Algebraic Foundations of Proof Refinement

Jonathan Sterling
Carnegie Mellon University

Robert Harper
Carnegie Mellon University

***Abstract*—We contribute a general apparatus for *dependent* tactic-based proof refinement in the LCF tradition, in which the statements of subgoals may express a dependency on the proofs of other subgoals; this form of dependency is extremely useful and can serve as an *algorithmic* alternative to extensions of LCF based on non-local instantiation of schematic variables. Additionally, we introduce a novel behavioral distinction between *refinement rules* and *tactics* based on naturality. Our framework, called Dependent LCF, is already deployed in the nascent RedPRL proof assistant for computational cubical type theory.**

I. Introduction

Interactive proof assistants are at their most basic level organized around some form of *proof refinement apparatus*, which defines the valid transitions between partial constructions, and induces a notion of *proof tactic*. The proof refinement tradition begins with Milner et al.'s *Logic for Computable Functions* [1], [2] and was further developed in Cambridge LCF, HOL and Isabelle [3], as well as the Nuprl family [4], [5], [6]; tactic-based proof refinement is also used in the highly successful Coq proof assistant [7] as well as the new Lean theorem prover [8].

Notation I.1. Throughout this paper, we employ a notational scheme where the active elements of mathematical statements or judgments are colored according to their *mode*, i.e. whether they represent inputs to a mathematical construction or outputs. Terms in input-mode are colored with **blue**, whereas terms in output-mode are colored with **maroon**.

A. Proof refinement and evidence semantics

At the heart of LCF-style proof refinement is the coding of the inference rules of a formal logic into partial functions which take a conclusion and return a collection of premises (subgoals); this is called *backward inference*. Often, this collection of subgoals is equipped with a *validation*, a function that converts evidence of the premises into evidence of the conclusion (which is called *forward inference*).

An elementary example of the LCF coding of inference rules can be found in the right rule for conjunction in an intuitionistic sequent calculus:

$$\dfrac{\overset{\mathcal{D}}{\Gamma \vdash P} \qquad \overset{\mathcal{E}}{\Gamma \vdash Q}}{\Gamma \vdash P \wedge Q}\ \wedge_{\mathfrak{R}}$$

$$\triangledown$$

$$\begin{array}{|ll} \Gamma \vdash A \wedge B & \mapsto \left\langle \begin{array}{l} [\Gamma \vdash A, \Gamma \vdash B], \\ \lambda[\mathcal{D},\mathcal{E}].\ \wedge_{\mathfrak{R}}(\mathcal{D},\mathcal{E}) \end{array} \right\rangle \\ \_ & \mapsto \textbf{raise}\ \mathsf{Fail} \end{array}$$

In the example above, the validation produces exactly a proof of $\Gamma \vdash A \wedge B$ in the intuitionistic sequent calculus, but in general it is not necessary for the meaning of the validations to correspond exactly with formal proofs in the logic being modeled.

In both Edinburgh LCF and Cambridge LCF, the implementation of a proof refinement logic was split between a trusted kernel which implemented *forward inference*, and a separate module which implemented backward inference (refinement) using the forward inference rules as validations [9]. Under this paradigm, every inference rule must be implemented twice, once in each direction.

This approach of duplicating formal rules in forward and backward inference is a *particular* characteristic of the LCF revolution as it actually occurred, rather than a *universal* principle which is to be applied dogmatically in every application of the LCF methodology. On the contrary, it is possible to view a collection of backward inference (refinement) rules as definitive in its own right, independent of any forward inference characterization of a logic; this insight, first achieved in the Nuprl proof assistant [4], enables a sharp distinction between formal proof and evidence (the former corresponding to derivations and the latter corresponding to realizers).

For instance, in the Nuprl family of proof assistants, the refinement logic is a formal sequent calculus for a variant of Martin-Löf's type theory (see [10]), but the validations produce programs in an untyped language of realizers with computational effects [11]. Validations, rather than duplicating the existing refinement rules in forward inference, form the "program extraction" mechanism in Nuprl; this deep integration of extraction into the refinement logic lies in stark contrast with the purely extra-theoretic treatments of program extraction in other proof assistants, such as Coq and Agda.

