6}$$
with $?h', ?r', ?f', ?k'$ fresh unification variables, $?f'$ with type $\forall k.\ \mathsf{partition}\ k\ ?r'$ and the rest with type heap.

2. Unify the type of the instance from (2.6) with the type of the expected instance (*i.e.*, the argument of $\mathsf{heap\_of}$) in (2.5). We know that $?f$ has type $\forall k : \mathsf{heap}.\ \mathsf{partition}\ k\ ?r$, and therefore that $?f\ (x \mapsto ?w)$ has type $\mathsf{partition}\ (x \mapsto ?w)\ ?r$. The type of (2.6) is $\mathsf{partition}\ ?k'\ (?h' \bullet ?r')$. Putting it all together, we get the equation
$$\mathsf{partition}\ ?k'\ (?h' \bullet ?r') \approx \mathsf{partition}\ (x \mapsto ?w)\ ?r$$

3. Unify the $\mathsf{heap\_of}$ projection of (2.6) with the right-hand side of (2.5), that is,
$$\mathsf{right\_tag}\ (?h' \bullet ?f'\ ?k') \approx \mathsf{right\_tag}\ (h_0 \bullet x \mapsto z)$$

4. Finally, unify the instances:
$$?f\ (x \mapsto ?w) \approx \mathsf{right\_struct}\ ?h'\ ?r'\ ?f'\ ?k'$$

Altogether, we get the following equations that Coq processes in order:

1. $?k' \approx x \mapsto ?w$
2. $?h' \bullet ?r' \approx ?r$
3. $?h' \approx h_0$
4. $?f'\ ?k' \approx x \mapsto z$
5. $?f\ (x \mapsto ?w) \approx \mathsf{right\_struct}\ ?h'\ ?r'\ ?f'\ ?k'$

The first three equations are solved immediately. The fourth one, if we expand the implicit projection and instantiate the variables, is
$$\mathsf{untag}\ \ (\mathsf{heap\_of}\ (?f'\ (x \mapsto ?w))) \approx x \mapsto z \tag{2.7}$$

Solving this recursively (we give the details of this part of the trace at the end of the section), Coq instantiates the following variables:

1. $?r' = \mathsf{empty}$

2. $?w = z$

3. $?f' = \mathsf{found\_struct}$

Note how the type of the instance found_struct actually matches the type of $?f'$, that is, $\forall k.\, \mathsf{partition}\ k\ \mathsf{empty}$.

Coq now can proceed to solve the last equation, which after instantiating the variables is

$$?f\ (x \mapsto z) \approx \mathsf{right\_struct}\ h_0\ \mathsf{empty}\ \mathsf{found\_struct}\ (x \mapsto z)$$

which means that it has to find a function for $?f$ such that, when given the singleton $x \mapsto z$, produces the instance on the right-hand side. As it is well known (Goldfarb, 1981), higher-order unification problems are in general undecidable, as they might have an infinite number of solutions, without any one being the most general one. For this example[1], Coq takes a commonly-used pragmatic solution of falling back to a kind of first-order unification: it tries to unify the functions and then the arguments on both side of the equation, which in this case immediately succeeds:

1. $?f \approx \mathsf{right\_struct}\ h_0\ \mathsf{empty}\ \mathsf{found\_struct}$

2. $(x \mapsto z) \approx (x \mapsto z)$

This is the key to understanding why the instances of partition all take the k parameter last: we want the k parameter to ultimately be unified with the argument of $?f$. If the k parameter did not come last, then Coq would try here to unify $x \mapsto z$ with whatever parameter *did* come last, which would clearly lead to failure.

Thus far, we have described how to construct the canonical solution of $?f$, but the mere construction is not sufficient to carry out the proof of bnd_writeR. For the proof, we further require an explicit invariant that $?f\ (x \mapsto v)$ produces a heap in which the contents of x is changed to v, but *everything else is unchanged* when compared to $?f\ (x \mapsto w)$.

This is the role of the parameter r, which is constrained by the invariant in the definition of partition to equal the "rest of the heap", that is

$$h = k \bullet r$$

[1] There exists a decidable fragment of higher-order unification called the "pattern fragment" (Miller, 1991a). If the problem at hand falls into this fragment, Coq will find the most general unifier. However, our example does not fall into this fragment.

With this invariant in place, we can vary the parameter k from $x \mapsto w$ in the conclusion of bnd_writeR to $x \mapsto v$ in the premise. However, r remains fixed by the type of $?f$, providing the guarantee that the only change to the heap was in the contents of x.

It may be interesting to note that, while our code computes an $?f$ that syntactically matches the parentheses and the order of summands in h (as this is important for using the lemma in practice), the above invariant on h, k and r is in fact a *semantic*, not a syntactic, equality. In particular, it does not guarantee that h and $k \bullet r$ are constructed from the same exact applications of $\mapsto$ and $\bullet$, since in HTT those are defined functions, not primitive constructors. Rather, it captures only equality up to commutativity, associativity and other semantic properties of heaps as partial maps. This suffices to prove bnd_writeR, but more to the point: the syntactic property, while true, cannot even be expressed in Coq's logic, precisely because it concerns the syntax and not the semantics of heap expressions.

To conclude the section, we note that the premise and conclusion of bnd_writeR both contain projections out of $?f$. As a result, the lemma may be used both in forward reasoning (out of hypotheses) and in backward reasoning (for discharging a given goal). For example, we can prove the goal

$$\mathsf{verify}\ (\mathsf{bind}\ (\mathsf{write}\ x_2\ 4)\ e)\ (i_1 \bullet (x_1 \mapsto 1 \bullet x_2 \mapsto 2))\ q$$

under hypothesis

$$H\ :\ \mathsf{verify}\ (e\ ())\ (i_1 \bullet (x_1 \mapsto 1 \bullet x_2 \mapsto 4))\ q$$

in two ways:

- *Backward*: By applying bnd_writeR to the goal. The goal will therefore be changed to exactly match H.
- *Forward*: By applying bnd_writeR $(x := x_2)$ $(w := 2)$ to the hypothesis H, thus obtaining the goal. Note how in this case we need to explicitly provide the instantiations of the parameters x and w because they cannot be inferred just from looking at H.

This kind of versatility is yet another advantage that lemmas based on canonical instances exhibit when compared to tactics. The latter, it seems, must be specialized to either forward or backward mode, and we have not managed to encode a tactic equivalent of bnd_writeR that is usable in both directions. It likewise appears difficult, if not impossible, to encode this bidirectional functionality using the style of proof by reflection we explored in Section 2.4.

Fleshing out the Trace: In the partial unification trace given above, we streamlined the presentation by omitting the part concerning the resolution of the unification equation (2.7). Since found_tag is the canonical instance of the tagged_heap structure, instance resolution will reduce this problem to:

$$\mathsf{heap\_of}\ (?f'\ (x \mapsto ?w)) \approx \mathsf{found\_tag}\ (x \mapsto z) \tag{2.8}$$

For this equation, Coq follows the same steps as in the processing of equation (2.5). It will:

1. Create a unification variable $?k''$ for the argument of found_struct.
2. Unify the type of found_struct $?k''$ with the type of $?f'\ (x \mapsto ?w)$, that is,

   $$\mathsf{partition}\ ?k''\ \mathsf{empty} \approx \mathsf{partition}\ (x \mapsto ?w)\ ?r'$$

   getting $?k'' = x \mapsto ?w$ and $?r' = \mathsf{empty}$.
3. Unify the heap_of projection of found_struct $?k''$ with the right-hand side of (2.8):

   $$\mathsf{found\_tag}\ ?k'' \approx \mathsf{found\_tag}\ (x \mapsto z)$$

   Unfolding the already known definition of $?k''$ as $x \mapsto ?w$, we get $?w = z$.
4. Unify the argument of heap_of with the instance. After applying the solutions found so far, this produces

   $$?f'\ (x \mapsto z) \approx \mathsf{found\_struct}\ (x \mapsto z)$$

   As before, Coq solves this higher-order problem by unifying

   (a) $?f' \approx \mathsf{found\_struct}$

   (b) $x \mapsto z \approx x \mapsto z$

2.5.2 Automatic Lemma Selection

In the previous section we saw how to automate the symbolic evaluation of the command write. In our implementation of separation logic in Coq, there are in fact several such commands (allocation, deallocation, read, write, etc.), each of which may appear in one of two contexts (either by itself or as part of a bind expression sequencing it together with other commands). We have created one automated lemma for each combination of command and context, but picking the right one to apply at each step of a proof can be

tedious, so we would like to automate this process by creating a procedure for selecting the right lemma in each case. We will now show how to build such an automated procedure.

For the running example we will use just two lemmas:

$$\begin{aligned}\mathsf{bnd\_writeR} : {} & \forall A\ B\ C.\, \forall x : \mathsf{ptr}.\, \forall v : A.\, \forall w : C.\, \forall e : \mathsf{unit} \to \mathsf{prog}\ B. \\ & \forall q : B \to \mathsf{heap} \to \mathsf{Prop}. \\ & \forall r : \mathsf{heap}.\, \forall f : (\forall k : \mathsf{heap}.\, \mathsf{partition}\ k\ r). \\ & \mathsf{verify}\ (e\ ())\ (f\ (x \mapsto v))\ q \to \\ & \mathsf{verify}\ (\mathsf{bind}\ (\mathsf{write}\ x\ v)\ e)\ (f\ (x \mapsto w))\ q\end{aligned}$$

$$\begin{aligned}\mathsf{val\_readR} : {} & \forall A.\, \forall x : \mathsf{ptr}.\, \forall v : A.\, \forall q : A \to \mathsf{heap} \to \mathsf{Prop}. \\ & \forall r : \mathsf{heap}.\, \forall f : (\mathsf{partition}\ (x \mapsto v)\ r). \\ & (\mathsf{def}\ f \to q\ v\ f) \to \\ & \mathsf{verify}\ (\mathsf{read}\ x)\ f\ q\end{aligned}$$

The first one is the automated lemma from the previous section. The second one executes the command read alone (not as part of a bind expression). The lemma val_readR states that, in order to show that read x satisfies postcondition q, we need to prove that q holds of the value v that x points to.

Consider a simple example, in which we have the following goal:

$$\mathsf{verify}\ (\mathsf{bind}\ (\mathsf{write}\ x\ 4)\ (\lambda\_.\ \mathsf{read}\ x))\ (x \mapsto 0)\ (\lambda r.\ \lambda\_.\ r = 4)$$

This goal states that after writing the value 4 in location x, we read x and get a result r that is equal to 4. Using the above lemmas directly, we would prove this as follows [2]:

```
apply: bnd_writeR
by apply: val_readR
```

The first application changes the goal to

$$\mathsf{verify}\ (\mathsf{read}\ x)\ (x \mapsto 4)\ (\lambda r.\ \lambda\_.\ r = 4)$$

while the second application performs the read and produces the trivial goal $4 = 4$.

[2]Note that we are using Ssreflect apply (*i.e.*, with colon) instead of Coq's native tactic. This is required since Coq's apply tactic might use two different and inconsistent unification algorithms.

```
Structure val_form (h : heap) (q : A → heap → Prop) (p : Prop) :=
  ValForm { val_pivot : prog A;
            _ : p → verify val_pivot h q }

Structure bnd_form (h : heap) (e : A → prog B) (q : B → heap → Prop) (p : Prop) :=
  BndForm { bnd_pivot : prog A;
            _ : p → verify (bind bnd_pivot e) h q }

Canonical val_bind_struct h e q p (f : bnd_form h e q p) :=
  ValForm (bind (bnd_pivot f) e) ...

step : ∀h. ∀q. ∀p. ∀f : val_form h q p. p → verify f h q
```

FIGURE 2.6: Definition of the overloaded step lemma.

Using the overloaded lemma step we will present below, we will instead be able to prove this as follows:

```
by do 2 apply: step
```

where do n *tactic* is the nth repeated application of *tactic*. When verifying a large program, an overloaded lemma like step becomes increasingly convenient to use, for all the usual reasons that overloading is useful in ordinary programming. This convenience is borne out in our source files (Ziliani, 2014), where the interested reader can find the verification of a linked list data type using step.

Intuitively, step works by inspecting the program expression being verified and selecting an appropriate lemma to apply. In our example, the first application of step will apply bnd_writeR, and the second one val_readR, exactly as in our manual proof.

Notational Convention 7. We use val_$*$ to name every lemma concerning the symbolic execution of a program expression consisting of a single command, like val_readR. We use bnd_$*$ to name every lemma concerning the symbolic execution of a command inside a bind expression, like bnd_writeR.

Figure 2.6 shows the main structure, val_form, for the overloaded lemma step. It has two fields: (1) a program expression, which we call val_pivot, and (2) a proof that, assuming precondition p, the postcondition q will hold after executing the pivot program in the initial heap h. Our overloaded lemma step is trivially proved by the projection of this second field.

When step is applied to a goal verify e h q, the system tries to construct an instance f : val_form h q p, whose val_pivot matches e. Figure 2.7 declares one such val_$*$ instance,

```
Canonical val_read_struct x v q r f :=
  ValForm (read x) (val_readR x v q r f)

Canonical bnd_write_struct x v w e q r f :=
  BndForm (write x v) (bnd_writeR x v w e q r f)
```

FIGURE 2.7: Registering individual lemmas with step.

val_read_struct, which is selected when e is a read x command, for some x. The second field of the instance declares the lemma that should be applied to the verify goal; in this case the val_readR lemma.

Thus, declaring instances of val_form corresponds to *registering* with step the lemmas that we want applied for specific forms of e. For example, we can register lemmas that apply when e is a primitive command such as alloc or dealloc. The only requirement is that the form of the registered lemma matches the second field of val_form; namely, the lemma has a conclusion verify $e\ h\ q$ and *one* premise p, for some e, h, q, and p.

When the command e is not a standalone command, but a bind composite, we redirect step to search for an approriate lemma among bnd_$*$ instances. That is achieved by declaring a new structure bnd_form and a canonical instance val_bnd_struct for val_form in Figure 2.6. When step is applied to a goal verify $e\ h\ q$, in which e is a bind, val_bnd_struct is selected as a canonical instance. But since val_bnd_struct is parameterized by a hypothesis f : bnd_form $h\ e\ q\ p$, this redirects the unification algorithm into solving for f.

Much as in the val_form case, we need to register the bnd_$*$ lemmas that the algorithm should apply depending on the first command of e. Figure 2.7 shows an example instance bnd_write_struct which registers lemma bnd_writeR to be applied by step whenever e *starts* with a write command.

In a similar way, we can register val_$*$ and bnd_$*$ lemmas for *user-defined* commands as well, thereby extending step at will as we implement new commands. In this sense, step is an *open-ended* automation procedure. Such open-endedness is yet another aspect of lemma overloading that does not seem to have a direct correspondent in tactic-based automation.

2.6 Flexible Composition and Application of Overloaded Lemmas

In this section we construct an overloaded version of the noalias lemma from the introduction. This example presents two main challenges: (1) composing two overloaded lemmas where the output of one is the input to the other one, and (2) making the resulting lemma applicable in both forward and backward reasoning.

Concerning the first problem, there are several ways to solve it. We present different alternative approaches to composing overloaded lemmas, equipping the interested reader with a handy set of techniques with varying complexity/flexibility tradeoffs.

Concerning the second problem, the key challenge is to ensure that the unification constraints generated during canonical structure inference are resolved in the intended order. This is important because the postponing of a certain constraint may underspecify certain variables, leading the system to choose a wrong intermediate value that will eventually fail to satisfy the postponed constraint. In the case of noalias, the problem is that a naive implementation will result in the triggering of a search for a pointer in a heap before we know what pointer we're searching for. Fortunately, it is possible to handle this problem very easily using a simple design pattern we call *parameterized tagging*.

In the following sections, we build several, progressively more sophisticated, versions of the noalias lemma, ultimately arriving at a lemma noaliasR that will be applicable both backwards and forwards. In the backwards direction, we will be able to use to it solve a goal such as

$$(x_1 \texttt{ != } x_2) \texttt{ \&\& } (x_2 \texttt{ != } x_3) \texttt{ \&\& } (x_3 \texttt{ != } x_1)$$

by rewriting repeatedly (notice the modifier "!"):

$$\texttt{by rewrite !(noaliasR } D\texttt{)}$$

Here, D is assumed to be a hypothesis describing the well-definedness of a heap containing three singleton pointers, one for each pointer appearing in the goal. Notice how, in order to rewrite repeatedly using the same lemma (noaliasR D), it is crucial that we do not need to explicitly specify the pointers involved in each application of the lemma, since each application involves a different pair of pointers. In our first versions of the lemma, this advanced functionality will not be available, and the pointers will need to be given explicitly, but eventually we will show how to support it.

```
Structure tagged_heap := Tag {untag : heap}
default_tag (h : heap) := Tag h
ptr_tag h := default_tag h
Canonical union_tag h := ptr_tag h

Structure scan (s : seq ptr) :=
  Scan { heap_of : tagged_heap;
         _ : def heap_of →
               uniq s ∧ ∀x. x ∈ s → x ∈ dom heap_of }

Canonical union_struct s1 s2 (f1 : scan s1) (f2 : scan s2) :=
  Scan (append s1 s2) (union_tag (f1 • f2)) ...

Canonical ptr_struct A x (v : A) :=
  Scan (x :: nil) (ptr_tag (x ↦ v)) ...

Canonical default_struct h := Scan nil (default_tag h) ...
```

FIGURE 2.8: Structure scan for computing a list of pointers syntactically appearing in a heap.

Before exploring the different versions of the lemma, we begin by presenting the infrastructure common to all of them.

2.6.1 Basic Infrastructure for the Overloaded Lemma

Given a heap h, and two pointers x and y, the algorithm for noalias proceeds in three steps: (1) scan h to compute the list of pointers s appearing in it, which must by well-definedness of h be a list of *distinct* pointers; (2) search through s until we find either x or y; (3) once we find one of the pointers, continue searching through the remainder of s for the other one. Figures 2.8–2.10 show the automation procedures for performing these three steps.

Step (1) is implemented by the scan structure in Figure 2.8. Like the ast structure from Section 2.4, scan returns its output using its *parameter* (here, s). It also outputs a proof that the pointers in s are all distinct (*i.e.*, uniq s) and that they are all in the domain of the input heap, assuming it was well-defined.

Step (2) is implemented by the search2 structure (named so because it searches for *two* pointers, both taken as parameters to the structure). It produces a proof that x and y are both distinct members of the input list s, which will be passed in through unification with the seq2_of projection. The search proceeds until either x or y is found, at which point the search1 structure (next paragraph) is invoked with the other pointer.

```
Structure tagged_seq2 := Tag2 {untag2 : seq ptr}
foundz (s : seq ptr) := Tag2 s
foundy s := foundz s
Canonical foundx s := foundy s

Structure search2 (x y : ptr) :=
  Search2 { seq2_of : tagged_seq2;
            _ : x ∈ seq2_of ∧ y ∈ seq2_of
                ∧(uniq seq2_of → x != y) }

Canonical x_struct x y (s1 : search1 y) :=
  Search2 x y (foundx (x :: s1)) ...

Canonical y_struct x y (s1 : search1 x) :=
  Search2 x y (foundy (y :: s1)) ...

Canonical z_struct x y z (s2 : search2 x y) :=
  Search2 x y (foundz (z :: s2)) ...
```

FIGURE 2.9: Structure search2 for finding two pointers in a list.

```
Structure tagged_seq1 := Tag1 {untag1 : seq ptr}
recurse_tag (s : seq ptr) := Tag1 s
Canonical found_tag s := recurse_tag s

Structure search1 (x : ptr) := Search1 { seq1_of : tagged_seq1;
                                        _ : x ∈ seq1_of }

Canonical found_struct (x : ptr) (s : seq ptr) :=
  Search1 x (found_tag (x :: s)) ...

Canonical recurse_struct (x y : ptr) (f : search1 x) :=
  Seach1 x (recurse_tag (y :: f)) ...
```

FIGURE 2.10: Structure search1 for finding a pointer in a list.

Step (3) is implemented by the search1 structure, which searches for a single pointer x in the remaining piece of s, returning a proof of x's membership in s if it succeeds. Its implementation is quite similar to that of the find structure from Section 2.2.

2.6.2 Naive Composition

The noalias lemma we wish to build is, at heart, a composition of two subroutines: one implemented by the structure scan and the other by the structure search2. Indeed, looking at the structures scan and search2, we notice that the output of the first one coincides with the input of the second one: scan computes the list of distinct pointers in

a heap, while search2 proves that, if the list output by scan contains distinct pointers, then the two pointers given as parameters are distinct.

Our first, naive attempt at building noalias is to (1) define overloaded lemmas corresponding to scan and search2, and (2) compose them using ordinary (function) composition, in the same that way that we composed the two lemmas simplify and cancelR in Section 2.4.3. As we will see, this direct approach does not quite work—*i.e.*, we will not get out a *general lemma* in the end—but it is instructive to see why.

First, we create the two overloaded lemmas, scan_it and search_them, whose proofs are merely the proof projections from the scan and search2 structures. As usual, we leave the structure projections (here of f and g) implicit:

$$\begin{aligned} \mathsf{scan\_it} : \forall s : \mathsf{seq\ ptr}.\, \forall f : \mathsf{scan}\ s.\\ \mathsf{def}\ f\ \rightarrow \mathsf{uniq}\ s \end{aligned}$$

$$\begin{aligned} \mathsf{search\_them} : \forall x\ y : \mathsf{ptr}.\, \forall g : \mathsf{search2}\ x\ y.\\ \mathsf{uniq}\ g \rightarrow x\ \mathsf{!=}\ y \end{aligned}$$

We can apply their composition in the same way as in Section 2.4.3. For example:

$$\begin{aligned} \text{Hyp. } D &: \mathsf{def}\ (i_1 \bullet (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2) \bullet (i_2 \bullet x_3 \mapsto v_3))\\ \text{Goal} &: x_1\ \mathsf{!=}\ x_3\\ \text{Proof} &: \mathsf{by\ apply:}\ (\mathsf{search\_them}\ x_1\ x_3\ (\mathsf{scan\_it}\ D)) \end{aligned}$$

During the typechecking of the lemma to be applied (*i.e.*, search_them x_1 x_3 (scan_it D)), Coq will first unify D's type with that of scan_it's premise (*i.e.*, def (untag (heap_of $?f$))), which forces the unification of the heap in the type of D with the implicit projection untag (heap_of $?f$). This in turn triggers an inference problem in which the system solves for the canonical implementation of $?f$ by executing the scan algorithm, thus computing the pointer list s (in this case, $[x_1, x_2, x_3]$). After obtaining uniq s as the output of scan_it, the search for the pointers x_1 and x_3 is initiated by unifying uniq s with the premise of search_them (*i.e.*, uniq (seq2_of (untag2 $?g$))), which causes s to be unified with seq2_of (untag2 $?g$), thus triggering the resolution of $?g$ by the search2 algorithm.

We may be tempted, then, to define the lemma noalias as a direct composition of the two lemmas. Unfortunately this will not work, because although we can compose the lemmas *dynamically* (*i.e.*, when applied to a particular goal), we cannot straightforwardly compose them *statically* (*i.e.*, in the proof of the general noalias lemma with unknown

parameters). To see why, let us try to build the lemma using the structures:

$$\mathsf{noaliasR\_fwd\_wrong} : \forall x\ y : \mathsf{ptr}.\, \forall s : \mathsf{seq\ ptr}.\, \forall f : \mathsf{scan}\ s.\, \forall g : \mathsf{search2}\ x\ y.\\ \mathsf{def}\ f \to x\ \mathsf{!=}\ y$$

The reason we cannot prove this lemma is that there is no connection between the output of the scan—that is, s—and the input sequence of search2. To put it another way, what we have is a proof that the list of pointers s appearing in the heap component of f is unique, but what we need is a way to prove that the list component of g is unique. We do not have any information telling us that these two lists should be equal, and in fact there is no reason for that to be true.

2.6.3 Connecting the Lemmas with an Equality Hypothesis

To prove the general noalias lemma, we clearly need a way to connect the output of scan with the input of search2. A simple way to achieve this is by adding an extra hypothesis representing the missing connection (boxed below), using which the proof of the lemma is straightforward. For clarity, we make the projection in this hypothesis explicit.

$$\mathsf{noaliasR\_fwd} : \forall x\ y : \mathsf{ptr}.\, \forall s : \mathsf{seq\ ptr}.\, \forall f : \mathsf{scan}\ s.\, \forall g : \mathsf{search2}\ x\ y.\\ \mathsf{def}\ f \to \boxed{s = (\mathsf{untag2}\ (\mathsf{seq2\_of}\ g))} \to x\ \mathsf{!=}\ y$$

We show how it works by applying it to the same example as before. We assume s, f, g implicit.

$$\begin{array}{lll} \text{Hyp. } D: & \mathsf{def}\ (i_1 \bullet (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2) \bullet (i_2 \bullet x_3 \mapsto v_3)) \\ \text{Goal} & : & x_1\ \mathsf{!=}\ x_3 \\ \text{Proof} & : & \mathsf{by\ apply:}\ (\mathsf{noaliasR\_fwd}\ x_1\ x_3\ D\ (\mathsf{erefl}\ \_)) \end{array}$$

Here, erefl x is the proof that $x = x$. The trace of the typechecker is roughly as follows. It:

1. Generates fresh unification variables for each of the arguments: $?x, ?y, ?s, ?f, ?g$.
2. Unifies $?x$ and $?y$ with the pointers given as parameters, x_1 and x_3, respectively.
3. Unifies the hypothesis D with the hypothesis of the lemma. More precisely, with $?f : \mathsf{scan}\ ?s$, it will unify

$$\mathsf{def}\ ?f \approx \mathsf{def}\ (i_1 \bullet (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2) \bullet (i_2 \bullet x_3 \mapsto v_3))$$

   starting the scan'ing of the heap. When scan is finished, we get $?s = [x_1, x_2, x_3]$.

4. Unifies the equality with the type of $\mathsf{erefl}\ \_$, where $\_$ is an implicit argument. More precisely, Coq will create a unification variable $?t$ for this implicit argument, and unify

$$(?t = ?t) \approx ([x_1, x_2, x_3] = \mathsf{untag2}\ (\mathsf{seq2\_of}\ ?g))$$

where $?g : \mathsf{search2}\ x_1\ x_3$. Next, it decomposes the equality, obtaining equations

$$?t \approx [x_1, x_2, x_3]$$
$$?t \approx \mathsf{untag2}\ (\mathsf{seq2\_of}\ ?g)$$

which effectively results in the equation we need for triggering the search for the two pointers in s:

$$[x_1, x_2, x_3] \approx \mathsf{untag2}\ (\mathsf{seq2\_of}\ ?g)$$

The astute reader may have noticed that, in the definition of noaliasR_fwd, we purposely arranged for the hypothesis $\mathsf{def}\ f$ to appear *before* the equality of s and g. If instead we had put it afterward, the last two steps would have been swapped, resulting in a doomed search for the pointers in s before the identity of s as $[x_1, x_2, x_3]$ was known.

In fact, note that we can actually arrange for $\mathsf{def}\ f$ to be hoisted out even further, outside the scope of the pointers x and y. The benefit of doing so is that we can scan a heap just once, and then use the resulting lemma to prove non-aliasing properties between different pairs of pointers without rescanning the heap each time. In particular, let our lemma be

$$\mathsf{noaliasR\_fwd} : \forall s : \mathsf{seq\ ptr}.\, \forall f : \mathsf{scan}\ s.\, \boxed{\forall d : \mathsf{def}\ f.}\, \forall x\ y : \mathsf{ptr}.\, \forall g : \mathsf{search2}\ x\ y.$$
$$s = \mathsf{untag2}\ (\mathsf{seq2\_of}\ g) \to x\ \mathsf{!=}\ y$$

and hypothesis D be as before. We can then make a local lemma abbreviation with the partially applied lemma

$$\mathsf{have}\ F := \mathsf{noaliasR\_fwd}\ D$$

Typechecking this local definition implicitly solves for s and f by scanning the heap defined by D, leaving only x, y, and g to be resolved by subsequent instantiation. We can then solve the following goal by rewriting several times using the partially-applied F:

$$\text{Goal} : x_1\ \mathsf{!=}\ x_3\ \&\&\ x_2\ \mathsf{!=}\ x_3$$
$$\text{Proof} : \mathsf{by\ rewrite}\ (F\ x_1\ x_3\ (\mathsf{erefl}\ \_))\ (F\ x_2\ x_3\ (\mathsf{erefl}\ \_))$$

2.6.4 Looking Backward, Not Forward

Note that, when applying the noaliasR_fwd lemma from the previous section, we need to instantiate the pointer parameters x and y explicitly, or else the search will fail. More precisely, the search will proceed without knowing which pointers to search for, and Coq will end up unifying x and y with (as it happens) the first two pointers in the list s (in the above example, x_1 and x_2). If they are not the pointers from the goal (as indeed in the example they are not), the application of the lemma will simply fail.

For many examples, like the cancelR example in Section 2.4 this is not a problem since the lemma is intended to be used only in *forward* mode. However, if we want noalias to be applicable also in *backward* mode—in particular, if we want to rewrite repeatedly with noalias D as shown at the beginning of this section—then we need to find a way of helping Coq infer the pointer arguments. The approach we present in this section will demonstrate yet another way of composing lemmas, which improves on the approach of the previous section in that it enables one (but not both) of the pointer arguments of noalias to be inferred. It thus serves as a useful bridge step to the final version of noalias in Section 2.6.5, which will be applicable both forwards and backwards.

The idea is to replace the equality hypothesis in the previous formulation of the lemma with a bespoke structure, check, which serves to unify the output of scan and the input of search2. To understand where we put the "trigger" for this structure, consider the noaliasR_fwd lemma from the previous section. There, the trigger was placed strategically after the hypothesis def f to fire the search after the list of pointers is computed. If we want to remove the equality hypothesis, then we do not have any other choice but to turn one of the *pointers* into the trigger.

We show first the reformulated lemma noaliasR_fwd to explain the intuition behind this change. As in our last version of the lemma, we move the hypothesis def f before the pointer arguments x and y to avoid redundant recomputation of the scan algorithm.

$$\begin{aligned}\mathsf{noaliasR\_fwd} : \forall s : \mathsf{seq\ ptr}.\, \forall f : \mathsf{scan}\ s.\, \forall d : \mathsf{def}\, f.\, \forall x\ y : \mathsf{ptr}.\, \boxed{\forall g : \mathsf{check}\ x\ y\ s.} \\ x\ \mathsf{!=}\ \mathsf{y\_of}\ g\end{aligned}$$

As before, when the lemma is applied to a hypothesis D of type def h, the heap h will be unified with the (implicit) projection untag (heap_of f). This will execute the scan algorithm, producing as output the pointer list s. However, when this lemma is subsequently applied in order to solve a goal of the form x' != y', the unification of that goal with the conclusion of the lemma will trigger the unification of y' with y_of ?g (the projection from check, which we have made explicit here for clarity), and that will in

turn initiate the automated search for the pointers in the list s. Note that we could use the unification with either x' or y' to trigger the search, but here we have chosen the latter.

We define the structure check and its canonical instance start as follows:

```
Structure check (x y : ptr) (s : seq ptr) :=
  Check { y_of : ptr;
          _ : y = y_of;
          _ : uniq s → x != y_of }
Canonical start x y (s2 : search2 x y) :=
  Check x y (untag2 (seq2_of s2)) y (erefl _) ...
```

The sole purpose of the canonical instance start for the check structure is to take the pointers x' and y' and the list s, passed in as parameters, and repackage them appropriately in the form that the search2 structure expects. In particular, while check is keyed on the right pointer (here, y'), search2 is keyed on the list of pointers s, so a kind of coercion between the two structures is necessary. Notice that this coercion is only possible if the structure's pointer parameter y is constrained to be *equal* to its y_of projection. Without this constraint, appearing as the second field (first proof field) of check, we obviously cannot conclude x != y_of from x != y.

Knowing that Coq unifies subterms in a left-to-right order, it should be clear that we can avoid mentioning the pointer x', since Coq will unify $?x$ with x' *before* unifying y_of g with y' and triggering the search. Consequently, if we have the goal

$$x_1 \texttt{ != } x_3 \texttt{ \&\& } x_2 \texttt{ != } x_3$$

and D has the same type as before, we can solve the goal by repeated rewriting as follows:

```
rewrite !(noaliasR_fwd D _ x3)
```

It is thus sensible to ask if we can avoid passing in the instantiation for the y parameter (here, x_3) explicitly. Unfortunately, we cannot, and the reason will become apparent by following a trace of application.

Assume D as before. We solve the goal

$$x_1 \texttt{ != } x_3$$

by

```
apply: (noaliasR_fwd D _ x3)
```

For the application to succeed, it must unify the goal with the inferred type for the lemma being applied. Let us write $?s$, $?f$, $?d$, $?x$, $?y$ and $?g$ for the unification variables that Coq creates for the arguments of noaliasR_fwd. As usual in type inference, the type of D must match the type of $?d$, *i.e.*, def $?f$. As mentioned above, this produces the unification problem that triggers the scanning:

$$\mathsf{untag}\ (\mathsf{heap\_of}\ ?f) \approx i_1 \bullet (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2) \bullet (i_2 \bullet x_3 \mapsto v_3)$$

In solving this problem, Coq instantiates variables $?f$ and $?s$ with I_f and $[x_1, x_2, x_3]$ respectively, where I_f stands for an instance that we omit here.

Now Coq continues the inference, instantiating $?y$ with x_3, and getting the following type for the conclusion of the lemma:

$$?x\ \mathsf{!=}\ \mathsf{y\_of}\ ?g$$

where $?g$: check $?x\ x_3\ [x_1, x_2, x_3]$. Coq proceeds to unify this type with the goal:

$$?x\ \mathsf{!=}\ \mathsf{y\_of}\ ?g \approx x_1\ \mathsf{!=}\ x_3$$

This generates two subproblems:

1. $?x \approx x_1$
2. y_of $?g \approx x_3$

The first one is solved immediately, and the second one triggers the search for an instance. The only available instance is start. After creating unification variables $?x'$, $?y'$ and $?s'_2$, one for each argument of the instance, Coq unifies the type of the instance with the expected type, *i.e.*,

$$\mathsf{check}\ ?x'\ ?y'\ (\mathsf{untag2}\ (\mathsf{seq2\_of}\ ?s'_2)) \approx \mathsf{check}\ x_1\ x_3\ [x_1, x_2, x_3]$$

therefore creating the following unification problems:

1. $?x' \approx x_1$
2. $?y' \approx x_3$
3. untag2 (seq2_of $?s'_2$) $\approx [x_1, x_2, x_3]$

After solving the first two problems right away, the third problem triggers the search for the pointers in the list. It is only now, after successfully solving these equations, that

Coq unifies the y_of projection of the instance with the expected one from the original equation, getting the trivial equation $x_3 \approx x_3$. Finally, it can assign the instance to $?g$, *i.e.*,

$$?g \approx \mathsf{start}\ x_1\ x_3\ ?s_2'$$

thus successfully proving the goal.

It should now hopefully be clear why we needed to pass x_3 as an explicit argument. If we had left it implicit, the unification problem #3 above would have triggered a search on an unknown $?y'$, even though the information about the identity of $?y'$ was made available in the subsequent step.

2.6.5 Reordering Unification Subproblems via Parameterized Tagging

Fortunately, there is a simple fix to our check structure to make our lemma infer *both* pointers from the goal without any intervention from the user.

The Pattern: In order to fix our check structure, we need a way to reorder the unification subproblems so that $?y$ gets unified with x_3 before the search algorithm gets triggered on the pointer list s. The trick for doing this is to embed the parameter y in the *type* of the projector y_of, thus ensuring higher priority in the unification order. Specifically, we will give y_of the type equals_ptr y, which (as the name suggests) will serve to constrain y_of to be equal to y and to ensure that this constraint is registered *before* the search algorithm is triggered. (Technically, equals_ptr y is *not* a singleton type, but canonical instance resolution will cause it to effectively behave like one.) We call this pattern *parameterized tagging*.

To illustrate, we present the structure equals_ptr, its canonical instance equate, and the requisite changes to the check structure (and its instance) according to the pattern:

```
Structure equals_ptr (z : ptr) := Pack {unpack : ptr}

Canonical equate (z : ptr) := Pack z z

Structure check (x y : ptr) (s : seq ptr) :=
  Check { y_of : equals_ptr y;
          _ : uniq s → x != unpack y_of }
Canonical start x y (s2 : search2 x y) :=
  Check x y (untag2 (seq2_of s2)) (equate y) ...
```

Here, the instance equate guarantees that the canonical value of type equals_ptr z is a package containing the pointer z itself.

We can now revise our statement of the overloaded noaliasR_fwd lemma ever so slightly to mention the new projector unpack (which could be made implicit). We also rename it, since this is the final presentation of the lemma:

$$\begin{aligned}\mathsf{noaliasR} : \forall s : \mathsf{seq\ ptr}.\, \forall f : \mathsf{scan}\ s.\, \forall d : \mathsf{def}\, f.\, \forall x\ y : \mathsf{ptr}.\, \forall g : \mathsf{check}\ x\ y\ s.\\ x\ \mathsf{!=}\ \mathsf{unpack}\ (\mathsf{y\_of}\ g)\end{aligned}$$

As before, suppose that noaliasR has already been applied to a hypothesis D of type def h, so that the lemma's parameter s has already been solved for. Then, when noaliasR is applied to a goal x_1 != x_3, the unification engine will unify x_1 with the argument $?x$ of noaliasR, and proceed to unify

$$\mathsf{unpack}\ (\mathsf{y\_of}\ ?g) \approx x_3$$

in a context where $?g : \mathsf{check}\ x_1\ ?y\ s$. In order to fully understand what is going on, we detail the steps involved in the instance search. The only instance applicable is equate. After opening the instance by creating the unification variable $?z$, Coq unifies the type of the instance with the type of y_of $?g$:

$$\mathsf{equals\_ptr}\ ?z \approx \mathsf{equals\_ptr}\ ?y$$

and obtains the solution $?z = ?y$. Then, the unpack projection from equate $?z$ (which is simply $?z$) is unified with the value it is supposed to match, namely x_3. This step is the key to understanding how we pick up x_3 from the goal. Replacing $?z$ with its solution $?y$, we thus get the equation

$$?y \approx x_3$$

Finally, Coq unifies the expected instance with the one computed:

$$\mathsf{y\_of}\ ?g \approx \mathsf{equate}\ x_3$$

which triggers the search for the pointers in s as before.

Applying The Lemma: The overloaded noaliasR lemma supports a number of modes of use: it can be applied, used as a rewrite rule, or composed with other lemmas. For example, assume that we have a hypothesis specifying a disjointness of a number of

heaps in a union:

$$D : \mathsf{def}\ (i_1 \bullet (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2) \bullet (i_2 \bullet x_3 \mapsto v_3))$$

Assume further that the arguments x, y, s, f and g of noaliasR are implicit, so that we can write simply (noaliasR D) when we want to partially instantiate the lemma with the hypothesis D. Then the following are some example goals, and proofs to discharge them, illustrating the flexibility of use. As can be seen, no tedious reordering of heap expressions by commutativity and associativity is needed.

1. The lemma can be used in backward reasoning. The type checker picks up x_1 and x_2 from the goal, and confirms they appear in D.