The ability for the notion of *evidence* to vary independently from the notion of *formal proof* is a major strength of the LCF design: the latter is defined by the assignment of premises to conclusions in trusted refinement rules, whereas the former is defined in the validations of these rules.

Constable first made precise this idea under the name of *computational evidence semantics* [12], [13]. In practice, this technique of separating the proof theory from the evidence semantics can be used to induce (sub)structural and behavioral invariants in programs from arbitrary languages *as they already exist*, regardless of whether these languages possess a sufficiently "proof-theoretic" character (embodied, for instance, in the much acclaimed "decidable typing" property).

This basic asymmetry between *proof* and *computational* *evidence* in refinement logics corresponds closely with the origin of the sequent calculus, which was designed as a theory of derivability for natural deduction. More generally, it reflects the distinction between the demonstration of a judgment and the construction it effects [14], [15].

B. Dependency or barbarism!

An apparently essential part of the LCF apparatus is that each subgoal may be stated independently of evidence for the others; this characteristic, originally identified as part of the *constructible subgoals* property by Harper in his dissertation [16], allows proof refinement to proceed in parallel and in any order.

This restriction, however, raises difficulties in the definition of a refinement logic for dependent type theory, where the statement of one subgoal may depend on the evidence for another subgoal (a state of affairs induced by families of propositions which are indexed in proofs other another proposition). The most salient example of this problem is given by the introduction rule for the dependent sum of a family of types [16, p. 35]. First, consider the standard type *membership* rules, which pose no problem:

$$\frac{M \in A \quad N \in B[M]}{\langle M, N\rangle \in (x : A) \times B[x]} \times^{=}_{\mathfrak{I}}$$

The premises can be stated purely on the basis of the conclusion, because M appears in the statement of the conclusion. However, if we try to convert this to a *refinement* rule, in which we do not have to specify the exact member $\langle M, N\rangle$ in advance, we immediately run into trouble:

$$\frac{\overset{M}{A}\ \textit{true} \quad \overset{N}{B[M]}\ \textit{true}}{\overset{\langle M, N\rangle}{(x : A) \times B[x]}\ \textit{true}} \times_{\mathfrak{R}}$$

Suspending for the moment one's suspicions about the above notation, the fundamental problem is that this inference rule cannot be translated into an LCF rule, because the subgoal $B[M]$ *true* cannot even be stated until it is known what M is, i.e. until we have somehow run the validation for the completed proof of the first subgoal.

To paraphrase the immortal words of the German revolutionary Rosa Luxemburg, we stand at a crossroads: either we shall account for dependency, or we must regress into barbarism.

1) Ignore the problem!: The resolution adopted by the first several members of the Nuprl family was to defer the solution of this problem to a later year, thereby requiring that the user solve the first subgoal in advance of applying the introduction rule. This amounted to defining a countable family of rules $\times_{\mathfrak{R}}\{M\}$, with the term M a parameter:

$$\frac{\overset{*}{M \in A} \quad \overset{N}{B[M]}\ \textit{true}}{\overset{\langle M, N\rangle}{(x : A) \times B[x]}\ \textit{true}} \times_{\mathfrak{R}}\{M\}$$

The subgoals of the above rule are independent of each other, and it can therefore easily be coded as an LCF rule. However, this has come at great cost, because the user of the proof assistant is no longer able to produce the proof of A using refinement, and must choose some witness before proceeding. This is disruptive to the interactive and incremental character of refinement proof, and we believe that the time is ripe to revisit this question.