   Goal : x_1 != x_2
   Proof : by apply: (noaliasR D)

2. The lemma can be used in iterated rewriting. The lemma is partially instantiated with D. It performs the initial scan of D once, but is then used three times to reduce each conjunct to true. There is no need *in the proof* to specify the input pointers to be checked for aliasing. The type checker can pick them up from the goal, in the order in which they appear in the conjunction.

   Goal : $(x_1$!= $x_2)$ && $(x_2$!= $x_3)$ && $(x_3$!= $x_1)$
   Proof : by rewrite !(noaliasR D)

3. The lemma can be composed with other lemmas, to form new rewrite rules. Again, there is no need to provide the input pointers in the proofs. For example, given the standard library lemma negbTE : $\forall b$:bool. $!b = \mathsf{true} \rightarrow b = \mathsf{false}$, we have:

   Goal : if $(x_2$ == $x_3)$ && $(x_1$!= $x_2)$ then false else true
   Proof : by rewrite (negbTE (noaliasR D))

4. That said, we can provide the input pointers in several ways, if we wanted to, which would correspond to forward reasoning. We can use the term selection feature of rewrite to reduce only the specified conjunct in the goal.

   Goal : $((x_1$!= $x_2)$ && $(x_2$!= $x_3)) = (x_1$!= $x_2)$
   Proof : by rewrite $[x_2$!= $x_3]$(noaliasR D) andbT

   Here a rewrite by andbT : $\forall b.\, b$ && true $= b$ is used to remove the true left in the goal after rewriting by noaliasR.

5. Or, we can supply one (or both) of the pointer arguments directly to noaliasR.

$$\begin{array}{ll} \text{Goal} : & ((x_1 \texttt{ != } x_2) \texttt{ \&\& } (x_2 \texttt{ != } x_3)) = (x_1 \texttt{ != } x_2) \\ \text{Proof} : & \texttt{by rewrite (noaliasR } (x := x_2)\ D) \texttt{ andbT} \end{array}$$

$$\begin{array}{ll} \text{Goal} : & ((x_1 \texttt{ != } x_2) \texttt{ \&\& } (x_2 \texttt{ != } x_3)) = (x_1 \texttt{ != } x_2) \\ \text{Proof} : & \texttt{by rewrite (noaliasR } (y := x_3)\ D) \texttt{ andbT} \end{array}$$

2.7 Characteristics of Lemma Overloading

We conclude this chapter by discussing the properties of Lemma Overloading, as they were laid out in the introduction of the dissertation.

Maintainability. Since overloaded lemmas are typed, changes in definitions are immediately spot by the typechecker. Moreover, overloaded lemmas, such as regular lemmas, can be easily packed in modules to help reusability.

Composability. As we have shown in the previous section, composing two overloaded lemmas can be a trivial or a highly convoluted task, depending on the problem at hand. For instance, in Figure 2.9 the search2 overloaded lemma calls the search overloaded lemma without needing any particular technical device; however, this was not the case when trying to compose scan with search2. But where the technique of Lemma Overloading really shines is when tactics are extended by simple composition with regular lemmas, as shown at the end of the previous section.

Interactive Tactic Programming. A very distinctive characteristic of Lemma Overloading (and, as it will turn out, Mtac), with respect to other tactic languages, is its integration with the Coq proof environment. This comes from the fact that each instance is a plain Coq term, only with a particular treatment during unification. Therefore, building an instance interactively is as easy as building any other term interactively.

The code shown below is an example of interactive tactic programming applied to the search algorithm from Figure 2.10. In this case we consider two new instances to handle list concatenation. The first instance searches for the element in the list on the left of the concatenation symbol, while the second one searches for it in the list on the right. We omit the definition of the new tags, app_right_tag and app_left_tag, but they are standard.

```
Program Canonical app_right_form x s (f : search x) :=
  Search x (app_right_tag (s ++ f)) _.
Next Obligation.
  case: f⇒/= r H.
  rewrite /axiom in H ×.
  by rewrite mem_cat H orbT.
Qed.

Program Canonical app_left_form x s (f : search x) :=
  Search x (app_left_tag (f ++ s)) _.
Next Obligation.
  case: f⇒/= r H.
  rewrite /axiom in H ×.
  by rewrite mem_cat H.
Qed.
```

In this code the instances are created very much as the previous instances in Figure 2.10, except for two things: (1) we prepended the keyword **Program** to the declaration of the instances, and (2) we provided underscores (_) as the values for the proof component of the search structure.

The **Program** keyword (Sozeau, 2007) works as follows: after elaborating a term, Coq will ask us to construct interactively every missing part of the term (*i.e.*, uninstantiated meta-variables). In the Coq terminology, every such meta-variable is an obligation we must instantiate by iterating the list of obligations (via the **Next Obligation** keyword). The original term is completed when all the obligations are done with.

In the code above, the **Program** keyword is used to enable interactive tactic programming. The two underscores (one for each proof component of an instance) are transformed into meta-variables by Coq at elaboration. Since these meta-variables are left uninstantiated (Coq cannot guess the proofs), Coq asks us for the proofs as obligations. We then provide the proofs interactively, with the proof scripts listed.

Simplicity and Formality. Clearly the main drawback of Lemma Overloading is the difficulty to build and compose complex tactics, as we saw in Section 2.6. The semantics of Lemma Overloading relies heavily on the order in which subproblems are considered by both the elaboration and the unification algorithm. About the former, the closest

formal description we are aware of, is the elaboration algorithm for the CIC based proof assistant HELM Sacerdoti Coen (2004, chp. 10). Although it is not precisely what Coq does, it helps grasping and intuitive idea of what it does. For the elaboration algorithm, this intuitive idea is, according to our experience, enough for building overloaded lemmas.

However, intuition is not enough when it comes to unification. Although the patterns presented in this chapter help building most of the idioms required in a real development, the proof developer still has to carefully consider the order of unification subproblems. This order is difficult to grasp by mere intuition, and to the best of our knowledge, there is no up-to-date documentation explaining unification in Coq. It is possible to get a rough approximation by combining the unification algorithm of HELM, described also in Sacerdoti Coen (2004, chp. 10), with the description of canonical structures in Saïbi (1997). However, taking these sources to understand the underlying mechanism of Lemma Overloading can result in a backshot, as the difference between these works and the actual algorithm in Coq is quite big.

In Chapter 4 we provide a full-fleshed description of a new unification algorithm for Coq, considering CIC with Canonical Structures. Although it is not intended to be a precise description of the current algorithm in Coq, but rather of an improved algorithm, it still covers the main aspects of the current algorithm. And the proof of soundness of the unification algorithm is, for the moment, a conjecture.

Performance. The method presented in this chapter has some performance issues, mainly due to the following reasons:

1. There are no tools available to control the size of proof terms, and therefore proof terms generated using this method tend to grow. In particular, the excessive use of structure arguments generates several copies of objects. This problem can be witnessed in the proof term shown at the end of Section 2.2, where the input pointer z is copied twice, once for each instance of the structure.

2. The programming model is a restricted form of dependently typed logic programming, without a cut operator to reduce the size of the search tree.

3. The programming model does not allow for effective imperative data structures.

To a certain extent, a highly experienced proof developer may encode the algorithms in certain ways to overcome the first two issues. For (1), sometimes (but not always!) it is possible to encode the algorithm in a way that it requires fewer arguments in the type of the structure. And for (2), it is possible to force the unification algorithm to fail early

and, therefore, to backtrack early. Still, for the most of the cases, there is no way of eliminating all of the data duplication and backtracking.

Extensibility. Canonical structures, like any other overloading mechanism, exists precisely to extend dynamically the cases of a generic function. We already saw a simple example of an extensible algorithm in Section 2.5.2, where adding a new case to the step overloaded lemma is just a matter of adding a new canonical instance.

Now, in the presence of overlapping instances, where the tagging pattern is used, this is not the case. That is, we cannot add an instance just like that, as the list of instances is hardcoded in the list of tags. There is, however, a simple fix to the problem: use the tagging pattern locally, to solve the disambiguation of the overlapping instances. This is, in fact, the spirit of the step overloaded lemma: all the bind_∗ instances, which overlap on the bind constructor, are treated separately in their own structure. The main structure then uses this auxiliary structure to disambiguate the overlapping instances.

In the case of the step overloaded lemma, each of the instances of the auxiliary structure is uniquely determined by the constructor (read, write, etc.). Sometimes, however, the structure of the problem at hand requires us to use the tagging pattern in the auxiliary structure. In that case, the problem requires a bit more of thinking when combining the main structure with the auxiliary structure.

To illustrate, we consider a fully extensible version of the indom overloaded lemma from Section 2.3 . The original version uses a fixed list of tags, and therefore does not allow for new instances to be added. If we look at the constants we need to disambiguate, we find that only the disjoint union of heaps is the problematic one. Therefore, we can push the disambiguation of instances to that constructor only, and use an auxiliary structure to help disambiguate the two possible cases (when the pointer is in the heap on the left or on the right).

However, as it turns out, adding the tagging pattern to the auxiliary structure is not a trivial task. The reader is invited to try a naive approach; here we present the final solution, which uses the parametrized tagging pattern to trigger the search in both sides of the disjoint union operator.

In Figure 2.11 we present the revised version of the indom overloaded lemma. In the first lines (01–06) we create the main structure find as before, except that now the heap is not tagged. Between lines 08–20 we create a new structure, findU, that uses the tagging pattern to disambiguate the two instances for the union operator. Note that this structure is keyed on the pointer being searched for instead of the heap (as in

find). The reason is rather technical, and will become evident after seeing the use of the parametrized tagging pattern.

The parametrized tagging pattern is used between lines 22–32, with the structure checkH as the "glue code" to trigger the algorithm encoded in the findU structure. The following lines show the two instances of the main algorithm: the one for points-to, and the one for disjoint union. The former is standard, but the latter requires a bit of explanation. It uses the auxiliary checkH structure to trigger the search of the pointer of both heaps. The search for an instance c of this structure is done when the heap on the right of the operator is unified with the heap_ofH (implicit) projector of c. Since the heap is also an input of the algorithm, the heap_ofH field of checkH must use the parametrized tagging pattern.

When the unique instance of checkH, startH, is invoked, it calls further the findU algorithm to perform the actual search. The call is done, as mentioned above, by matching the pointer with the ptr_ofU (implicit) projector of findU. This is to make sure that both heaps are known before the search, since in startH the pointer is unified last, after the heaps h_1 and h_2.

The astute reader may wonder why we are not using the parametrized tagging pattern again, since we need the pointer as input of the algorithm in findU. The answer is: we can, but we do not need to in this particular case, for rather technical reasons that will become obvious when we look in details the unification algorithm in Chapter 4.

With the new definition of indom we can create a new heap operator and add a new instance for it. For instance, the code below defines the function array, that given an initial pointer x, a length n, and an initial value v, creates n adjoined pointers, starting from x, pointing to n copies of value v. The notation $x.+m$ creates a pointer m places higher in memory than x. Then, we add a new instance stating that the last pointer of the array is in the domain. (This instance is just for illustration purposes only; a real application will search for inclusion of any pointer between x and $x.+n$. In Section 3.3.3 we discuss a solution for a similar problem.)

```
Definition array A (x : ptr) (n : nat) (v : A) :=
  iteri n (λ m h. x .+ m ↦ v • h) empty.

Canonical array_last A m y (v : A) :=
  @Form (y .+ m) (array y m.+1 v) ...
```

```
01 Definition invariant x (h : heap) :=
02   def h → x ∈ dom h.
03
04 Structure find (x : ptr) :=
05   Form { heap_of :> heap;
06          _ : invariant x heap_of }.
07
08 Structure tagged_ptrU := TagU { untagU :> ptr }.
09 Definition right_tagU := TagU.
10 Canonical left_tagU x := right_tagU x.
11
12 Structure findU (h : heap) :=
13   FormU { ptr_ofU :> tagged_ptrU;
14           _ : invariant ptr_ofU h }.
15
16 Canonical search_leftU x (h : heap) (f : find x) :=
17   FormU (f • h) (left_tagU x) ...
18
19 Canonical search_rightU x (h : heap) (f : find x) :=
20   FormU (h • f) (right_tagU x) ...
21
22 Structure equals_heap (h : heap) :=
23   PackH { unpackH :> heap;
24           _ : unpackH = h }.
25 Canonical equateH h := PackH (erefl h).
26
27 Structure checkH (h1 h2 : heap) x :=
28   CheckH { heap_ofH :> equals_heap h2;
29            _ : invariant x (h1 • h2) }.
30
31 Canonical startH h1 h2 (f : findU (h1 • h2)) :=
32   CheckH h1 h2 f (equateH h2) ...
33
34 Canonical ptr_found A x (v : A) := Form x (x ↦ v) ...
35
36 Canonical findU_bridge x (h1 h2 : heap) (c : checkH h1 h2 x) :=
37   Form x (h1 • c) ...
38
39 indom (x : ptr) (f : find x) : def f → x ∈ dom f
```

FIGURE 2.11: Tagging pattern and parametrized tagging pattern combined to locally disambiguate instances.

Chapter 3

Mtac

We saw in the previous chapter how to encapsulate automation routines as *overloaded lemmas*. Like an ordinary lemma, an overloaded lemma has a precise formal specification in the form of a (dependent) Coq type. The key difference is that an overloaded lemma—much like an overloaded function in Haskell—is not proven (*i.e.*, implemented) once and for all up front; instead, every time the lemma is applied to a particular goal, the system will run a user-specified automation routine in order to construct a proof on the fly for that particular instance of the lemma. To program the automation routine, one uses canonical structure to declare a set of proof-building rules—implemented as Coq terms—that will be fired in a predictable order by the Coq unification algorithm (but may or may not succeed). In effect, one encodes one's automation routine as a *dependently typed logic program* to be executed by Coq's type inference engine.

The major benefit of this approach is its integration into Coq: it enables users to program tactics in Coq directly, rather than in a separate language, while at the same time offering significant additional expressive power beyond what is available in the base logic of Coq. The downside, however, is that the logic-programming style of canonical structures is in most cases not as natural a fit for tactics as a *functional-programming* style would be.[1] Moreover, canonical structures provide a relatively low-level language for writing tactics. The control flow of sophisticated canonical structure programs depends closely on how Coq type inference is implemented, and thus writing even simple tactics requires one to think at the level of the Coq unification algorithm, sometimes embracing its limitations and sometimes working around them. The series of "design patterns" presented in the previous chapter were designed precisely for programming effectively with canonical structures. While these design patterns are clearly useful, the desire for them nonetheless suggests that there is a *high-level language* waiting to be born.

[1] In terms of expressivity, there are tradeoffs between the two styles—for further discussion, see Section 3.7.

3.1 Mtac: A Monad for Typed Tactic Programming in Coq

In this chapter, we present a new language—**Mtac**—for typed tactic programming in Coq. Like Beluga and VeriML, Mtac supports general-purpose tactic programming in a direct functional style. Unlike those languages, however, Mtac is not a separate language, but rather a simple extension to Coq. As a result, Mtac tactics (or as we call them, **Mtactics**) have access to all the features of ordinary Coq programming *in addition to* a new set of tactical primitives. Furthermore, like overloaded lemmas, their (partial) correctness is specified statically within the Coq type system itself, and they are fully integrated into Coq, so they can be programmed and used interactively. Mtac is thus, to our knowledge, the first language to support interactive, dependently typed tactic programming.

The key idea behind Mtac is dead simple. We encapsulate tactics in a *monad*, thus avoiding the need to change the base logic and trusted kernel typechecker of Coq *at all*. Then, we modify the Coq infrastructure so that it executes these monadic tactics, when requested to do so, *during* type inference (*i.e.*, during interactive proof development or when executing a proof script).

More concretely, Mtac extends Coq with:

1. An inductive type family $\bigcirc\tau$ (read as "maybe τ") classifying Mtactics that—if they terminate successfully—will produce Coq terms of type τ. The constructors of this type family essentially give the syntax for a monadically-typed tactic language: they include the usual monadic return and bind, as well as a suite of combinators for tactic programming with fixed points, exception handling, pattern matching, and more. (Note: the definition of the type family $\bigcirc\tau$ does not *per se* require any extension to Coq—it is just an ordinary inductive type family.)

2. A primitive *tactic execution* construct, **run** t, which has type τ assuming its argument t is a tactic of type $\bigcirc\tau$. When (our instrumentation of) the Coq type inference engine encounters **run** t, it executes the tactic t. If that execution terminates, it will either produce a term u of type τ (in which case Coq will rewrite **run** t to u) or else an uncaught exception (which Coq will report to the user). If a proof passes entirely through type inference without incurring any uncaught exceptions, that means that all instances of **run** in the proof must have been replaced with standard Coq terms. Hence, there is no need to extend the trusted kernel typechecker of Coq to handle **run**.

```
01 Definition search (x : A) :=
02   mfix f (s : seq A) :=
03     mmatch s as s' return ○(x ∈ s') with
04     | [l r] l ++ r ⇒
05       mtry
06         il ← f l;
07         ret (in_or_app l r x (or_introl il))
08       with _ ⇒
09         ir ← f r;
10         ret (in_or_app l r x (or_intror ir))
11       end
12     | [s'] (x :: s') ⇒ ret (in_eq _ _)
13     | [y s'] (y :: s') ⇒
14       r ← f s';
15       ret (in_cons y r)
16     | _ ⇒ raise NotFound
17     end.
```

FIGURE 3.1: Mtactic for searching in a list.

Example: Searching in a List. To get a quick sense of what Mtac programming is like, consider the example in Figure 3.1. Here, search is a tactical term of type $\forall x : A.\ \forall s :$ seq $A.\ \bigcirc(x \in s)$. When executed, search x s will search for an element x (of type A) in a list s (of type seq A), and if it finds x in s, it will return a proof that $x \in s$. Note, however, that search x s itself is just a Coq term of monadic type $\bigcirc(x \in s)$, and that the execution of the tactic will only occur when this term is **run**.

The implementation of search relies on four new features of Mtac that go beyond what is possible in ordinary Coq programming: it iterates using a potentially unbounded fixed point **mfix** (line 2), it case-analyzes the input list s using a new **mmatch** constructor (line 3), it **raise**-s an exception NotFound if the element x was not found (line 16), and this exception is caught and handled (for backtracking purposes) using **mtry** (line 5). These new features, which we will present in detail in Section 3.2, are all constructors of the inductive type family $\bigcirc\tau$. Regarding **mmatch**, the reason it is different from ordinary Coq **match** is that it supports pattern-matching not only against primitive datatype constructors (*e.g.*, [] and ::) but also against arbitrary terms (*e.g.*, applications of the ++ function for concatenating two lists). For example, search starts out (line 4) by checking whether s is an application of ++ to two subterms l and r. If so, it searches for x first in l and then in r. In this way, **mmatch** supports case analysis of the intensional syntactic structure of open terms, in the manner of VeriML's **holcase** (Stampoulis and Shao, 2010) and Beluga's **case** (Pientka, 2008).

By **run**-ning search, we can now, for example, very easily prove the following lemma establishing that z is in the list $[x; y; z]$:

$$\textbf{Lemma}\ \mathsf{z\_in\_xyz}\ (x\ y\ z : A) : z \in [x; y; z] := \textbf{run}\ (\mathsf{search}\ \_\ \_)$$

Note here that we did not even need to supply the inputs to search explicitly: they were picked up from context, namely the goal of the lemma ($z \in [x; y; z]$), which Coq type inference proceeds to unify with the output type of the Mtactic search.

3.1.1 Chapter Overview

In the remainder of this chapter, we will:

- Describe the design of Mtac in detail (§3.2).
- Give a number of examples to concretely illustrate the benefits of Mtac programming (§3.3).
- Present the formalization of Mtac, along with meta-theoretic results such as type safety (§3.4).
- Explore some technical issues regarding the integration of Mtac into Coq (§3.5).
- Extend the language to support *stateful* Mtactics (§3.6).

The Coq patch and the examples can be downloaded from:

`http://plv.mpi-sws.org/mtac`

3.2 Mtac: A Language for Proof Automation

In this section, we describe the syntax and typing of Mtac, our language for typed proof automation.

Syntax of Mtac. Mtac extends CIC, the Calculus of (co-)Inductive Constructions (see *e.g.*, Bertot and Castéran (2004)), with a monadic type constructor $\bigcirc\tau$, representing tactic computations returning results of type τ, along with suitable introduction and elimination forms for such computations. We define $\bigcirc$: Type $\to$ Prop as a normal CIC inductive predicate with constructors reflecting our syntax for tactic programming, which are shown in Fig. 3.2. (We prefer to define $\bigcirc$ inductively instead of axiomatizing

$$
\begin{array}{ll}
\bigcirc & : \mathsf{Type} \to \mathsf{Prop} \\
\textbf{ret} & : \forall A.\ A \to \bigcirc A \\
\textbf{bind} & : \forall A\ B.\ \bigcirc A \to (A \to \bigcirc B) \to \bigcirc B \\
\textbf{raise} & : \forall A.\ \mathsf{Exception} \to \bigcirc A \\
\textbf{mtry} & : \forall A.\ \bigcirc A \to (\mathsf{Exception} \to \bigcirc A) \to \bigcirc A \\
\textbf{mfix} & : \forall A\ P.\ ((\forall x : A.\ \bigcirc(P\ x)) \to (\forall x : A.\ \bigcirc(P\ x))) \\
 & \quad \to \forall x : A.\ \bigcirc(P\ x) \\
\textbf{mmatch} & : \forall A\ P\ (t : A).\ \mathsf{seq}\ (\mathsf{Patt}\ A\ P) \to \bigcirc(P\ t) \\
\textbf{print} & : \forall s : \mathsf{string}.\ \bigcirc\mathsf{unit} \\
\textbf{nu} & : \forall A\ B.\ (A \to \bigcirc B) \to \bigcirc B \\
\textbf{abs} & : \forall A\ P\ x.\ P\ x \to \bigcirc(\forall y : A.\ P\ y) \\
\textbf{is\_var} & : \forall A.\ A \to \bigcirc\mathsf{bool} \\
\textbf{evar} & : \forall A.\ \bigcirc A \\
\textbf{is\_evar} & : \forall A.\ A \to \bigcirc\mathsf{bool} \\
\\
\mathsf{Patt} & : \forall A\ (P : A \to \mathsf{Type}).\ \mathsf{Type} \\
\mathsf{Pbase} & : \forall A\ P\ (p : A)\ (b : \bigcirc(P\ p)).\ \mathsf{Patt}\ A\ P \\
\mathsf{Ptele} & : \forall A\ P\ C.\ (\forall x : C.\ \mathsf{Patt}\ A\ P) \to \mathsf{Patt}\ A\ P
\end{array}
$$

FIGURE 3.2: The $\bigcirc$ and Patt inductive types.

it in order to cheaply ensure that we do not affect the logical consistency of CIC.) The $\bigcirc$ constructors include standard monadic return and bind (**ret**, **bind**), primitives for throwing and handling exceptions (**raise**, **mtry**), a fixed point combinator (**mfix**), a pattern matching construct (**mmatch**), and a printing primitive useful for debugging Mtactics (**print**). Mtac also provides more specialized operations for handling parameters and unification variables (**nu**, **abs**, **is_var**, **evar**, **is_evar**), but we defer explanation of those features until §3.3.2.

First, let us clear up a somewhat technical point. The reason we define $\bigcirc$ as an inductive *predicate* (*i.e.*, whose return sort is Prop rather than Type) has to do with the handling of **mfix**. Specifically, in order to satisfy Coq's syntactic positivity condition on inductive definitions, we cannot declare **mfix** directly with the type given in Figure 3.2, since that type mentions the monadic type constructor $\bigcirc$ in a negative position. To work around this, in the inductive definition of $\bigcirc\tau$, we replace the **mfix** constructor with a variant, **mfix′**, in "Mendler style" (Hur et al., 2013, Mendler, 1991), *i.e.*, in which references to $\bigcirc$ are substituted with references to a *parameter* $\odot$:

$$
\begin{array}{l}
\textbf{mfix}' : \forall A\ P\ \odot.\ (\forall x : A.\ \odot(P\ x) \to \bigcirc(P\ x)) \to \\
\qquad ((\forall x{:}A.\ \odot(P\ x)) \to (\forall x{:}A.\ \odot(P\ x))) \to \forall x{:}A.\ \bigcirc(P\ x)
\end{array}
$$

The **mfix** from Figure 3.2 is then recovered simply by instantiating the $\odot$ parameter of **mfix′** with $\bigcirc$, and instantiating its first value parameter with the identity function. However, due to the inherent "circularity" of this trick, it only works if the type $\bigcirc\tau$

belongs to an impredicative sort like Prop. (In particular, if $\bigcirc\tau$ were defined in Type, then while **mfix′** would be well-formed, applying **mfix′** to the identity function in order to get an **mfix** would cause a universe inconsistency.) Fortunately, defining $\bigcirc\tau$ in Prop has no practical impact on Mtac programming. Note that, in CIC, Prop : Type; so it is possible to construct nested types such as $\bigcirc(\bigcirc\tau)$.

Now, on to the features of Mtac. The typing of monadic **ret** and **bind** is self-explanatory. The exception constructs **raise** and **mtry** are also straightforward: their types assume the existence of an exception type Exception. It is easy to define such a type, as well as a way of declaring new exceptions of that type, in existing Coq (see §3.5.4 for details). The print statement **print** takes the string to print onto the standard output and returns the trivial element.

Pattern matching, **mmatch**, expects a term of type A and a sequence of pattern matching clauses of type Patt A P, which match objects x of type A and return results of type P x. Binding in the pattern matching clauses is represented as a *telescope*: Pbase p b describes a ground clause that matches the constant p and has body b, and Ptele($\lambda x.$ pc) adds the binder x to the pattern matching clause pc. So, for example, Ptele($\lambda x.$ Ptele($\lambda y.$ Pbase $(x + y)$ b) represents the clause that matches an addition expression, binds the left subexpression to x and the right one to y, and then returns some expression b which can mention both x and y. Another example is the clause, Ptele($\lambda x.$ Pbase x b) which matches any term and returns b.

Note that it is also fine for a pattern to mention free variables bound in the ambient environment (*i.e.*, not bound by the telescope pattern). Such patterns enable one to check that (some component of) the term being pattern-matched is unifiable with a specific term of interest. We will see examples of this in the search2 and lookup Mtactics in §3.3.

Just as we did in the previous chapter, we often omit inferrable type annotations. Also, in our examples and in our Coq development we use the following notation to improve

readability of Mtactics:

$x \leftarrow t; t'$	denotes	**bind** t $(\lambda x.\ t')$
mfix f $(x : \tau) : \bigcirc\tau' := t$	denotes	**mfix** $(\lambda x : \tau.\ \tau')$ $(\lambda f.\ \lambda x.\ t)$
$\nu x : A.\ t$	denotes	**nu** $(\lambda x : A.\ t)$
mmatch t	denotes	**mmatch** $(\lambda x.\ \tau)$ t
as x **return** $\bigcirc\tau$ **with**		[
$\mid [\overline{x_1}]\ p_1 \Rightarrow b_1$		Ptele $\overline{x_1}$ (Pbase p_1 b_1),
...		...
$\mid [\overline{x_m}]\ p_m \Rightarrow b_m$		Ptele $\overline{x_m}$ (Pbase p_m b_m)
end		]
mtry t	denotes	**mtry** t $(\lambda x.$
with ps **end**		**mmatch** x **with** ps **end**)

where Ptele $x_1 \cdots x_n$ p means Ptele$(\lambda x_1.\ \cdots.$ Ptele$(\lambda x_n.\ p)\cdots)$. Both type annotations in the **mfix** and in the **mmatch** notation (in the latter, denoted **as** x **return** $\bigcirc\tau$) are optional and can be omitted, in which case the returning type is left to the type inference algorithm to infer. The **mfix** construct accepts up to 5 arguments.

Running Mtactics. Defining $\bigcirc$ as an inductive predicate means that terms of type $\bigcirc\tau$ can be destructed by case analysis and induction. Unlike other inductive types, $\bigcirc$ supports an additional destructor: *tactic execution*. Formally, we extend Coq with a new construct, **run** t, that takes an Mtactic t of type $\bigcirc\tau$ (for some τ), and runs it *at type-inference time* to return a term u of type τ.

$$\frac{\Gamma \vdash t : \bigcirc\tau \qquad \Gamma \vdash t \rightsquigarrow^* \textbf{ret}\ u}{\Gamma \vdash \textbf{run}\ t : \tau}$$

We postpone the definition of the tactic evaluation relation, $\rightsquigarrow$, as well as a precise formulation of the rule, to §3.4, but note that since tactic evaluation is type-preserving, u has type τ, and thus τ is inhabited.

3.3 Mtac by Example

In this section, we offer a gentle introduction to the various features of Mtac by working through a sequence of proof automation examples.

3.3.1 noalias in Mtac

Our first example is a new version of the example that motivated Section 2.6, the lemma noalias about non-aliasing of disjointly allocated pointers. The goal is to prove that two pointers are distinct, given the assumption that they appear in the domains of disjoint subheaps of a well-defined memory.

In Section 2.6, the noalias example was used to illustrate a rather subtle and sophisticated design pattern for composition of overloaded lemmas. Here, it will help illustrate the main characteristics of Mtac, while at the same time emphasizing the relative simplicity and readability of Mtactics compared to previous approaches.

Motivating Example. Let us state a goal we would like to solve automatically:

$$\frac{D : \mathsf{def}\ (h_1 \bullet (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2) \bullet (h_2 \bullet x_3 \mapsto v_3))}{x_1\ \mathsf{!=}\ x_2 \wedge x_2\ \mathsf{!=}\ x_3}$$

Above the line is a hypothesis concerning the well-definedness of a heap mentioning x_1, x_2, and x_3, and below the line is the goal, which is to show that x_1 is distinct from x_2, and x_2 from x_3.

Intuitively, the truth of the goal follows obviously from the fact that x_1, x_2, and x_3 appear in disjoint subheaps of a well-defined heap. This intuition is made formal with the following lemma (in plain Coq):

$$\begin{aligned}\mathsf{noalias\_manual} : \forall(h{:}\mathsf{heap})\ (y_1\ y_2{:}\mathsf{ptr})\ (w_1{:}A_1)\ (w_2{:}A_2).\\ \mathsf{def}\ (y_1 \mapsto w_1 \bullet y_2 \mapsto w_2 \bullet h) \to y_1\ \mathsf{!=}\ y_2\end{aligned}$$

Unfortunately, we cannot apply this lemma using hypothesis D as it stands, since the heap that D proves to be well-defined is not of the form required by the premise of the lemma—that is, with the pointers in question (x_1 and x_2, or x_2 and x_3) at the very *front* of the heap expression. It is of course possible to solve the goal by: (a) repeatedly applying rules of associativity and commutativity for heap expressions in order to rearrange the heap in the type of D so that the relevant pointers are at the front of the heap expression; (b) applying the noalias_manual lemma to solve the first inequality; and then repeating (a) and (b) to solve the second inequality.

But we would like to do better. What we really want is an Mtactic that will solve these kinds of goals automatically, no matter where the pointers we care about are located inside the heap. One option is to write an Mtactic to perform all the rearrangements necessary to put the two pointers at the front, and then apply the lemma above. This

```
01 Record form h := Form {
02         seq_of :> seq ptr;
03         axiom_of : def h → uniq seq_of
04                         ∧ ∀ x. x ∈ seq_of → x ∈ dom h }.
05
06 Definition scan :=
07   mfix f (h : heap) : ○(form h) :=
08     mmatch h with
09     | [x A (v:A)] x ↦ v ⇒ ret (Form [x] ...)
10     | [l r] l • r ⇒
11       rl ← f l;
12       rr ← f r;
13       ret (Form (seq_of rl ++ seq_of rr) ...)
14     | [h'] h' ⇒ ret (scan_h [] ...)
15     end.
```

FIGURE 3.3: Mtactic for scanning a heap to obtain a list of pointers.

approach has two drawbacks: it is computationally expensive, and it generates big proof terms. Both problems comes from the observation that the algorithm has to rearrange the heap twice for each inequality, moving each pointer in the inequality to the front of the heap by multiple applications of the commutativity and associativity lemmas.

Instead, we pursue a solution analogous to the one in Section 2.6, breaking the problem into two smaller Mtactics scan and search2, combined in a third Mtactic, noalias.

The Mtactic scan. Figure 3.3 presents the Mtactic scan, the functional analogous to the overloaded lemma with the same name. It scans its input heap h to produce a list of the pointers x appearing in singleton heaps $x \mapsto v$ in h. More specifically, it returns a dependent record[2] containing a list of pointers (seq_of, of type seq ptr), together with a proof that, if h is well-defined, then (1) the list seq_of is "unique" (denoted uniq seq_of), meaning that all elements in it are distinct from one another, and (2) its elements all belong to the domain of the heap.

To do this, scan inspects the heap and considers three different cases. If the heap is a singleton heap $x \mapsto v$, then it returns a singleton list containing x. If the heap is the disjoint union of heaps l and r, it proceeds recursively on each subheap and returns the concatenation of the lists obtained in the recursive calls. Finally, if the heap does not match any of the previous cases, then it returns an empty list. Note that this case analysis is not possible using Coq's standard **match** mechanism, because **match** only pattern-matches against primitive datatype constructors. In the case of heaps, which

[2]The keyword **Record** is a synonym of **Structure**, but we use the former to avoid confusion with the structures that we use for overloading.

```
01 Definition search2 x y :=
02   mfix f (s : seq ptr) : ○(uniq s → x != y) :=
03     mmatch s with
04     | [s'] x :: s' ⇒ r ← search y s'; ret (foundx_pf x r)
05     | [s'] y :: s' ⇒ r ← search x s'; ret (foundy_pf y r)
06     | [z s'] z :: s' ⇒ r ← f s'; ret (foundz_pf z r)
07     | _ ⇒ raise NotFound
08     end.
```

FIGURE 3.4: Mtactic for searching for two pointers in a list.

are really finite maps from pointers to values, $x \mapsto v$ and $l \bullet r$ are applications not of primitive datatype constructors but of defined functions ($\mapsto$ and $\bullet$). Thus, in order to perform our desired case analysis, we require the ability of Mtac's **mmatch** mechanism to pattern-match against the *syntax* of heap expressions.

In each case, scan also returns a proof that the output list obeys the aforementioned properties (1) and (2). For presentation purposes, we follow the notation convention 5 and omit these proofs (denoted with ... in the figures), but they are proven as standard Coq lemmas. (We will continue to omit proofs in this way throughout the chapter when they present no interesting challenges. The reader can find them in the source files (Ziliani, 2014).)

The Mtactic search2. Figure 3.4 presents the Mtactic search2. It takes two elements x and y and a list s as input, and searches for x and y in s. If successful, search2 returns a proof that, if s is unique, then x is distinct from y. Similarly to scan, this involves a syntactic inspection and case analysis of the input list s.

When s contains x at the head (*i.e.*, s is of the form $x :: s'$), search2 searches for y in the tail s', using the Mtactic search from Section 3.1. If this search is successful, producing a proof $r : y \in s'$, then search2 concludes by composing this proof together with the assumption that s is unique, using the easy lemma foundx_pf:

$$\begin{aligned} \mathsf{foundx\_pf} : \forall x\ y : \mathsf{ptr}.\ \forall s : \mathsf{seq\ ptr}.\\ y \in s \to \mathsf{uniq}\ (x :: s)\ \to x\ \mathsf{!=}\ y \end{aligned}$$

(In the code, the reader will notice that foundx_pf is not passed the arguments y and s explicitly. That is because we treat y and s as implicit arguments because they are inferrable from the type of r.)

```
Definition noalias h (D : def h) : ○(∀ x y. ○(x != y)) :=
  sc ← scan h;
  ret (λ x y.
    s2 ← search2 x y (seq_of sc);
    ret (combine s2 D)).
```

FIGURE 3.5: Mtactic for proving that two pointers do not alias.

If s contains y at the head, search2 proceeds analogously. If the head element is different from both x and y, then it calls itself recursively with the tail. In any other case, it throws an exception.

Note that, in order to test whether the head of s is x or y, we rely crucially on the ability of patterns to mention free variables from the context. In particular, the difference between the first two cases of search2's **mmatch** and the last one is that the first two do *not* bind x and y in their telescope patterns (thus requiring the head of the list in those cases to be syntactically unifiable with x or y, respectively), while the third *does* bind z in its telescope pattern (thus enabling z to match anything).

The Mtactic noalias. The Mtactics shown above are a direct translation from the ones in Section 2.6. But the Mtactic noalias presented in Figure 3.5, which stitches scan and search2 together, shows how easy is the composition of other Mtactics is. Compare the code from this figure with the code required for the same problem in Section 2.6.5!

The type of noalias is as follows:

$$\forall h : \mathsf{heap}.\ \mathsf{def}\ h \to \bigcirc(\forall x\ y.\ \bigcirc(x\ \mathsf{!=}\ y))$$

As the two occurrences of ○ indicate, this Mtactic is *staged*: it takes as input a proof that h is defined and first runs the scan Mtactic on h, producing a list of pointers sc, but then it immediately returns *another* Mtactic. This latter Mtactic in turn takes as input x and y and searches for them in sc. The reason for this staging is that we may wish to prove non-aliasing facts about different pairs of pointers in the same heap. Thanks to staging, we can apply noalias to some D just once and then reuse the Mtactic it returns on many different pairs of pointers, thus avoiding the need to rescan h redundantly.

At the end, the proofs returned by the calls to scan and search2 are composed using a combine lemma with the following type:

```
Lemma combine h x y (sc : form h) :
  (uniq (seq_of sc) → x != y) → def h → x != y.
```

This lemma is trivial to prove by an application of the cut rule.

Applying the Mtactic noalias**.** The following script shows how noalias can be invoked in order to solve the motivating example from the beginning of this section:

$$\begin{array}{l}\mathsf{pose}\ F := \textbf{run}\ (\mathsf{noalias}\ D)\\ \mathsf{by}\ \mathsf{split};\ \mathsf{apply}\text{:}\ \textbf{run}\ (F\ \_\ \_)\end{array}$$

When Coq performs type inference on the **run** in the first line, that forces the execution of (the first scan-ning phase of) the Mtactic noalias on the input hypothesis D, and the standard pose mechanism then binds the result to F. This F has the type

$$\forall x\ y : \mathsf{ptr}.\ \bigcirc(x\ \mathsf{!=}\ y)$$

In the case of our motivating example, F will be an Mtactic that, when passed inputs x and y, will search for those pointers in the list $[x_1; x_2; x_3]$ output by the scan phase.

The script continues with Coq's standard split tactic, which generates two subgoals, one for each proposition in the conjunction. For our motivating example, it generates subgoals x_1 != x_2 and x_2 != x_3. We then solve both goals by executing the Mtactic F. When F is **run** to solve the first subgoal, it will search for x_1 and x_2 in $[x_1; x_2; x_3]$ and succeed; when F is **run** to solve the second subgoal, it will search for x_2 and x_3 in $[x_1; x_2; x_3]$ and succeed. QED. Note that we provide the arguments to F *implicitly* (as _). As in the proof of the z_in_xyz lemma from §3.1, these arguments are inferred from the respective goals being solved. (We will explain how this inference works in more detail in §3.5.)

3.3.1.1 Developing Mtactics Interactively

As with Lemma Overloading, Mtac shares the key advantage that it works very well with the rest of Coq, allowing us among other things to develop Mtactics interactively.