2) Non-local unification: choosing barbarism: The most commonly adopted solution is to introduce a notion of *existential variable* and solve these non-locally in a proof via unification or spontaneous resolution. The basic idea is that you proceed with the proof of the second premise without yet knowing what M is, and then at a certain point, it will hopefully become obvious what it *must* be, on the basis of some goal in the subtree of this premise:

$$\frac{\overset{*}{?\mathfrak{m} \in A} \quad \overset{N}{B[?\mathfrak{m}]}\ \textit{true}}{\overset{\langle ?\mathfrak{m}, N\rangle}{(x : A) \times B[x]}\ \textit{true}}$$

An essential characteristic of this approach is that when it is known what $?\mathfrak{m}$ must be, this knowledge propagates immediately to all nodes in the proof that mention it. This is a fundamentally *imperative* operation, and makes it very difficult to reason about proof scripts; moreover, it complicates the development of a clean and elegant semantics (denotational or otherwise) for proof refinement. At the very least, a fully formal presentation of the transition rules for such a system will be very difficult, both to state and to understand.

Another potential problem with using unification to solve goals is that one must be very cautious about how it is applied: using unification *uncritically* in the refinement apparatus can change the character of the object logic. One of the more destructive examples of this happening in practice was when overzealous use of unification in the Agda proof assistant [17] led to the injectivity of type constructors being derivable, whence the principle of the excluded middle could be refuted.

Uncritical use of unification in type theories like that of Nuprl might even lead to inconsistency, since greater care must be taken to negotiate the intricacies of subtyping and functionality which arise under the semantic typing discipline. As Cockx, Devriese and Piessens point out in their recent refit of Agda's unification theory, unification must be integrated tightly with the object logic in order to ensure soundness [18].

3) Our solution: adopt a dependent refinement discipline: Taking a step back, there are two more things that should make us suspicious of the use of existential variables and unification in the above:

1) We *still* do not exhibit A using refinement rules, but rather simply hope that at some point, we shall happen upon (by chance) a suitable witness for which we can check membership. In many cases, it will be possible to inhabit $B[?\mathfrak{m}]$ regardless of whether $?\mathfrak{m}$ has any referent, which will leave us in much the same position as we were in

Nuprl: we must cook up some arbitrary A independently without any help from the refinement rules.

2) It is a basic law of type-theoretic praxis that whatever structure exists at the propositional level should mirror a form of construction that already exists at the judgmental level, adding to it only those characteristics which distinguish truth from the evidence of an arbitrary judgment (e.g. functionality, local character, fibrancy, etc.). In this case, the use of existential variables and unification seems to come out of nowhere, whereas we would expect a dependent sum at the type level to be defined in terms of some notion of dependent sum at the judgmental level.

With the above in mind, we are led to try and revise the old LCF apparatus to support a *dependent* refinement discipline, relaxing the constructible subgoals property in such a way as to admit a coding for the rule given at the beginning of this section, $\times_{\mathfrak{R}}$.

II. Survey of Related Works

A. Semantics of proof refinement in Nuprl

The most detailed denotational semantics for tactic-based proof refinement that we are aware of is contained in Todd Knoblock's PhD thesis [19] vis-à-vis the Nuprl refinement logic. Knoblock's purpose was to endow Nuprl with a tower of metalogics, each reflecting the contents of the previous ones, enabling internal reasoning about the proof refinement process itself; this involved specifying semantic domains for (reflected) Nuprl judgments, proofs and proof tactics in the Nuprl logic.

A detailed taxonomy of different forms of proof tactic was considered, including *search tactics* (analogous to *valid tactics* in LCF), *partial tactics* (tactics whose domain of applicability is circumscribed a priori), and *complete tactics* (partial tactics which produce no subgoals when applied in their domain of applicability).

Spiritually, our apparatus is most closely related to Knoblock's contributions, in light of the purely semantical and denotational approach which he pursued. The fact that Knoblock's semantic universe for proof refinement was the Nuprl logic itself enabled users of the system to prove internally the general effectiveness of a tactic on some class of Nuprl sequents once and for all; then, such a tactic could be applied without needing to be executed.

B. Isabelle as a meta-logical framework

Isabelle, a descendent of Cambridge LCF, is widely considered the gold standard in tactic-based proof refinement today; at its core, it is based on a version of Intuitionistic higher-order logic called Isabelle/Pure, which serves as the logical framework for all other Isabelle theories, including the famous Isabelle/HOL theory for classical higher-order logic.