For instance, consider the code shown in Figure 3.6. This is an interactive development of the search2 Mtactic, where the developer knows the overall search structure in advance, but not the exact proof terms to be returned, as this can be difficult in general. Here, we have prefixed the definition with the keyword **Program** (Sozeau, 2007), which allows us to omit certain parts of the definition by writing underscores. **Program** instructs the type inference mechanism to treat these underscores as unification variables, which—unless instantiated during type inference—are exposed as proof obligations. In our case, none of these underscores is resolved, and so we are left with three proof obligations. Each

```
01 Program Definition interactive_search2 x y :=
02   mfix f (s : seq ptr) : ○(uniq s → x != y) :=
03     mmatch s with
04     | [s'] x :: s' ⇒ r ← search y s'; ret _
05     | [s'] y :: s' ⇒ r ← search x s'; ret _
06     | [z s'] z :: s' ⇒ r ← f s'; ret _
07     | _ ⇒ raise NotFound
08     end.
09 Next Obligation. ... Qed.
10 Next Obligation. ... Qed.
11 Next Obligation. ... Qed.
```

FIGURE 3.6: Interactive construction of search2 using **Program**.

of these obligations can then be solved interactively within a **Next Obligation** ... **Qed** block.

Finally, it is worth pointing out that within such blocks, as well as within the actual definitions of Mtactics, we could be running other more primitive Mtactics.

3.3.2 tauto: A Simple First-Order Tautology Prover

With this next example, we show how Mtac provides a simple but useful way to write tactics that manipulate contexts and binders. Specifically, we will write an Mtactic implementing a rudimentary tautology prover, modeled after those found in the work on VeriML (Stampoulis and Shao, 2010) and Chlipala's CPDT textbook (Chlipala, 2011a). Compared to VeriML, our approach has the benefit that it does not require any special type-theoretic treatment of contexts: for us, a context is nothing more than a Coq list. Compared to Chlipala's Ltac version, our version is typed, offering a clear static specification of what the tautology prover produces, if it succeeds.

To ease the presentation, we break the problem in two. First, we show a simple propositional prover that uses the language constructs we have presented so far. Second, we extend this prover to handle first-order logic, and we use this extension to motivate some additional features of Mtac.

Warming up the Engine: A Simple Propositional Prover. Figure 3.7 displays the Mtactic for a simple propositional prover, taking as input a proposition p and, if successful, returning a proof of p:

$$\mathsf{prop\text{-}tauto} : \forall p : \mathsf{Prop}.\ \bigcirc p$$

```
01 Definition prop-tauto :=
02   mfix f (p : Prop) : ○p :=
03     mmatch p as p' return ○p' with
04     | True ⇒ ret I
05     | [p1 p2] p1 ∧ p2 ⇒
06         r1 ← f p1;
07         r2 ← f p2;
08         ret (conj r1 r2)
09     | [p1 p2] p1 ∨ p2 ⇒
10         mtry
11           r1 ← f p1; ret (or_introl r1)
12         with _ ⇒
13           r2 ← f p2; ret (or_intror r2)
14         end
15     | _ ⇒ raise NotFound
16     end.
```

FIGURE 3.7: Mtactic for a simple propositional tautology prover.

The Mtactic only considers three cases:

- p is True. In this case, it returns the trivial proof I.
- p is a conjunction of p_1 and p_2. In this case, it proves both propositions and returns the introduction form of the conjunction (conj r_1 r_2).
- p is a disjunction of p_1 and p_2. In this case, it tries to prove the proposition p_1, and if that fails, it tries instead to prove the proposition p_2. The corresponding introduction form of the disjunction is returned (or_introl r_1 or or_intror r_2).
- Otherwise, it raises an exception, since no proof could be found.

Extending to First-Order Logic. We now extend the previous prover to support first-order logic. This extension requires the tactic to keep track of a context for hypotheses, which we model as a list of (dependent) pairs pairing hypotheses with their proofs. More concretely, each element in the hypothesis context has the type dyn $= \Sigma p :$ Prop. p. (In Coq, this is encoded as an inductive type with constructor Dyn p x, for any $x : p$.)

Figure 3.8 shows the first-order logic tautology prover tauto. The fixed point takes the proposition p and is additionally parameterized over a context (c : seq dyn). The first three cases of the **mmatch** are similar to the ones in Figure 3.7, with the addition that the context is passed around in recursive calls.

Before explaining the cases for $\to$, $\forall$ and $\exists$, let us start with the last one (line 29), since it is the easiest. In this last case, we attempt to prove the proposition in question by

```
01  Definition tauto' :=
02    mfix f (c : seq dyn) (p : Prop) : ○p :=
03      mmatch p as p' return ○p' with
04      | True ⇒ ret I
05      | [p1 p2] p1 ∧ p2 ⇒
06            r1 ← f c p1 ;
07            r2 ← f c p2 ;
08            ret (conj r1 r2)
09      | [p1 p2] p1 ∨ p2 ⇒
10            mtry
11              r1 ← f c p1 ; ret (or_introl r1)
12            with _ ⇒
13              r2 ← f c p2 ; ret (or_intror r2)
14            end
15      | [p1 p2 : Prop] p1 → p2 ⇒
16            ν (y:p1).
17              r ← f (Dyn p1 y :: c) p2;
18              abs y r
19      | [A (q:A → Prop)] (∀ x:A. q x) ⇒
20            ν (y:A).
21              r ← f c (q y);
22              abs y r
23      | [A (q:A → Prop)] (∃ x:A. q x) ⇒
24            X ← evar A;
25            r ← f c (q X);
26            b ← is_evar X;
27            if b then raise ProofNotFound
28            else ret (ex_intro q X r)
29      | [p':Prop] p' ⇒ lookup p' c
30      end.
```

FIGURE 3.8: Mtactic for a simple first-order tautology prover.

simply searching for it in the hypothesis context. The search for the hypothesis p' in the context c is achieved using the Mtactic lookup shown in Figure 3.9. lookup takes a proposition p and a context and traverses the context linearly in the hope of finding a dependent pair with p as the first component. If it finds such a pair, it returns the second component. Like the Mtactic search2 from §3.3.1, this simple lookup routine depends crucially on the ability to match the propositions in the context *syntactically* against the p for which we are searching.

Returning to the tautology prover, lines 15–18 concern the case where $p = p_1 \to p_2$. Intuitively, in order to prove $p_1 \to p_2$, one would (1) introduce a *parameter* y witnessing the proof of p_1 into the context, (2) proceed to prove p_2 having $y : p_1$ as an assumption, and (3) abstract any usage of y in the resulting proof. The rationale behind this last step is that if we succeed proving p_2, then the result is *parametric* over the proof of p_1,

```
Definition lookup (p : Prop) :=
  mfix f (s : seq dyn) : ○p :=
    mmatch s return ○p with
    | [x s'] (Dyn p x) :: s' ⇒ ret x
    | [d s'] d :: s' ⇒ f s'
    | _ ⇒ raise ProofNotFound
    end.
```

FIGURE 3.9: Mtactic to look up a proof of a proposition in a context.

in the sense that any proof of p_1 will suffice to prove p_2. Steps (1) and (3) are performed by two of the operators we have not yet described: **nu** and **abs** (the former is denoted by the νx binder). In more detail, the three steps are:

Line 16: It creates a parameter $y : p_1$ using the constructor **nu**. This constructor has type

$$\textbf{nu} : \forall(A\ B : \mathsf{Type}).\ (A \to \bigcirc B) \to \bigcirc B$$

(where A and B are left implicit). It is similar to the operator with the same name in Nanevski (2002) and Schürmann et al. (2005). Operationally, $\nu x : \tau.\ f$ (which is notation for **nu** $(\lambda x : \tau.\ f)$) creates a parameter y with type A, pushes it into the local context, and executes $f\{y/x\}$ (where $\cdot\{\cdot/\cdot\}$ is the standard substitution) in the hope of getting a value of type B. If the value returned by f refers to y, then it causes the tactic execution to fail: such a result would lead to an ill-formed term because y is not bound in the ambient context. This line constitutes the first step of our intuitive reasoning: we introduce the parameter y witnessing the proof of p_1 into the context.

Line 17: It calls tauto' recursively, with context c extended with the parameter y, and with the goal of proving p_2. The result is bound to r. This line constitutes the second step.

Line 18: The result r created in the previous step has type p_2. In order to return an element of the type $p_1 \to p_2$, we abstract y from r, using the constructor

$$\begin{aligned}\textbf{abs} : \forall(A : \mathsf{Type})\ (P : A \to \mathsf{Type})\ (y : A).\\ P\ y \to \bigcirc(\forall x : A.\ P\ x)\end{aligned}$$

(with A, P implicit). Operationally, **abs** $y\ r$ checks that the first parameter y is indeed a variable, and returns the function

$$\lambda x : A.\ r\{x/y\}$$

In this case, the resulting element has type $\forall x : p_1.\ p_2$, which, since p_2 does not refer to x, is equivalent to $p_1 \to p_2$. This constitutes the last step: by abstracting over y in the result, we ensure that the resulting proof term no longer mentions the ν-bound variable (as required by the use of **nu** in line 16).

Lines 19–22 consider the case that the proposition is an abstraction $\forall x : A.\ q\ x$. Here, q is the body of the abstraction, represented as a function from A to Prop. We rely on Coq's use of higher-order pattern unification (Miller, 1991b) to instantiate q with a faithful representation of the body. The following lines mirror the body of the previous case, except for the recursive call. In this case we do not extend the context with the parameter y, since it is not a proposition. Instead, we try to recursively prove the body q replacing x with y (that is, applying q to y).

If the proposition is an existential $\exists x : A.\ q\ x$ (line 23), then the prover performs the following steps:

Line 24: It uses Mtac's **evar** constructor to create a fresh unification variable called X.

Line 25: It calls tauto′ recursively, replacing x for X in the body of the existential.

Lines 26–28: It uses Mtac's **is_evar** mechanism to check whether X is still an uninstantiated unification variable. If it is, then it raises an exception, since no proof could be found. If it is not—that is, if X was successfully instantiated in the recursive call—then it returns the introduction form of the existential, with X as its witness. Note that the instantiation of X may occur during the lookup of a hypothesis to solve the goal. That is, if the goal refers to X, and there is a hypothesis similar to the goal, but with some term t instead of X, then the pattern matching performed during the lookup will instantiate X with t.

Now we are ready to prove an example, where P : nat → Prop:

$$\textbf{Definition}\ \mathsf{exmpl} : \forall P\ x.\ P\ x \to \exists y.\ P\ y := \textbf{run}\,(\mathsf{tauto}\ []\ \_).$$

The proof term generated by **run** is

$$\mathsf{exmpl} = \lambda P\ x\ (H : P\ x).\ \mathsf{ex\_intro}\ P\ x\ H$$

3.3.3 Inlined Proof Automation

Due to the tight integration between Mtac and Coq, Mtactics can be usefully employed in definitions, notations and other Coq terms, in addition to interactive proving. In

this respect, Mtac differs from the related systems such as VeriML (Stampoulis and Shao, 2010) and Beluga (Pientka, 2008), where, to the best of our knowledge, such expressiveness is not currently available due to the strict separation between the object logic and the automation language.

In this section, we illustrate how Mtactics can be invoked from Coq proper. To set the stage, consider the scenario of developing a library for n-dimensional integer vector spaces, with the main type vector n defined as a record containing a list of nats and a proof that the list has size n:

```
Record vector (n : nat) := Vector {
  seq_of : seq nat;
  _ : size seq_of = n}.
```

One of the important methods of the library is the accessor function ith, which returns the i-th element of the vector, for $i < n$. One implementation possibility is for ith to check at run time if $i < n$, and return an option value to signal when i is out of bounds. The downside of this approach is that the clients of ith have to explicitly discriminate against the option value. An alternative is for ith to explicitly request a proof that $i < n$ as one of its arguments, as in the following type ascription:

$$\mathsf{ith} : \forall n{:}\mathsf{nat}.\mathsf{vector}\ n \to \forall i{:}\mathsf{nat}.i < n \to \mathsf{nat}$$

Then the clients have to construct a proof of $i < n$ before invoking ith, but we show that in some common situations, the proof can be constructed automatically by Mtac and then passed to ith.

Specifically, we describe an Mtactic compare, which automatically searches for a proof that two natural numbers n_1 and n_2 satisfy $n_1 \leq n_2$. compare is incomplete, and if it fails to find a proof, because the inequality does not hold, or because the proof is too complex, it raises an exception.

Once compare is implemented, it can be composed with ith as follows. Given a vector v whose size we denote as vsize v, and an integer i, we introduce the following notation, which invokes compare to automatically construct a proof that $i+1 \leq$ vsize v (equivalent to $i <$ vsize v).

```
Notation "[ 'ith' v i ]" :=
  (@ith _ v i (run (compare (i+1) (vsize v)))).
```

```
Program Definition compare (n1 n2 : nat) : ○(n1 ≤ n2) :=
  r1 ← to_ast [] n1;
  r2 ← to_ast (ctx_of r1) n2;
  match cancel (ctx_of r2) (term_of r1) (term_of r2)
        return ○(n1 ≤ n2) with
  | true ⇒ ret (@sound n1 n2 r1 r2 _)
  | _ ⇒ raise NotLeqException
  end.
Next Obligation. ... Qed.
```

FIGURE 3.10: Mtactic for proving inequalities between nat's.

The notation can be used in definitions. For example, given vectors v_1, v_2 of fixed size 2, we could define the inner product of v_1 and v_2 as follows, letting Coq figure out automatically that the indices $0, 1$ are within bounds.

```
Definition inner_prod (v1 v2 : vector 2) :=
          [ith v1 0] × [ith v2 0] + [ith v1 1] × [ith v2 1].
```

If we tried to add the summand $[\text{ith } v_1\ 2] \times [\text{ith } v_2\ 2]$, where the index 2 is out of bounds, then compare raises an exception, making the whole definition ill-typed. Similarly, if instead of vector 2, we used the type vector n, where n is a variable, the definition will be ill-typed, because there is no guarantee that n is larger than 1. On the other hand, the following is a well-typed definition, as the indices k and n are clearly within the bound $n + k + 1$.

```
Definition indexing n k (v : vector (n + k + 1)) :=
  [ith v k] + [ith v n].
```

We proceed to describe the implementation of compare, presented in Figure 3.10. compare is implemented similarly to the heap cancellation overloaded lemma from Section 2.4, using two main helper functions. The first is the Mtactic to_ast which *reflects*[3] the numbers n_1 and n_2. More concretely, to_ast takes an integer expression and considers it as a syntactic summation of a number of components. It *parses* this syntactic summation into an explicit list of summands, each of which can be either a constant or a free variable (subexpressions containing operations other than + are treated as free variables).

The second helper is a CIC function cancel which cancels the common terms from the syntax lists obtained by reflecting n_1 and n_2. If all the summands in the syntax list of

[3] In the literature this step is also known as *reification*.

n_1 are found in the syntax list of n_2, then it must be that $n_1 \leq n_2$ and cancel returns the boolean true. Otherwise, cancel does not search for other ways of proving $n_1 \leq n_2$ and simply returns false to signal the failure to find a proof. This failure ultimately results in compare raising an exception. Notice that cancel cannot directly work on n_1 and n_2, but has to receive their syntactic representation from to_ast (in the code of compare these are named term_of r_1 and term_of r_2, respectively). The reason is that cancel has to compare names of variables appearing in n_1 and n_2, and has to match against the occurrences of the (non-constructor) function +, and such comparisons and matchings are not possible in CIC.

Alternatively, we could use **mmatch** to implement cancel in Mtac, but there are good reasons to prefer a purely functional Coq implementation when one is possible, as is the case here. With a pure cancel, compare can return a very short proof term as a result (*e.g.*, (sound n_1 n_2 r_1 r_2 _) in the code of compare). An Mtac implementation would have to expose the reasoning behind the soundness of the Mtactic at a much finer granularity, resulting in a larger proof.

We next describe the implementations of the two helpers.

Data Structures for Reflection. There are two main data structures used for reflecting integer expressions. As each expression is built out of variables, constants and +, we syntactically represent the sum as term containing a list of syntactic representations of variables appearing in the expression, followed by a nat constant that sums up all the constants from the expression. We also need a type of variable contexts ctx, in order to determine the syntactic representation of variables. In our case, a variable context is simply a list of nat expression, each element standing for a different variable, and the position of the variable in the context serves as the variable's syntactic representative.

Definition ctx := seq nat
Record var := Var **of** nat
Definition term := (seq var) × nat

Example 3.1. *The expression* $n = (1 + x) + (y + 3)$ *may be reflected using a variable context* $c = [x, y]$*, and a* term ([Var 0, Var 1], 4). Var 0 *and* Var 1 *correspond to the two variables in* c *(*x *and* y*, respectively).* 4 *is the sum of the constants appearing in* n*.*

Example 3.2. *The syntactic representations of* 0*, successor constructor* S*, addition and an individual variable may be given as following* term *constructors. We use* _.1 *and* _.2 *to denote projections out of a pair, following the standard* Ssreflect *notation.*

```
Definition syn_zero : term := ([], 0).
Definition syn_succ (t : term) := (t.1, t.2 + 1).
Definition syn_add (t1 t2 : term) :=
  (t1.1 ++ t2.1, t1.2 + t2.2).
Definition syn_var (i : nat) := ([Var i], 0).
```

In prose, 0 *is reflected by an empty list of variable indexes, and* 0 *as a constant term; if* t *is a term reflecting* n*, then the successor* S n *is reflected by incrementing the constant component of* t*, etc.*

We further need a function interp that takes a variable context c and a term t, and *interprets* t into a nat, as follows.

```
interp_vars (c : ctx) (t : list var) :=
  if t is (Var j) :: t' then
    if (vlook c j, interp c t') is (Some v, Some e)
      then Some (v + e) else None
  else Some 0.

interp (c : ctx) (t : term) :=
  if interp_vars c t.1 is Some e
  then Some (e + t.2) else None.
```

First, interp_vars traverses the list of variable indices of t, turning each index into a natural number by looking it up in the context c, and summing the results. The lookup function vlook c j is omitted here, but it either returns Some j-th element of the context c, or None if c has less than j elements. Then, interp simply adds the result of interp_vars to the constant part of the term. For example, if the context $c = [x, y]$ and term $t = ([\text{Var } 0, \text{Var } 1], 4)$, then interp c t equals Some $(x + y + 4)$.

Reflection by to_ast**.** The to_ast Mtactic is applied twice in compare: once to reflect n_1, and again to reflect n_2. Each time, to_ast is passed as input a variable context, and extends this context with new variables encountered during reflection. To reflect n_1 in compare, to_ast starts with the empty context [], and to reflect n_2, it starts with the context obtained after the reflection of n_1. This ensures that if the reflections of n_1 and n_2 encounter the same variables, they will use the same syntactic representations for them.

```
Record ast (c : ctx) (n : nat) :=
  Ast {term_of : term;
       ctx_of : ctx;
       _ : interp ctx_of term_of = Some n ∧ prefix c ctx_of}

Definition to_ast : ∀ c n. ○(ast c n) :=
  mfix f c n :=
    mmatch n with
    | 0 ⇒ ret (Ast c 0 syn_zero c ...)
    | [n'] S n' ⇒
      r ← f c n';
      ret (Ast c (S n') (syn_succ (term_of r))
                (ctx_of r) ...)
    | [n1 n2] n1 + n2 ⇒
      r1 ← f c n1; r2 ← f (ctx_of r1) n2;
      ret (Ast c (n1 + n2)
        (syn_add (term_of r1) (term_of r2)) (ctx_of r2) ...)
    | _ ⇒
      ctx_index ← find n c;
      ret (Ast c n (syn_var ctx_index.2) ctx_index.1 ...)
    end.
```

FIGURE 3.11: Mtactic for reflecting nat expressions.

The invariants associated with to_ast are encoded in the data structure ast (Figure 3.11). ast is indexed by the input context c and the number n to be reflected. Upon successful termination of to_ast, the term_of field contains the term reflecting n, and the ctx_of field contains the new variable context, potentially extending c. The third field of ast is a proof formalizing the described properties of term_of and ctx_of.

Referring to Figure 3.11, the Mtactic to_ast takes the input variable context c and the number n to be reflected, and traverses n trying to syntactically match the head construct of n with 0, S or $+$, respectively. In each case it returns an ast structure containing the syntactic representation of n, e.g.: syn_zero, syn_succ or syn_add, respectively. In the n_1+n_2 case, to_ast recurses into n_2 by using the variable context returned from reflection of n_1 as an input, similar as in compare. In each case, the Ast constructor is supplied a proof that we omit but can be found in the sources. In the default case, when no constructor matches, n is treated as a variable. The Mtactic find n c (omitted here), searches for n in c, and returns a ctx $\times$ nat pair. If n is found, the pair consists of the old context c, and the position of n in c. If n is not found, the pair consists of a new context in which n is cons-ed to c, and the index k, where k is the index of n in the new context. to_ast then repackages the context and the index into an ast structure.

```
Fixpoint cancel_vars (s1 s2 : list var) : bool :=
  if s1 is v :: s1' then v ∈ s2 &&
    cancel_vars s1' (remove_var v s2)
  else true.

Definition cancel (t1 t2 : term) : bool :=
  cancel_vars t1.1 t2.1 && t1.2 ≤ t2.2.
```

FIGURE 3.12: Algorithm for canceling common variables from terms.

Canceling Common Variables. The cancel function is presented in Figure 3.12. It takes terms t_1 and t_2 and tries to determine if t_1 and t_2 syntactically represent two $\leq$-related expressions by cancelling common terms, as we described previously. First, the helper cancel_vars iterates over the list of variable representations of t_1, trying to match each one with a variable representation in t_2 (in the process, removing the matched variables by using yet another helper function remove_vars, omitted here). If the matching is successful and all variables of t_1 are included in t_2, then cancel merely needs to check if the constant of t_1 is smaller than the constant of t_2.

We conclude the example with the statement of the correctness lemma of cancel, which is the key component of the soundness proof for compare. We omit the proof here, but it can be found in our Coq files (Ziliani, 2014).

```
Lemma sound n1 n2 (a1 : ast [] n1) (a2 : ast (ctx_of a1) n2) :
      cancel (term_of a1) (term_of a2) →
      n1 ≤ n2.
```

In prose, let a_1 and a_2 be reflections of n_1 and n_2 respectively, where the reflection of a_1 starts in the empty context, and the reflection of a_2 starts in the variable context returned by a_1. Then running cancel in the final context of a_2 over the reflected terms of a_1 and a_2 returns true only when it is correct to do so; that is, only when $n_1 \leq n_2$.

3.4 Operational Semantics and Type Soundness

In this section we present the operational semantics and type soundness of Mtac. After presenting the rules of the semantics (§3.4.1), we provide more motivation for the use of contextual types (§3.4.2), before giving details of the type soundness proof (§3.4.3). Along the way, we motivate by example a number of our design decisions.

3.4.1 Rules of the Semantics

First, some preliminaries. Following the conventions used throughout this thesis, we write A, B, C for CIC type variables, τ for CIC types, ρ for CIC type *predicates* (functions returning types), f for CIC functions. We depart ever so slightly from the conventions when it comes to terms: since we need to distinguish any CIC term from Mtactics, *i.e.*, CIC terms of type $\bigcirc\tau$ for some type τ. For the former we use e while for the latter we use t from *tactic*. The operational semantics of Mtac defines the judgment form

$$\Gamma \vdash (\Sigma; t) \rightsquigarrow (\Sigma'; t')$$

where, as before, Γ is the typing context containing parameters and (let-bound) local definitions, and Σ and Σ' are contexts for meta-variables $?x$. We remind the reader that both kinds of contexts contain both variable declarations (standing for parameters and uninstantiated meta-variables, respectively) and definitions (let-bound variables and instantiated meta-variables, respectively). These contexts are needed for weak head reduction of CIC terms ($\Sigma; \Gamma \vdash e \overset{\mathsf{whd}}{\rightsquigarrow} e'$), but also for some of Mtac's constructs. The types of meta-variables are annotated with a "local context", as in $\tau[\Gamma]$, in which Γ should bind a superset of the free variables of τ. As we mentioned in Section 1.2, such local contexts prevents ill-typed instantiations of meta-variables. Below (§3.4.2) we discuss Mtac specific examples of how these local contexts prevent Mtac from being unsound.

As is usual in Coq, we omit function arguments that can be inferred by the type inference engine (typically, the type arguments). In some cases, however, it is required, or clearer, to explicitly flesh out all of the arguments, in which case we adopt another convention from Coq: prepending the @ symbol to the function being applied.

We assume that the terms are well-typed in their given contexts, and we ensure that this invariant is maintained throughout execution. Tactic computation may either (a) terminate successfully returning a term, **ret** e, (b) terminate by throwing an exception, **raise** e, (c) diverge, or (d) get blocked. (We explain the possible reasons for getting blocked below.) Hence we have the following tactic values:

Definition 1 (Values)**.** $v \in \text{Values} ::= \textbf{ret}\ e \mid \textbf{raise}\ e$.

Figure 3.13 shows our operational semantics. The first rule (EReduc) performs a CIC weak head reduction step. As mentioned, weak head reduction requires both contexts because, among other things, it will unfold definitions of variables and meta-variables in head position. For a precise description of Coq's standard reduction rules, revisit §1.2. It is important to note that the decision to evaluate lazily instead of strictly is

not arbitrary. Weak head reduction preserves the structure of terms, which is crucial to avoid unnecessary reductions, and to produce small proof terms. Take for instance the search example in §3.1. The proof showing that x is in the list $[z; y; w]{+}{+}[x]$ consists of one application of the lemma in_or_app to the lemma in_eq, while the proof showing the same element is in the list $[x; y; w; x]$, resulting from forcing the execution of the append function, consists of three applications of the lemma in_cons to the lemma in_eq. That is, a call-by-value reduction forces the reduction of the append and results in a larger proof term.

The next seven rules, concerning the semantics of Mtac fixed points, **bind**, and **mtry**, are all quite standard.

The most complex rule is the subsequent one concerning pattern matching (EMMATCH). It matches the term e with some pattern described in the list ps. Each element ps_i of ps is a pair containing a pattern p and a body b, abstracted over a list of (dependent) variables $\overline{x : \tau}$. Since patterns are first-class citizens in CIC, ps_i is first reduced to weak head normal form in order to expose the pattern and the body. The normalization relation is written $\overset{\mathsf{whd}}{\rightsquigarrow}{}^*$ and, as with the weak head reduction relation, it requires the two contexts. Then, we replace each variable x with a corresponding meta-variable $?y$ in p, and proceed to unify the result with term e. For this, the context Σ is extended with the freshly created meta-variables $\overline{?y}$. For each $?y_k$, the type τ_k is created within the context Γ'_k, which is the original context Γ extended with the variables appearing to the left of variable x_k in the list $\overline{x : \tau}$. After unification is performed, a new meta-variable context is returned that might not only instantiate the freshly generated meta-variables $\overline{?y}$, but may also instantiate previously defined meta-variables. (Instantiating such meta-variables is important, for instance, to instantiate the existentials in the tautology prover example of §3.3.2). The meta-variables appearing in the body of the pattern are substituted with their definitions, denoted as $\Sigma'(b')$, where b' is the body after substituting the pattern variables with the meta-variables, *i.e.*, $b' = b\{?y_1[\mathrm{id}_{\Gamma'_1}]/x_1\}\cdots\{?y_n[\mathrm{id}_{\Gamma'_n}]/x_n\}$. Here, $[\mathrm{id}_{\Gamma'_k}]$ refers to the identity substitution for variables in Γ'_k; its use will be explained in detail in the discussion of Example 3.4 (§3.4.2 below). Finally, we require that patterns are tried in sequence, *i.e.*, that the scrutinee, e, should not be unifiable with any previous pattern ps_j. In case no patterns match the scrutinee, the **mmatch** is blocked.

The semantics for pattern matching is parametric with respect to the unification judgment and thus does not rely on any particular unification algorithm. (Our implementation allows choosing between two algorithms: Coq's standard unification algorithm and our unification algorithm from Chapter 4.) We observe that our examples, however, implicitly depend on *higher-order pattern unification* (Miller, 1991b). Higher-order unification is in general undecidable, but Miller identified a decidable subset of problems,

$$\frac{\Sigma;\Gamma \vdash t \overset{\mathsf{whd}}{\rightsquigarrow} t'}{\Gamma \vdash (\Sigma;t) \rightsquigarrow (\Sigma;t')}\ \textsc{EReduc} \qquad \frac{}{\Gamma \vdash (\Sigma;\mathbf{mfix}\ f\ t) \rightsquigarrow (\Sigma;f\ (\mathbf{mfix}\ f)\ t)}\ \textsc{EFix}$$

$$\frac{\Gamma \vdash (\Sigma;t) \rightsquigarrow (\Sigma';t')}{\Gamma \vdash (\Sigma;\mathbf{bind}\ t\ f) \rightsquigarrow (\Sigma';\mathbf{bind}\ t'\ f)}\ \textsc{EBindS}$$

$$\frac{\Gamma \vdash (\Sigma;t) \rightsquigarrow (\Sigma';t')}{\Gamma \vdash (\Sigma;\mathbf{mtry}\ t\ f) \rightsquigarrow (\Sigma';\mathbf{mtry}\ t'\ f)}\ \textsc{ETryS}$$

$$\frac{}{\Gamma \vdash (\Sigma;\mathbf{bind}\ (\mathbf{ret}\ e)\ f) \rightsquigarrow (\Sigma;f\ e)}\ \textsc{EBindR}$$

$$\frac{}{\Gamma \vdash (\Sigma;@\mathbf{bind}\ \tau\ \tau'\ (@\mathbf{raise}_{\tau}\ e)\ \ f) \rightsquigarrow (\Sigma;@\mathbf{raise}_{\tau'}\ e)}\ \textsc{EBindE}$$

$$\frac{}{\Gamma \vdash (\Sigma;\mathbf{mtry}\ (\mathbf{ret}\ e)\ f) \rightsquigarrow (\Sigma;\mathbf{ret}\ e)}\ \textsc{ETryR} \qquad \frac{}{\Gamma \vdash (\Sigma;\mathbf{mtry}\ (\mathbf{raise}\ e)\ f) \rightsquigarrow (\Sigma;f\ e)}\ \textsc{ETryE}$$

$$\frac{\begin{array}{ll}\Sigma;\Gamma \vdash ps_i \overset{\mathsf{whd}_*}{\rightsquigarrow} \mathsf{Ptele}\ (\overline{x:\tau})\ (\mathsf{Pbase}\ p\ b) & \forall k.\ \Gamma'_k = \Gamma, x_1:\tau_1,\ldots,x_{k-1}:\tau_{k-1} \\ \Sigma, \overline{?y:\tau[\Gamma']};\Gamma \vdash p\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\} \approx e \ \triangleright\ \Sigma' & \forall j < i.\ ps_j \text{ does not unify with } e\end{array}}{\Gamma \vdash (\Sigma;\mathbf{mmatch}\ e\ ps) \rightsquigarrow (\Sigma';\Sigma'(b\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\}))}\ \textsc{EMmatch}$$

$$\frac{?x \notin \mathsf{dom}(\Sigma)}{\Gamma \vdash (\Sigma;\mathbf{evar}_{\tau}) \rightsquigarrow (\Sigma, ?x:\tau[\Gamma];\mathbf{ret}\ ?x[\mathrm{id}_{\Gamma}])}\ \textsc{EEvar}$$

$$\frac{\Sigma;\Gamma \vdash e \overset{\mathsf{whd}_*}{\rightsquigarrow} ?x[\sigma] \quad (?x := \_) \notin \Sigma}{\Gamma \vdash (\Sigma;\mathbf{is\_evar}\ e) \rightsquigarrow (\Sigma;\mathbf{ret}\ \mathsf{true})}\ \textsc{EIsVarT}$$

$$\frac{\Sigma;\Gamma \vdash e \overset{\mathsf{whd}_*}{\rightsquigarrow} e' \quad e' \text{ not unif. variable}}{\Gamma \vdash (\Sigma;\mathbf{is\_evar}\ e) \rightsquigarrow (\Sigma;\mathbf{ret}\ \mathsf{false})}\ \textsc{EIsVarF}$$

$$\frac{\Gamma, x:\tau \vdash (\Sigma;t) \rightsquigarrow (\Sigma';t')}{\Gamma \vdash (\Sigma;\nu x:\tau.\ t) \rightsquigarrow (\Sigma';\nu x:\tau.\ t')}\ \textsc{ENuS} \qquad \frac{x \notin \mathsf{FV}(v)}{\Gamma \vdash (\Sigma;(\nu x.\ v)) \rightsquigarrow (\Sigma;v)}\ \textsc{ENuV}$$

$$\frac{\Sigma;\Gamma \vdash e \overset{\mathsf{whd}_*}{\rightsquigarrow} x \quad \Gamma = \Gamma_1, x:\tau', \Gamma_2 \quad x \notin \mathsf{FV}(\Gamma_2, \rho)}{\Gamma \vdash (\Sigma;@\mathbf{abs}\ \tau\ \rho\ e\ e') \rightsquigarrow (\Sigma;\mathbf{ret}\ (\lambda y.\ e'\{y/x\}))}\ \textsc{EAbs}$$

$$\frac{\textit{(* print s to stdout *)}}{\Gamma \vdash (\Sigma;\mathbf{print}\ s) \rightsquigarrow (\Sigma;\mathbf{ret}\ \langle\rangle)}\ \textsc{EPrint}$$

Figure 3.13: Operational small-step semantics.

the so-called *pattern fragment*, where unification variables appear only in equations of the form $?f\ x_1\ \ldots\ x_n \approx e$, with $x_1, \ldots, x_n$ distinct variables. The $\forall$ and $\exists$ cases of the tautology prover (§3.3.2) fall into this pattern fragment, and their proper handling depends on higher-order pattern unification.

Another notable aspect of Coq's unification algorithm is that it equates terms up to definitional equality. In particular, if a pattern match at first does not succeed, Coq will take a step of reduction on the scrutinee, try again, and repeat. Thus, the ordering of two patterns in a **mmatch** matters, even if it seems the patterns are syntactically non-overlapping. Take for instance the search example in §3.1. If the pattern for concatenation of lists were moved *after* the patterns for consing, then the consing patterns would actually match against (many) concatenations as well, since the concatenation of two lists is often reducible to a term of the form $h :: t$.

Related to this, the last aspect of Coq's unification algorithm that we depend on is its *first-order approximation*. That is, in the presence of an equation of the form $c\ e_1\ \ldots\ e_n \approx c\ e'_1\ \ldots\ e'_n$, where c is a constant, the unification algorithm tries to equate each $e_i \approx e'_i$. While this may cause Coq to miss out on some solutions, it has the benefit of being simple and predictable. For instance, consider the equation

$$?l{+}{+}?r \approx []{+}{+}(h :: t)$$

that might result from matching the list $[]{+}{+}(h :: t)$ with the pattern for concatenation of lists in the search example from §3.1, with $?l$ and $?r$ fresh meta-variables. Here, although there exist many solutions, the algorithm assigns $?l := []$ and $?r := (h :: t)$, an assignment that is intuitively easy to explain. [4]

Coming back to the rules, next is the rule for $\textbf{evar}_\tau$, which simply extends Σ with a fresh uninstantiated meta-variable of the appropriate type. Note that it is always possible to solve a goal by returning a fresh meta-variable. But a proof is only complete if it is closed and, therefore, before QED time, every meta-variable has to be instantiated. The two following rules govern **is_evar** e and check whether an expression (after reduction to weak head normal form) is an uninstantiated meta-variable.

The next two rules define the semantics of the νx binder: the parameter x is pushed into the context, and the execution proceeds until a value is reached. The computed value is simply returned if it does not contain the parameter, x; otherwise, $\nu x.\ v$ is blocked. The latter rule is for abstracting over parameters. If the first non-implicit argument

[4] For the Ltac *connoisseur*, there are several differences between our **mmatch** and Ltac's **match**. In particular, the latter (1) does not reduce terms during unification, (2) uses an unsound unification algorithm, unlike Mtac (§3.4.3), and (3) backtracks upon (any kind of) failure, making it hard to understand the flow of the program.

of **abs** weak-head reduces to a parameter, then we abstract it from the second (also non-implicit) argument of **abs**, thereby returning a function. In order to keep a sound system, we need to check that the return type and the local context do not depend on the variable being abstracted, as discussed below in Example 3.5.

The astute reader may wonder why we decided to have νx and **abs** instead of one single constructor combining the semantics of both. Such a combined constructor would always abstract the parameter x from the result, therefore avoiding the final check that the parameter is not free in the result. The reason we decided to keep **nu** and **abs** separate is simple: it is not always desirable to abstract the parameters in the same order in which they were introduced. This is the case, for instance, in the Mtactic skolemize for skolemizing a formula (provided in the Mtac distribution). Moreover, sometimes the parameter is not abstracted at all, for instance in the Mtactic fv for computing the list of free variables of a term (also provided in the Mtac distribution).

Finally, the last rule (EPrint) replaces a printing command with the trivial value $\langle\rangle$. Informally, we also print out the string s to the standard output, although standard I/O is not formally modeled here.

Example 3.3. *We show the trace of a simple example to get a grasp of the operational semantics. In this example,* $\Gamma = \{h : \mathsf{nat}\}$.

$$\textbf{let } s := (h :: [])\text{++}[] \textbf{ in } \mathsf{search}\ h\ s$$

We want to show that the final term produced by running this Mtactic expresses the fact that h was found at the head of the list on the left of the concatenation, that is,

$$\mathsf{in\_or\_app}\ (h :: [])\ []\ (\mathsf{or\_introl}\ (\mathsf{in\_eq}\ h\ []))$$

First, the **let** is expanded, obtaining

$$\mathsf{search}\ h\ ((h :: [])\text{++}[])$$

Then, after expanding the definition of search and β-reducing the term, we are left with the fixpoint being applied to the list:

$$(\textbf{mfix } f\ (s : \mathsf{seq}\ A) := \ldots)\ ((h :: [])\text{++}[])$$

At this point the rule for **mfix** triggers, exposing the **mmatch**:

$$\textbf{mmatch } ((h :: [])\text{++}[]) \textbf{ with} \ldots \textbf{end}$$

Thanks to first-order approximation, the case for append is unified, and its body is executed:

$$\textsf{mtry}\ il \leftarrow f\ (h :: []);\ \textsf{ret}\ \ldots\ \textsf{with}\ \_ \Rightarrow \ldots\ \textsf{end} \tag{3.1}$$

where f stands for the fixpoint. The rule ETryS executes the code for searching for the element in the sublist $(h :: [])$:

$$il \leftarrow f\ (h :: []);\ \textsf{ret}\ (\textsf{in\_or\_app}\ (h :: [])\ []\ h\ (\textsf{or\_introl}\ il)) \tag{3.2}$$

The rule EBindS triggers, after which the fixpoint is expanded and a new **mmatch** exposed:

$$\textsf{mmatch}\ (h :: [])\ \textsf{with}\ \ldots\ \textsf{end}$$

This time, the rule for append fails to unify, but the second case succeeds, returning the result in_eq h []. Coming back to (3.2), il is replaced with this result, getting the expected final result that is in turn returned by the **mtry** of (3.1).

As a last remark, notice how at each step the selected rule is the only applicable one: the semantics of Mtac is deterministic.

3.4.2 Meta-variables in Mtac

So far we mentioned that is indispensable for Mtac's type soundness that meta-variables have *contextual types*, but did not explain why.

In the rules in Figure 3.13, there are two places where contextual types and suspended substitutions are relevant: the rule for creation of meta-variables (EEvar) and the rule for pattern matching (EMmatch). In EEvar, the new meta-variable $?x$ is created with contextual type $\tau[\Gamma]$, where Γ is a copy of the local context coming from the evaluation judgment. The returned value is $?x$ applied to the identity suspended substitution id_Γ. In EMmatch, every pattern variable $?y_k$ is created with the contextual type $\tau_k[\Gamma'_k]$. Γ'_k is the context Γ extended with the prior variables in the telescope $x_1, \ldots, x_{k-1}$. In order to marry the context in the contextual type with the local context Γ, each meta-variable $?y_k$ is applied, as before, with the identity suspended substitution.

There is another, more subtle use of contextual types in EMmatch: the unification condition. In Mtac, the unification algorithm is in charge of instantiating meta-variables. As we saw in the example from Section 1.2.1.1, it cannot instantiate meta-variables arbitrarily, as it may end up with an ill-typed meta-substitution.

But not only contextual types are useful to ensure soundness of unification. In particular, the following examples show two different potential sources of unsoundness related to the

abs constructor. Fortunately, thanks to contextual types, plus the restrictions enforced by the rule EAbs, neither example presents any risk to the soundness of Mtac.

Example 3.4. *Example showing how contextual types preclude the assignment of different types to the same term.*

```
01 Definition abs_evar (A : Type) :=
02   X ← evar A;
03   f ← @abs Type (λ B. B) A X : ○(∀ B : Type. B);
04   ret (f nat, f bool).
```

In short, this example takes a type A and creates a meta-variable X of type A. Then, it abstracts its type, creating the function f, which is then applied to the types nat and bool. For readability, we annotated the type of f: $\forall B : \mathsf{Type}.\ B$, and therefore the returned terms f nat and f bool have type nat and bool, respectively. However, they both compute to X, the meta-variable, so at first sight it looks like the same expression can be typed with different and incompatible types!

No need to panic here, contextual types solve the conundrum! Let's see what really happens when we execute the Mtactic. First, note that, in order for the **abs**traction in line 3 to succeed, the Mtactic should instantiate the argument A with a type variable. Otherwise, the computation will get *blocked*, as we are only allowed to abstract variables. So let's assume abs_evar is executed in a local context with only variable $A' : \mathsf{Type}$, which is then passed in as the argument. When executing line 2, the meta-variable $?u$ is created with contextual type $A'[A' : \mathsf{Type}]$. Then, the variable X is bound to $?u[A']$.

Next, f is bound to the value resulting from abstracting the type A' from the meta-variable, that is,

$$f = \lambda B : \mathsf{Type}.\ ?u[A']\{B/A'\},$$

which after applying the substitution is equal to

$$f = \lambda B : \mathsf{Type}.\ ?u[B].$$

Therefore, the value resulting from applying f to the different types is

$$(?u[\mathsf{nat}], ?u[\mathsf{bool}]).$$

Since the type of the meta-variable *depends on* A', the return type is $\mathsf{nat} \times \mathsf{bool}$. But the substitution distinguishes both terms in the pair, so even when they refer to the same

meta-variable, they're actually referring to different *interpretations* of the value carried by $?u$. It might be instructive at this point to go back to the typing rule for meta-variables in Figure 1.5 to fully understand how $?u$ gets two different types according to the substitution applied.

Thus, contextual types prevents us from providing a value to the meta-variable that will only comply to one of the interpretations. That is, whatever value $?u$ holds, it must have the abstract type A', such that each substitution to which $?u$ is applied provides a proper interpretation for the type of the value (in the example, nat or bool).

To conclude with this example, observe that $?u$ can only be instantiated with a falsity. In fact, the contextual type $A'[A' : \mathsf{Type}]$ is equivalent to the standard type for falsity $\forall A' : \mathsf{Type}.\ A'$.

Example 3.5. *In this example we show why it is necessary to severely restrict occurrences of the variable being abstracted.*

In the previous example the variable being abstracted occurred only in the term (more precisely, in the substitution). If it instead occurred in the return type or in the local context (Γ), then this could lead to another potential source of unsoundness. To see what can go wrong, let us consider the following example:

```
Definition abs_dep (A : Type) (x : A) :=
  X ← evar nat;
  @abs Type (λ _. nat) A X : ○(Type → nat).
```

After running the first line of the function, X is bound to the term $?u[A, x]$, where $?u$ is a fresh meta-variable with (contextual) type $\mathsf{nat}[A : \mathsf{Type}, x : A]$. If we allow the abstraction of A in X, then we arrive at the ill-typed term

$$\lambda\ B.\ ?u[B, x]$$

Note that x should have as type the first component of the substitution, which is B, but it has type A instead. For this reason we impose the restriction that the abstracted variable (A in this case) must not occur anywhere in the context aside from the point where it is bound. It is not hard to see what goes wrong when the abstracted variable appears in the return type, so we prohibit this case as well. The technical reasons for these restrictions, as we will see in the following section, can be understood as follow: after the abstraction of the variable, we should be able to strength the local context by removing the variable.

3.4.3 Proof of Soundness

As mentioned earlier, Mtactic execution can block. Here, we define exactly the cases when execution of a term is blocked.

Definition 2 (Blocked terms)**.** A term t is *blocked* if and only if the subterm in reduction position satisfies one of the following cases:

- It is not an application of one of the $\bigcirc$ constructors and it is not reducible using the standard CIC reduction rules ($\overset{\mathsf{whd}}{\rightsquigarrow}$).
- It is $\nu x.\ v$ and $x \in \mathsf{FV}(v)$.
- It is $\mathbf{abs}\ e\ e'$ and $e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ e''$ and $(e'' : \_) \notin \Gamma$.
- It is $@\mathbf{abs}\ \tau\ \rho\ e\ e'$ and $e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ x$ and $\Gamma = \Gamma_1, x : \tau, \Gamma_2$ and $x \in \mathsf{FV}(\Gamma_2, \rho)$.
- It is $\mathbf{mmatch}\ e\ ps$ and no pattern in ps unifies with e.

With this definition, we can then establish a standard type soundness theorem for Mtac.

Theorem 3.1 (Type soundness)**.** *If* $\Sigma; \Gamma \vdash t : \bigcirc\tau$*, then either* t *is a value, or* t *is blocked, or there exist* t' *and* Σ' *such that* $\Gamma \vdash (\Sigma; t) \rightsquigarrow (\Sigma'; t')$ *and* $\Sigma'; \Gamma \vdash t' : \bigcirc\tau$ *.*

In order to prove it, we take the typing judgment for Coq terms from Chapter 1:

$$\Sigma; \Gamma \vdash e : \tau$$

and also assume the following standard properties:

Postulate 1 (Convertibility)**.** If $\Sigma; \Gamma \vdash e : \tau$ and $\Sigma; \Gamma \vdash \tau \equiv \tau'$ (τ and τ' are convertible up to CIC reduction) then

$$\Sigma; \Gamma \vdash e : \tau'$$

Postulate 2 (Type preservation of reduction)**.** If $\Sigma; \Gamma \vdash e : \tau$ and $\Sigma; \Gamma \vdash e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ e'$ then

$$\Sigma; \Gamma \vdash e' : \tau$$

Postulate 3 (Meta-substitution)**.** If $\Sigma; \Gamma \vdash e : \tau$ then $\Sigma; \Gamma \vdash \Sigma(e) : \tau$.

Postulate 4 (Substitution)**.** If $\Sigma; \Gamma_1, x : \tau', \Gamma_2 \vdash e : \tau$ and $\Sigma; \Gamma_1 \vdash e' : \tau'$ then

$$\Sigma; \Gamma_1, \Gamma_2\{e'/x\} \vdash e\{e'/x\} : \tau\{e'/x\}$$

Postulate 5 (Weakening of meta-context)**.** If $\Sigma; \Gamma_1, \Gamma_2 \vdash e : \tau$ and $\Sigma; \Gamma_1 \vdash \tau' : \mathsf{Type}$ and $?x \notin \mathsf{dom}(\Sigma)$ then

$$\Sigma, ?x : \tau'[\Gamma_1]; \Gamma_1, \Gamma_2 \vdash e : \tau$$

Postulate 6 (Strengthening)**.** If $\Sigma; \Gamma_1, x : \tau, \Gamma_2 \vdash e : \tau'$ and $x \notin \mathsf{FV}(\Gamma_2, e, \tau')$ then

$$\Sigma; \Gamma_1, \Gamma_2 \vdash e : \tau'$$

Postulate 7 (Weakening)**.** If $\Sigma; \Gamma_1, \Gamma_2 \vdash e : \tau'$ and $\Sigma; \Gamma_1 \vdash \tau : \mathsf{Type}$ and $x \notin \mathsf{dom}(\Gamma_1, \Gamma_2)$ then

$$\Sigma; \Gamma_1, x : \tau, \Gamma_2 \vdash e : \tau'$$

We also need the following relation between meta-contexts:

Definition 3 (Meta-context extension)**.** We say that Σ' extends Σ, written $\Sigma' \geq \Sigma$, when every meta-variable in Σ occurs in Σ', with the same type, and in case it has been instantiated, instantiated with the same term. Formally,

$$\begin{aligned}\Sigma' \geq \Sigma \mathrel{\widehat{=}} \forall ?x : \tau[\Gamma] \in \Sigma.\ ?x : \tau[\Gamma] \in \Sigma' \vee ?x : \tau[\Gamma] := e \in \Sigma' \\ \wedge \forall ?x : \tau[\Gamma] := e \in \Sigma.\ ?x : \tau[\Gamma] := e \in \Sigma'\end{aligned}$$

We postulate that meta-context extension does not alter typechecking:

Postulate 8 (Preservation under meta-context extension)**.** If $\Sigma; \Gamma \vdash e : \tau$ and $\Sigma' \geq \Sigma$, then

$$\Sigma'; \Gamma \vdash e : \tau$$

We require from unification the following:

Postulate 9 (Soundness of unification)**.** If $\Sigma; \Gamma \vdash e : \tau$ and $\Sigma; \Gamma \vdash e' : \tau'$ and $\Sigma; \Gamma \vdash e \approx e' \triangleright \Sigma'$ then

$$\Sigma' \equiv ee' \quad \text{and} \quad \Sigma' \equiv \tau\tau' \quad \text{and} \quad \Sigma' \geq \Sigma$$

We start by proving type preservation:

Theorem 3.2 (Type preservation)**.** *If* $\Gamma \vdash (\Sigma; t) \rightsquigarrow (\Sigma'; t')$ *and* $\Sigma; \Gamma \vdash t : \bigcirc\tau$*, then* $\Sigma' \geq \Sigma$ *and* $\Sigma'; \Gamma \vdash t' : \bigcirc\tau$*.*

Proof. By induction on the reduction relation. We will expand (some of) the implicit parameters for clarity.

Case EReduc: It follows by preservation of reduction (Postulate 2).

Case EFix: We have $t = @\textsf{mfix}\ \tau'\ \rho\ f\ t''$. Remember the type of the fixpoint:

$$\textsf{mfix} : \forall A\ P.\ ((\forall x : A.\ \bigcirc(P\ x)) \to (\forall x : A.\ \bigcirc(P\ x))) \to \forall x : A.\ \bigcirc(P\ x)$$

By hypothesis we know t is well-typed, and as a consequence, t'' has type τ'. We have to show that

$$\Sigma; \Gamma \vdash f\ (@\textsf{mfix}\ \tau'\ \rho\ f)\ t'' : \bigcirc(\rho\ t'')$$

It follows immediately from the types of the subterms:

$$f : (\forall x : \tau'.\ \bigcirc(\rho\ x)) \to (\forall x : \tau'.\ \bigcirc(\rho\ x))$$

and

$$\textsf{mfix}\ f : \forall x : \tau'.\ \bigcirc(\rho\ x)$$

and $t'' : \tau'$.

Case EBindS: We have $t = @\textsf{bind}\ \tau'\ \tau''\ t''\ f$. By the premise of the rule, there exists t''' such that

$$\Gamma \vdash (\Sigma; t'') \rightsquigarrow (\Sigma'; t''')$$

Remember the type of **bind**:

$$\textsf{bind} : \forall A\ B.\ \bigcirc A \to (A \to \bigcirc B) \to \bigcirc B$$

We have that t has type $\bigcirc\tau''$. We have to show that

$$\Sigma'; \Gamma \vdash @\textsf{bind}\ \tau'\ \tau''\ t'''\ f : \bigcirc\tau''$$

By the inductive hypothesis, we know that $\Sigma' \geq \Sigma$ and

$$\Sigma'; \Gamma \vdash t''' : \bigcirc\tau'$$

We conclude by noting that, by Postulate 8 (Preservation under meta-context), the type of f is not changed in the new meta-context Σ'.

Case EBindR: We have $t = @\textsf{bind}\ \tau'\ \tau''\ (\textsf{ret}\ e)\ f$. We have to show that

$$\Sigma; \Gamma \vdash f\ e : \bigcirc\tau''$$

Immediate by type of f and e.

Case EBindE: We have $t = @\textbf{bind}\ \tau'\ \tau''\ (@\textbf{raise}_{\tau'}\ e)\ f$. We have to show that

$$\Sigma; \Gamma \vdash @\textbf{raise}_{\tau''}\ e : \bigcirc \tau''$$

Immediate by type of $@\textbf{raise}_{\tau''}\ e$.

Cases ETryS, ETryR, ETryE: Analogous to the previous cases.

Case EMmatch: We have $t = @\textbf{mmatch}\ \tau'\ \rho\ e\ ps$. The type of **mmatch** is

$$\textbf{mmatch} : \forall A\ P\ (t : A).\ \mathsf{seq}\ (\mathsf{Patt}\ A\ P) \to \bigcirc (P\ t)$$

By the premises of the rule:

$$\Sigma; \Gamma \vdash ps_i \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ \mathsf{Ptele}\ (\overline{x : \tau''})\ (\mathsf{Pbase}\ p\ b) \tag{3.3}$$

$$\Sigma, \overline{?y : \tau''[\Gamma']}; \Gamma \vdash p\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\} \approx e\ \triangleright\ \Sigma' \tag{3.4}$$

$$\forall j < i.\ ps_j \text{ does not unify with } e \tag{3.5}$$

That is, reducing ps leads to a list whose i-th element is a *telescope* (see §3.2) abstracting variables $\overline{x : \tau''}$ from a pattern p and body b. This pattern, after replacing its abstraction with unification variables, is unifiable with term e, producing a new meta-context Σ'. Every unification variable $?y_k$ is created with type τ_k in a context resulting form extending the context Γ with the variables to the left of the telescope, that is

$$\Gamma'_k = \Gamma, x_1 : \tau_1, \ldots, x_{k-1} : \tau_{k-1}$$

We have to show that

$$\Sigma'; \Gamma \vdash \Sigma'(b\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\}) : \bigcirc (\rho\ e)$$

Given the hypothesis that t is well-typed, we know that

$$\Sigma; \Gamma \vdash e : \tau' \tag{3.6}$$

$$\Sigma; \Gamma, \overline{x : \tau''} \vdash p : \tau' \tag{3.7}$$

$$\Sigma; \Gamma, \overline{x : \tau''} \vdash b : \bigcirc (\rho\ p) \tag{3.8}$$

Taking Equation 3.7 and by multiple weakenings of the meta-context we obtain

$$\Sigma, \overline{?y : \tau''[\Gamma']}; \Gamma, \overline{x : \tau''} \vdash p : \tau' \tag{3.9}$$

By Postulate 4 (Substitution), noting that the variables $\overline{x}$ cannot appear free in τ' or in Γ,

$$\Sigma, \overline{?y : \tau''[\Gamma']}; \Gamma \vdash p\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\} : \tau' \tag{3.10}$$

Therefore, by Postulate 9 (soundness of unification) and equations 3.4, 3.6, and 3.10, we obtain that both sides of the unification are convertible under the new context Σ'.

Similarly as before, by Equation 3.8, weakening of meta-context, and Substitution postulate, we obtain

$$\Sigma'; \Gamma \vdash b\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\} : \bigcirc(\rho\ p\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\})$$

(noting that ρ actually does not have any x_k in its set of free variables).

By convertibility, this is equal to

$$\Sigma'; \Gamma \vdash b\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\} : \bigcirc(\rho\ e)$$

We conclude by Postulate 3 (Meta-substitution).

Case EEvar: We have $t = \mathsf{evar}_\tau$. It is trivial,

$$\Sigma, ?x : \tau[\Gamma]; \Gamma \vdash \mathsf{ret}\ ?x[\mathrm{id}_\Gamma] : \bigcirc\tau$$

Cases EIsEvarT and EIsEvarF: Trivial, both return a boolean value.

Case ENuS: Desugaring and making parameters explicit, we have $t = @\mathsf{nu}\ \tau'\ \tau\ (\lambda x : \tau'.\ t'')$.

The type of **nu** is

$$\mathsf{nu} : \forall A\ B.\ (A \to \bigcirc B) \to \bigcirc B$$

Therefore,

$$\Sigma; \Gamma, x : \tau' \vdash t'' : \bigcirc\tau \tag{3.11}$$

By the premise of the rule, $\Gamma, x : \tau' \vdash (\Sigma; t'') \rightsquigarrow (\Sigma'; t''')$. By the inductive hypothesis with Equation 3.11,

$$\Sigma'; \Gamma, x : \tau' \vdash t''' : \bigcirc\tau$$

and $\Sigma' \geq \Sigma$. Therefore

$$\Sigma'; \Gamma \vdash \lambda x : \tau'.\ t''' : \tau' \to \bigcirc\tau$$

which allows us to conclude that

$$\Sigma'; \Gamma \vdash \nu x : \tau'.\ t''' : \bigcirc\tau$$

Case ENuV: As in the previous case, we have $t = @\mathbf{nu}\ \tau'\ \tau\ (\lambda x : \tau'.\ v)$. We have to show that

$$\Sigma; \Gamma \vdash v : \bigcirc\tau$$

Since t is well-typed, we know that

$$\Sigma; \Gamma, x : \tau' \vdash v : \bigcirc\tau$$

By the premise of the rule, $x \notin \mathsf{FV}(v)$, and it cannot appear free in τ, so by strengthening (Postulate 6),

$$\Sigma; \Gamma \vdash v : \bigcirc\tau$$

Case EAbs: We have $t = @\mathbf{abs}\ \tau'\ \rho\ e\ e'$. Remember the type of **abs**:

$$\mathbf{abs} : \forall A\ P\ x.\ P\ x \to \bigcirc(\forall y : A.\ P\ y)$$

We need to show that

$$\Sigma; \Gamma \vdash \mathbf{ret}\ (\lambda y : \tau'.\ e'\{y/x\}) : \bigcirc(\forall y : \tau'.\ \rho\ y)$$

where, by the premises of the rule, e reduces to x. By preservation of reduction (Postulate 2), we know that x has type τ', and therefore Γ is equivalent (*i.e.*, convertible) to $\Gamma_1, x : \tau', \Gamma_2$, for some context Γ_1 and Γ_2. Also, by the premises of the rule, $x \notin \mathsf{FV}(\Gamma_2, \rho)$.

By t being well-typed, we know

$$\Sigma; \Gamma \vdash e' : \rho\ e$$

which by convertibility is equivalent to (expanding Γ)

$$\Sigma; \Gamma_1, x : \tau', \Gamma_2 \vdash e' : \rho\ x$$

By weakening (Postulate 7),

$$\Sigma; \Gamma_1, x : \tau', \Gamma_2, y : \tau' \vdash e' : \rho\ x$$

and by Postulate 4 (Substitution), noting that by the premise of the rule x does not appear free in ρ nor in Γ_2,

$$\Sigma; \Gamma_1, \Gamma_2, y : \tau' \vdash e'\{y/x\} : \rho\ y$$

By weakening,

$$\Sigma; \Gamma, y : \tau' \vdash e'\{y/x\} : \rho\ y$$

We can conclude that

$$\Sigma; \Gamma \vdash \lambda y : \tau'.\ e'\{y/x\} : (\forall y : \tau'.\ \rho\ y)$$

which is precisely what we have to show.

Case EPrint: Immediate.

□

As a corollary, we have the main theorem of this section:

Theorem 3.1 (Type soundness)**.** *If* $\Sigma; \Gamma \vdash t : \bigcirc\tau$*, then either* t *is a value, or* t *is blocked, or there exist* t' *and* Σ' *such that* $\Gamma \vdash (\Sigma; t) \rightsquigarrow (\Sigma'; t')$ *and* $\Sigma'; \Gamma \vdash t' : \bigcirc\tau$*.*

Proof. Since t is well-typed with type $\bigcirc\tau$, then the following two cases have to occur: either the head constructor of t is one of the constructors of $\bigcirc$—and it is fully applied—or it is another CIC construct (another constant, a let-binding, etc.).

In the first case we have further three sub-cases:

1. It is a value (of the form **ret** e or **raise** e).
2. It is *blocked*, that is, is of any of the following forms:
   - It is $\nu x.\ v$ and $x \in \mathsf{FV}(v)$.
   - It is **abs** $e\ e'$ and $\Sigma; \Gamma \vdash e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*} e''$ and $(e'' : \_) \notin \Gamma$.
   - It is @**abs** $\tau'\ \rho\ e\ e'$ and $e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*} x$ and $\Gamma = \Gamma_1, x : \tau', \Gamma_2$ and $x \in \mathsf{FV}(\Gamma_2, \rho)$.
   - It is **mmatch** $e\ ps$ and no pattern in ps unifies with e.
3. There exist a t' and Σ' such that $\Gamma \vdash (\Sigma; t) \rightsquigarrow (\Sigma'; t')$.

In the first two cases there is nothing left to prove. In the last case we conclude by applying Theorem 3.2.

If t's head constant is another CIC construct, then it either reduces using the standard CIC reduction rules, in which case the proof follows by preservation of CIC reduction rules, or it does not reduce and hence it is blocked. □

3.5 Implementation

This section presents a high-level overview of the architecture of our Mtac extension to Coq, explaining our approach for guaranteeing soundness even in the possible presence of bugs in our Mtac implementation.

The main idea we leverage in integrating Mtac into Coq is that Coq distinguishes between fully and partially type-annotated proof terms: Coq's type inference (or *elaboration*) algorithm transforms partially annotated terms into fully annotated ones, which are then fed to Coq's kernel type checker. In this respect Coq follows the typical architecture of interactive theorem provers, ensuring that all proofs are ultimately certified by a small trusted kernel. Assuming that the kernel is correct, no code outside this kernel may generate incorrect proofs. Thus, our Mtac implementation modifies only the elaborator lying outside of Coq's kernel, and leaves the kernel type checker untouched.

3.5.1 Extending Elaboration

The typing judgment used by Coq's elaboration algorithm (Sacerdoti Coen, 2004, Saïbi, 1997) takes a partially type-annotated term e, a local context Γ, a unification variable context Σ, and an optional expected type τ', and returns its type τ, and produces a fully annotated term e', and updated unification variable context Σ'.

$$\Sigma; \Gamma \vdash_{\tau'} e \hookrightarrow e' : \tau \ \triangleright \ \Sigma'$$

If an expected type τ', is provided, then the returned type τ will be convertible to it, possibly instantiating any unification variables appearing in both τ and τ'. The elaboration judgment serves three main purposes that the kernel typing judgment does not support:

1. To resolve implicit arguments. We have already seen several cases where this is useful (*e.g.*, in §3.1), allowing us to write underscores and let Coq's unification mechanism replace them with the appropriate terms.

2. To insert appropriate coercions. For example, Ssreflect (Gonthier et al., 2008) defines the coercion $\mathsf{is\_true} : \mathsf{bool} \to \mathsf{Prop} := (\lambda b.\ b = \mathsf{true})$. So whenever a term of type Prop is expected and a term b of type bool is encountered, elaboration will insert the coercion, thereby returning the term $\mathsf{is\_true}\ b$ having type Prop.

3. To perform canonical structure resolution. We have seen ample examples of canonical structures in Chapter 2. As we mentioned in that chapter, the instances are resolved during unification, which is a subprocess of elaboration.

We simply extend the elaboration mechanism to perform a fourth task, namely to run Mtactics. We achieve this by adding the following rule for handling **run** t terms:

$$\frac{\Sigma; \Gamma \vdash_{\bigcirc\tau'} t \hookrightarrow t' : \bigcirc\tau \ \triangleright\ \Sigma' \qquad \Gamma \vdash (\Sigma'; t') \rightsquigarrow^* (\Sigma''; \textbf{ret}\ e)}{\Sigma; \Gamma \vdash_{\tau'} \textbf{run}\ t \hookrightarrow e : \tau \ \triangleright\ \Sigma''}$$

This rule first recursively elaborates the tactic body, while also unifying the return type τ of the tactic with the expected goal τ' (if present). This results in the refinement of t to a new term t', which is then executed. If execution terminates successfully returning a term e (which from Theorem 3.1 will have type τ), then that value is returned. Therefore, as a result of elaboration, all **run** t terms are replaced by the terms produced when running them, and thus the kernel type checker does not need to be modified in any way.

3.5.2 Elaboration and the apply Tactic

We have just seen *how* the elaborator coerces the return type τ of an Mtactic to be equivalent to the goal τ', but we did not stipulate in what situations the knowledge of τ' is available. Our examples so far assumed τ' was given, and this was indeed the case thanks to the specific ways we invoked Mtac. For instance, at the end of §3.1 we proved a lemma by *direct definition*—*i.e.*, providing the proof term directly—and in §3.3.1 we proved the goal by calling the Ssreflect tactic apply: (note the colon!). In both these situations, we were conveniently relying on the fact that Coq passed the knowledge of the goal being proven into the elaboration of **run**.

Unfortunately, not every tactic does this. In particular, the standard Coq tactic apply (without colon) does not provide the elaborator with the goal as expected type, so if we had written apply (**run** (F _ _)), the Mtactic F would have been executed on unknown parameters, resulting in a different behavior from what we expect. (Specifically, it would have unified the implicits with the first two pointers appearing in the heap, succeeding only if, luckily, these are the pointers in the goal.)

To ensure that information about the goal is available when running Mtactics, we recommend installing Ssreflect (Gonthier et al., 2008). However, we note that using the standard Coq tactic refine instead of apply also works.

One last point about tactics: Mtac is intended as a typed alternative to Ltac for developing custom automation routines, and it is neither intended to replace the built-in tactics (like apply) nor to subsume all uses of existing Coq tactics. For example, the OCaml tactic vm_compute enables dramatic efficiency gains for reflection-based proofs (Grégoire

```
01 Class runner A (t : ○A) := { eval : A }.
02
03 Hint Extern 20 (runner ?t) ⇒
04       (exact (Build_runner t (run t)))
05       : typeclass_instances.
```

FIGURE 3.14: Type class for delayed execution of Mtactics.

and Leroy, 2002), but its performance depends critically on being *compiled*. Mtac is interpreted, and it is not clear how it could be compiled, given the interaction between Mtac and Coq unification.

3.5.3 Delaying Execution of Mtactics for Rewriting

Consider the goal from §3.3.1, after doing pose F := **run** (noalias D), unfolding the implicit is_true coercions for clarity:

$$\frac{D : \mathsf{def}\ (h_1 \bullet (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2) \bullet (h_2 \bullet x_3 \mapsto v_3)) \qquad F : \forall x\ y.\ \bigcirc((x\ \texttt{!=}\ y) = \mathsf{true})}{(x_1\ \texttt{!=}\ x_2) = \mathsf{true} \wedge (x_2\ \texttt{!=}\ x_3) = \mathsf{true}}$$

Previously we solved this goal by applying the Mtactic F twice to the two subgoals x_1 != x_2 and x_2 != x_3. An alternative way in which a Coq programmer would hope to solve this goal is by using Coq's built-in rewrite tactic. rewrite enables one to apply a lemma one or more times to reduce various *subterms* of the current goal. In particular, we intuitively ought to be able to solve the goal in this case by invoking rewrite !(**run** (F _ _)), where the ! means that the Mtactic F should be applied repeatedly to solve any and all pointer inequalities in the goal. Unfortunately, however, this does not work, because—like Coq's apply tactic—rewrite typechecks its argument without knowledge of the expected type from the goal, and only later unifies the result with the subterms in the goal. Consequently, just as with apply, F gets run prematurely.

Fortunately, we can circumvent this problem, using a cute trick based on Coq's type class resolution mechanism.

Type classes are an advanced Coq feature similar to canonical structures, with the crucial difference that their resolution is triggered by proof search *at specific points* in the tactics and commands of Coq (Sozeau and Oury, 2008). We exploit this functionality in Figure 3.14, by defining the class runner, which is parameterized over an Mtactic t with return type τ and provides a value, eval, of the same type. We then declare a **Hint**

instructing the type class resolution mechanism how to build an instance of the runner class, which is precisely by running t.

The details of this implementation are a bit of black magic and, although we will provide some details below for the interested reader, it is beyond the scope of this thesis to explain fully. Intuitively, all that is going on is that eval is *delaying* the execution of its Mtactic argument until type class resolution time, at which point information about the goal to be proven is available.

Returning to our example, we can now use the following script:

$$\mathsf{rewrite}\ !(\mathsf{eval}\ (F\ \_\ \_))\,.$$

This will convert the goal to $\mathsf{is\_true}\ \mathsf{true} \wedge \mathsf{is\_true}\ \mathsf{true}$, which is trivially solvable.

In fact, with eval we can even employ the standard apply tactic, with the caveat that eval creates slightly bigger proof terms, as the final proof term will also contain the unevaluated Mtactic inside it.

Turning Black Magic into Grey Magic. As promised, we provide the interested reader with some of the key ingredients to understand how the trick works. In particular, we are interested in two things: (1) the time at which type class resolution is triggered, and (2) the dynamic construction of a type class instance. About the former, we mentioned already that type class resolution happens at specific points in the tactics and commands of Coq. More precisely, Coq sources include an Ocaml primitive called solve_typeclasses, which takes all the meta-variables with class type from the meta-context, and proceeds to find an instance for each of them. This primitive, for instance, is called by the apply tactic *after* the tactic has unified the goal with the conclusion of the lemma. Similarly, and key to understanding why eval works in the example above, the rewrite tactic calls the primitive after *each* unification of the left/right hand side of the equation with the subterms of the goal. At this point, the Mtactic knows which is the pointer inequality of interest, and is able to provide a proof for it.

About the dynamic construction of instances, the idea is to execute an Mtactic at the moment of instance resolution. Therefore, when presented with the problem of finding an instance for some meta-variable $?X$ with type runner t, we construct it by **run**ing the Mtactic t. Obviously, we can achieve this by extending the type class resolution algorithm with the capability to execute Mtactics, but fortunately this is not needed. Instead, we use the existing command **Hint Extern**, which allows for the dynamic construction of an instance by executing a given tactic. In our case, the tactic is the simple variant of apply, exact, providing the term Build_runner t (**run** t), where Build_runner is

```
01 Definition MyException (s : string) : Exception.
02   exact exception.
03 Qed.
04
05 Definition AnotherException : Exception.
06   exact exception.
07 Qed.
08
09 Definition test_ex e :=
10   mtry (raise e) with
11     | AnotherException ⇒ ret ""
12     | MyException "hello" ⇒ ret "world"
13     | [s] MyException s ⇒ ret s
14   end.
```

FIGURE 3.15: Exceptions in Mtac.

the constructor of the class. This effectively results in the execution of the Mtactic t at instance resolution time.

3.5.4 A Word about Exceptions

In ML, exceptions have type exn and their constructors are created via the keyword **exception**, as in

exception MyException **of** string

Porting this model into Coq is difficult as it is not possible to define a type without simultaneously defining its constructors. Instead, we opted for a simple yet flexible approach. We define the type Exception as isomorphic to the unit type, and to distinguish each exception we create them as *opaque*, that is, irreducible. Figure 3.15 shows how to create two exceptions, the first one parameterized over a string. What is crucial is the sealing of the definition with **Qed**, signaling to Coq that this definition is opaque. The example test_ex illustrates the catching of different exceptions.

3.5.5 Controlling the Size of Terms

Some tactics tend to generate unnecessarily big terms. Take for instance the to_ast Mtactic from Figure 3.11. In the case of an addition, this Mtactic constructs an instance of the ast structure using the values projected out from calling the Mtactic recursively. That means that the final proof term will contain several copies of a structure, one for

each projector called on that structure. As a matter of fact, for a type that admits a term of size n we can end up constructing a term with size as big as n^2!

For this reason, Mtac is equipped with three different unit operators: **ret**, **retS**, **retW**. The first one we have already seen in the rest of the work; the other two are new and are actually used extensively in the source files (Ziliani, 2014). They both reduce the term prior to returning it: **retS** using the simplification strategy from the simpl tactic, and **retW** using the weak head reduction strategy.

For instance, **run**-ning the following terms produces different outputs:

$$\begin{aligned} \textbf{ret}\ (1+1) &\rightsquigarrow 1+1 \\ \textbf{retS}\ (1+1) &\rightsquigarrow 2 \\ \textbf{retW}\ (1+1) &\rightsquigarrow \mathsf{S}\ (0+1) \end{aligned}$$

As mentioned above, **retS** is critical to reduce the size of terms containing nested projections of structures. For instance, given a context G (in the sense of Section 3.3.3, that is, an element of type ctx) the term

$$@\mathsf{term\_of}\ G\ 0\ (@\mathsf{Ast}\ G\ 0\ \mathsf{syn\_zero}\ G\ (\mathsf{zero\_pf}\ G)),$$

which may result as intermediate step of the to_ast Mtactic, can be drastically reduced to

$$\mathsf{syn\_zero}$$

by using simplification. Note the 4 copies of G in the original term!

For details on the reduction strategies see Section 1.2.2 or The Coq Development Team (2012).

3.6 Stateful Mtactics

So far, we have seen how Mtactics support programming with a variety of effects, including general recursion, syntactic pattern matching, exceptions, and unification, that are not available in the base logic of Coq. If, however, we compare these to the kinds of effects available in mainstream programming languages, we will immediately spot a huge omission: mutable state! This omission is costly: the lack of imperative data structures can in some cases preclude us from writing efficient Mtactics.

One example is the tautology prover from Section 3.3.2, whose running time can be asymptotically improved by using a hashtable. To see this, consider the complexity of

$$
\begin{array}{ll}
\bigcirc & : \mathsf{Type} \to \mathsf{Prop} \\
\ldots & \\
\textbf{hash} & : \forall A.\ A \to \mathsf{N} \to \bigcirc\mathsf{N} \\
\textbf{array\_make} & : \forall A.\ \mathsf{N} \to A \to \bigcirc(\mathsf{array}\ A) \\
\textbf{array\_get} & : \forall A.\ \mathsf{array}\ A \to \mathsf{N} \to \bigcirc A \\
\textbf{array\_set} & : \forall A.\ \mathsf{array}\ A \to \mathsf{N} \to A \to \bigcirc\mathsf{unit} \\
\textbf{array\_length} & : \forall A.\ \mathsf{array}\ A \to \mathsf{N}
\end{array}
$$

FIGURE 3.16: The new array primitives of the $\bigcirc$ inductive type.

proving the following class of tautologies:

$$p_1 \to \ldots \to p_n \to p_1 \wedge \ldots \wedge p_n$$

in a context where $p_1\ \ldots\ p_n : \mathsf{Prop}$. First, the tautology prover will perform n steps to add hypotheses $h_1 : p_1, \ldots, h_n : p_n$ to the context. Then, it will proceed to decompose the conjunctions and prove each p_i by searching for it in the context. Since the lookup Mtactic performs a linear search on the context, the overall execution will take $O(n^2)$ steps in total. Had we used a more clever functional data structure for implementing the context, such as a balanced tree, we could reduce the lookup time to logarithmic and the total running time to $O(n \log n)$. By using a hashtable to represent the context, however, we can do much better: we can achieve a constant-time lookup, thereby reducing the total proof search time to $O(n)$.

Rather than directly adding hashtables to Mtac, in this section, we show how to extend Mtac with two more primitive constructs: arrays and a hashing operator (§3.6.1). With these primitives, we implement a hashtable in Mtac (§3.6.2) and use it to implement a more efficient version of the tautology prover (§3.6.3), obtaining the expected significant speedup (§3.6.6).

The model of mutable state that we decided to incorporate to Mtac is very flexible and allows us to write complex imperative structures, but that comes at a price. First of all, combining mutable state with parameters (*i.e.*, the **nu** and **abs** operators) and syntax inspection (*i.e.*, the **mmatch** operator) is tricky and requires special care in the design of the operational semantics (§3.6.4). Second, the interactive nature of a proof assistant like Coq enforces certain constraints (§3.6.5). Despite these difficulties, the language enhanced with mutable state remains sound (§3.6.7).

3.6.1 New Language Constructs

Figure 3.16 shows the new language constructs, where N stands for binary encoded natural numbers and array τ is an abstract type representing arrays of type τ (this type will be discussed later). The new constructs are explained next:

- **hash** e n takes term e and natural number n and returns the *hash* of e as a natural number between 0 and $n-1$, ensuring a fair distribution.
- **array_make** n e creates an array with n elements initialized to e.
- **array_get** a i returns the element stored in position i in the array a. If the position is out of bounds, it raises an exception.
- **array_set** a i e stores element e in position i in the array a. If the position is out of bounds, it raises an exception.
- **array_length** a returns the size of array a. Note that it's not part of the ○ inductive type.

For each of these constructs, the type parameter A is treated as an implicit argument.

In the rest of the section we will use *references*, which are arrays of length 1. We will use a notation resembling that of ML: ref e creates a reference of type Ref τ, where $e : \tau$, $!r$ returns the current contents of reference r, and $r ::= e$ updates r to e.

3.6.2 A Dependently Typed Hashtable in Mtac

The aforementioned primitives are enough to build stateful algorithms and data structures, in particular a dependently typed hashtable. We present the hashtable below, introducing first a required module:

The Array module. Figure 3.17 presents the wrapper module Array, emulating the one in OCaml. The motivation for this module is merely to pack together all array operations into one module, and to provide the developer with a familiar interface. Besides the four primitive operations on arrays (make, length, get, set, mere synonyms of the primitives shown in Figure 3.16), the module also provides an iterator iter, an array constructor with an initialization function init (which throws an exception if asked to create a 0-length array), a conversion function from array to seq, and a copy function that copies the elements from the first array into the second one, if there's enough space, or fails otherwise. We omit the code for these, but they are standard and can be found in the source files (Ziliani, 2014).

Definition t A := array A.

Definition make $\{A\}$ n $(c : A)$:= array_make n c.
Definition length $\{A\}$ $(a :$ t $A)$:= array_length a.
Definition get $\{A\}$ $(a :$ t $A)$ i := array_get a i.
Definition set $\{A\}$ $(a :$ t $A)$ i $(c : A)$:= array_set a i c.

Definition iter $\{A\}$ $(a :$ t $A)$ $(f :$ N $\to A \to$ ○unit$)$: ○unit := ...
Definition init $\{A\}$ n $(f :$ N $\to$ ○$A)$: ○(t A) := ...
Definition to_list $\{A\}$ $(a :$ t $A)$: ○(seq A) ⇒:= ...
Definition copy $\{A\}$ $(a$ $b :$ t $A)$: ○unit := ...

FIGURE 3.17: The Array module.