In Isabelle, tactics generally operate on a full proof state (as opposed to the "local" style pioneered in LCF); a tactic is a partial function from proof states to *lazy sequences* of proof states. Note that the sequences here are used to accomodate sophisticated, possibly non-deterministic search schemata. In contrast to other members of the LCF family, the notion of *validation* has been completely eschewed; this has relieved Isabelle of the need to duplicate rules in both forward and backward inference, and simplifies the correctness conditions for a tactic.

Isabelle does not address the issue of dependent refinement, instead relying heavily on instantiation of schematic variables by higher-order unification. Because HOL is Isabelle's main theory, this is perhaps not so bad a state of affairs, since the most compelling uses of dependent refinement arise in proof systems for dependent type theory, in which a domain of discourse is identified with the inhabitants of a type or proposition. In non-type-theoretic approaches to logic, a proposition is proved using inference rules, whereas an element of the domain of discourse is constructed according to a *grammar*.

With that said, instantiation of schematic variables at higher type in Isabelle is not always best done by unification alone, and often requires manual guiding. A pathological case is the instantiation of predicates shaped like $?\mathfrak{m}[?\mathfrak{n}]$, where it is often difficult to proceed without further input [20, §4.2.2].

Under a dependent refinement discipline, however, the instantiation of schematic variables ranging over higher predicates can be pursued with the rules of the logic as a guide, in the same way that all other objects under consideration are constructed: in the validations of backward inference rules.

This insight, which is immanent in the higher-order propositions-as-types principle, is especially well-adapted for use in implementations of Martin-Löf-style dependent type theory, where it is often the case that the logic can guide the instantiation of a predicate variable in ways that pure unification cannot. This is essentially the difference between a direct and *algorithmic* treatment of synthetic judgment, and its *declarative* simulation as analytic judgment [21].

We stress that higher-order unification is an extremely useful technique, but it appears to *complement* rather than obviate a proper treatment of dependent refinement. Though it is not the topic of this paper, we believe that a combination of the two techniques would prove very fruitful.

C. OLEG & Epigram: tactics as admissible rules

Emanating from Conor McBride's dissertation work is a unique approach to proof elaboration which has been put to great use in several proof assistants for dependent type theory including OLEG, Epigram and Idris [22], [23], [24], [25], [26]. A more accessible introduction to McBride's methodology is given in [27].

Like ours, McBride's approach rests upon the specification of judgments in context which are stable under context substitution; crucially, McBride's apparatus was the first treatment of proof refinement to definitively rule out invalid scoping of schematic variables, a problem which plagued early implementations of "holes". In particular, McBride's framework is well suited to the development of type checkers, elaborators and type inferencers for formal type theories and programming languages.

One of the ways in which our contribution differs from McBride's system is that we treat rules of inference *algebraically*, i.e. as first-class entities in a semantic domain; then, following the LCF tradition, we develop a menagerie of combinators (tacticals) by which we can combine these rules into composite proofs. In this sense, our development is a treatment of *derivability* relative to a trusted basis of (backward) inference rules.

McBride's approach is, on the contrary, to take the trusted basis of inference rules as given *ambiently* rather than algebraically, and then to develop a theory of proof tactic based on *admissibility* with respect to this basis theory.

Both approaches have been shown to be useful in implementations of type theory, and we hope to better understand through practice the various trade-offs which they induce.

D. Typed tactics with Mtac

The idea of capturing proof tactics using a monad is put to use in the Mtac language, which is an extension to Coq which supports typed tactic programming [28].

E. Dependent subgoals in Coq 8.5

In his PhD thesis, Arnaud Spiwack addressed the lack of dependent refinement in the Coq proof assistant by redesigning its tactic apparatus [29]; Spiwack's efforts culminated in the release of Coq 8.5 in early 2016, which incorporates his new design, including support for "dependent subgoals" (which we call *dependent refinement*) and tactics that operate simultaneously on multiple goals (which we call *multitactics*).

Spiwack's work centered around a new formulation of LCF-style tactics which was powerful enough to support a number of useful features, including backtracking and subgoals expressing dependencies on other subgoals; the latter is effected through an imperative notion of existential variable (in contrast to the purely functional semantics for subgoal dependency that we give in this paper).

F. Our contributions

We take a very positive view of Spiwack's contributions in this area, especially in light of the successful concrete realization of his ideas in the Coq proof assistant. As far as engineering is concerned, we consider Spiwack to have *definitively* resolved the matter of dependent refinement for Coq.

At the same time, we believe that there is room for a mathematical treatment of dependent refinement which abstracts from the often complicated details of real-world implementations, and is completely decoupled from specific characteristics of a particular logic or proof assistant; our experience suggests that the development of a semantics for proof refinement along these lines can also lead to a cleaner, more reusable concrete realization.

Our contribution is a precise, compositional and purely functional semantics for dependent proof refinement which is also immediately suitable for implementation; we have also introduced a novel behavioral distinction between refinement rules and tactics based on *naturality*. Our framework is called Dependent LCF, and our Standard ML implementation has already been used to great effect in the new **RedPRL** proof assistant for computational cubical type theory [6], [30].

III. Preliminaries

A. Lawvere Theories

We wish to study the algebraic structure of dependent proof refinement for a fixed language of constructions or evidence. To abstract away from the bureaucratic details of a particular encoding, we will work relative to some multi-sorted Lawvere theory $\mathbb{T}$, a strictly associative cartesian category whose objects can be viewed as sorts or contexts (finite products of sorts) and whose morphisms may be viewed as terms or substitutions.

Definition III.1 (Lawvere theory)**.** To define the notion of a multi-sorted Lawvere theory, we fix a set of sorts $\mathcal{S}$; let $\mathcal{S}_\times$ be the free strict associative cartesian category on $\mathcal{S}$. Then, an $\mathcal{S}$-sorted Lawvere theory is a strictly associative cartesian category $\mathbb{T}$ equipped with a cartesian functor $\mathsf{k} : \mathcal{S}_\times \to \mathbb{T}$.

We will write $\Gamma, \Delta, \Xi : \mathbb{T}$ for the objects of $\mathbb{T}$ and $\mathfrak{a}, \mathfrak{b}, \mathfrak{c} : \Gamma \Rightarrow \Delta$ for its morphisms. We will freely interchange "context" and "sort" (and "substitution" and "term") when one is more clear than the other. We will sometimes write $\Gamma, x : \Delta$ for the context $\Gamma \times \Delta$, and then use x elsewhere as the canonical projection $\mathsf{p} : \Gamma \times \Delta \Rightarrow \Delta$.

Remark III.2 (Second-order theories)*.* In the simplest case, a Lawvere theory $\mathbb{T}$ forms the category of contexts and substitutions for some *first-order* language. However, as Fiore and Mahmoud have shown, this machinery scales up perfectly well to the case of *second-order* theories (theories with binding) [31].

In that case, the objects are contexts of second-order variables associated to *valences* (a sort together with a list of sorts of bound variables), and the maps are second-order substitutions; when the output of a map is a single valence $\vec{\sigma}.\tau$, the map can be read as a term binder.

One of our reasons for specifying no more about $\mathbb{T}$ than we have done so far is to ensure that our apparatus generalizes well to the case of second-order syntax, which is what is necessary in nearly every concrete application of this work.

In what follows, we will often refer to variables as *schematic variables* in order to emphasize that these are variables which range over evidence in the proof refinement apparatus, as opposed to variables from the object logic. In the first-order case, all variables are schematic variables; in the second-order case, the second-order variables (called *metavariables* by Fiore et al) are the schematic variables, and the object variables are essentially invisible to our development.

B. Questions Concerning a Semantic Universe

Our main task is to define a semantic universe in which we can build objects indexed in $\mathbb{T}$, which respect substitutions of schematic variables. Some kind of presheaf category, then, seems to be what we want—and then proof refinement rules should be natural transformations in this presheaf category.