The HashTbl module. Figure 3.18 presents the module HashTbl, which implements a (rudimentary) dependently typed hashtable. At a high level, given types τ and $\rho : \tau \to$ Type, a hashtable maps each key x of type τ to a term of type $\rho\ x$. More concretely, it consists of a pair of references, one containing the load of the hash (how many elements were added), and the other containing an array of "buckets". Each bucket, following the standard *open hashing* strategy for conflict resolution, is a list of Σ-types packing a key with its element. In open hashing, each bucket holds all the values in the table whose keys have the same hash, which is the index of that bucket in the array.

The hashtable is created with an initial size (initial_size) of 16 buckets, and every time it gets expanded it increases by a factor (inc_factor) of 2. The threshold to expand the hashtable is when its load reaches 70% of the buckets.

The function quick_add adds key x, mapped to element y, to the table. It does so by hashing x, obtaining a position i for a bucket l, and adding the existential package containing both x and y to the head of the list. Note that this function does not check whether the threshold has been reached and is thus not intended to be called from the outside—it is just used within the HashTbl module as an auxiliary function.

The function expand first creates a new array of buckets doubling the size of the original one, and then it iterates through the elements of the original array, adding them to the new array. We omit the code of the iterator function iter for brevity.

The function add first checks if the load exceeds the threshold and, if this is the case, proceeds to expand the table. It then adds the given key-element pair to the table. Finally, the count of the load is increased by one. Since the load is a binary natural, and for binary naturals the successor (succ) operation is a function, we force the evaluation of succ *load* by simplifying it with **retS**.

```
Definition t A (P : A → Type) :=
  (Ref N × Ref (Array.t (seq {x : A & P x}))).

Definition initial_size := 16.
Definition inc_factor := 2.
Definition threshold := 7.

Definition create A P : ○(t A P) :=
  n ← ref 0; a ← Array.make initial_size []; ra ← ref a;
  ret (n, ra).

Definition quick_add {A P} a (x : A) (y : P x) : ○unit :=
  let n := Array.length a in i ← hash x n; l ← Array.get a i;
  Array.set a i (existT _ x y :: l).

Definition iter {A P} (h : t A P) (f : ∀ x : A. P x → ○unit) : ○unit := ...

Definition expand {A P} (h : t A P) : ○unit :=
  let (_, ra) := h in a ← !ra; let size := Array.length a in
  let new_size := size × inc_factor in new_a ← Array.make new_size [];
  iter h (λ x y. quick_add new_a x y);; ra ::= new_a.

Definition add {A P} (h : t A P) (x : A) (y : P x) :=
  let (rl, ra) := h in load ← !rl; a ← !ra; let size := Array.length a in
  (if threshold × size ≤ 10 × load then expand h else ret tt);;
  a ← !ra; quick_add a x y;;
  new_load ← retS (N.succ load); rl ::= new_load.

Definition find {A P} (h : t A P) (x : A) : ○(P x) :=
  let (_, ra) := h in a ← !ra;
  let size := Array.length a in
  i ← hash x size; l ← Array.get a i;
  list_find x l.

Definition remove {A P} (h : t A P) (x : A) : ○unit :=
  let (rl, ra) := h in a ← !ra;
  let size := Array.length a in
  i ← hash x size; l ← Array.get a i;
  l' ← list_remove x l;
  Array.set a i l';;
  load ← !rl; new_load ← retS (N.pred load);
  rl ::= new_load.
```

FIGURE 3.18: The HashTbl module.

The function list_find first obtains the bucket l corresponding to the hashed index i of the key x, and then performs a linear search using the function

$$\mathsf{list\_find}\ (A : \mathsf{Type})\ (P : A \to \mathsf{Type})\ (x : A)\ (l : \mathsf{seq}\ \{z : A\ \&\ P\ z\}) : \bigcirc(P\ x)$$

We omit the code of this function since it is similar to the lookup function from §3.3.2.

Similarly, the function list_remove removes an element from the hashtable by obtaining the bucket l of the key x and removing the element using the auxiliary function

$$\begin{aligned}&\mathsf{list\_remove}\ (A : \mathsf{Type})\ (P : A \to \mathsf{Type})\ (x : A)\ (l : \mathsf{seq}\ \{z : A\ \&\ P\ z\})\\ &\quad : \bigcirc(\mathsf{seq}\ \{z : A\ \&\ P\ z\})\end{aligned}$$

This function throws the exception NotFound if the element is not in the list, and only removes one copy if the key x appears more than once.

3.6.3 The Tautology Prover Revisited

The code for the new tautology prover with hashing of hypotheses is listed in Figure 3.19. The type used to represent contexts is defined in the first line: it is a dependently typed hashtable whose keys are propositions and whose elements are proofs of those propositions. The prover itself begins on line 3. The cases for the trivial proposition, conjunction and disjunction (lines 5–17), as well as for $\forall$ (lines 23–26), are similar to the original ones, except that the context is not threaded through recursive calls but rather updated imperatively.

The implication, existential, and base cases require more drastic changes. The implication case is modified to extend the hashtable (line 20) with the mapping of parameter x with hypothesis p_1. In order to avoid leaving garbage in the hashtable, the added element is removed after the recursive call. Since the recursive call may raise an exception, the removal is performed before re-throwing the exception.

The base case (line 36) is modified to perform the lookup in the hashtable.

Finally, the existential case (line 27), requires a bit of explanation. It starts, as in the previous prover, by creating a unification variable for the witness X (line 28). Then it differs substantially. The reason why we need to change its behavior is simple: in the previous prover we were expecting it to find the solution for the witness by unification during the lookup in the base case. Since now we have a hash table for the lookup, there is no unification process going on, and therefore no instantiation for the witness can occur.

Here instead we do the following: we still try to find a solution recursively (line 30) for when the solution is trivial or does not depend on the witness—for instance, consider the case $\exists p : \mathsf{Prop}.\ p$ with trivial solution

$$\mathsf{ex\_intro}\ (\lambda p : \mathsf{Prop}.\ p)\ \mathsf{True}\ \mathsf{I}$$

If no solution is found (*i.e.*, an exception ProofNotFound is raised), then we create another unification variable for the proof r and return the proof term $\mathsf{ex\_intro}\ q\ X\ r$ (line 34). That is, we return a proof with a hole, expecting the proof developer to fill the hole later on. For instance, if we run the tautology prover on the example

$$\forall x : \mathsf{nat}.\ \exists y : \mathsf{nat}.\ y \leq x$$

then Coq will ask, as a subgoal, for a proof of $?X \leq x$. The proof developer can proceed to provide the standard proof of $0 \leq x$, thereby instantiating $?X$ with 0.

More to the point, this example focuses primarily on the use of a hashtable for quick lookups for hypotheses. It is possible to create a more involved prover combining both a hashtable- and a list-based context to regain the previous functionality for existentials, paying the price of linear lookup only when needed.

3.6.4 Operational Semantics

As we mentioned in the introduction of this section, we need to take special care when combining mutable state with parameters and syntax inspection.

Mutable State and Parameters: The combination of these two features requires us to adjust the operational semantics for the **nu** (ν) operator in order to preserve soundness. More precisely, we need to ensure that if we store a parameter x in an array or reference, then we should not be able to read from that location outside the scope of x. Take for instance the following example:

Definition wrong $:= r \leftarrow$ **ref** 0; (ν x:nat. $r ::= x$);; $!r$.

In this code, first a reference r is created, then a new parameter x is created and assigned to r. Later, outside of the scope of x, r is dereferenced and returned. Without checking the context of the element being returned, the result of this computation would be undefined.

```
01 Definition ctx := HashTbl.t Prop (λ x. x).
02
03 Definition tautoh' (c : ctx) := mfix f (p : Prop) : ○p :=
04   mmatch p as p' return ○p' with
05   | True ⇒ ret I
06   | [p1 p2] p1 ∧ p2 ⇒
07         r1 ← f p1 ;
08         r2 ← f p2 ;
09         ret (conj r1 r2)
10   | [p1 p2] p1 ∨ p2 ⇒
11         mtry
12           r1 ← f p1 ;
13           ret (or_introl r1)
14         with _ ⇒
15           r2 ← f p2 ;
16           ret (or_intror r2)
17         end
18   | [p1 p2 : Prop] p1 → p2 ⇒
19         ν (x:p1).
20           HashTbl.add c p1 x;;
21           mtry r ← f p2; HashTbl.remove c p1;; abs x r
22           with [e] e ⇒ HashTbl.remove c p1;; raise e end
23   | [A (q:A → Prop)] (∀ x:A, q x) ⇒
24         ν (x:A).
25           r ← f (q x);
26           abs x r
27  | [A (q:A → Prop)] (∃ x : A. q x) ⇒
28        X ← evar A;
29        mtry
30          r ← f (q X);
31          ret (ex_intro q X r)
32        with ProofNotFound ⇒
33          r ← evar (q X);
34          ret (ex_intro q X r)
35        end
36   | [x] x ⇒ HashTbl.find c x
37   end.
38
39 Definition tautoh P :=
40   c ← HashTbl.create Prop (λ x. x);
41   tautoh' c P.
```

FIGURE 3.19: Tautology prover with hashing of hypotheses.

With unification variables we also have this problem—we can encode a similar example using **evar** instead of **ref** and **mmatch** instead of the assignment. But with unification variables, the contextual type of the unification variable would prevent us from performing the instantiation, therefore effectively ensuring that "nothing goes wrong". For mutable state, on the other hand, it would be too restrictive to restrict assignments to only include variables coming from the context where the array was created. Take for instance the tautology prover: there, in the implication case, the parameter x is added to a hashtable that was created outside the scope of x.

Thus, for mutable state we take a different, more "dynamic" approach: before returning from a νx binder we *invalidate* all the cells in the state that refer (in the term or in the type) to x. If later on a read is performed on any such cell, the system simply gets blocked.

Mutable State and Syntax Inspection: The problem with combining mutable state and the **mmatch** constructor concerns (lack of) abstraction. In short, we would like to be able to prevent the user from breaking abstraction and injecting a spurious reference (a location that is not in the domain of the store or with a different type from the one in the store) in the execution of a Mtactic, as that would result in a violation of type soundness. However, we have been unable to find any simple and elegant way of preventing the user from doing that, and so instead we choose to incur the cost of dynamic safety checks when performing stateful operations.

In order to understand the problem, let us explain first how it is handled in Haskell, and why we cannot simply port the same approach to Mtac. In Haskell, the **runST** command (Launchbury and Peyton Jones, 1994) enables one to escape the stateful ST monad by executing the stateful computation inside, much as with Mtac's **run** and ○ monad.

Since **runST** uses an effectful computation to produce a term of a pure Haskell type, it is essential to the soundness of **runST** that its computation runs inside a fresh store. Moreover, it is important to prevent the user from attempting to access an old reference within a new call to **runST**, as in the following code (here in Mtac syntax).

let r := **run** (**ref** 0) **in run** !r

After creating the reference r in the first **run**, the reference is read in the second. In Haskell, this situation is prevented by giving **runST** the following more restrictive type:

$$\forall A.\ (\forall S.\ \mathsf{ST}\ S\ A) \to A$$

where S is a phantom type parameter representing the state and A is the return type of the monad, which may not mention S. By making the monad parametric in the state type, the typechecker effectively ensures that the computation being executed can neither leak any references to nor depend on any references from its environment. For example, in the code above, the command **ref** 0 has type $\mathsf{ST}\ S\ (\mathsf{Ref}\ S\ \mathsf{nat})$ for some S, but this cannot be generalized to **runST**'s argument type $(\forall S.\ \mathsf{ST}\ S\ A)$ because $A = \mathsf{Ref}\ S\ \mathsf{nat}$ mentions S.

Crucially, Haskell's built-in abstraction mechanisms also hide the *constructor* for the Ref type. That is, in Haskell the user is not entitled to pattern match an element of the Ref type simply because its constructor is hidden from the user.

Unfortunately, the same techniques will not help us in Mtac. The problem arises from the following observation: for any constant c with type τ, if Mtac can create it, Mtac can inspect it. That is, we can always pattern match a term of type τ and get a hold of its head constant (*e.g.*, c). The same happens with references. Say that we create a new type $\mathsf{Ref}\ A$ with constructor

$$\mathsf{cref} : \forall A.\ \mathsf{Loc} \to \mathsf{Ref}\ A$$

Say further that we hide the constructor from the user (by using the module system of Coq), so that the proof developer is not entitled to write the following code:

$$\textbf{mmatch}\ x\ \textbf{with}\ [l : \mathsf{Loc}]\ \mathsf{cref}\ l\ \Rightarrow \ldots\ \textbf{end}$$

There is still a way of achieving the same behavior with the following code:

$$\textbf{mmatch}\ x\ \textbf{with}\ [(c : \mathsf{Loc} \to \mathsf{Ref}\ A)\ (l : \mathsf{Loc})]\ c\ l\ \Rightarrow \ldots\ \textbf{end}$$

since, if Mtac is (somehow) allowed to construct a cref, then nothing can prevent it from matching a meta-variable with a cref.

And this can be disastrous from the perspective of soundness. Imagine, for instance, that the location of a reference with type A is accessed at type B:

mmatch $(x,\ y)$ **with**
| $[(c_1 : \mathsf{Loc} \to \mathsf{Ref}\ A)\ (c_2 : \mathsf{Loc} \to \mathsf{Ref}\ B)\ l_1\ l_2]\ (c_1\ l_1,\ c_2\ l_2) \Rightarrow\ !(c_2\ l_1)$
end

One option for solving this problem might be to forbid **mmatch** from pattern matching an element containing a reference. However, this solution is too strict, as it would disallow legitimate matches on terms of data types containing references.

Instead, the solution that we have adopted in Mtac is not to hide cref at all, but rather to bite the bullet and perform the necessary dynamic checks to ensure that the code above gets blocked. More precisely, under Mtac semantics, the read of $c_2\ l_1$ will get blocked as it is reading from the location l_1 at a different type (B) from the expected one (A). Similarly, in the previous example, the attempt to read the reference r in **run** $!r$ will get blocked since **run** $!r$ executes in a fresh store and r is not in the domain of that store.

With these considerations in mind, we can now move on to explain the actual rules. We extend the judgment from Section 3.4 to include input and output stores. A store σ is a map from locations l to arrays annotated with the type τ of their elements. An array element d is either null or a Coq term:

$$\sigma ::= \cdot \mid l \mapsto [d; \ldots; d]_\tau, \sigma$$
$$d ::= \mathsf{null} \mid e$$

The new judgment is

$$\Sigma; \Gamma; \sigma \vdash t \rightsquigarrow (\Sigma'; \sigma'; t')$$

On the Coq side, we have an inductive type of locations Loc and an inductive array type, where the number in carray represents the length of the array:

$$\begin{aligned} &\mathsf{array} \ : \mathsf{Type} \to \mathsf{Type} \\ &\mathsf{carray} \ : \forall A.\ \mathsf{Loc} \to \mathsf{N} \to \mathsf{array}\ A \end{aligned}$$

Since we need to transform numbers and locations from Coq to ML and vice versa, we write $\overline{e}$ for the interpretation of Coq numbers/locations e into ML, implicitly reducing e to normal form, and $\underline{e}$ for the interpretation of ML numbers/locations e into Coq.

Figure 3.20 shows the new rules concerning effectful computations. The first rule presents the aforementioned modifications to the original rule for the νx binder. The changes (highlighted) show that, upon return of the νx binder, we invalidate all the arrays whose type contains x, and all the array positions referring to x. The invalidate judgment is given in Figure 3.21.[5]

[5]For performance reasons, in the implementation we don't traverse the entire state. Instead, we keep a map relating each parameter x introduced by a νx binder with a list of array positions. When the parameter x is going out of scope, all the array positions associated with it are invalidated. The map is

$$\frac{x \notin \mathsf{FV}(v) \quad \mathsf{invalidate}\ \sigma\ x = \sigma'}{\Sigma; \Gamma;\ \sigma\ \vdash (\nu x.\ v) \rightsquigarrow (\Sigma;\ \sigma'\ ; v)}\text{ENuV}$$

$$\frac{h = \mathsf{hash}(e) \bmod \overline{s}}{\Sigma; \Gamma; \sigma \vdash \textbf{hash}\ e\ s \rightsquigarrow (\Sigma; \sigma; \textbf{ret}\ \underline{h})}\text{EHash}$$

$$\frac{l \text{ fresh}}{\Sigma; \Gamma; \sigma \vdash \textbf{array\_make}\ \tau\ n\ e \rightsquigarrow (\Sigma; l \mapsto [e; \overset{n-2}{\ldots}; e]_\tau, \sigma; \textbf{ret}\ (@\mathsf{carray}\ \tau\ \underline{l}\ n))}\text{EAMake}$$

$$\frac{\Sigma; \Gamma \vdash a \overset{\mathsf{whd}*}{\rightsquigarrow} @\mathsf{carray}\ \tau'\ l\ \_ \quad \overline{i} < n \quad e_{\overline{i}+1} \neq \mathsf{null} \quad \Sigma; \Gamma \vdash \tau \approx \tau' \rhd \Sigma'}{\Sigma; \Gamma; \sigma, \overline{l} \mapsto [e_1, \ldots, e_n]_\tau \vdash \textbf{array\_get}\ a\ i \rightsquigarrow (\Sigma'; \sigma; \textbf{ret}\ e_{\overline{i}+1})}\text{EAGetR}$$

$$\frac{\Sigma; \Gamma \vdash a \overset{\mathsf{whd}*}{\rightsquigarrow} \mathsf{carray}\ l\ \_ \quad \overline{i} \geq n}{\Sigma; \Gamma; \sigma, \overline{l} \mapsto [e_1, \ldots, e_n]_\tau \vdash \textbf{array\_get}\ a\ i \rightsquigarrow (\Sigma; \sigma; \textbf{raise}\ \mathsf{OutOfBounds})}\text{EAGetE}$$

$$\frac{\Sigma; \Gamma \vdash a \overset{\mathsf{whd}*}{\rightsquigarrow} @\mathsf{carray}\ \tau'\ l\ \_ \quad \overline{i} < n \quad \Sigma; \Gamma \vdash \tau \approx \tau' \rhd \Sigma'}{\Sigma; \Gamma; \sigma, \overline{l} \mapsto [e_1, \ldots, e_n]_\tau \vdash \textbf{array\_set}\ a\ i\ e \rightsquigarrow (\Sigma'; \sigma, \overline{l} \mapsto [e_1, \overset{\overline{i}-1}{\ldots}, e, \ldots, e_n]_\tau; \textbf{ret}\ \langle\rangle)}\text{EASetR}$$

$$\frac{\Sigma; \Gamma \vdash a \overset{\mathsf{whd}*}{\rightsquigarrow} \mathsf{carray}\ l\ \_ \quad \overline{i} \geq n}{\Sigma; \Gamma; \sigma, \overline{l} \mapsto [e_1, \ldots, e_n]_\tau \vdash \textbf{array\_set}\ a\ i\ e \rightsquigarrow (\Sigma; \sigma; \textbf{raise}\ \mathsf{OutOfBounds})}\text{EASetE}$$

FIGURE 3.20: Operational small-step semantics of references.

The second rule performs the **hash**-ing of a term e, limited by s. The element is hashed using an ML function hash, which we omit here.

The next rule creates an n-element array initialized with element e. A fresh location pointing to the new array is appended to the state, and this location, together with n, are returned as part of the carray constructor.

The next two rules describe the behavior of the getter. In both rules the array is first weak head-reduced in order to obtain the carray constructor applied to location l. Then the rules differ according to the case: if the index i is within the bounds of the array, and the element at index i is defined (*i.e.*, not null), then the type of the array τ is unified with the type coming from the carray constructor (τ'), potentially instantiating new meta-variables. If the index is outside the bounds of the array, an error OutOfBounds is raised.

The two rules for the setter are similar: we first check that the index is within the bounds and that the type of the array unifies with the one from the constructor, and if so, the position i of the array is updated. Otherwise, an exception is raised.

populated in each operation array_set, where we take the set of free parameters and add the position to their lists of positions.

$$\mathsf{invalarr}\ [e_1, \ldots, e_n]_\tau\ x = [e'_1, \ldots, e'_n]_\tau \qquad \forall i \in [1, n].\ e'_i = \begin{cases} \mathsf{null} & \text{if } x \in \mathsf{FV}(e) \\ e_i & \text{otherwise} \end{cases}$$

$$\begin{array}{rll} \mathsf{invalidate}\ []\ x = [] & & \\ \mathsf{invalidate}\ (l \mapsto a_\tau, \sigma)\ x = \mathsf{invalidate}\ \sigma\ x & & x \in \mathsf{FV}(\tau) \\ \mathsf{invalidate}\ (l \mapsto a_\tau, \sigma)\ x = (l \mapsto \mathsf{invalarr}\ a_\tau\ x, \mathsf{invalidate}\ \sigma\ x) & & x \notin \mathsf{FV}(\tau) \end{array}$$

FIGURE 3.21: Invalidation of array positions whose contents are out of scope.

3.6.5 Use Once and Destroy

Allowing state to persist across multiple Mtactic **run**s is possible but technically challenging. For the present work, we have favored a simple implementation, and therefore restricted Mtactics to only use mutable state internally, as in Launchbury and Peyton Jones (1994).

If we were to consider generalizing to handle persistent state, we would encounter at least two key challenges. First, since Coq is interactive, the user is entitled to undo and redo operations. Allowing state to persist across multiple **run**s would thus require the ability to rollback the state accordingly. While this is doable using *persistent arrays* (Baker, 1991) with reasonable amortized space and time complexity, there is still a technical challenge: how to know to which state to rollback.

A second challenge arises from the module system of Coq. In particular, a developer may create references and modify them in different modules. Then, importing a module from another file should replay the effects in that file in order to ensure a consistent view of the storage among different modules.

As mentioned above, we decided to leave this problem for future work and throw away the state after the execution of a Mtactic. While this decision may sound restrictive, it is flexible enough for us to be able to encode interesting examples, such as the one presented in this section.

The rule for elaborating the **run** operator is modified to start the evaluation of the Mtactic with an empty state:

$$\frac{\Sigma; \Gamma \vdash_{\bigcirc\tau'} t \hookrightarrow t' : \bigcirc\tau \ \triangleright\ \Sigma' \qquad \Sigma'; \Gamma; [] \vdash t' \rightsquigarrow^* (\Sigma''; \sigma; \mathbf{ret}\ e)}{\Sigma; \Gamma \vdash_{\tau'} \mathbf{run}\ t \hookrightarrow e : \tau \ \triangleright\ \Sigma''}$$

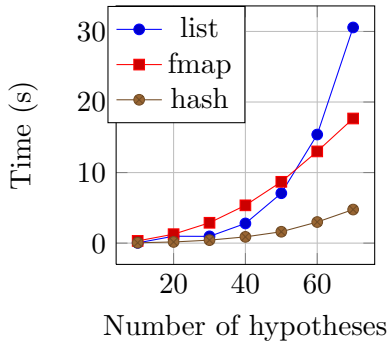

FIGURE 3.22: Performance of three different implementations of tautology provers: using a list, a finite map over a balanced tree, and a hashtable.

3.6.6 A Word on Performance

Despite the various dynamic checks performed by our stateful extensions to Mtactics, we can obtain a significant speedup by using mutable state. To support our claim, we implemented three different versions of the tautology prover, using different data structures (both functional and imperative). Figure 3.22 shows the time it takes for each prover to solve a certain class of tautologies, of the form $p_1 \to \ldots \to p_n \to p_1 \wedge \ldots \wedge p_n$, on an increasing number n of hypotheses. The slowest one is the functional prover from Section 3.3.2, which takes constant time for extending the context and linear time (in the size of the context) for lookup.

The second prover uses a functional map implemented with a balanced tree. As keys for the map we use the natural numbers coming from hashing the hypotheses, similarly to what the prover from Figure 3.19 does. Both extending and querying the context take logarithmic time to compute.

Finally, the third prover is the one presented in this section, which performs both extension and lookup of the context in amortized constant time. As expected, it greatly outperforms the other two.

Before moving on to prove soundness of the system, we want to stress that the safety check done in the getter and setter rules (*i.e.*, the unification of the type of the array with the one coming from the carray constructor) does not result in a significant performance cost. In particular, it is possible to cache the results, effectively avoiding the overhead in the most common cases (which cover almost all of the accesses). Even in our rather naive implementation, our experimental results shows that the invalidation of cells and the unification of types in the getter and setter operations cause no significant slowdowns (less than 5%).

3.6.7 Soundness in the Presence of State

The soundness theorem must now be extended to consider the store. We start by extending the definition of blocked terms to consider the new cases:

Definition 4 (Blocked terms)**.** A term t is *blocked* if and only if the subterm in reduction position satisfies one of the cases of Definition 2, or:

- It is an array operation (get or set), and the array or the index do not normalize to the array constructor or a natural number, respectively.
- It is an array operation (get or set), the array and the index normalize to $@\mathsf{carray}\ \tau\ l\ \_$ and i respectively, for some τ, l and i, but either
  1. l is not in the state σ, or
  2. there exists $l \mapsto a_{\tau'} \in \sigma$, but τ does not unify with τ' or $a_i = \mathsf{null}$.

To state the preservation theorem, we need to say what it means for a store to be *valid*. A store σ is *valid* in contexts Γ and Σ if for every array a with element type τ in σ, every element of a is either null or has type τ. Formally,

$$\frac{}{\Sigma; \Gamma \vdash \cdot\ \mathsf{valid}} \qquad \frac{\forall j \in [1..n].\ e_j = \mathsf{null} \ \vee\ \Sigma; \Gamma \vdash e_j : \tau \qquad \Sigma; \Gamma \vdash \sigma\ \mathsf{valid}}{\Sigma; \Gamma \vdash (l \mapsto [e_1, \ldots, e_n]_\tau, \sigma)\ \mathsf{valid}}$$

We now extend the proof of type preservation, where we need to also ensure preservation of the validity of the store.

Theorem 3.3 (Type preservation with references)**.** *If* $\Sigma; \Gamma \vdash t : \bigcirc\tau$ *and* $\Sigma; \Gamma \vdash \sigma\ \mathsf{valid}$ *and* $\Sigma; \Gamma; \sigma \vdash t \rightsquigarrow (\Sigma'; \sigma'; t')$*, then* $\Sigma'; \Gamma \vdash t' : \bigcirc\tau$ *and* $\Sigma'; \Gamma \vdash \sigma'\ \mathsf{valid}$*.*

Proof. The proof poses little challenge. We only consider the new or modified cases.

Case ENuV: We have $t = \nu x.\ v$. Since x goes out of scope in the returning term, in order to have a valid state we need to remove all the arrays with types referring to x, and invalidate all array positions with contents referring to x. This is precisely what the $\mathsf{invalidate}$ judgment does.

Case EHash: Trivial, as it is returning a number.

Case EAMake: We have $t = \mathsf{array\_make}\ \tau'\ n\ e$. Let l be a fresh location. By hypothesis we know that e has type τ', so the new store $\sigma' = l \mapsto [e; \overset{n-2}{\ldots}; e]_{\tau'}, \sigma$ contains an array of elements of type τ'. Therefore, the new state is valid, and the returned value $@\mathsf{carray}\ \tau'\ l\ n$ has type $\mathsf{array}\ \tau'$.

Case EAGetR: We have $t = \textsf{array\_get}\ \tau'\ a\ i$. By the premise of the rule, a normalizes to $@\textsf{carray}\ \tau'\ l\ \_$ and i to i' such that $l \mapsto [e_1, \ldots, e_n]_\tau \in \sigma$ for some $e_1, \ldots, e_n$. Also by hypothesis, $i' < n$, τ unifies with τ', and $e_{i'+1} \neq \textsf{null}$. Therefore, $e_{i'+1}$ has type (convertible with) τ'. Note that, by soundness of unification (Postulate 9), $\Sigma' \geq \Sigma$.

Case EAGetE: Trivial.

Case EASetR: We have $t = \textsf{array\_set}\ \tau'\ a\ i\ e$. As in the EAGetR case, we also have that $a \overset{\textsf{whd}}{\rightsquigarrow}{}^* @\textsf{carray}\ \tau'\ l\ \_$ and there exist $e_1, \ldots, e_n$ such that $l \mapsto [e_1, \ldots, e_n]_\tau \in \sigma$. We need to show that $\Sigma; \Gamma \vdash \sigma'\ \textsf{valid}$, where $\sigma' = l \mapsto [e_1, \overset{\vec{i}-1}{\ldots}, e, \ldots, e_n]_\tau, \sigma$. By hypothesis and the premises of the rule, we know that e has type τ', unifiable with τ. Therefore, updating the i-th position cannot invalidate the store. Note that, as in the previous case, by soundness of unification (Postulate 9), $\Sigma' \geq \Sigma$.

Case EASetE: Trivial.

□

As before, our main theorem follows as an immediate corollary:

Theorem 3.4 (Type soundness with references)**.** *If* $\Sigma; \Gamma \vdash t : \bigcirc\tau$ *and* $\Sigma; \Gamma \vdash \sigma\ \textsf{valid}$*, then either* t *is a value, or* t *is blocked, or there exist* t'*,* Σ' *and* σ' *such that* $\Sigma; \Gamma; \sigma \vdash t \rightsquigarrow (\Sigma'; \sigma'; t')$ *and* $\Sigma'; \Gamma \vdash t' : \bigcirc\tau$ *and* $\Sigma'; \Gamma \vdash \sigma'\ \textsf{valid}$*.*

As a second corollary, we have that the new rule for **run** t (in §3.6.5) is also sound: since the initial empty store under which t is run is trivially valid, Theorem 3.4 tells us that the type τ of t is preserved, and hence that the returned term e has type τ.

3.7 Characteristics of Mtac and Conclusions

Maintainability. Mtac shares with Lemma Overloading the benefits of typed tactic programming. There is a difference though: the dynamic nature of the νx binder makes, in a sense, Mtactics be "less typed" than overloaded lemmas—*i.e.*, they may *fail* (block) in ways overloaded lemmas are not allowed.

Composability. The noalias Mtactic from Section 3.3.1 was presented to show how trivial is to compose Mtactics, in particular in comparison with Lemma Overloading. We have not mentioned how, or if, we can compose easily Mtactics with standard Coq

lemmas. The answer is: yes, we can! For instance, the Mtactic noalias can be used in exactly the same way as the overloaded lemma from Section 2.6. We show here one of the examples shown at the end of that section, and the others are equally solvable. Assume F as before, with type $\forall x\ y$: ptr. $\bigcirc(x$!= $y)$. We can compose the execution of F with the negbTE lemma and solve the following goal:

Goal : if $(x_2$ == $x_3)$ && $(x_1$!= $x_2)$ then false else true
Proof : by rewrite (negbTE (eval $(F$ _ _)))

Interactive Tactic Programming. As mentioned in Section 3.3.1.1, Mtactics can be built interactively.

Simplicity. The semantics of Mtac is quite simple, as shown by the multiple examples throughout this chapter, most notably the noalias Mtactic from Section 3.3.1. With Lemma Overloading, this example required several advanced features to work properly (Section 2.6). And for those corner cases where intuition is not enough, and we require a more precise description of the semantics, we can turn our attention to figures 3.13 and 3.20.

Formality. As mentioned in the previous paragraph, figures 3.13 and 3.20 provides the semantics of the language. Moreover, in sections 3.4.3 and 3.6.7 we provide the proof of soundness of the system. The only weakness in the semantics of Mtac, as with Lemma Overloading, is the dependency on the unification algorithm. As such, the formalization of such algorithm is crucial if we want to understand fully how Mtactics are run, and this is why we took the endeavor of building and describing a new unification algorithm in Chapter 4.

Performance. Being an interpreted language, Mtac has a competitive performance with, for instance, Ltac (another interpreted language). Furthermore, the addition of stateful tactics allows for a significant reduction of the complexity of Mtactics, as shown in Figure 3.22.

But we think we can do better. In the future we envision two orthogonal improvements to Mtac, which combined should greatly improve the overall performance of the language. The first one is to extract Ocaml code from Mtactics and compile it, borrowing ideas from Claret et al. (2013). The main challenge is how to amortize the cost of dynamic compilation and linking. The second improvement is related to the unification algorithm, which currently is one of the major bottlenecks of Mtac. This thesis makes the first step

of explaining the process of unification (Chapter 4); the next step is to introduce the necessary mechanisms to improve its performance.

Extensibility. Mtac alone does not provide the tools to write extensible tactics. However, it is possible to combine Mtac with overloading and build extensible tactics. The idea is dead simple, although it requires some thinking to make it work. Recall from Section 3.2 the definition of the pattern matching construct, **mmatch**, which takes a *Coq list* of patterns. If we manage to construct this list *dynamically*, then we can easily obtain an extensible tactic.

And how do we construct the list of patterns dynamically? Using overloading! As argued in Section 2.7, overloading is perfect to add cases in an algorithm, so we can as well use it to add elements in a list—in this case the list of patterns of an Mtactic!

Figure 3.23 shows an extensible version of the search tactic from Section 3.1. It uses overloading for building the cases of the algorithm. For presentation purposes, we decided to use type classes (Sozeau and Oury, 2008), instead of canonical structures, since the former, in this particular case, leads to shorter code, allowing us to concentrate on the relevant aspect of the algorithm we intend to highlight. Further down, and in Section 5.1.2, we discuss the key differences between the two overloading mechanism in Coq. For the moment, the important point to take is that this code is just a proof of concept for extensible Mtactics.

The code starts defining the type of a case, CaseT, which takes the element we are looking for, x, the function f to make recursive calls, and the list s, and returns an element of type Patt, the type for an **mmatch** pattern. Next, in line 5, the class Case is defined, parametrized over a natural number and with only one field, the_case, with an element of CaseT. The natural number will be used as the key to find instances, as will be discussed below in detail.

In line 7 we find the build_cases Mtactic that, as its name suggests, is in charge of building the list of cases. It does so by increasing a number n (starting from 0), at each step constructing an unknown instance (meta-variable) X with type Case n. For each X it triggers the type class resolution mechanism in order to obtain an instantiation for X. If such instance exists, then it appends the case in X to the list of already found cases l and recurses with the next number. If not, it returns the list l constructed so far.

Before explaining why this works, we should explain how canonical structures and type classes difere in two relevant aspects of overloading: (1) the triggering of instance resolution, and (2) the selection of a particular instance. In canonical structures, as we saw in Chapter 2, the resolution process starts at unification, when a projector of an

```
01 Definition CaseT :=
02   ∀ (x : A) (f : ∀ s' : seq A, ○((λ s : seq A. x ∈ s) s')) (s : seq A),
03   Patt (seq A) (λ s' : seq A. x ∈ s') s.
04
05 Class Case (n : nat) := { the_case : CaseT }.
06
07 Definition build_cases : ○(seq CaseT) :=
08   let cases :=
09     mfix f (n : nat) (l : seq CaseT) : ○(seq CaseT) :=
10       X ← evar (Case n); solve_typeclasses;;
11       b ← is_evar X;
12       if b then ret l
13       else f (S n) (@the_case _ X :: l)
14   in cases 0 nil.
15
16 Definition next_no : ○nat := l ← build_cases; retS (length l).
17
18 Instance default_inst : Case 0 :=
19   {| the_case := λ x f s. (| [l] l ⇒ raise NotFound |) |}.
20
21 Instance iterate_inst : Case (run next_no) :=
22   {| the_case := λ x f s.
23         (| [y s'] (y :: s') ⇒ r ← f s'; ret (in_cons y _ _ r) |) |}.
24
25 Instance found_inst : Case (run next_no) :=
26   {| the_case := λ x f s. (| [s'] (x :: s') ⇒ ret (in_eq _ _) |) |}.
27
28 Definition search_ext (x : A) s :=
29   cs ← build_cases;
30   (mfix f (s : seq A) : ○(x ∈ s) :=
31     cs ← retS (map (λ c. c x f s) cs);
32     mmatch (λ s'. x ∈ s') s cs) s.
```

FIGURE 3.23: Extensible search Mtactic.

unknown instance is unified with some particular value. In that case, the instance is selected based on the head constructor of the value. In type classes, instead, the resolution process is triggered *for every* unknown instance, at specific points in the Ocaml code of Coq. For instance, the refiner triggers it after refining a term, some tactics trigger it at certain points when the instances might be needed, etc. The selection of an instance is performed by unifying the type of the unknown instance with the type of the instances in the database.

In our code, we trigger the resolution process of type classes explicitly, by calling the new Mtac constructor **solve_typeclasses**. The call is done right after the creation of the

meta-variable X with type Case n, in the hope that the resolution process will find an instance for it. The instances of the class are created with different, increasing numbers, starting from 0. The result is that, after adding m instances, the Mtactic build_cases will build a list containing cases $m-1,\ldots,0$, in that specific order.

To make things easier with the numbering, we defined the next_no Mtactic in line 16, which simply creates the list of cases and returns its length. This is just to help providing unique numbers to instances, although, of course, it does not solve the known "diamond dependency problem", intrinsic to type classes.

In the code, only three instances are added, in lines 18–26. First, the default instance, added to fail gracefully when the element is not found, then the case for when the element is not in the head of the list, and finally the case for when the element is in the head of the list. Compared to the code in Figure 3.1, these instances corresponds to the last three instances of the search Mtactic, but in reverse order. In the code we use the notation $(\!|p|\!)$ to create pattern p using the notation for patterns provided in Section 3.2. Note the use of **run** next_no in order to dynamically obtain the number for the instance.

Finally, in lines 28–32 we have the extensible Mtactic search_ext. This Mtactic builds the set of cases and creates the **mfix**point iterating and pattern matching the list, similarly to the one in Figure 3.1. But unlike in the non-extensible version, here the **mmatch** is constructed with the list of cases built in line 29. Since the type of the cases depends on the element x, the fixpoint function f, and list s, we need to first apply each case to these values, which we do using the standard map function in line 31.

With this function we can prove, so far, that some element is in a list resulting from cons-ing elements. For instance, the first of the two examples below searches for x_2 in a list resulting form concatenating list $[x_1, x_2]$ with an unknown list s. Since the concatenation function iterates the list on the left, the unification process taking place in the pattern matching of the search_ext Mtactic reduces the list to $(x_1 :: x_2 :: s)$, allowing the Mtactic to make progress and eventually succeed to prove the statement.

Example ex_app_l $(x_1\ x_2 : \mathsf{nat})\ s : x_2 \in ([x_1; x_2] +\!\!+ s) :=$
  run (search_ext _ _).

Fail Example ex_app_r $(x_1\ x_2 : \mathsf{nat})\ s : x_1 \in (s +\!\!+ [x_1; x_2]) :=$
  run (search_ext _ _).

The second example is similar, although the unknown list s is now at the left of the concatenation function. Since we have not added a case for concatenation, and the list

cannot be reduced, the Mtactic fails, as noted with the **Fail** command. In order to make it succeed, we just need to add the case for concatenation:

```
Instance app_inst : Case (run next_no) :=
  {| the_case := λ A x f s. (|
     [ l r ] l ++ r ⇒
       mtry
         il ← f l;
         ret (in_or_app l r x (or_introl il))
       with NotFound ⇒
         ir ← f r;
         ret (in_or_app l r x (or_intror ir))
       end
  |) |}
```

Note that there is nothing special about the search Mtactic; we can as well use the same pattern to build any other extensible Mtactic.