The question of which indexing category to choose is a subtle one; in order to construct our *proof states monad*, we will need to work with something like presheaves over $\mathbb{T}$, i.e. "variable sets" which implement all substitutions. However, most interesting *refinement rules* that we wish to define will not commute with substitutions in all cases, which is the content of *naturality*. This corresponds to the fact that a refinement rule may fail to be applied if there is a schematic variable in a certain position, but may succeed if that variable is substituted for by some suitable term.

Essentially the same problem arises in the context of coalgebraic logic programming [32], [33]; several methods have been developed to deal with this behavior, including switching to an order-enriched semantic universe and using *lax natural transformations* for the operational semantics; another approach, called *saturation* involves trivializing naturality by treating $\mathbb{T}$ as a discrete category $|\mathbb{T}|$ (by taking the free category on the set of objects of $\mathbb{T}$), and then "saturating" constructions along the adjunction $i^* : \widehat{\mathbb{T}} \to \widehat{|\mathbb{T}|} \dashv i_* : \widehat{|\mathbb{T}|} \to \widehat{\mathbb{T}}$.

In the context of general dependent proof refinement, the lax semantics are the most convenient; we will apply a variation on this approach here, which also incorporates discrete reindexing for interpreting tactics.

Notation III.3. Following the notation of the French school [34, p. 25], we write $\widehat{\mathbb{X}}$ for the category of presheaves $\mathbb{SET}^{\mathbb{X}^{op}}$ on a category $\mathbb{X}$.

Notation III.4. We will write $\mathsf{Ctx} : \widehat{\mathbb{T}}$ for the constant presheaf of $\mathbb{T}$-objects, $\mathsf{Ctx}(\Gamma) \triangleq \mathsf{ob}(\mathbb{T})$. We may also write $\Gamma \Vdash X : F$ to mean $X \in F(\Gamma)$ when $F : \widehat{\mathbb{T}}$.

We will frequently have need for a presheaf of terms of an appropriate sort relative to a particular context, $(\Gamma \vdash \Delta) : \widehat{\mathbb{T}}$. This we can define informally as follows:

$$\frac{\Xi, \Gamma \vdash a : \Delta}{\Xi \Vdash a : (\Gamma \vdash \Delta)}$$

Formally, this is the exponential $\mathcal{H}(\Delta)^{\mathcal{H}(\Gamma)}$ with $\mathcal{H}(-) : \mathbb{T} \to \widehat{\mathbb{T}}$ the Yoneda embedding; this perspective is developed in Appendix A.

C. Presheaves and lax natural transformations

Let $\mathbb{POS}$ be the order-enriched category of partially ordered sets; arrows are endowed with an order by pointwise approximation: $f \preccurlyeq g$ iff $x \preccurlyeq y \Rightarrow f(x) \preccurlyeq g(y)$.

Definition III.5 (Presheaves and lax natural transformations)**.** A $\mathbb{POS}$*-valued presheaf* on $\mathbb{C}$ is a functor from $\mathbb{C}^{op}$ into $\mathbb{POS}$.

A *lax natural transformation* ϕ between two such presheaves P, Q is a collection of components whose naturality square commutes up to approximation in the following sense:

$$\begin{tikzcd} P(d) \arrow[r, "P(f)"] \arrow[d, "\phi_d"'] \arrow[dr, phantom, "\preccurlyeq"] & P(c) \arrow[d, "\phi_c"] \\ Q(d) \arrow[r, "Q(f)"'] & Q(c) \end{tikzcd}$$

In other words, we need have only that $Q(f) \circ \phi_d \preccurlyeq \phi_c \circ P(f)$ in the above diagram.

Notation III.6. We will write $\widetilde{\mathbb{C}}$ for the category of $\mathbb{POS}$-valued presheaves and lax natural transformations on $\mathbb{C}$. *Note that the presheaves themselves are strict; only natural transformations between the presheaves are lax.* A $\mathbb{SET}$-valued presheaf $P : \widehat{\mathbb{C}}$ can be silently regarded as a $\mathbb{POS}$-valued presheaf $P : \widetilde{\mathbb{C}}$ by endowing each fiber with the discrete order.

IV. A Framework for Proof Refinement

We will now proceed to develop the Dependent LCF theory by specifying the semantic objects under consideration, namely *judgment structures*, *proof states*, *refinement rules*, and *proof tactics*.

A. Judgment Structures

A *judgment* is an intention toward a particular form of construction; that is, a form of judgment is declared by specifying the $\mathbb{T}$-object (that is, sort or context) which classifies what it intends to construct. It is suggestive to consider this object the *output* of a judgment, in the sense that if the judgment is derived, it will emit a substitution of the appropriate sort which can be used elsewhere.

For example, in a Martin-Löf-style treatment of intuitionistic logic (see [35]), the judgment P *true* constructs objects of sort exp, where exp is the sort of expressions in a basic programming language.

In the dependent proof refinement discipline, the statement of a judgment may be interrupted by a schematic variable (e.g. the judgment $P(x)$ *true*), which ranges over the evidence of some other judgment, and may be substituted for by a term of the appropriate sort. This behavior captures the ubiquitous case of existential instantiation, where we have a predicate applied to a schematic variable which stands for an element of the domain of discourse, to be refined in the course of verifying another judgment.

To make this precise, we can define a notion of "judgment structure" as a collection of "judgments" which varies over contexts and substitutions, along with an assignment of sorts to judgments: the sort assigned to a judgment is then the sort of object that the judgment intends to construct.

Then, a homomorphism between judgment structures would be a natural transformation of presheaves which preserves sort assignments. In Section IV-C will capture *refinement rules* as homomorphisms between certain kinds of judgment structures.

Definition IV.1 (Judgment structures)**.** Formally, we define the category of judgment structures $\mathbb{J}$ on $\mathbb{T}$ as the slice category $\widetilde{\mathbb{T}}/\mathsf{Ctx}$; expanding definitions, an object $J : \mathbb{J}$ is a presheaf $J : \widetilde{\mathbb{T}}$ together with an assignment $\pi_J : J \to \mathsf{Ctx}$. Then, a map from J_0 to J_1 is a natural transformation that preserves π, in the sense that the following triangle commutes:

$$\begin{tikzcd} J_0 \arrow[rr, "\phi"] \arrow[dr, "\pi_{J_0}"'] & & J_1 \arrow[dl, "\pi_{J_1}"] \\ & \mathsf{Ctx} & \end{tikzcd}$$

It will usually be most clear to define a judgment structure inductively in syntactic style, by specifying the (meta)judgment $\Gamma \Vdash X : J \rightsquigarrow \Delta$, pronounced "$X$ is a J-judgment in context Γ, constructing a substitution for Δ", which will mean $X \in J(\Gamma)$ and $\pi_J^\Gamma(X) = \Delta$.

Remark IV.2. It is easiest to understand this judgment in the special case where Δ is a unary context $x : \tau$; then the judgment means that the construction induced by the J-judgment X (i.e. it's "output") will be a term of sort τ; in general, we allow multiple outputs to a judgment, which corresponds to the case that Δ is a context with multiple elements.

When defining a judgment structure J, its order will be specified using the (meta)judgment $\Gamma \Vdash X \preccurlyeq Y : J$ (pronounced "X approximates Y as a J-judgment in context Γ"), which presupposes both $\Gamma \Vdash X : J \rightsquigarrow \Delta$ and $\Gamma \Vdash Y : J \rightsquigarrow \Delta$; unless otherwise specified, when defining a judgment structure, we usually assume the discrete order.

In practice, a collection of inference rules in (meta)judgments $\Gamma \Vdash X : J \rightsquigarrow \Delta$ and $\Gamma \Vdash X \preccurlyeq Y : J$ should be understood as defining the *least* judgment structure closed under those rules, unless otherwise stated.

Example IV.3 (Cost dynamics of basic arithmetic)**.** A simple example of a judgment structure can be given by considering the cost dynamics for a small language of arithmetic expressions [36, Ch. 7.4].

We will fix two syntactic sorts, num and exp; num will be the sort of numerals, and exp will be the sort of arithmetic expressions; the Lawvere theory generated from these sorts and suitable operations that we define in Figure 1 will be called $\mathbb{A}$. We will write $\mathbb{J}_\mathbb{A}$ for the category of judgment structures over $\mathbb{A}$, namely the slice category $\widetilde{\mathbb{A}}/\mathsf{Ctxs}_\mathbb{A}$.