We have answered the question on *how to build* extensible Mtactics, but we have not motivated *the need for* extensible Mtactics. After all, Lemma Overloading is already a method that facilitates the construction of extensible tactics. Why do we need extensible tactics in Mtac? The answer is simple: To build easy-to-use, easy-to-compose extensible *functional* tactics.

We conclude this chapter noting that we have succeeded in building a language for proof automation with all the characteristics enumerated in the introduction of this thesis.

Chapter 4

An Improved Unification Algorithm for Coq

The unification algorithm is at the heart of a proof assistant like Coq. In particular, it is a key component of the *refiner* (the algorithm that has to infer implicit terms and missing type annotations), of certain tactics, and, as we saw in previous chapters, of proof automation.

Despite playing a central role in the proof development process, there is no good source of documentation to understand Coq's unification algorithm. We mentioned earlier (§2.7) two different works on this topic: Sacerdoti Coen (2004, chp. 10) and Saïbi (1997). The first one describes a unification algorithm for CIC, and the second one introduces (briefly) canonical structures. However, neither suffices to really understand the current state of affairs, in particular certain design decisions that shaped the current algorithm in Coq.

Unification is an inherently undecidable problem for CIC, as it must deal with higher-order problems up to conversion. For instance, how should the meta-variable $?f$ be instantiated to satisfy the equation $?f\ 1 \approx 41 + 1$? This problem has infinitely many solutions, as $?f$ can be instantiated with any function that returns number 42 when the input is 1. Many works in the literature (*e.g.*, Abel and Pientka, 2011, Miller, 1991b, Reed, 2009) are devoted to the creation of unification algorithms returning a *Most General Unifier* (MGU), that is, a *unique* solution that serves as a representative for all *convertible* solutions. In the example above, no such MGU exists, as the problem has (infinitely many) non-convertible solutions: *e.g.*, $?f := \lambda x.\ 41 + x$ and $?f := \lambda x.\ 42$.

However, in Coq, restricting the algorithm to return only MGUs is impractical. Consider for instance the search Mtactic from Figure 3.1. Let's assume we are searching for an

element in the list [2]++[1]. We expect the Mtactic to match the concatenation case, searching for the element first in the list on the left and then on the right. For that, the semantics of **mmatch** must unify the term $?l{+}{+}?r$, for fresh meta-variables $?l$ and $?r$, with the list [2]++[1]. This unification should result in assigning $?l := [2]$ and $?r := [1]$. However, there exist other solutions, like assigning the whole list to $?l$, and the empty list to $?r$. If the unification algorithm is restricted to only return MGUs, it will fail in this case, rendering the search Mtactic unusable!

MGUs are not only impractical in Mtac, but also in almost any other process involving unification in Coq. For this reason, Coq's unification algorithm includes heuristics to provide what we call a "natural" unifier, that is, a solution as close as possible to what the proof developer expects (this is not a formal definition, but rather a consequence of years of development).

However, while certain heuristics are essential, there are others that introduce a certain brittleness to the algorithm. An example of this will be discussed in Section 4.3, but for the moment the important point to take away is the tension existing in the design of an unification algorithm: more heuristics means more unification problems are solved, but less predictably.

For this reason, we decided to build a new algorithm from scratch, instead of just performing reverse engineering on the existing algorithm. The process helped us identify the algorithm's relevant design decisions we have to take into account, which we diligently enumerate in this chapter. To the best of our knowledge, this is the first document describing such design decisions.

In the remainder of the chapter we go straight to the rules of the new algorithm (§4.1), explaining each of the design decisions we made. We expect this algorithm to be *sound*, for a specific definition of soundness (§4.2). As mentioned above, our algorithm does not incorporate all of the heuristics that exist in the current algorithm. Probably the most controversial omission is *constraint postponement* (Reed, 2009), which helps disambiguating solutions when multiple ones exist, but at the cost of making the algorithm harder to reason about (§4.3). However, our results compiling the Mathematical Components library (Gonthier et al., 2008) using our algorithm show that this heuristic is not essential after all (§4.4). Still, there are many problems our algorithm cannot solve, but that are easily solvable by incorporating a much simpler heuristic than constraint postponement (§4.5).

4.1 The Algorithm

We proceed to describe our proposal in the following pages, emphasizing every non-standard design decision. The unification judgment is of the form

$$\Sigma; \Gamma \vdash t_1 \approx t_2 \triangleright \Sigma'$$

It unifies terms t_1 and t_2, given a well-formed meta-variable context Σ and a well-formed local context Γ. There is an implicit well-formed global environment E.

We assume that the two terms are well-typed, according to the rules in §1.2.3. That is, there must exist T_1 and T_2 such that

$$\Sigma; \Gamma \vdash t_i : T_i \qquad \text{for } i \in [1, 2]$$

As a result, the algorithm returns a new meta-variable context Σ' with instantiations for the meta-variables appearing in the terms or in the contexts. The algorithm ensures that, upon success, terms t_1 and t_2 are *weakly* well-typed and convertible under the new meta-context. *Weak typing* is similar to typing, but without checks like the *guardedness condition*. We discuss the reason for such weakening in §4.2.

Our algorithm closely relates to the original algorithm of Coq. In particular, it shares the following practical heuristics:

First-order approximation: As mentioned in the introduction of this chapter, when faced with a problem of the form

$$c\ t_1\ \ldots\ t_n \approx c\ u_1\ \ldots\ u_n$$

it decomposes the problem into n subproblems $t_i \approx u_i$, for $0 < i \leq n$. For instance, in order to solve the equation $?l{+}{+}?r \approx [2]{+}{+}[1]$, for fresh meta-variables $?l$ and $?r$, it obtains two unification problems, $?l \approx [2]$ and $?r \approx [1]$.

Backtracking: When first-order approximation fails, in an effort to find a solution to the equation, it reduces the terms. For instance, in the equation $?x :: ?s \approx [2]{+}{+}[1]$, for $?x$ an unknown element and $?s$ an unknown list, it reduces the right-hand side, obtaining the (solvable) equation $?x :: ?s \approx 2 :: (1 :: [])$.

Reduction of the goal first: The unification algorithm is used, for instance, for the application of lemmas. When applying a lemma with conclusion t to a goal u, it results in the equation $t \approx u$, that is, with the goal on the right-hand side.

Typically, we need to reduce the goal to adapt to the lemma being applied than the other way around, and for this reason the algorithm gives priority to reduction on the right-hand side.

Delay unfolding of definitions: The previous heuristic, reduction on the right-hand side first, has an exception. When the right-hand side is a constant or variable definition, before expanding it, the algorithm tries to make progress on the left-hand side, in the hope of encountering the same constant/variable also on the left-hand side. As we are going to see in Section 4.1.2, this heuristic has also desirable consequences in canonical structures resolution.

Canonical structures resolution: As mentioned several times throughout this manuscript (*e.g.*, Section 1.3 and Chapter 2), the unification algorithm is in charge of instantiating unknown instances of structures with canonical values. More precisely, when it encounters a problem of the form $\mathsf{proj}\ ?s \approx c\ t$, with proj being a projector of a structure, $?s$ and unknown instance of that structure, and c some constant, it instantiates $?s$ with the canonical value stored in the database for the pair (proj, c).

As for the differences with the current algorithm in Coq, we can mention the following:

Uniform treatment of applications: When faced with an equation of the form

$$t\ t_1\ \ldots\ t_n \approx u\ u_1\ \ldots\ u_n$$

the original algorithm performs a case analysis on terms t and u, ultimately unifying also each subterm t_i and u_i according to the first-order approximation heuristic. In our algorithm, instead, we first compare t and u using a restricted set of rules, and then proceed uniformly to unify the subterms. Ultimately, both algorithms behave similarly, but ours allows for a cleaner presentation (in paper and in code).

No constraint postponement: As mentioned in the introduction of this chapter, we purposely did not incorporate constraint postponement (Reed, 2009). This heuristic allows the postponement of unification problems with multiple solutions, in the hope that, at a later stage, more information may come to help disambiguate the solutions. The reason to not include this heuristic will be discussed in Section 4.3.

Meta-variable instantiation: As we are going to see, solving a problem of the form $?f\ t_1\ \ldots\ t_n \approx u$ can be quite challenging. In the original algorithm of Coq, the heuristic implemented to solve these sort of equations is quite convoluted, so we have implemented our own, following closely the work by Abel and Pientka (2011).

$$\frac{}{\Sigma; \Gamma \vdash_0 \mathsf{Prop} \approx \mathsf{Prop} \triangleright \Sigma} \textsc{Prop-Same} \qquad \frac{i = j}{\Sigma; \Gamma \vdash_0 \mathsf{Type}(i) \approx \mathsf{Type}(j) \triangleright \Sigma} \textsc{Type-Same}$$

$$\frac{\Sigma_0; \Gamma \vdash T \approx U \triangleright \Sigma_1 \qquad \Sigma_1; \Gamma, x : T \vdash t \approx u \triangleright \Sigma_2}{\Sigma; \Gamma \vdash_0 \lambda x : T.\ t \approx \lambda x : U.\ u \triangleright \Sigma_2} \textsc{Lam-Same}$$

$$\frac{\Sigma_0; \Gamma \vdash T_1 \approx U_1 \triangleright \Sigma_1 \qquad \Sigma_1; \Gamma, x : T_1 \vdash T_2 \approx U_2 \triangleright \Sigma_2}{\Sigma_0; \Gamma \vdash_0 \forall x : T_1.\ T_2 \approx \forall x : U_1.\ U_2 \triangleright \Sigma_2} \textsc{Prod-Same}$$

$$\frac{\begin{array}{c}\Sigma_0; \Gamma \vdash T \approx U \triangleright \Sigma_1 \qquad \Sigma_1; \Gamma \vdash t_2 \approx u_2 \triangleright \Sigma_2 \\ \Sigma_2; \Gamma, x := t_2 \vdash t_1 \approx u_1 \triangleright \Sigma_3\end{array}}{\Sigma_0; \Gamma \vdash_0 \mathbf{let}\ x := t_2 : T\ \mathbf{in}\ t_1 \approx \mathbf{let}\ x := u_2 : U\ \mathbf{in}\ u_1 \triangleright \Sigma_3} \textsc{Let-Same}$$

$$\frac{\Sigma; \Gamma \vdash t_1\{t_2/x\} \approx u_1\{u_2/x\} \triangleright \Sigma'}{\Sigma; \Gamma \vdash_0 \mathbf{let}\ x := t_2 : T\ \mathbf{in}\ t_1 \approx \mathbf{let}\ x := u_2 : U\ \mathbf{in}\ u_1 \triangleright \Sigma'} \textsc{Let-Par}\zeta$$

$$\frac{h \in \mathcal{V} \cup \mathcal{C} \cup \mathcal{I} \cup \mathcal{K}}{\Sigma; \Gamma \vdash_0 h \approx h \triangleright \Sigma} \textsc{Rigid-Same}$$

$$\frac{\Sigma_0; \Gamma \vdash T \approx U \triangleright \Sigma_1 \qquad \Sigma_1; \Gamma \vdash t \approx u \triangleright \Sigma_2 \qquad \Sigma_2; \Gamma \vdash \overline{b} \approx \overline{b'} \triangleright \Sigma_3}{\Sigma_0; \Gamma \vdash_0 \mathbf{match}_T\ t\ \mathbf{with}\ \overline{b}\ \mathbf{end} \approx \mathbf{match}_U\ u\ \mathbf{with}\ \overline{b'}\ \mathbf{end} \triangleright \Sigma_3} \textsc{Case-Same}$$

$$\frac{\Sigma_0; \Gamma \vdash \overline{T} \approx \overline{U} \triangleright \Sigma_1 \qquad \Sigma_1; \Gamma \vdash \overline{t} \approx \overline{u} \triangleright \Sigma_2}{\Sigma_0; \Gamma \vdash_0 \mathbf{fix}_j\ \{\overline{x/n : T := t}\} \approx \mathbf{fix}_j\ \{\overline{x/n : U := u}\} \triangleright \Sigma_2} \textsc{Fix-Same}$$

FIGURE 4.1: Unifying terms sharing the same head constructor.

The unification algorithm is quite involved, so to help readability we split the rules across four different subsections. Roughly, in §4.1.1 we consider the case when the two terms being unified have no arguments, and share the same head constructor; in §4.1.2 we consider terms having arguments; in §4.1.3 we consider meta-variable unification; and in §4.1.4 we consider canonical structures resolution. For each set of rules presented in each subsection, the algorithm tries them in the order presented. However, this implicit ordering is not enough to understand completely the algorithm's strategy, which backtracks in some particular cases. We devote the last subsection (§4.1.5) to explain in detail the algorithm's strategy.

4.1.1 Same Constructor

Figure 4.1 shows the rules that apply when both terms share the same constructor. We need to distinguish this set of rules from the others rules in the algorithm, so we annotate

them with a 0 as subscript of the turnstile ($\vdash_0$). The reasons will become evident when we look at the rules in the next subsection.

The rules Prop-Same and Type-Same unify two sorts, either Prop or $\mathsf{Type}(i)$ for some universe level i. The current implementation supports optionally *sort subtyping*, that is, allowing the universe level in the sort on the left to be less than or equal to the one on the right (*i.e.*, $\mathsf{Type}(i) \approx \mathsf{Type}(j)$ if $i \leq j$). In this work we will ignore sort subtyping and only consider sort equality.

For abstractions (Lam-Same) and products (Prod-Same), we first unify the types of the arguments, and then the body of the binder, with the local context extended with the bound variable. When unifying two **let**s, the rule Let-Same compares first the type of the definitions, then the definitions themselves, and finally the body. In the last case, it augments the local context with the definition on the left (taking the one on the left is somehow arbitrary, but after unification both definitions are convertible, so it does not really matter which one is used). If the definitions fail to unify, then Let-Parζ unfolds both definitions in the respective terms.

Rigid-Same equates the same variable, constant, inductive type, or constructor. The following two rules (Case-Same and Fix-Same) unify **match**es and **fix**points, respectively. In both cases we just unify pointwise every component of the term constructors.

4.1.2 Reduction

The previous subsection considered only the cases when both terms have no arguments and share the same constructor. If that is not the case, as we mentioned several times, the algorithm first tries first-order approximation (rule App-FO in Figure 4.2). This rule, when considering two applications with the same number of arguments (n), compares the head element (t and t', using only the rules in Figure 4.1), and then proceeds to unify each of the arguments. As customary, we denote multiple applications as a *spine* (Cervesato and Pfenning, 2003), using the form $t\ \overline{u_n}$ to represent the term $(\ldots(t\ u_1)\ \ldots\ u_n)$. We call t the *head* of the term.

If the rules in Figure 4.1 plus App-FO fail to apply, then the algorithm tries different reduction strategies. Except in some particular cases, the algorithm first tries reducing the right-hand side (rules ending with R) and then the left-hand side (rules ending with L). Except where noted, every L rule is just the mirror of the corresponding R rule, swapping the positions of the terms being unified. We will often omit the last letter (R or L), and simply write *e.g.*, Meta-δ when referring to both rules.

$$\frac{\Sigma_0; \Gamma \vdash_0 t \approx u \triangleright \Sigma_1 \qquad n \geq 0 \qquad \Sigma_1; \Gamma \vdash \overline{t_n} \approx \overline{u_n} \triangleright \Sigma_2}{\Sigma_0; \Gamma \vdash t\ \overline{t_n} \approx u\ \overline{u_n} \triangleright \Sigma_2}\ \textsc{App-FO}$$

$$\frac{\Sigma; \Gamma \vdash u \overset{\text{w}}{\rightsquigarrow}_{\delta\Sigma} u' \qquad \Sigma; \Gamma \vdash t \approx u' \triangleright \Sigma'}{\Sigma; \Gamma \vdash t \approx u \triangleright \Sigma'}\ \textsc{Meta-}\delta\text{R} \qquad \frac{\Sigma; \Gamma \vdash t \overset{\text{w}}{\rightsquigarrow}_{\delta\Sigma} t' \qquad \Sigma; \Gamma \vdash t' \approx u \triangleright \Sigma'}{\Sigma; \Gamma \vdash t \approx u \triangleright \Sigma'}\ \textsc{Meta-}\delta\text{L}$$

$$\frac{\Sigma; \Gamma \vdash u \overset{\text{w}}{\rightsquigarrow}_{\beta} u' \qquad \Sigma; \Gamma \vdash t \approx u' \triangleright \Sigma'}{\Sigma; \Gamma \vdash t \approx u \triangleright \Sigma'}\ \textsc{Lam-}\beta\text{R} \qquad \frac{t\text{'s head not a } \textbf{let-in} \qquad \Sigma; \Gamma \vdash u \overset{\text{w}}{\rightsquigarrow}_{\zeta} u' \qquad \Sigma; \Gamma \vdash t \approx u' \triangleright \Sigma'}{\Sigma; \Gamma \vdash t \approx u \triangleright \Sigma'}\ \textsc{Let-}\zeta\text{R}$$

$$\frac{t \text{ is } \textbf{fix} \text{ or } \textbf{match} \qquad \Sigma; \Gamma \vdash t \downarrow^{\text{w}}_{\beta\iota\theta} t' \qquad t \neq t' \qquad \Sigma; \Gamma \vdash u \approx t' \triangleright \Sigma'}{\Sigma; \Gamma \vdash u \approx t \triangleright \Sigma'}\ \textsc{Case-}\iota\text{R} \qquad \frac{\Sigma; \Gamma \vdash t \overset{\text{w}}{\rightsquigarrow}_{\beta} t' \qquad \Sigma; \Gamma \vdash t' \approx u \triangleright \Sigma'}{\Sigma; \Gamma \vdash t \approx u \triangleright \Sigma'}\ \textsc{Lam-}\beta\text{L}$$

$$\frac{u\text{'s head not a } \textbf{let-in} \qquad \Sigma; \Gamma \vdash t \overset{\text{w}}{\rightsquigarrow}_{\zeta} t' \qquad \Sigma; \Gamma \vdash t' \approx u \triangleright \Sigma'}{\Sigma; \Gamma \vdash t \approx u \triangleright \Sigma'}\ \textsc{Let-}\zeta\text{L} \qquad \frac{t \text{ is } \textbf{fix} \text{ or } \textbf{match} \qquad \Sigma; \Gamma \vdash t \downarrow^{\text{w}}_{\beta\iota\theta} t' \qquad t \neq t' \qquad \Sigma; \Gamma \vdash t' \approx u \triangleright \Sigma'}{\Sigma; \Gamma \vdash t \approx u \triangleright \Sigma'}\ \textsc{Case-}\iota\text{L}$$

$$\frac{(c := t : A) \in \Gamma \cup E \qquad \text{not } \Sigma; \Gamma \vdash \text{is\_stuck } (c\ \overline{t_n}) \qquad \Sigma; \Gamma \vdash u \approx t\ \overline{t_n} \triangleright \Sigma'}{\Sigma; \Gamma \vdash u \approx c\ \overline{t_n} \triangleright \Sigma'}\ \textsc{Cons-}\delta\textsc{NotStuck}\text{R}$$

$$\frac{(c := t : A) \in \Gamma \cup E \qquad \Sigma; \Gamma \vdash \text{is\_stuck } u \qquad \Sigma; \Gamma \vdash t\ \overline{t_n} \approx u \triangleright \Sigma'}{\Sigma; \Gamma \vdash c\ \overline{t_n} \approx u \triangleright \Sigma'}\ \textsc{Cons-}\delta\textsc{Stuck}\text{L}$$

$$\frac{(c := t : A) \in \Gamma \cup E \qquad \Sigma; \Gamma \vdash u \approx t\ \overline{t_n} \triangleright \Sigma'}{\Sigma; \Gamma \vdash u \approx c\ \overline{t_n} \triangleright \Sigma'}\ \textsc{Cons-}\delta\text{R} \qquad \frac{(c := t : A) \in \Gamma \cup E \qquad \Sigma; \Gamma \vdash t\ \overline{t_n} \approx u \triangleright \Sigma'}{\Sigma; \Gamma \vdash c\ \overline{t_n} \approx u \triangleright \Sigma'}\ \textsc{Cons-}\delta\text{L}$$

$$\frac{\Sigma_0; \Gamma \vdash u : U \qquad \begin{array}{c} u\text{'s head is not an abstraction} \\ \Sigma_0, ?v : \mathsf{Type}(i)[\Gamma, y : T]; \Gamma \vdash U \approx \forall y : T.\ ?v[\mathsf{id}_{\Gamma,y}] \triangleright \Sigma_1 \end{array} \qquad \Sigma_1; \Gamma, x : T \vdash u\ x \approx t \triangleright \Sigma_2}{\Sigma_0; \Gamma \vdash u \approx \lambda x : T.\ t \triangleright \Sigma_2}\ \textsc{Lam-}\eta\text{R}$$

$$\frac{\Sigma_0; \Gamma \vdash u : U \qquad \begin{array}{c} u\text{'s head is not an abstraction} \\ \Sigma_0, ?v : \mathsf{Type}(i)[\Gamma, y : T]; \Gamma \vdash U \approx \forall y : T.\ ?v[\mathsf{id}_{\Gamma,y}] \triangleright \Sigma_1 \end{array} \qquad \Sigma_1; \Gamma, x : T \vdash t \approx u\ x \triangleright \Sigma_2}{\Sigma_0; \Gamma \vdash \lambda x : T.\ t \approx u \triangleright \Sigma_2}\ \textsc{Lam-}\eta\text{L}$$

FIGURE 4.2: Reduction steps attempted during unification.

The first reduction the algorithm tries is reduction of meta-variables, $\delta\Sigma$, as described in the rules Meta-δR and Meta-δL. Actually, as we are going to see in Section 4.1.5, this is what the algorithm tries prior to any other rule, including App-FO and the rules in Figure 4.1.

Next, the algorithm tries β, ζ, and ι reduction, respectively (Lam-β, Let-ζ, and Case-ι). In the first two cases, it performs one-step (weak) head reduction. In the case of **let-in**s, we make sure to avoid overlapping with the Let-Parζ rule.

More interesting are the cases for $\delta\Gamma, \delta E$ and ι reductions. The high level idea is that we need to unfold constants carefully, because they may be used for the resolution of canonical instances. In the case of a **match** or a **fix** (rule Case-ιR), we want to be able to reduce the scrutinee using all reduction rules, including δE-reduction, and then (if applicable), continue reducing the corresponding branch of the **match** or the body of the **fix**, but avoiding the δE-reduction rule.

We illustrate this desired behavior with a simple example. Consider the environment $E = \{\mathsf{d} := 0, \mathsf{c} := \mathsf{d}\}$, where there is also a structure with projector proj. Suppose further that there is a canonical instance i registered for proj and d. Then, the algorithm should succeed finding a solution for the following equation:

$$\textbf{match } \mathsf{c} \textbf{ with } 0 \Rightarrow \mathsf{d} \mid \_ \Rightarrow 1 \textbf{ end} \approx \mathsf{proj}\ ?f \tag{4.1}$$

where $?f$ is an unknown instance of the structure. More precisely, we expect the left-hand side to be reduced as

$$\mathsf{d} \approx \mathsf{proj}\ ?f$$

therefore enabling the use of the canonical instance i to solve for $?f$.

This is done in the rule Case-ιL by weak-head normalizing the left-hand side using the standard $\beta\iota$ rules plus a new reduction rule, θ, that weak-head normalizes scrutinees:

$$\begin{array}{lll} \textbf{match}_T\ t \textbf{ with } \overline{k\ \overline{x} \Rightarrow t'} \textbf{ end} \rightsquigarrow_\theta \textbf{match}_T\ k_j\ \overline{a} \textbf{ with } \overline{k\ \overline{x} \Rightarrow t'} \textbf{ end} & & t \downarrow^{\mathrm{w}}_{\beta\zeta\delta\iota} k_j\ \overline{a} \\ \textbf{fix}_j\ \{F\}\ a_1\ \ldots\ a_{n_j} \rightsquigarrow_\theta \textbf{fix}_j\ \{F\}\ a_1\ \ldots\ a_{n_j-1}\ (k\ \overline{b}) & & a_{n_j} \downarrow^{\mathrm{w}}_{\beta\zeta\delta\iota} k\ \overline{b} \end{array}$$

Note that we really need this new reduction rule: we cannot consider weak-head reducing the term using δE rule, as it will destroy the constant d in the example above, nor restrict reduction of the scrutinee to not include δE, as it will be too restrictive (disallowing δE in the reduction on the left-hand side makes Equation 4.1 not unifiable).

In Equation 4.1 we have a **match** on the left-hand side, and a constant on the right-hand side (the projector). By giving priority to the ι reduction strategy over the δE one

we can be sure that the projector will not get unfolded beforehand, and therefore the canonical instance resolution mechanism will work as expected. Different is the situation when we have constants on both sides of the equation.

For instance, consider the following equation:

$$\mathsf{c} \approx \mathsf{proj}\ ?f \tag{4.2}$$

in the same context as before. Since there is no instance defined for c, we expect the algorithm to unfold it, uncovering the constant d. Then, it should solve the equation, as before, by instantiating $?f$ with i.

If the projector is unfolded first instead, then the algorithm will not find the solution. The reason is that the projector unfolds to a case on the unknown $?f$:

$$\mathsf{c} \approx \textbf{match}\ ?f\ \textbf{with}\ \mathsf{Constr}\ a_1\ \ldots\ a_n \Rightarrow a_j\ \textbf{end}$$

(Assuming the projector proj corresponds to the j-th field in the structure, and Constr is the constructor of the structure.) Now the canonical instance resolution will fail to see that the right-hand side is (was) a projector, so after unfolding c and d on the left, the algorithm will give up and fail.

In this case we cannot just simply rely on the ordering of rules, since that will make the algorithm sensitive to the position of the terms. In order to solve Equation 4.2 above, for instance, we need to prioritize reduction on the left-hand side over the right-hand side, but this prioritization will have a negative impact on similar equations with the projector occurring in the term on the left instead of in the term on the right. The solution is to unfold a constant on the right-hand side *only if the term does not "get stuck"*, that is, does not evaluate to certain values, like an irreducible **match**. More precisely, we define the concept of "being stuck" as

$$\text{is\_stuck}\ t \mathrel{\widehat{=}} \exists t'\ t''.\ t \rightsquigarrow^{0..1}_{\delta E} t' \wedge t' \downarrow^{\mathrm{w}}_{\beta\iota\zeta\theta} t'' \text{ and } t'' \text{ head is a variable, case, fix, or abstraction}$$

that is, after performing an (optional) δE step and $\beta\iota\zeta\theta$-weak head reducing the definition, the head element of the result is tested to be a **match**, **fix**, variable, or a λ-abstraction. Note that the reduction will effectively stop at the first head constant, without unfolding it further. This is important, for instance, when having a definition that reduces to a projector of a structure. If the projector is not exposed, and is instead reduced, then some canonical solution may be lost.

The rule Cons-δNotStuckR unfolds the right-hand side constant only if it will not get stuck. If it is stuck, then the rule Cons-δStuckL triggers and unfolds the left-hand side,

which is precisely what happened in the example above. The rules Cons-δ are triggered as a last resort.

When none of the rules above applies, the algorithm tries η-expansion (Lam-η rules). Note that we have the premise that u's head is not an abstraction to avoid overlapping with the Lam-Same rules, otherwise it is possible to build an infinite loop together with the rules Lam-β. The hypotheses inside the grey boxes ensure that u has product type with T as domain. It is interesting to point out that the current unification algorithm of Coq lacks these hypotheses, which makes the algorithm unsound (*c.f.*, §4.2.2).

4.1.3 Meta-Variable Instantiation

The rules for meta-variable instantiation are considered in Figure 4.3, most of which are closely inspired by Abel and Pientka (2011). There are, however, several differences between their work and ours, since we have a different base logic (CIC instead of LF), and a different assumption on the types of the terms: they require the terms being unified to have the same (known) type, while we do not (types play no role in our unification judgment).

For presentation purposes, we only present the rules having a meta-variable on the right-hand side of the equation, but the algorithm also includes the rules with the terms swapped.

Same Meta-Variable: If both terms are the same meta-variable $?x$, we have two distinct cases: if their substitution is exactly the same, the rule Meta-Same-Same applies, in which the arguments of the meta-variable are compared point-wise. Note that we require the elements in the substitution to be the same, and not just convertible. If, instead, their substitution is different, then the rule Meta-Same is attempted. To better understand this rule, let's look at an example. Say $?z$ has type $T[x_1 : \mathsf{nat}, x_2 : \mathsf{nat}]$ and we have to solve the equation

$$?z[y_1, y_2] \approx ?z[y_1, y_3]$$

where y_1, y_2 and y_3 are defined in the local context. From this equation we cannot know yet what value $?z$ will hold, but at least we know it cannot refer to the second parameter, x_2, since that will render the equation above false. This reasoning is reflected in the rule Meta-Same in the hypothesis

$$\Psi_1 \vdash \sigma \cap \sigma' \triangleright \Psi_2$$

$$\frac{\Sigma; \Gamma \vdash \overline{t} \approx \overline{u} \triangleright \Sigma'}{\Sigma; \Gamma \vdash ?x[\sigma]\ \overline{t} \approx ?x[\sigma]\ \overline{u} \triangleright \Sigma'}\ \textsc{Meta-Same-Same}$$

$$\frac{\begin{array}{c}?x : T[\Psi_1] \in \Sigma \qquad \Psi_1 \vdash \sigma \cap \sigma' \triangleright \Psi_2 \qquad \Sigma \vdash \Psi_2 \qquad \mathsf{FV}(T) \subseteq \Psi_2 \\ \Sigma \cup \{?y : T[\Psi_2], ?x := ?y[\mathsf{id}_{\Psi_2}]\}; \Gamma \vdash \overline{t} \approx \overline{u} \triangleright \Sigma'\end{array}}{\Sigma; \Gamma \vdash ?x[\sigma]\ \overline{t} \approx ?x[\sigma']\ \overline{u} \triangleright \Sigma'}\ \textsc{Meta-Same}$$

$$\frac{\begin{array}{c}?x : T[\Psi] \in \Sigma_0 \qquad t', \xi_1 = \mathsf{remove\_tail}(t; \xi') \qquad t' \downarrow_\beta^{\mathrm{w}} t'' \\ \Sigma_0 \vdash \mathsf{prune}(\xi, \xi_1; t'') \triangleright \Sigma_1 \\ \Sigma_1; \Gamma \vdash \xi_1 : \overline{U} \qquad t''' = \lambda y : U\{\xi, \xi_1/\hat{\Psi}, \overline{y}\}^{-1}.\ \Sigma_1(t'')\{\xi, \xi_1/\hat{\Psi}, \overline{y}\}^{-1} \\ \Sigma_1; \Psi \vdash t''' : T' \qquad \Sigma_1; \Psi \vdash T' \approx T \triangleright \Sigma_2 \qquad ?x \notin \mathsf{FMV}(t''')\end{array}}{\Sigma_0; \Gamma \vdash t \approx ?x[\xi]\ \xi' \triangleright \Sigma_2 \cup \{?x := t'''\}}\ \textsc{Meta-InstR}$$

$$\frac{\begin{array}{c}?x : T[\Psi] \in \Sigma_0 \qquad 0 < n \qquad \Sigma_0; \Gamma \vdash u \approx ?x[\sigma] \triangleright \Sigma_1 \\ \Sigma_1; \Gamma \vdash \overline{u_n} \approx \overline{t_n} \triangleright \Sigma_2\end{array}}{\Sigma_0; \Gamma \vdash u\ \overline{u_n} \approx ?x[\sigma]\ \overline{t_n} \triangleright \Sigma_2}\ \textsc{Meta-FOR}$$

$$\frac{?u : T[\Psi] \in \Sigma_0 \qquad t \overset{\mathrm{w}}{\rightsquigarrow}_\delta^{0..1} t' \qquad t' \downarrow_{\beta\iota\theta}^{\mathrm{w}} t'' \qquad \Sigma_0; \Gamma \vdash t'' \approx ?u[\sigma]\ \overline{t_n} \triangleright \Sigma_1}{\Sigma_0; \Gamma \vdash t \approx ?u[\sigma]\ \overline{t_n} \triangleright \Sigma_1}\ \textsc{Meta-ReduceR}$$

FIGURE 4.3: Meta-variable instantiation.

$$\frac{}{\cdot \vdash \cdot \cap \cdot \triangleright \cdot} \qquad \frac{\Psi \vdash \sigma \cap \sigma' \triangleright \Psi'}{\Psi, x : A \vdash \sigma, t \cap \sigma', t \triangleright \Psi', x : A}$$

$$\frac{\Psi \vdash \sigma \cap \sigma' \triangleright \Psi'}{\Psi, x := u : U \vdash \sigma, t \cap \sigma', t \triangleright \Psi', x := u : U} \qquad \frac{\Psi \vdash \sigma \cap \sigma' \triangleright \Psi' \qquad y \neq z}{\Psi, x : T \vdash \sigma, y \cap \sigma', z \triangleright \Psi'}$$

$$\frac{\Psi \vdash \sigma \cap \sigma' \triangleright \Psi' \qquad y \neq z}{\Psi, x := u : U \vdash \sigma, y \cap \sigma', z \triangleright \Psi'}$$

FIGURE 4.4: Intersection of substitutions.

This judgment performs an intersection of both substitutions, filtering out those positions from the context of the meta-variable Ψ_1 where the substitutions disagree, resulting in Ψ_2.

The intersection judgment is defined in Figure 4.4. The definition is conservative: it only filters different *variables*. The judgment is undefined if the two substitutions have different *terms* (not variables) in some position. Of course, a more aggressive approach is possible, checking for convertibility of the terms instead of just syntactic equality, but it is not clear whether the few more cases covered compensates for the potential loss in

performance.

Coming back to the rule META-SAME, by filtering out the disagreeing positions of the substitution, we obtain a new context Ψ_2, which is a subset of Ψ_1. Then, we restrict $?x$ to only refer to the variables in Ψ_2. We do this by creating a new meta-variable $?y$ with the type of $?x$, but in the context Ψ_2. We further instantiate $?x$ with $?y$. Both the creation of $?y$ and the instantiation of $?x$ in the context Σ is expressed in the fragment $\Sigma \cup \{?y : T[\Psi_2], ?x := ?y[\mathsf{id}_{\Psi_2}]\}$ of the last hypothesis. We use this new context to compare point-wise the arguments of the meta-variable.

There are two other hypotheses that ensure that nothing goes wrong. Again, we explain them by means of example. The hypothesis

$$\mathsf{FV}(T) \subseteq \Psi_2$$

ensures that the type T of $?x$ (and therefore, of $?y$), is well formed in the new (shorter) context Ψ_2. This condition is enclosed in a grey box because it's optional: if the terms on both sides of the equation are assumed to have the same type, it can be omitted. In Abel and Pientka (2011), for instance, the condition is not present, because in their setting this assumption holds. But if the terms are allowed to have different types, then it's possible for the intersection judgment to return an ill-formed context. To illustrate, consider the equation

$$?f[y] \approx ?f[z]$$

under contexts

$$\Sigma = \{?f : x[x : \mathsf{Type}]\} \qquad \Gamma = \{y : \mathsf{Type}, z : \mathsf{Type}\}$$

The intersection of both substitutions will return an empty context. But we cannot create a new meta-variable $?f'$ with type x in the empty context! The problem comes from the fact that the terms have different types (y and z respectively). By ensuring that every free variable in the type of the meta-variable is in the context Ψ_2 we prevent this problem.

More subtle is the premise ensuring the well-formedness of Ψ_2:

$$\Sigma \vdash \Psi_2$$

The intersection judgment may return an invalid context; a context where some type refers to a variable that was wiped out from the context. This can only happen if the two substitutions agree on a term whose type depends on a variable where the two substitutions disagree. This sounds odd, but convertibility makes it possible.

For example, consider contexts

$$\Sigma = \{?v : \mathsf{Prop}[x : \mathsf{Type}, p : \mathsf{fst}(\mathsf{Prop}, x)]\} \qquad \Gamma = \{y : \mathsf{Type}, z : \mathsf{Type}, w : \mathsf{Prop}\}$$

and the equation

$$?v[y, w] \approx ?v[z, w]$$

Both substitutions agree on the value for p, w, which has type Prop. But the type of p, convertible to Prop, depends also on x, a type variable where both substitutions disagree (with values y and z respectively). After performing the intersection, we get the ill-formed context $[p : \mathsf{fst}(\mathsf{Prop}, x)]$.

A more sophisticated intersection judgment may detect a situation like this and reduce the type to avoid the dependency.

Meta-Variable Instantiation: The META-INST rules instantiate a meta-variable applying a variation of *higher-order pattern unification* (HOPU) (Miller, 1991b). They unify a meta-variable $?x$ with some term t. As required by HOPU, the meta-variable is applied to a suspended substitution mapping variables to variables, ξ, and a spine of arguments ξ', of only variables. Assuming $?x$ has (contextual) type $T[\Psi]$, this rule must find a term t''' to instantiate $?x$ such that, after performing the suspended substitution ξ and applying arguments ξ' (formally, $t'''\{\xi/\hat{\Psi}\}\ \xi'$), results in a term convertible to t.

However, we do not want *any* solution; we want the *most natural* one, often meaning also the *most general* one. We show with examples what we mean.

Example 4.1 (Equation without a most general solution)**.** *This example shows a case where no most general unifier exists. In this case, we expect the rule to fail to instantiate the meta-variable.*

Consider the equation

$$?f[x]\ x \approx \mathsf{addn}\ x\ 0 \tag{4.3}$$

where $?f$ *is an uninstantiated meta-variable with type* $\mathsf{nat} \to \mathsf{nat}[y : \mathsf{nat}]$*, and* addn *is the addition operator for natural numbers. This equation has the following* incomparable *solutions:*

1. $?f := \lambda z.\ \mathsf{addn}\ z\ 0$
2. $?f := \lambda \_.\ \mathsf{addn}\ y\ 0$

(Note that the solution should have all its free variables included in the local context of the type of $?f$, $\{y\}$. Unfolding each solution of $?f$ in Equation 4.3, after applying the suspended substitution $[x]$, we obtain terms convertible to $\mathsf{addn}\ x\ 0$*.)*

The problem with the previous example is that x occurs twice, once in the substitution and once in the arguments of the unification variable. However this is not always a problem, as the following example illustrates.

Example 4.2 (Equation with the same variable occurring twice, but having a most general solution)**.** *Consider $?f$ now with type* $\mathsf{nat} \to \mathsf{nat}[y_1 : \mathsf{nat}, y_2 : \mathsf{nat}]$ *in the following equation:*

$$?f[x, z]\ x \approx \mathsf{addn}\ z\ 0 \tag{4.4}$$

This equation has the following most general solution:

$$?f := \lambda\_.\ \mathsf{addn}\ y_2\ 0$$

Since x does not occur in the term, it does not matter that it occurs twice in the arguments of the meta-variable.

In theory, a solution and its η-expansion are comparable. However, from a practical perspective, an η-expanded term is *less natural*, and it potentially *hides* the head constant from the canonical structures resolution algorithm. Consider the following example:

Example 4.3 (Equation whose solutions are η-convertible)**.** *Consider the equation*

$$?f[]\ x\ y \approx \mathsf{addn}\ x\ y \tag{4.5}$$

This equation has the following solutions convertible up to η-expansion:

1. $?f := \mathsf{addn}$
2. $?f := \lambda z.\ \mathsf{addn}\ z$
3. $?f := \lambda z.\ \lambda w.\ \mathsf{addn}\ z\ w$

However, the last two solutions in this example have two problems. First, they are not intuitive. The proof developer will find awkward that $?f$ was not assigned the more compact and simple term. Second, and more importantly, a solution may be lost if $?f$ is used in another equation later on. For instance, consider the following equation:

$$\mathsf{proj}\ ?s \approx ?f$$

where $?s$ is an unknown instance of a structure with projector proj, and $?f$ has been instantiated with the second or third solution from Equation 4.5. Then, if there is a canonical instance with key $(\mathsf{proj}, \mathsf{addn})$, it will not be taken into account, since the λ-abstractions are "hiding" the constant addn.

In the previous example there was a direct correspondance between the arguments of the meta-variable and the addition function. This will not always be the case, but we still make an effort to obtain the shortest possible term. The following example illustrates this point.

Example 4.4 (Equation with most general solution, with only one out of two arguments in common)**.** *Consider equation:*

$$?f[z]\ x\ y \approx \mathsf{addn}\ z\ y$$

This equation has the following η-convertible solutions:

1. $?f := \lambda x.\ \mathsf{addn}\ z$
2. $?f := \lambda x\ y.\ \mathsf{addn}\ z\ y$

Our algorithm will favor the first solution.

With these examples in mind, we come back to the Meta-Inst rules, more specifically, the Meta-InstR rule. This rule should find a solution for the equation

$$t \approx ?x[\xi]\ \xi'$$

under contexts Σ_0 and Γ. t is crafted into a new term t''' following these steps:

1. To avoid η-expanded solutions, the term t and arguments ξ' are decomposed using the function $\mathsf{remove\_tail}(\cdot;\cdot)$:

$$\begin{aligned} \mathsf{remove\_tail}(t\ x; \xi, x) &\mathrel{\widehat{=}} \mathsf{remove\_tail}(t; \xi) && \text{if } x \notin \mathsf{FV}(t) \\ \mathsf{remove\_tail}(t; \xi) &\mathrel{\widehat{=}} (t, \xi) && \text{in any other case} \end{aligned}$$

   This function, applied to t and ξ', returns a new term t' and a list of variables ξ_1, where there exists ξ_2 such that $t = t'\ \xi_2$ and $\xi' = \xi_1, \xi_2$, and ξ_2 is the longest such list. For instance, in Example 4.3, ξ_1 will be empty, and ξ_2 will be x, y, while in Example 4.4, ξ_1 will be x and ξ_2 will be y.

The check that $x \notin \mathsf{FV}(t)$ in the first case above ensures that no solutions are erroneously discarded. Consider the following equation:

$$?f[]\ x \approx \mathsf{addn0}\ x\ x$$

If we remove the argument of the meta-variable, we will end up with the unsolvable equation:

$$?f[] \approx \mathsf{addn0}\ x$$

2. The term obtained in the previous step is weak head β normalized, noted $t' \downarrow_{\beta}^{\mathrm{w}} t''$. This is performed in order to remove false dependencies, like variable x in $(\lambda\_.\ 0)\ x$.

3. The meta-variables in t'' are *pruned*. This process is quite involved, so we defer the explanation for next subsection (§4.1.3.1). At high level, the pruning judgment ensures that the term t'' has no "offending variables", that is, free variables outside of those occurring in the substitution ξ, ξ_1. It does so by restricting meta-variables occurring in t'' whose suspended substitutions have such offending variables. The output of this judgment is a new meta-context Σ_1.

4. The final term t''' is constructed as

$$\overline{\lambda y : U\{\xi, \xi_1/\hat{\Psi}, \overline{y}\}^{-1}}.\ \Sigma_1(t'')\{\xi, \xi_1/\hat{\Psi}, \overline{y}\}^{-1}$$

First, note from Example 4.4 that t''' has to be a function taking n arguments $\overline{y}$, where $n = |\xi_1|$. For the moment, let's forget about the types of each y_j.

The body of this function is the term obtained from the second step, t'', after performing a few changes. First, all of its defined meta-variables are normalized with respect to the meta-context obtained in the previous step, Σ_1, in order to replace the meta-variables with the pruned ones. This step effectively removes false dependencies on variables not occurring in ξ, ξ_1.

Then, the *inversion* of substitution $\xi, \xi_1/\hat{\Psi}, \overline{y}$ is performed. This inversion ensures that all free variables in $\Sigma_1(t'')$ are replaced by variables in Ψ and $\overline{y}$. More precisely, it replaces every variable in $\Sigma_1(t'')$ appearing *only once* in the image of the substitution (ξ, ξ_1) by the corresponding variable in the domain of the substitution $(\hat{\Psi}, \overline{y})$. If a variable appears multiple times in the image and occur in term t'', then inversion fails.

The type of each argument y_j of the function is the type U_j, obtained from the j-th element in ξ_1, after performing the inversion substitution (with the caveat that the substitution includes only the $j - 1$ elements in $\overline{y}$).

5. The type of t''', which now only depends on the context Ψ, is computed as T'. This introduces a penalty in the performance of the algorithm, but since t''' is well typed (since t is assumed to be, and every step above preserves its type), then we can perform a fast *re-typing*. Re-typing assumes terms are well-typed, therefore omitting to typecheck arguments of functions: for example, to compute the type of the term $t_1\ t_2$, it suffices to compute the type of t_1, say $\forall x : A.\ B$, and substitute t_2 for x in B. There is no need to compute the type of t_2.

6. The type obtained in the previous step, T', is unified with the type of $?x$, obtaining a new meta-context Σ_2. These last two steps are shown inside a grey box in the rules, in order to indicate that they can be omitted if the types of the terms are unified prior to the term themselves.

7. Finally, an *occurs check* is performed to prevent illegal solutions, making sure $?x$ does not occur in t'''.

The algorithm outputs the new context Σ_2, instantiating $?x$ with t'''.

First-Order Approximation: The meta-variable in the rules Meta-Inst is restricted to have only variables in the spine of arguments. This can be quite restrictive, in particular when meta-programming. Consider for instance the following equation that tries to unify an unknown function, applied to an unknown argument, with the term 1 (expanded to $\mathsf{S}\ 0$):

$$\mathsf{S}\ 0 \approx ?f[]\ ?y[]$$

As usual, such equation have multiple solutions, but one that is "more natural": assign S to $?f$ and 0 to $?y$. However, since the argument to the meta-variable is not a variable, it does not fall into the higher-order pattern fragment, and therefore is not considered by the Meta-Inst rules. In an scenario like this, the Meta-FO rules perform a *first order approximation*, unifying the meta-variable ($?f$ in the equation above) with the term on the left-hand side without the last n arguments (S), which are in turn unified pointwise with the n arguments in the spine of the meta-variable (0 and $?y$, respectively). Note that the rule App-FO does not subsume this rule, as it requires both terms being equated to have the same number of arguments.

Eliminating Dependencies: If none of the rules above apply, then the algorithm makes a last attempt reducing the term, as this may remove dependencies (Meta-Reduce). For instance, the following example has a solution, even when the term refers

$$\frac{h \in \mathcal{S} \cup \mathcal{C}}{\Sigma \vdash \mathsf{prune}(\xi; h) \triangleright \Sigma} \textsc{Prune-Constant} \qquad \frac{x \in \xi}{\Sigma \vdash \mathsf{prune}(\xi; x) \triangleright \Sigma} \textsc{Prune-Var}$$

$$\frac{\Sigma \vdash \mathsf{prune}(\xi, x; t) \triangleright \Sigma'}{\Sigma \vdash \mathsf{prune}(\xi; \lambda x.\ t) \triangleright \Sigma'} \textsc{Prune-Lam} \qquad \frac{\Sigma \vdash \mathsf{prune}(\xi, x; t) \triangleright \Sigma'}{\Sigma \vdash \mathsf{prune}(\xi; \forall x.\ t) \triangleright \Sigma'} \textsc{Prune-Prod}$$

$$\frac{\Sigma_0 \vdash \mathsf{prune}(\xi; t) \triangleright \Sigma_1 \qquad \Sigma_i \vdash \mathsf{prune}(\xi; t_i) \triangleright \Sigma_{i+1} \qquad i \in [1, n]}{\Sigma_0 \vdash \mathsf{prune}(\xi; t\ \overline{t_n}) \triangleright \Sigma_{i+1}} \textsc{Prune-App}$$

$$\frac{\Sigma_1 \vdash \mathsf{prune}(\xi; t_2) \triangleright \Sigma_2 \qquad \Sigma_2 \vdash \mathsf{prune}(\xi, x; t_1) \triangleright \Sigma_3}{\Sigma_1 \vdash \mathsf{prune}(\xi; \textbf{let}\ x := t_2\ \textbf{in}\ t_1) \triangleright \Sigma_3} \textsc{Prune-Let}$$

$$\frac{?x : T[\Psi] \in \Sigma \qquad \Psi \vdash \mathsf{prune\_ctx}(\xi; \sigma) \triangleright \Psi}{\Sigma \vdash \mathsf{prune}(\xi; ?x[\sigma]) \triangleright \Sigma} \textsc{Prune-Meta-NoPrune}$$

$$\frac{?x : T[\Psi] \in \Sigma \qquad \Psi \vdash \mathsf{prune\_ctx}(\xi; \sigma) \triangleright \Psi' \qquad \Sigma \vdash \Psi' \qquad \Sigma \vdash \mathsf{prune}(\mathsf{id}_{\Psi'}; T) \triangleright \Sigma'}{\Sigma \vdash \mathsf{prune}(\xi; ?x[\sigma]) \triangleright \Sigma', ?y : \Sigma'(T)[\Psi'] \cup \{?x := ?y[\mathsf{id}_{\Psi'}]\}} \textsc{Prune-Meta}$$

Figure 4.5: Pruning of meta-variables.

$$\frac{}{\cdot \vdash \mathsf{prune\_ctx}(\xi; \cdot) \triangleright \cdot} \textsc{PruneCtx-Nil}$$

$$\frac{\mathsf{FV}(t) \in \xi \qquad \Psi \vdash \mathsf{prune\_ctx}(\xi; \sigma) \triangleright \Psi'}{\Psi, x : A \vdash \mathsf{prune\_ctx}(\xi; \sigma, t) \triangleright \Psi', x : A} \textsc{PruneCtx-NoPrune}$$

$$\frac{y \notin \xi \qquad \Psi \vdash \mathsf{prune\_ctx}(\xi; \sigma) \triangleright \Psi'}{\Psi, x : A \vdash \mathsf{prune\_ctx}(\xi; \sigma, y\ \overline{t_n}) \triangleright \Psi'} \textsc{PruneCtx-Prune}$$

Figure 4.6: Pruning of contexts.

to a variable that is not in the context of the meta-variable:

$$\mathsf{fst}\ (0, x) \approx ?g[]$$

4.1.3.1 Pruning

The idea behind pruning can be understood with an example. Say we want to unify terms

$$?w[x, y] \approx c\ ?u[z, ?v[y]] \tag{4.6}$$

A solution exists, although z is a free variable in the right-hand side not appearing in the image of the substitution of the left-hand side. The solution has to restrict $?u$ to ensure that it does not depend on the first element of the substitution. This can be done by meta-substituting $?u$ with a new $?u'$ having a smaller context, as we did with the intersection of substitutions in the rule META-SAME. That is, if $?u : U[z_1 : T_1, z_2 : T_2]$, then a fresh meta-variable $?u'$ is created with type $U[z_2 : T_2]$, and $?u := ?u'[z_2]$. The result of this process in Equation 4.6 is

$$?w[x, y] \approx c\ ?u'[?v[y]]$$

which can now be easily solved.

But not every offending variable occurring inside a substitution can be pruned. Consider a similar example as Equation 4.6, where the offending variable z now occurs inside the substitution of $?v$:

$$?w[x, y] \approx c\ ?u[x, ?v[z]]$$

z cannot be pruned, since a solution may exists by pruning z from $?v$, or by pruning the second argument ($?v[z]$) from $?u$.

There is an extra consideration to take into account. The offending variable can only occur at the head of the term in the suspended substitution, otherwise the pruning may lose solutions, and fail in unexpected places. The following example illustrates this point.

Example 4.5 (Bad pruning)**.**

$$\textsf{let}\ p := (x, y)\ \textsf{in}\ (?u[x], ?v[p]) \approx \textsf{let}\ p := (x, y)\ \textsf{in}\ (c\ ?v[(x, y)], \textsf{fst}\ p)$$

After unifying the definition of the **let**, it introduces the definition $p := (x, y)$ in the local context and proceeds to pairwise unify the components of the pair, obtaining the following equations:

$$?u[x] \approx c\ ?v[(x, y)]$$
$$?v[p] \approx \textsf{fst}\ p$$

When considering the first equation, it can be tempting to prune the argument of $?v$, as it has the offending variable y, thus instantiating $?v$ with $?v'[]$, where $?v'$ is a fresh meta-variable having an empty context. However, such pruning renders the second equation unsolvable. After unfolding the definition of $?v$, we obtain the equation

$$?v'[] \approx \textsf{fst}\ p$$

which has no solution. In this example it is easy to see where things went wrong, but in general it is a bad idea to fail at the wrong place, as it forces the developer to trace the algorithm to find out where the problem was originated. The original algorithm of Coq performs such ill-prunings, and in this example incorrectly flags the problem as coming from the second equation.

We can now get into the formal description of the pruning process (Figure 4.5). The pruning judgment is noted

$$\Sigma \vdash \mathsf{prune}(\xi; t) \triangleright \Sigma'$$

It takes a meta-context Σ, a list of variables ξ, the term to be pruned t, and returns a new meta-context Σ', which is an extension of Σ where all the meta-variables with offending variables in their suspended substitution are instantiated with pruned ones.

For brevity, we only show rules for the Calculus of Constructions fragment of CIC, *i.e.*, without considering pattern matching and fixpoints. The missing rules are easy to extrapolate from the given ones. The only interesting case is when the term t is a meta-variable $?x$ applied to the suspended substitution σ. We have two possibilities: either every variable from every term in σ is included in ξ, in which case we do not need to prune (Prune-Meta-NoPrune), or there exists some terms which have to be removed (pruned) from σ (Prune-Meta).

These two rules use an auxiliary judgment to prune the local context of the meta-variable Ψ. This judgment has the form

$$\Psi \vdash \mathsf{prune\_ctx}(\xi; \sigma) \triangleright \Psi'$$

and its rules are described in Figure 4.6. Basically, it filters out every variable in Ψ where σ has an *offending term*, that is, a term whose head variable is not in ξ. Ψ' is the result of this process.

Coming back to the rules in Figure 4.5, in Prune-Meta-NoPrune we have the condition that the pruning of context Ψ resulted in the same context (no need for a change). More interestingly, when the pruning of Ψ results in a new context Ψ', Prune-Meta does the actual pruning of $?x$. At high level, it creates a new meta-variable $?y$ with type T, the type of $?x$, but with local context Ψ', and then it instantiates $?x$ with $?y$ applied to the identity substitution of Ψ'. Similar to Meta-Same, we have to make sure that the new context is well-typed (condition $\Sigma \vdash \Psi'$), since the pruning of the local context may return an ill-typed context.

There is a little burden required to ensure that the type T respects the new context Ψ'. This is reflected in the pruning of T with respect to the variables of Ψ'. Note that

$$\frac{\begin{array}{c}(p_j, c, \iota) \in \Delta_{\text{db}} \qquad \iota := \lambda\overline{x : T}.\ k\ \overline{a'}\ \overline{v} \qquad v_j = c\ \overline{u'} \\ \Sigma_1 = \Sigma_0, \overline{?y : T} \qquad \Sigma_1; \Gamma \vdash \overline{a} \approx \overline{a'\{\overline{?y}/\overline{x}\}} \triangleright \Sigma_2 \\ \Sigma_2; \Gamma \vdash \overline{u} \approx \overline{u'\{\overline{?y}/\overline{x}\}} \triangleright \Sigma_3 \qquad \Sigma_3; \Gamma \vdash i \approx \iota\ \overline{?y} \triangleright \Sigma_4 \qquad \Sigma_4; \Gamma \vdash \overline{t} \approx \overline{t'} \triangleright \Sigma_5\end{array}}{\Sigma_0; \Gamma \vdash c\ \overline{u}\ \overline{t'} \approx p_j\ \overline{a}\ i\ \overline{t} \triangleright \Sigma_5}\ \text{CS-ConstR}$$

$$\frac{\begin{array}{c}(p_j, \to, \iota) \in \Delta_{\text{db}} \qquad \iota := \lambda\overline{x : T}.\ k\ \overline{a'}\ \overline{v} \qquad v_j = u \to u' \\ \Sigma_1 = \Sigma_0, \overline{?y : T} \qquad \Sigma_1; \Gamma \vdash \overline{a} \approx \overline{a'\{\overline{?y}/\overline{x}\}} \triangleright \Sigma_2 \qquad \Sigma_2; \Gamma \vdash t \approx u\{\overline{?y}/\overline{x}\} \triangleright \Sigma_3 \\ \Sigma_3; \Gamma \vdash t' \approx u'\{\overline{?y}/\overline{x}\} \triangleright \Sigma_4 \qquad \Sigma_4; \Gamma \vdash i \approx \iota\ \overline{?y} \triangleright \Sigma_5\end{array}}{\Sigma_0; \Gamma \vdash t \to t' \approx p_j\ \overline{a}\ i \triangleright \Sigma_5}\ \text{CS-ProdR}$$

$$\frac{\begin{array}{c}(p_j, s, \iota) \in \Delta_{\text{db}} \qquad \iota := \lambda\overline{x : T}.\ k\ \overline{a'}\ \overline{v} \qquad \Sigma_1 = \Sigma_0, \overline{?y : T} \\ \Sigma_1; \Gamma \vdash \overline{a} \approx \overline{a'\{\overline{?y}/\overline{x}\}} \triangleright \Sigma_2 \qquad \Sigma_2; \Gamma \vdash i \approx \iota\ \overline{?y} \triangleright \Sigma_3\end{array}}{\Sigma_0; \Gamma \vdash s \approx p_j\ \overline{a}\ i \triangleright \Sigma_3}\ \text{CS-SortR}$$

$$\frac{\begin{array}{c}(p_j, \_, \iota) \in \Delta_{\text{db}} \qquad \iota := \lambda\overline{x : T}.\ k\ \overline{a'}\ \overline{v} \qquad v_j = x_l \\ \Sigma_1 = \Sigma_0, \overline{?y : T} \qquad \Sigma_1; \Gamma \vdash \overline{a} \approx \overline{a'\{\overline{?y}/\overline{x}\}} \triangleright \Sigma_2 \\ \Sigma_2; \Gamma \vdash x_l\{\overline{?y}/\overline{x}\} \approx t \triangleright \Sigma_3 \qquad \Sigma_3; \Gamma \vdash i \approx \iota\ \overline{?y} \triangleright \Sigma_4\end{array}}{\Sigma_0; \Gamma \vdash t \approx p_j\ \overline{a}\ i \triangleright \Sigma_3}\ \text{CS-DefaultR}$$

FIGURE 4.7: Canonical structures resolution.

pruning not only prunes meta-variables, but also ensures that the term does not contain offending variables (see rule PRUNE-VAR). The pruning of T results in the new meta-context Σ'. The result of the whole process is Σ' extended with the fresh meta-variable $?y$, and with $?x$ instantiated with this new meta-variable.

4.1.4 Canonical Structures Resolution

Canonical structures resolution is listed in Figure 4.7. The rules are rather involved, but they follow the informal description already given in §2.3.2. Assume an equation of the form

$$t'' \approx p_j\ \overline{a}\ i\ \overline{t}$$

where p_j is a projector of a structure, $\overline{a}$ are the arguments of the structure, i is the instance (usually a meta-variable), and $\overline{t}$ are the arguments of the projected value, in the case when it has product type. In order to solve this equation the algorithm proceeds as follows:

1. First, the instance ι is selected from the Canonical Instances Database (Δ_{db}), keying on the projector p_j and the head element h of t''. According to the class of h, the algorithm considers different rules:

   (a) CS-Const if h is a constant c.

   (b) CS-Prod if h is a non-dependent product $t \to t'$.

   (c) CS-Sort if h is a sort s.

   If these do not apply, then it tries CS-Default.

   The instance ι stored in the database is a function taking arguments $\overline{x : T}$ and returning the term $k\ \overline{a'}\ \overline{v}$, with k the constructor of the structure, $\overline{a'}$ the arguments of the structure, and $\overline{v}$ the values for each of the fields of the structure.

2. Then, the arguments of the expected structure $\overline{a}$ are unified with the arguments in the instance $\overline{a'}$, after replacing every argument x with a corresponding fresh unification variable $?y$.

3. Next, for CS-Const and CS-Prod only, the subterms in t'' are unified with those in the value of the instance for the j-th field. If t'' is a constant c applied to arguments $\overline{u}$, and the value v_j of the j-th field of ι is c applied to $\overline{u'}$, then $\overline{u}$ and $\overline{u'}$ are unified. If t'' is a product with premise t and conclusion t', they are unified with the corresponding terms (u and u') in v_j.

4. The instance of the structure i is unified with the instance found in the database, ι, applied to the meta-variables $\overline{y}$. Typically, i is a meta-variable, and this step results in instantiating the meta-variable with the constructed instance.

5. Finally, for CS-Const only, if the j-th field of the structure has product type, and is applied to $\overline{t'}$ arguments, then these arguments are unified with the arguments $\overline{t}$ of the projector.

As with the rules for meta-variable instantiation, we only show the rules in one direction, with the projector on the right-hand side, but the algorithm also includes the rules in the opposite direction.

4.1.5 Rule Priorities and Backtracking

The figures shown above does not precisely nail the priority of the rules, nor when the algorithm backtracks. Below we show the precise order of application of the rules, where the rules in the same line are tried in the given order *without* backtracking (the first matching conclusion is used). Rules in different lines (when overlapping) are tried *with* backtracking (if one fails to apply, the next is tried).

1. If a term has a *defined* meta-variable in its head position:

   (a) Meta-δR, Meta-δL

2. If both terms has *the same undefined* meta-variable in its head position:

   (a) Meta-Same-Same, Meta-Same

3. If one term has an *undefined* meta-variable, and the other term does not have the same meta-variable in its head position:

   (a) Meta-InstR

   (b) Meta-FOR

   (c) Meta-ReduceR

   (d) Lam-ηR

   (e) Meta-InstL

   (f) Meta-FOL

   (g) Meta-ReduceL

   (h) Lam-ηL

4. Else:

   (a) If the two terms have different head constants:

      i. CS-ConstR, CS-ProdR, CS-SortR

      ii. CS-DefaultR

      iii. CS-ConstL, CS-ProdL, CS-SortL

      iv. CS-DefaultL

   (b) App-FO, the head being compared using the rules in Figure 4.1 in the order shown, backtracking only in the rules Let-Same and Let-ζ.

   (c) Rules in Figure 4.2, in the order specified without backtracking.

4.2 Formalities

Ideally, we expect the unification algorithm to satisfy the following:

Conjecture 1 (Incorrect Conjecture of Soundness of Unification)**.** If

$$\Sigma; \Gamma \vdash t_i : T_i \qquad \text{for } i \in [1, 2]$$

and

$$\Sigma; \Gamma \vdash t_1 \approx t_2 \triangleright \Sigma'$$

then

$$\Sigma'; \Gamma \vdash t_i : T_i \qquad \text{for } i \in [1, 2]$$

and

$$\Sigma'; \Gamma \vdash t_1 \equiv t_2$$

That is, if two well-typed terms t_1 and t_2 unify under meta-context Σ, resulting in a new meta-context Σ', both terms should also be well typed under Σ'. More over, both terms should be convertible under Σ'.

However, as the name suggests, this conjecture is false—for both the current algorithm implemented in Coq, and the one described in this chapter. We present the reason why this conjecture is false, and provide a weaker conjecture we expect our algorithm to support. As it turns out, the current algorithm in Coq does not support the weaker conjecture. At the end of this section we present an example showing how to build an ill-typed term with the current unification algorithm of Coq.

4.2.1 Proving False

The following code exploits unification to *almost* prove False: except for the **Qed** command in the last line, every other line is happily accepted by Coq. It is important to remark that it is possible to encode the same example as one big unification problem. We decided to present it with a proof script to help readability.

```
01 Theorem False_proof : False.
02 Proof.
03   evar (h : (nat → False) → nat → False).
04   pose (T := fix f (x:nat) : False := h f x).
05   assert (H : h = @id (nat → False)).
06     unfold h.
07     reflexivity.
08   apply (T 0).
09 Qed.
```

The example works as follows: First, it creates a meta-variable $?h$ with type (nat → False) → nat → False. Then, it defines T as a fixpoint f, with type nat → False, whose body is the meta-variable $?h$ applied to f and the argument x. The assertion and its

proof in lines 5–7 has the effect of instantiating $?h$ with the identity function. Finally, at line 8, it applies the fixpoint T to 0, getting a complete proof term (*i.e.*, without meta-variables) with type False. Coq announces that the proof is complete, although, as mentioned before, in the last line the **Qed** command disagrees, saying that we have constructed an unbounded fixpoint.

The culprit is in the unification problem in line 7, which at first sight looks harmless:

$$?h[] \approx \mathsf{id}\ (\mathsf{nat} \to \mathsf{False})$$

The unification algorithm has no reason to distrust this equation, so it innocently proceeds to instantiate $?h$ with the identity function. However, it failed to notice that $?h$ is included in the body of the fixpoint,

$$\textbf{fix}\ f\ (x : \mathsf{nat}) : \mathsf{False} := ?h\ f\ x$$

After reducing $?h$ and the id function, we obtain the infamous infinite loop that allows us to prove False!

$$\textbf{fix}\ f\ (x : \mathsf{nat}) : \mathsf{False} := f\ x$$

We leave for future work the study of a solution to avoid this problem. A possible solution could be to *taint* every meta-variable occurring in the body of a fixpoint, performing the guardedness condition check prior to instantiation on those tainted meta-variables.

This example shows that we have to weaken the conjecture. In Sacerdoti Coen (2004) they use a notion of *weak typing*, which are the rules for typing given in Section 1.2.3 without the guardedness check on incomplete terms. That is, the body of a fixpoint is not checked for termination if it contains uninstantiated meta-variables. We borrow this notion here and note

$$\Sigma; \Gamma \vdash_\omega t : T$$

for term t having type T with the weak-typing relation. We also need the definition of a *valid* extension:

Definition 5 (Valid Extension)**.** For two meta-contexts Σ' and Σ such that $\Sigma' \geq \Sigma$ (*c.f.*, Definition 3), Σ' is a *valid* extension of Σ for term t and context Γ, noted $\Sigma' \geq_{\Gamma \vdash t} \Sigma$, iff for every body u of a fixpoint in t, with uninstantiated meta-variables in Σ, either u still have uninstantiated meta-variables in Σ', or $\Sigma'(u)$ satisfy the guardedness condition.

Now we are ready to present the conjecture we expect the algorithm to satisfy:

Conjecture 2 (Soundness of Unification)**.** If

$$\Sigma; \Gamma \vdash_\omega t_i : T_i \qquad \text{for } i \in [1,2]$$

and

$$\Sigma; \Gamma \vdash t_1 \approx t_2 \triangleright \Sigma'$$

then $\Sigma' \geq \Sigma$ and, if

$$\Sigma' \geq_{\Gamma \vdash t_i} \Sigma \qquad \text{for } i \in [1,2]$$

then

$$\Sigma'; \Gamma \vdash_\omega t_i : T_i \qquad \text{for } i \in [1,2]$$

and

$$\Sigma'; \Gamma \vdash t_1 \equiv t_2$$

We leave the proof of this conjecture for future work.

4.2.2 A Bug in the Original Algorithm

In the process of writing this thesis we found a bug in the current algorithm of Coq (version 8.4). The bug exploits two rules, Lam-η and Let-Same, which in Coq are weaker than in Figure 4.2. In Coq, Let-Same does not unify the types of the definitions prior to unifying the definitions, and Lam-η does not verify that the term is a product.

In the following example the unification algorithm instantiates a meta-variable with an ill-typed term. Luckily, the kernel realizes the mistakes and does not allow the definition to enter the pristine environment of Coq!

```
Definition it_fails (c : bool → nat) :
  (let z1 := λ x:nat ⇒ (_ : nat) in 0) =
  (let z2 := c in 0)
:= eq_refl.
```

Without getting into the details, the main unification problem in this example is as follows:

$$\textsf{let } z_1 := \lambda x : \textsf{nat}.\ ?e[c, x] \textsf{ in } 0 \approx \textsf{let } z_2 := c \textsf{ in } 0$$

with $\Sigma = \{?e : \textsf{nat}[c : \textsf{bool} \to \textsf{nat}, x : \textsf{nat}]\}$ and $\Gamma = \{c : \textsf{bool} \to \textsf{nat}\}$. Note that applying the rule Let-Parζ both sides reduce to 0 and the problem is solved (but leaving $?e$

uninstantiated). This is what our algorithm does. Instead, Coq 8.4 applies the weak version of the rule Let-Same, which equates the definitions without checking their types:

$$\lambda x : \mathsf{nat}.\ ?e[c, x] \approx c$$

In our algorithm, since c does not have type $\mathsf{nat} \to \mathsf{nat}$, Let-Same fails, backtracking to Let-Parζ. Instead, the algorithm of Coq applies the rule Let-ηL, getting the following equation:

$$?e[c, x] \approx c\ x$$

(in the new context $\Gamma, x : \mathsf{nat}$). Note that $c\ x$ is ill-typed! Since the meta-variable $?e$ expects something of type nat, and $c\ x$ is (incorrectly) re-typed with type nat, it instantiates $?e := c\ x$. As mentioned above, the kernel finds the problem and complains:

```
Error: Illegal application (Type Error):
The term "c" of type "bool -> nat" cannot be applied
  to the term "x" : "nat"
```

4.3 A Missing Heuristic: Constraint Postponement

So far we have described what our algorithm *does*, but not what it *does not* (with respect to the original algorithm of Coq). The most important piece we have purposely left out is *constraint postponement* (Reed, 2009). This heuristic, widely used in many proof assistants, has negative consequences in the context of Coq. In this section we present the motivation for why it exists in Coq today, followed by a discussion of its drawbacks. In the next section we will see that, actually, it is not as important as one might think of.

Sometimes the unification algorithm is faced with an equation that has multiple solutions, in a context where there should only be one possible candidate. For instance, consider the following term witnessing an existential quantification:

$$\mathsf{exist}\ \_\ 0\ (\mathsf{le\_n}\ 0) : \exists x.\ x \leq x$$

where exist is the constructor of the type $\exists x.\ P\ x$, with P a predicate over the (implicit) type of x. More precisely, exist takes a predicate P, an element x, and a proof that P holds for x, that is, $P\ x$. In the example above we are providing an underscore in place of P, since we want Coq to find out the predicate, and we annotate the term with a typing constraint (after the colon) to specify that we want the whole term to be a proof

that there exists a number that is lesser or equal to itself. In this case, we provide 0 as such number, and the proof le_n 0 which has type $0 \leq 0$.

During typechecking, Coq first typechecks the term on the left of the colon, and only then it verifies that its type is compatible (*i.e.*, convertible) with the typing constraint. More precisely, Coq will create a fresh meta-variable for the predicate P, let's call it $?P$, and unify $?P\ 0$ with $0 \leq 0$. Without any further information, Coq has four different (incomparable) solutions for P: $\lambda x.\ 0 \leq 0, \lambda x.\ x \leq 0, \lambda x.\ 0 \leq x, \lambda x.\ x \leq x$.

When faced with such an ambiguity, Coq delays the equation in the hope that further information will help disambiguate the problem. In this case, the necessary information is given later on through the typing constraint, which narrows the set of solution to a unique solution.

The inclusion of such heuristic has its consequences, though: On one hand, the algorithm can solve more unification problems and hence fewer typing annotations are required (in this case, we do not need to specify P). On the other hand, since constraints are delayed, the algorithm becomes hard to debug and, at times, slow.

The code below exemplifies both these problems at once.

```
Definition postponing :
  let f : nat → nat := _ in
  let x : f 0 = 0 := eq_refl in
  1000 + (f 0) = 1000 := eq_refl.
```

This code creates a meta-variable $?f$ and tries to solve two different equations:

$$?f\ 0 \approx 0 \tag{4.7}$$

$$1000 + (?f\ 0) \approx 1000 \tag{4.8}$$

Note that the second equation amounts to solve the same problem as in the first one, since evaluating the addition we obtain a thousand successors applied to $?f\ 0$, and on the right we have a thousand successors applied to 0.

Since this problem has multiple solutions, and since it does not fall into the decidable higher-order pattern fragment, the current algorithm of Coq postpones both equations. The consequences of this decision are two: first, it shows the error in the second equation, although the problem appears in the first equation. Second, in order to obtain the unsolvable equation in the second equation, it has to spend time reducing it.

For these reasons we decided to remove constraint postponement from the algorithm, making the example above fail immediately, correctly pointing out the error in the first equation. And, as we show in the next section, this does not shrink much the set of solvable problems.

4.4 Evaluation of the Algorithm

We tested our algorithm compiling the Mathematical Components library (Gonthier et al., 2008), version 1.4. We chose this library as test-bench for three reasons: (1) It is a huge development, with a total of 62 files. (2) It uses canonical structures heavily, providing us with several examples of canonical structures idioms that our algorithm should support. (3) It uses its own set of better behaved tactics. This last point is extremely important, although a bit technical. Truth be told, Coq has actually two different unification algorithms. One of these algorithms is mainly used by the type inference mechanism, and it outputs a sound substitution (up to bugs, like the one shown in §4.2.2). This is the one mentioned in this thesis as "the original unification algorithm of Coq". The other algorithm is used by most of Coq's original tactics (like apply or rewrite), and it is unsound, with convoluted semantics. Ssreflect's tactics use the former, better-behaved algorithm, which for this evaluation we have replaced with our own.

Since, as we saw in the previous section, our algorithm does not incorporate certain heuristics, it is reasonable to expect that it will not solve all the unification problems posed by the theories in the library. Surprisingly, only a very small set of problems were not solved by our algorithm. In total, only about 300 lines out of about 67,000 required changes. To give a sense of the magnitude of the task, our algorithm is called about 32 million times, of which only a very small fraction of times did it fail to find a solution when the original algorithm did (0.005%).

In the following paragraphs we group the different issues we found, ordered by number of affected lines. After presenting the issues, we discuss different solutions to solve them.

Non-dependent if−then−elses: Most notably, two thirds of the problematic lines can be easily solved with a minor change in the way Ssreflect handles **if**−**then**−**else**s. Indeed, 200 out of those 300 lines are **if**−**then**−**else** constructions in which the return type does not depend on the conditional. While in standard Coq this is not a problem, in Ssreflect the type of the branches is assumed to be dependent on the conditional. The

following example illustrates this point:

$$\textbf{if } b \textbf{ then } 0 \textbf{ else } 1$$

It features a simple **if**−**then**−**else** with conditional $b : \mathsf{bool}$, and branches with type nat. In standard Coq (modified to use our algorithm) this example typechecks, but it does not if the Ssreflect library is imported. With Ssreflect, the typechecker generates a fresh meta-variable $?T$ for the type of the branches, and proceeds to unify this meta-variable with the actual type of each branch. More precisely, $?T$ is created with contextual type $\mathsf{Type}[b : \mathsf{true}]$, and when unifying it with the actual type of each branch, b is substituted by the corresponding boolean constructor. This results in the following equations:

$$?T[\mathsf{true}] \approx \mathsf{nat}$$
$$?T[\mathsf{false}] \approx \mathsf{nat}$$

Since they are not of the form required by higher-order pattern unification, our algorithm fails. If $?T$ were created without depending on b, these equations can be easily solved.

False dependency in the in **modifier:** Another issue affecting 35 lines comes from the in modifier in Ssreflect's rewrite tactic. This modifier allows the selection of a portion of the goal to perform the rewrite. For instance, if the goal is

$$1 + x = x + 1$$

and we want to apply commutativity of addition on the term on the right, we can perform the following rewrite:

$$\mathsf{rewrite}\ [\mathsf{in}\ X\ \mathsf{in}\ \_ = X]\mathsf{addnC}$$

The standard unification algorithm of Coq instantiates X with the right hand side of the equation, and rewrite applies commutativity only to that portion of the goal. With our algorithm, however, rewrite fails. Again, the culprit is in the way meta-variables are created, with false dependencies forbidding the algorithm to apply higher-order pattern unification. In this case, the implicit (_) is replaced by a meta-variable $?y$, which is assumed to depend on X. But X is also replaced by a meta-variable, $?z$, therefore the unification problem becomes

$$?y[x, ?z[x]] = ?z[x] \ \approx\ 1 + x = x + 1$$

that, in turn, poses the equation $?y[x, ?z[x]] \approx 1 + x$, which does not have an MGU.

Non-dependent products: 30 lines required a simple typing annotation to remove dependencies in products. Consider the following Coq term:

$$\forall P\ x.\ (P\ (\mathsf{S}\ x) = \mathsf{True})$$

When Coq refines this term, it first assigns P and x two unknown types, $?T$ and $?U$ respectively, the latter depending on P. Then, it refines the term on the left of the equal sign, obtaining further information about the type $?T$ of P: it has to be a (possibly dependent) function $\forall y : \mathsf{nat}.\ ?T'[y]$. The type of the term on the left is the type of P applied to $\mathsf{S}\ x$, that is, $?T'[\mathsf{S}\ x]$. After refining the term on the right and finding out it is a Prop, it unifies the types of the two terms, obtaining the equation

$$?U'[\mathsf{S}\ x] \approx \mathsf{Prop}$$

Since, again, this equation does not fall into the HOPU fragment, our algorithm fails.

Explicit duplicated dependencies: In 11 lines, the proof developer wrote explicitly a dependency that duplicates an existing one. Consider for instance the following rewrite statement:

$$\mathsf{rewrite}\ [\_ * \_\ w]\mathsf{mulrC}$$

Here, the proof developer tries to rewrite using the commutativity of rings (mulrC), on a fragment of the goal matching the pattern $\_ * \_\ w$. Let's assume that in the goal there are several occurrences of the $*$ operator, but only one has w occurring in the right-hand side. For illustration purposes, let's assume this occurrence has the form $t * (w + u)$, for some terms t and u. The proof dev intends to select this particular expression, however, the pattern fails to match the expression.

The problem comes from the way implicit arguments are refined: An implicit _ is refined as a meta-variable depending on the entire local context. In this case, the local context will include w, so the pattern will be refined as $?y[w] * ?z[w]\ w$ (assuming no other variables appear in the local context). When unifying the pattern with the desired occurrence of the $*$ operator we obtain the problem:

$$?z[w]\ w \approx w + u$$

As we saw in Example 4.1, this equation does not have a most general unifier.

Other issues: The remaining 14 lines have other issues, and in most cases they require attention from an Ssreflect expert to identify the origin of the problem.

There are two possible solutions to solve these problems: (1) to modify the Ssreflect library to avoid these dependencies (to solve the first two cases), and the refiner to avoid creating spurious dependencies (to solve the third and fourth cases), or (2) to extend the class of problems handled by the unification algorithm. The former solution is out of the scope of this work, although we plan to explore it in the near future. We are convinced that it is not the role of the unification algorithm to decide which dependencies should be ignored. Nevertheless, in the following section (§4.5) we explore the second solution, obtaining very good results.

4.4.1 Performance

We have not yet looked into improving the performance of the algorithm. Our preliminary findings show that compiling the whole Ssreflect library takes about 35% more time with our algorithm than with the original algorithm of Coq. Note that our algorithm performs more checks than the original algorithm (*e.g.*, the hypotheses inside the grey box in Lam-η). We are certain that, with a careful profiling of the algorithm, we will detect major issues, and bring the compilation time closer to the current algorithm. In the long term, we aim to find ways to drastically improve the performance of the algorithm.

4.5 A Heuristic to Remove Potentially Spurious Dependencies

The algorithm presented so far is a bit *conservative*: when instantiating a meta-variable, in order to cope with problems outside the higher-order pattern fragment, it only attempts the first-order approximation heuristic. For this reason, as we saw in the previous section, many equations with "false dependencies" are rejected right away, without any further attempt to find, if not the most general solution, at least *some* solution. In the context of the Ssreflect library, we demonstrated that this conservative approach is acceptable, since only a few lines required changes to work with our algorithm. And, moreover, in those cases, it is arguable who's to blame, if our algorithm for not finding a solution, or the tactics and the refiner for posing equations with such false dependencies.

However, we should not jump to conclusions: the Ssreflect library was conceived for a specific problem (verification of mathematical components), and with specific programming patterns, coherently shared among the whole set of files. Other libraries may have different programming patterns, therefore expecting different results from the unification algorithm. For this reason, we decided to compile the files from the book Certified

Programming with Dependent Types (CPDT) (Chlipala, 2011a) , which provides several examples of functional programming with dependent types, in a radically different style from that of Ssreflect.

The result is rather discouraging: from the total of 6.200 lines, more than 60 lines (about 0.01%) require modifications. The problem is exacerbated if we consider that this library uses standard Coq tactics instead of Ssreflect's ones. As mentioned in the introduction of the previous section, most of Coq's original tactics use a different unification algorithm than the one we are replacing, so our algorithm is being called to solve fewer cases than when compiling Ssreflect.

We are left with two options: either we change the refiner and tactics to make sure that fewer equations get false dependencies, or we introduce a new rule to the algorithm to brutally chop off those dependencies, therefore cherry-picking one specific solution from the set of possible solutions. Since the first option is out of the scope of this work, we pursue the second one. In particular, we extend the algorithm with the rule shown below, which simply removes all elements in the suspended substitution σ of a meta-variable $?x$ that are not variables, or that occurs more than once in the substitution. Given a list of indices l, we note σ_l as the substitution obtained after filtering out those elements in σ whose indices are not in l. Similarly, we denote Ψ_l as the filtering of local context Ψ with respect to l.

$$\frac{\begin{array}{c} ?x : T[\Psi] \in \Sigma \qquad l = [i | \sigma_i \text{ is variable and } \nexists j > i.\ \sigma_i = (\sigma, \overline{u})_j] \\ \Psi' = \Psi_l \qquad \Sigma \vdash \Psi' \qquad \mathsf{FV}(T) \subseteq \Psi' \\ \Sigma \cup \{?y : T[\Psi'], ?x := ?y[\mathsf{id}_{\Psi'}]\}; \Gamma \vdash t \approx ?y[\sigma_l]\ \overline{u} \triangleright \Sigma' \end{array}}{\Sigma; \Gamma \vdash t \approx ?x[\sigma]\ \overline{u} \triangleright \Sigma'}\ \textsc{Meta-PruneR}$$

What this rule does, then, is to take each position i in σ such that σ_i is a variable with no duplicated occurrence in $\sigma, \overline{u}$. The list containing those positions l is used to filter out the local context of the meta-variable, obtaining the new context Ψ'. After making sure this context is valid, a fresh meta-variable $?y$ is created in this restricted local context, and $?x$ is instantiated with this meta-variable. The new meta-context obtained after this instantiation is used to recursively call the unification algorithm to solve the problem $?y[\sigma_l]\ \overline{u} \approx t$.

We illustrate the effect of this rule with the examples from Section 4.4. The first issue was with non-dependent **if**−**then**−**else**s, which produced equations of the form

$$?T[\mathsf{true}] \approx \mathsf{nat}$$
$$?T[\mathsf{false}] \approx \mathsf{nat}$$

In this case, the rule Meta-PruneR will remove the dependency to the conditional, obtaining the following solvable equations, for a fresh meta-variable $?T'$:

$$?T'[] \approx \mathsf{nat}$$
$$?T'[] \approx \mathsf{nat}$$

The second issue was with meta-variable dependency due to incorrect dependency injection by the in modifier in Ssreflect's rewrite tactic. In the example, we ended up with the following equation:

$$?y[x, ?z[x]] \approx 1 + x$$

The rule Meta-PruneR will remove the second dependency, obtaining the solvable equation

$$?y'[x] \approx 1 + x$$

The third issue was with non-dependen products. In the example, the equation that fall outside the higher-order pattern fragment was:

$$?U'[\mathsf{S}\ x] \approx \mathsf{Prop}$$

Again, Meta-PruneR will remove the dependency and obtain the equation:

$$?U''[] \approx \mathsf{Prop}$$

The fourth issue was with explicit duplicated dependencies. The problematic equation there was:

$$?z[w]\ w \approx t * (w + u)$$

where the second w in the left-hand side corresponds to an explicit written dependency (as in $\_\ w$). Our algorithm, extended with the rule Meta-PruneR, will chop off the first occurrence, obtaining the equation:

$$?z'[]\ w \approx t * (w + u)$$

As expected, with the addition of this rule the Ssreflect library compiles almost out of the box, with only 23 lines requiring modifications. We also obtain a significant reduction in the number of lines requiring modifications in the CPDT book: only 14 lines had to be revised. In most of the cases, the problem comes from our algorithm picking a different solution to the one expected. This is unsurprising, since we are deciding early on a solution, instead of postponing the equation until more information is obtained. And even when the selected solution works most of the times, it simply cannot work always. We are certain that, with a smarter refiner and tactics, the number of unsolved problems can be drastically reduced. The question that we leave open for future work is if such modifications to the refiner and tactics are enough, or if this heuristic is, after all, essential.

Chapter 5

Related Work

5.1 Overloading

5.1.1 Canonical Structures

To the best of our knowledge, the first application of *term classes*, that is, canonical structures keying on terms rather than just types, appeared in Bertot et al. (2008). Term classes are used for encoding iterated versions of classes of algebraic operators.

Gonthier (2011) describes a library for matrix algebra in Coq, which introduces a variant of the tagging and lemma selection patterns (§2.2 and §2.5.2, respectively), but briefly, and as a relatively small part of a larger mathematical development. In contrast, in Chapter 2, we give a more abstract and detailed presentation of the general tagging pattern and explain its operation with a careful trace. We also present several other novel design patterns for canonical structures, and explore their use in reasoning about heap-manipulating programs.

Asperti et al. (2009) present unification hints, which generalize Coq's canonical structures by allowing a canonical solution to be declared for any class of unification equations, not only for equations involving a projection out of a structure. Hints are shown to support applications similar to our reflection pattern from Section 2.4. However, they come with limitations; for example, the authors comment that hints cannot support backtracking. Thus, we believe that the design patterns that we have developed in Chapter 2 are not obviated by the additional generality of hints, and would be useful in that framework as well.

5.1.2 Type Classes

Sozeau and Oury (2008) present type classes for Coq, which are similar to canonical structures, but differ in a few important respects. The most salient difference is that inference for type class instances is not performed by unification, but by general proof search. This proof search is triggered after unification, and it is possible to give a weight to each instance to prioritize the search. This leads to somewhat simpler code, since no tagging is needed, but, on the other hand, it seems less expressive. For instance, we were not able to implement the search-and-replace pattern of Section 2.5 using Coq type classes, due to the lack of connection between proof search and unification. We *were* able to derive a different solution for bnd_writeR using type classes, but the solution was more involved (requiring two specialized classes to differentiate the operations such as write which perform updates to specific heaps, from the operations which merely inspect pointers without performing updates). More importantly, we were not able to scale this solution to more advanced lemmas from our implementation of higher-order separation logic. In contrast, canonical structures did scale, and we provide the overloaded code for these lemmas in our source files (Ziliani, 2014).

In the end, we managed to implement all the examples in this paper using Coq type classes, demonstrating that lemma overloading is a useful high-level concept and is not tied specifically to canonical structures. (The implementations using type classes are included in our source files as well (Ziliani, 2014).) Nevertheless, we have not yet arrived at a full understanding of how Coq type classes perform instance resolution. In particular, its semantics are hard to grasp, since type classes uses a completely unsound unification algorithm. In contrast, the semantics of canonical structures were thoroughly described in chapters 2 and 4.

Besides this, ultimately, it may turn out that the two formalisms (canonical structures and type classes) are interchangeable in practice, but we need more experience with type classes to confirm this. In the future, we hope the unification algorithm presented in this thesis will be also used by the type class mechanism. This will add predictability to type classes, and allow us to frame the semantics of type classes in the same semantical framework.

Using Coq type classes, Spitters and van der Weegen (2011) present a reflection algorithm based on the example of Asperti *et al.* discussed above. In addition, they consider the use of type classes for overloading and inheritance when defining abstract mathematical structures such as rings and fields. They do not, however, consider lemma overloading more generally as a means of proof automation, as we have presented in Chapter 2.

Finally, in the context of Haskell, Morris and Jones (2010) propose an alternative design for a type class system, called `ilab`, which is based on the concept of *instance chains*. Essentially, instance chains avoid the need for overlapping instances by allowing the programmer to control the order in which instances are considered during constraint resolution and to place conditions on when they may be considered. Our tagging pattern (Section 2.2) can be seen as a way of coding up a restricted form of instance chains directly in existing Coq, instead of as a language extension, by relying on knowledge of how the Coq unification algorithm works. `ilab` also supports *failure clauses*, which enable one to write instances that can only be applied if some constraint fails to hold. Our approach does not support anything directly analogous, although (as Morris and Jones mention) failure clauses can be encoded to some extent in terms of more heavyweight type class machinery.

5.2 Languages for Typechecked Tactics

In the last five years there has been increasing interest in languages that support safe tactics to manipulate proof terms of dependently typed logics. Delphin (Poswolsky and Schürmann, 2009), Beluga (Cave and Pientka, 2012, Pientka, 2008, Pientka and Dunfield, 2008), and VeriML (Stampoulis and Shao, 2010, 2012) are languages that, like Lemma Overloading (LO) and Mtac, fall into this category. By "safe" we mean that, if the execution of a tactic terminates, then the resulting proof term has the type specified by the tactic.

But, unlike LO and Mtac, these prior systems employ a strict separation of languages: the computational language (the language used to write tactics) is completely different from the logical language (the language of proofs), making the meta-theory heavier than in our work. Indeed, our proof of type soundness is completely straightforward, as it inherits from CIC all the relevant properties such as type preservation under substitution. Having a simple meta-theory is particularly important to avoid precluding future language extensions—indeed, extensions of the previous systems have often required a reworking of their meta-theory (Cave and Pientka, 2012, Stampoulis and Shao, 2012). In comparison, the extension of the Mtac language to support state (§3.6) did not require much rewriting of the original proof of soundness.

Another difference between these languages and ours is the logical language they support. For Delphin and Beluga it is LF (Harper et al., 1993), for VeriML it is λHOL (Barendregt and Geuvers, 2001), and for LO and Mtac it is CIC (Bertot and Castéran, 2004). CIC is the only one among these that provides support for computation at the term and type level, thereby enabling proofs by reflection (*e.g.*, see §3.3.3). Instead, in previous

systems term reduction must be witnessed explicitly in proofs. To work around this, VeriML's computational language includes a construct letstatic that allows one to stage the execution of tactics, so as to enable equational reasoning at typechecking time. Then, proofs of (in-)equalities obtained from tactics can be directly injected in proof terms generated by tactics. This is similar to our use of **run** in the example from §3.3.3, with the caveat that letstatic cannot be used within definitions, as we did in the inner_prod example, but rather only inside tactics.

In Beluga and VeriML the representation of objects of the logic in the computational language is based on Contextual Modal Type Theory (Nanevski et al., 2008c).[1] Therefore, every object is annotated with the context in which it is immersed. For instance, a term t depending only on the variable x is written in Beluga as $[x.\ t]$, and the typechecker enforces that t has only x free. In Mtac, it is only possible to perform this check dynamically, writing an Mtactic to inspect a term and rule out free variables not appearing in the set of allowed variables. (The interested reader may find an example of this Mtactic in the distributed files.) On the other hand, the syntax of the language and the meta-theory required to account for contextual objects are significantly heavier than those of Mtac.

Delphin shares with Mtac the $\nu x : A$ binder from Nanevski (2002), Schürmann et al. (2005). In Delphin, the variable x introduced by this binder is distinguished with the type $A^{\#}$, in order to statically rule out offending terms like $\nu x : A.\ \mathbf{ret}\ x$. In Mtac, instead, this check gets performed dynamically. Yet again, we see a tension between the simplicity of the meta-theory and the static guarantees provided by the system. In Mtac we favor the former.

Of all these systems, VeriML is the only one that provides ML-style references at the computational level. Our addition of mutable state to Mtac is clearly inspired by the work of Stampoulis and Shao (2010), although, as we do not work with Contextual Modal Type Theory, we are able to keep the meta-theory of references quite simple.

Beluga's typechecker is constantly growing in complexity in order to statically verify the completeness and correctness of tactics (through *coverage* and termination checking). If a tactic is proved to cover all possible shapes of the inspected object, and to be terminating, then there is no reason to execute it: it is itself a meta-theorem one can trust. This concept, also discussed below, represents an interesting area of future research for Mtac.

[1] The contextual types of CMTT are not to be confused with the lightweight "contextual types" that LO and Mtac assigns to unification variables (*e.g.*, §3.4.2). In our setting, we only use contextual types to ensure soundness of unification, inheriting the mechanism from Coq. Coq's contextual types are essentially hidden from the user, whereas in VeriML and Beluga they are explicit in the computational language.

Finally, a key difference between our work and all the aforementioned systems is the ability to program tactics interactively, as shown in §2.7 and §3.3.1. None of the prior systems support this.

5.3 Dependent Types Modulo Theories

Several recent works have considered enriching the term equality of a dependently typed system to natively admit inference modulo theories. One example is Strub *et al.*'s CoqMT (Barras et al., 2011, Strub, 2010), which extends Coq's typechecker with *first-order equational* theories. Another is Jia *et al.*'s language $\lambda^{\cong}$ (pronounced "lambda-eek") (Jia et al., 2010), which can be instantiated with various abstract term-equivalence relations, with the goal of studying how the theoretical properties (*e.g.*, the theory of contextual equivalence) vary with instantiations. Also related are Braibant *et al.*'s AAC tactics for rewriting modulo associativity and commutativity in Coq (Braibant and Pous, 2010).

In the two systems for proof automation presented in this thesis, we do not change the term equality of Coq. Instead, we allow user-supplied algorithms to be executed when desired, rather than by default whenever two terms have to be checked for equality. Moreover, these algorithms do not have to be only decision procedures, but can implement general-purpose computations.

5.4 Typechecked Tactics Through Reflection

There is a large, mostly theoretical, body of work on using the theory of a proof assistant to reason about tactics written for the same proof assistant. The high-level idea is to *reflect* (a fraction of) the object language of the proof assistant into a Term datatype inside the same proof assistant. Tactics are then constructed using this datatype, and can be verified just like any other procedure built inside the proof assistant. If the tactic is proven to be correct, then it can be safely used as an axiom, without having to spend time executing it, or checking its result.

While this idea is appealing, the circularity that comes from reasoning about the logic of a proof assistant within itself endangers the soundness of the logic, and therefore special care must be taken. In theory, one can avoid this circularity by restricting the objects of the language that can be reflected, and by establishing a hierarchy of assistants: an assistant at level k can reason about tactics for the assistant at level

$k-1$ or below.[2] This concept is discussed in depth, from a theoretical point of view, in Allen et al. (1990), Artëmov (1999), Constable (1992), Harrison (1995), Howe (1993), Pollack (1995). More recently, Devriese and Piessens (2013) provide, to the best of our knowledge, the first attempt at implementing it, but without arriving yet at a practical and sound implementation.

More pragmatically, the programming language Agda suports a lightweight implementation of reflection.[3] It does so with two primitives, **quoteGoal** and **unquoteGoal**. The first one reflects the current goal into a Term datatype, and the second one solves the current goal by the proof term that results from interpreting a term of the same datatype. This mechanism avoids the circularity problem mentioned above in two ways: first, as mentioned above, by restricting the object language supported and, second, by dynamically typechecking the proof term before solving a goal with it. That is, like in LO and Mtac, a tactic is never trusted, and therefore cannot be used as an axiom. An example of proof automation using Agda's reflection mechanism, and its limitations, can be found in van der Walt and Swierstra (2013).

All of the aforementioned works require tactics to be written using a de Bruijn encoding of objects, in contrast to the direct style promoted in LO and Mtac. In Agda this can be annoying, but is not fatal. However, if tactics have to be proven correct, as in Devriese and Piessens (2013), then the overhead of verifying a tactic quickly negates the benefit of using it. Another disadvantage is that tactics are restricted to the pure language of the assistant, and therefore cannot use effects like non-termination and mutable state.

Recent work by Malecha et al. (2014) restricts the use of reflection to reflective *hints*. Hints, in their setting, are lemmas reflected into an inductive datatype, similar to what the reflection mechanism of Agda does, packed together with a proof of soundness. These hints are then used by a specialized auto tactic that reflects the goal and tries to prove it automatically using the reflective hints. This work shares with the reflection mechanism of Agda the de Bruijn encoding of terms, but, unlike in Devriese and Piessens (2013), the soundness proof is local to the hint: the auto tactic will be in charge of performing the recursion, so the effort required to verify a hint is significantly smaller. Unlike Mtac, this work does not aim at a full language for meta-programming.

5.5 Simulable Monads

Claret et al. (2013) present *Cybele*, a framework for building more flexible proofs by reflection in Coq. Like Mtac, it provides a monad to build effectful computations,

[2] This is reminiscent of the universe level hierarchy in Type Theory.

[3] http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.Reflection

although these effects are compiled and executed in OCaml. Upon success, the OCaml code creates a *prophecy* that is injected back into Coq to simulate the effects in pure CIC. On the one hand, since the effects Cybele supports must be replayable inside CIC, it does not provide meta-programming features like Mtac's **mmatch**, **nu**, **abs**, and **evar**, which we use heavily in our examples. On the other hand, for the kinds of effectful computations Cybele supports, the proof terms it generates ought to be smaller than those Mtac generates, since Cybele enforces the use of proof by reflection. The two systems thus offer complementary benefits, and can in principle be used in tandem.

5.6 Effectful Computations in Coq

There is a large body of work incorporating or modeling effectful computations in Coq. Concerning the former, in Armand et al. (2010) the kernel of Coq is extended to handle machine integers and *persistent arrays* (Baker, 1991). As discussed in Section 3.6, in Mtac we favored the more generic reference model à la ML, and we did not need to modify the kernel.

Concerning the latter, there are two different approaches when it comes to modeling state in Coq. The more traditional one, exemplified by Miculan and Paviotti (2012), Nanevski et al. (2008a,b, 2010), is to encapsulate the effectful computations in a monad, and provide two different views of it. Inside the prover, this monad performs the computation in an inefficient functional (state-passing) way: easy to verify, but slow to execute. Then, the verified code is extracted into a programming language with imperative features (typically Haskell or OCaml) using the efficient model of imperative computation that these languages provide.

Vafeiadis (2013) argues that the monadic style of programming imposed by the previous approach is inconvenient, as it drags all functions that use stateful operations into the monadic tarpit, from which they cannot escape even if they are observably pure. He proposes an alternative called *adjustable references*, to enable the convenient use of references within code that is not observably stateful. An adjustable reference is like an ML reference cell with the addition of an invariant ensuring that updates to the cell are not observable. As with the previous approach, the code written in the prover is extracted into efficient OCaml code.

In contrast to the above approaches, stateful Mtactics do not offer any formal model for reasoning about code that uses references. State is restricted to the language of Mtactics, whose operational behavior cannot be reasoned about within Coq itself.

5.7 The Expression Problem

The examples at the end of Section 2.7 and Section 3.7 for extensible overloaded lemmas and Mtactics, respectively, can be seen as representatives for a subset of problems encompassed in The Expression Problem.[4] As originally formulated by P. Wadler, The Expression Problem focuses on dynamically extending a datatype with new constructors and, accordingly, in extending the functions acting on that datatype to handle the new constructors. In Haskell, the expression problem was solved using type classes in "Data types à la Carte" (Swierstra, 2008). In Coq, it is possible to port this idea, extending it further to support proofs, as shown in "Meta-theory à la Carte" (Delaware et al., 2013a). In this work, improved in Delaware et al. (2013b), the extensibility of datatypes and functions is used to modularly prove meta-theorems of languages with a different set of features.

In our setting, however, the problem is significantly simpler: an extensible tactic solely extends a meta-function acting on program expressions of a given type. That is, the type of programs itself does not change. For this reason, the solution offered in Section 2.7 and Section 3.7 are significantly simpler than that of Delaware et al. (2013a), Swierstra (2008). It might be fruitful to combine our work with that of Delaware et al. (2013a) to obtain "Tactics à la Carte": given a set of features for a programming language, in addition to the set of modular meta-theorems, one can obtain a set of tactics to help proving that programs in that language met a specification.

5.8 Unification

The first formal introduction of the problem of unification is due to Robinson (1965), 50 years ago, making the task of listing related work on the area a rather dull and daunting task. Instead, we focus our attention on a set of works that inspired our work, in the narrower area of higher-order unification, and refer the reader to different books and surveys (Baader and Nipkow, 1998, Baader and Siekmann, 1994, Huet, 2002, Knight, 1989).

Metaphorically speaking, Chapter 4, the chapter on unification, is a "coqtail" with the following recipe:

Ingredients:

[4] http://en.wikipedia.org/wiki/Expression_problem

1 part Sacerdoti Coen (2004, chap. 10).
1 part Abel and Pientka (2011).
1 part Saïbi (1999, chap. 4).
1 part Homegrown rules.
1 Cherry.

Preparation:

1. Pour into the glass the description of *first-order* unification for CIC from Sacerdoti Coen (2004, chap. 10). Use a strainer to filter out meta-variable instantiation, since it does not perform higher-order pattern unification.
2. Slowly add the treatment for meta-variable instantiation from Abel and Pientka (2011), including the pruning and substitution intersection judgments. At the same time, pay special attention to avoid dropping into the mix constraint postponement (an ingredient coming from Reed (2009)), and unnecessary η-expansions.
3. Prior to mixing in the treatment of canonical structures from Saïbi (1999, chap. 4), fix the order of unification of the subproblems to match the current algorithm in Coq. Note that, although it is shipped all the way from France, this ingredient has a richer description of canonical structures than Saïbi (1997).
4. Finish the mix by adding the following homegrown rules: (i) missing cases for canonical structures (CS-Prod, CS-Sort, and CS-Default), and (ii) heuristics for meta-variable instantiation (Meta-FO, Meta-Reduce).
5. Garnish with cherry: a novel precise description of how reduction and backtracking is handled (§4.1.2 and §4.1.5).

The cherry provides a sort of sweet-sour flavor, which, depending on the meal, might lend to a bit of reflux. Consider pairing the mix with the following food:

$$\mathsf{fst}'\ M\ N \approx \mathsf{fst}'\ P\ Q$$

where fst' is a constant defined as $\lambda x\ y.\ x$, and M and P are non-unifiable terms, heavy on fats. Because of reduction, the food will reflux as

$$M \approx P$$

which is processed again to ultimately fail to digest. In fact, it is not hard to come up with a meal with exponential reflux. In Pfenning and Schürmann (1998) they propose

a strict diet to avoid reflux; statically analyzing definitions to see if they expose their arguments, therefore ensuring that a failure to unify arguments implies a failure to unify the definition. In the example above, for instance, fst' exposes the first argument, so a failure to unify that argument implies that no further reduction is necessary: the problem is unsolvable.

A simpler option to solve similar issues is to implement a cache of failures. In the future, we are considering experimenting with different ingredients of this sort, in the interest of obtaining a gentler drink.

Chapter 6

Conclusions and Future Work

The work presented in this dissertation shines a new light on typed languages for tactic development, based on the following key observation:

> *The dependently typed logic of Coq (CIC) provides the necessary ingredients to provide (dependent-)types—within the same logic—to tactics.*

From this observation, I have explored two different idioms for (dependently) typed tactic languages:

Lemma Overloading: With overloading—in the form of type classes or, actually, *term* classes—one obtains a logic programming language.

Mtac: With an inductive type resembling a monad, and a simple interpreter baked into the refiner, one obtains a functional programming language, which may include imperative features.

Both idioms, in the context of the Coq *interactive* proof assistant, allow the construction of tactics *interactively*. I have studied several examples to show the trade-off between the two idioms, proving that, actually, it is fruitful to combine the two (§3.7). This combination gives raise to *extensible* functional tactics. Putting it all together, one arrives at the first formulation of a language for *extensible interactive typechecked functional tactic programming with imperative features*.

The semantics of both programming idioms rely on Coq's unification algorithm. For this reason, I devoted part of this dissertation to describing and building a novel unification algorithm for CIC, more predictable than the current algorithm of Coq. To the best of my knowledge, this is the first work introducing a unification algorithm for CIC including

canonical structures and *first order approximation* with backtracking and reduction. In this algorithm I purposely left out a heuristic known as *constraint postponement* (Reed, 2009), which makes the algorithm unpredictable and, in some cases, slow. Despite this omission, I managed to compile the Mathematical Components library (Gonthier et al., 2008) using our algorithm, having to change only a small number of lines. This result shows that predictability is not at odds with practicality.

To conclude, I enumerate different interesting directions for future work, some of which were already mentioned in the corpus of this dissertation.

Persistent State in Mtac: In §3.6.5 I mentioned the challenges maintaining persistent state across multiple executions of Mtactics. I believe that overcoming these challenges is a very promising direction for future work, since persistent state will allow Mtac to maintain tables of information similar to those used for overloading.

Compilation of Mtactics: In §6 I mentioned already the possibility to compile Mtactics into Ocaml to build more efficient tactics.

Coverage and Termination Checker for Mtactics: In §5.2 I mentioned that Beluga performs static checks to promote tactics into meta-theorems, avoiding the need to execute them (such tactics will always return a proof of the appropriate type). It is interesting to study how to incorporate those checks into Mtac.

Extensions to the Mtac Language: Mtac suffers from several limitations: it cannot (directly) manipulate goals, it cannot inspect inductive types, etc. These extensions would help Mtac cover the functionality provided by Ltac, and more.

Formalization of Unification: This thesis makes the first steps in the direction of a new, sound unification algorithm. A natural next step is to fully formalize in Coq the algorithm, thereby proving Conjecture 2. The main challenge lies in the encoding of Coq within Coq. Some steps in this direction were already given by Bruno Barras in his contribution CoqInCoq[1].

Formalization of the Refiner: The refiner is another key component of an interactive proof assistant. Once the unification algorithm is formalized, formalizing the refiner should not be a challenging task, and it would provide stronger guarantees for the tools outside Coq's kernel.

Strengthening Unification: Somewhat related to the previous two points, the conjecture made in §4.2 is not as strong as one would hope, as it does not take into account the guardedness condition for meta-variables occurring inside fixpoints.

[1] http://coq.inria.fr/pylons/pylons/contribs/view/CoqInCoq/v8.4

One option could be for the refiner to taint those meta-variables, so unification can perform the check upon instantiation.

Improvements in the Performance of Unification: At the end of §5.8 I mentioned that unification backtracks too much, trying hard to unify non-unifiable terms. It is critical to find methods to avoid unnecessary backtracking, without losing predictability, and without missing solutions. Somewhat related to this, in cases where a term is unified with multiple terms, as in a **mmatch**, it would be interesting to figure out how to reuse some of the steps performed by previous unification problems.

Bibliography

Abel, A. and Pientka, B. (2011). Higher-order dynamic pattern unification for dependent types and records. In *Proceedings of the 10th international conference on Typed lambda calculi and applications*, TLCA'11, pages 10–26, Berlin, Heidelberg. Springer-Verlag.

Allen, S. F., Constable, R. L., Howe, D. J., and Aitken, W. E. (1990). The semantics of reflected proof. In *LICS*.

Appel, K. and Haken, W. (1976). Every map is four colourable. In *Bulletin of the American Mathematical Society*, volume 82, pages 711–712.

Armand, M., Grégoire, B., Spiwack, A., and Théry, L. (2010). Extending Coq with imperative features and its application to SAT verification. In *ITP*.

Artëmov, S. N. (1999). On explicit reflection in theorem proving and formal verification. In *CADE*.

Asperti, A., Ricciotti, W., Coen, C. S., and Tassi, E. (2009). Hints in unification. In *TPHOLs*, volume 5674 of *LNCS*, pages 84–98.

Baader, F. and Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA.

Baader, F. and Siekmann, J. H. (1994). Handbook of logic in artificial intelligence and logic programming. chapter Unification Theory, pages 41–125. Oxford University Press, Inc., New York, NY, USA.

Baker, H. G. (1991). Shallow binding makes functional arrays fast. *SIGPLAN Notices*, 26(8):145–147.

Barendregt, H. and Geuvers, H. (2001). Proof-assistants using dependent type systems. In Robinson, A. and Voronkov, A., editors, *Handbook of automated reasoning*.

Barras, B., Jouannaud, J.-P., Strub, P.-Y., and Wang, Q. (2011). CoqMTU: a higher-order type theory with a predicative hierarchy of universes parametrized by a decidable first-order theory. In *LICS*, pages 143–151.

Bertot, Y. and Castéran, P. (2004). *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer.

Bertot, Y., Gonthier, G., Ould Biha, S., and Pasca, I. (2008). Canonical big operators. In *TPHOLs*, volume 5170 of *LNCS*, pages 86–101.

Braibant, T. and Pous, D. (2010). Rewriting modulo associativity and commutativity in Coq. In *Second Coq workshop*.

Cave, A. and Pientka, B. (2012). Programming with binders and indexed data-types. In *POPL*.

Cervesato, I. and Pfenning, F. (2003). A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688.

Chlipala, A. (2011a). *Certified Programming with Dependent Types*. MIT Press. http://adam.chlipala.net/cpdt/.

Chlipala, A. (2011b). Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245.

Chlipala, A. (2013). *Certified programming with dependent types*. MIT Press.

Claret, G., del Carmen González Huesca, L., Régis-Gianas, Y., and Ziliani, B. (2013). Lightweight proof by reflection using a posteriori simulation of effectful computation. In *ITP*.

Constable, R. L. (1992). Metalevel programming in constructive type theory. In Broy, M., editor, *Programming and Mathematical Method*, volume 88 of *NATO ASI Series*, pages 45–93.

Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T., and Smith, S. F. (1986). *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Delahaye, D. (2000). A tactic language for the system coq. In *Proceedings of the 7th international conference on Logic for programming and automated reasoning*, LPAR'00, pages 85–95, Berlin, Heidelberg.

Delaware, B., d. S. Oliveira, B. C., and Schrijvers, T. (2013a). Meta-theory à la carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 207–218, New York, NY, USA. ACM.

Delaware, B., Keuchel, S., Schrijvers, T., and Oliveira, B. C. (2013b). Modular monadic meta-theory. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 319–330, New York, NY, USA. ACM.

Devriese, D. and Piessens, F. (2013). Typed syntactic meta-programming. In *ICFP*.

Goldfarb, W. D. (1981). The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230.

Gonthier, G. (2008). Formal proof — the four-color theorem. *Notices of the AMS*, 55(11):1382–93.

Gonthier, G. (2011). Point-free, set-free concrete linear algebra. In *ITP*, pages 103–118.

Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., O'Connor, R., Ould Biha, S., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., and Théry, L. (2013a). A machine-checked proof of the odd order theorem. In *ITP*.

Gonthier, G. and Mahboubi, A. (2010). An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152.

Gonthier, G., Mahboubi, A., and Tassi, E. (2008). A small scale reflection extension for the Coq system. Technical report, INRIA.

Gonthier, G., Ziliani, B., Nanevski, A., and Dreyer, D. (2013b). How to make ad hoc proof automation less ad hoc. *JFP*, 23(4):357–401.

Gordon, A. D. (1994). A mechanisation of name-carrying syntax up to alpha-conversion. In *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, HUG '93, pages 413–425, London, UK, UK. Springer-Verlag.

Gordon, M. J. C. and Melham, T. F., editors (1993). *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, New York, NY, USA.

Grégoire, B. and Leroy, X. (2002). A compiled implementation of strong reduction. In *ICFP*.

Grégoire, B. and Mahboubi, A. (2005). Proving equalities in a commutative ring done right in Coq. In *TPHOLs*, pages 98–113.

Hall, C., Hammond, K., Jones, S. P., and Wadler, P. (1996). Type classes in Haskell. *TOPLAS*, 18(2):241–256.

Harper, R., Honsell, F., and Plotkin, G. (1993). A framework for defining logics. *J. ACM*, 40(1):143–184.

Harrison, J. (1995). Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK.

Howe, D. J. (1993). *Reflecting the semantics of reflected proof*. Cambridge University Press.

Huet, G. P. (2002). Higher order unification 30 years later. In *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '02, pages 3–12, London, UK, UK. Springer-Verlag.

Hur, C.-K., Neis, G., Dreyer, D., and Vafeiadis, V. (2013). The power of parameterization in coinductive proof. In *POPL*.

Jedynak, W. (2013). Operational semantics of Ltac. Master's thesis, Institute of Computer Science, University of Wrocław.

Jedynak, W., Biernacka, M., and Biernacki, D. (2013). An operational foundation for the tactic language of coq. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, pages 25–36, New York, NY, USA. ACM.

Jia, L., Zhao, J., Sjöberg, V., and Weirich, S. (2010). Dependent types and program equivalence. In *POPL*, pages 275–286.

Knight, K. (1989). Unification: A multidisciplinary survey. *ACM Comput. Surv.*, 21(1):93–124.

Launchbury, J. and Peyton Jones, S. L. (1994). Lazy functional state threads. In *PLDI*.

Leroy, X. (2009). Formal verification of a realistic compiler. *CACM*, 52(7):107–115.

Luo, Z. and Pollack, R. (1992). The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh.

Malecha, G., Chlipala, A., and Braibant, T. (2014). Compositional computational reflection. In *ITP*.

Mendler, N. P. (1991). Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172.

Miculan, M. and Paviotti, M. (2012). Synthesis of distributed mobile programs using monadic types in coq. In *ITP*.

Miller, D. (1991a). Unification of simply typed lambda-terms as logic programming. In *ICLP*, pages 255–269.

Miller, D. (1991b). Unification of simply typed lamda-terms as logic programming. In *ICLP*.

Morris, J. G. and Jones, M. P. (2010). Instance chains: Type class programming without overlapping instances. In *ICFP*, pages 375–386.

Nanevski, A. (2002). Meta-programming with names and necessity. In *ICFP*.

Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., and Birkedal, L. (2008a). Ynot: Dependent types for imperative programs. In *ICFP*.

Nanevski, A., Morrisett, J. G., and Birkedal, L. (2008b). Hoare type theory, polymorphism and separation. *JFP*, 18(5-6):865–911.

Nanevski, A., Pfenning, F., and Pientka, B. (2008c). Contextual modal type theory. *ACM Trans. Comput. Logic*, 9(3).

Nanevski, A., Vafeiadis, V., and Berdine, J. (2010). Structuring the verification of heap-manipulating programs. In *POPL*.

Paulson, L. (1994). *Isabelle: A Generic Theorem Prover*. Springer.

Pfenning, F. and Schürmann, C. (1998). Algorithms for equality and unification in the presence of notational definitions. In *Types for Proofs and Programs*, page 1657. Springer-Verlag LNCS.

Pfenning, F. and Schürmann, C. (1999). System description: Twelf - a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction: Automated Deduction*, CADE-16, pages 202–206, London, UK, UK. Springer-Verlag.

Pientka, B. (2008). A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL*.

Pientka, B. and Dunfield, J. (2008). Programming with proofs and explicit contexts. In *PPDP*.

Pientka, B. and Pfenning, F. (2003). Optimizing higher-order pattern unification. In *Automated Deduction–CADE-19*, pages 473–487. Springer Berlin Heidelberg.

Pollack, R. (1995). On extensibility of proof checkers. In *TYPES*.

Poswolsky, A. and Schürmann, C. (2009). System description: Delphin – a functional programming language for deductive systems. *ENTCS*, 228:113–120.

Reed, J. (2009). Higher-order constraint simplification in dependent type theory. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '09, pages 49–56, New York, NY, USA. ACM.

Reynolds, J. C. (2002). Separation logic: a logic for shared mutable data structures. In *LICS*, pages 55–74.

Robertson, N., Sanders, D., Seymour, P., and Thomas, R. (1997). The four-colour theorem. *J. Comb. Theory Ser. B*, 70(1):2–44.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41.

Saaty, T. and Kainen, P. (1977). *The four-color problem: assaults and conquest.* Advanced Book Program. McGraw-Hill International Book Co.

Sacerdoti Coen, C. (2004). *Mathematical Knowledge Management and Interactive Theorem Proving.* PhD thesis, University of Bologna.

Saïbi, A. (1997). Typing algorithm in type theory with inheritance. In *POPL*.

Saïbi, A. (1999). *Outils Generiques de Modelisation et de Demonstration pour la Formalisation des Mathematiques en Theorie des Types. Application a la Theorie des Categories.* PhD thesis, University Paris 6.

Schürmann, C., Poswolsky, A., and Sarnat, J. (2005). The ∇-calculus. Functional programming with higher-order encodings. In *TLCA*.

Sørensen, M. H. and Urzyczyn, P. (2006). *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics).* Elsevier Science Inc., New York, NY, USA.

Sozeau, M. (2007). Subset coercions in Coq. In *TYPES*.

Sozeau, M. and Oury, N. (2008). First-class type classes. In *TPHOLs*.

Spitters, B. and van der Weegen, E. (2011). Type classes for mathematics in type theory. *MSCS*, 21(4):795–825.

Stampoulis, A. (2012). *VeriML: A dependently-typed, user-extensible and language-centric approach to proof assistants.* PhD thesis, Yale University.

Stampoulis, A. and Shao, Z. (2010). VeriML: Typed computation of logical terms inside a language with effects. In *ICFP*.

Stampoulis, A. and Shao, Z. (2012). Static and user-extensible proof checking. In *POPL*.

Strub, P.-Y. (2010). Coq modulo theory. In *CSL*, pages 529–543.

Swierstra, W. (2008). Data types à la carte. *Journal of Functional Programming*, 18:423–436.

The Coq Development Team (2012). *The Coq Proof Assistant Reference Manual – Version V8.4*.

Vafeiadis, V. (2013). Adjustable references. In Blazy, S., Paulin-Mohring, C., and Pichardie, D., editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 328–337. Springer Berlin Heidelberg.

van der Walt, P. and Swierstra, W. (2013). Engineering proof by reflection in Agda. In *Implementation and Application of Functional Languages*.

Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *POPL*, pages 60–76.

Ziliani, B. (2014). Interactive typed tactic programming in the coq proof assistant, thesis material.

Ziliani, B., Dreyer, D., Krishnaswami, N., Nanevski, A., and Vafeiadis, V. Mtac: A monad for typed tactic programming in coq. *Submitted to JFP*, ??(?):??–??