Then, we define a *judgment structure* $J_\mathbb{A} : \mathbb{J}_\mathbb{A}$ for our theory by specifying the following forms of judgment:

1) $\mathsf{eval}(e)$ means that the arithmetic expression $e : \mathsf{exp}$ can be evaluated; its evidence is the numeral value of e and the cost k of evaluating e (i.e. the number of steps taken).
2) $\mathsf{add}(m; n)$ means that the numerals $m, n : \mathsf{num}$ can be added; the evidence of this judgment is the numeral which results from their addition.

The judgment structure $J_\mathbb{A}$ summarized above is defined schematically in Figure 1.

We will use the above as our running example, and after we have defined a suitable notion of *refinement rule*, we will define the appropriate rules for the judgment structure $J_\mathbb{A}$.

B. Telescopes and Proof States

1) Telescopes: An ordered sequence of judgments in which each induces a variable of the appropriate sort which the rest of the sequence may depend on is called a *telescope* [37]. The notion of a telescope will be the primary aspect in our definition of proof states later on, where it will specify the collection of judgments which still need to be proved.

Remark IV.4. If "dependent refinement" were replaced with "independent refinement", then the telescope data-structure could be replaced with lists. This design choice characterizes the LCF family of proof refinement apparatus.

We intend telescopes to themselves be an endofunctor on judgment structures, analogous to an iterated dependent sum; we will define the judgment structure endofunctor $\mathsf{Tl} : \mathbb{J} \to \mathbb{J}$ inductively.

For a judgment structure $J : \mathbb{J}$, a J-telescope is either $*$ (the empty telescope), or $x : X. \Psi$ with X a J-judgment and Ψ a J-telescope with x bound; the sort that is synthesized by a telescope is the product of the sorts synthesized by its constituent judgments. The precise rules for forming telescopes are given in Figure 2.

Note how in the above, we have used variable binding notation; formally, as can be seen in Appendix A, this corresponds to exponentiation by a representable functor, which is the usual way to account for variable binding in higher algebra. To be precise, given a presheaf $P : \widehat{\mathbb{T}}$ and a variable context $\Gamma : \mathbb{T}$, by exponentiation we can construct a new presheaf $P^{\mathcal{H}(\Gamma)} : \widehat{\mathbb{T}}$ (with $\mathcal{H}(-) : \mathbb{T} \to \widehat{\mathbb{T}}$ the Yoneda embedding) whose values are *binders* that close over the variables in Γ.

Henceforth, we are justified in adopting the *variable convention*, by which terms are identified up to renamings of their bound variables.

2) Proof States Monad: We define another endofunctor on judgment structures for proof states, $\mathsf{St} : \mathbb{J} \to \mathbb{J}$. Fixing a judgment structure $J : \mathbb{J}$, there are three ways to form a J-proof state:

1) When Ψ is a J-telescope that synthesizes sorts Ξ, and $\mathfrak{a}$ is a substitution from Ξ to Δ, then $\Psi \rhd \mathfrak{a} :: \Delta$ is a proof state; as an object in a judgment structure, this proof state synthesizes Δ. Intuitively, Ψ is the collection of subgoals and $\mathfrak{a}$ is the *validation* which constructs evidence on the basis of the evidence for those subgoals. We will usually write $\Psi \rhd \mathfrak{a}$ instead of $\Psi \rhd \mathfrak{a} :: \Delta$ when it is clear from context.
2) For any Δ, ✠$[\Delta]$ is a proof state that synthesizes Δ; this state represents *persistent failure*. We will always write ✠ instead of ✠$[\Delta]$.
3) For any Δ, $\bot[\Delta]$ is a proof state that synthesizes Δ; this state represents what we call *unsuccess*, or failure which may not be persistent. As above, we will always write $\bot$ instead of $\bot[\Delta]$.

Moreover, we impose the approximation ordering $\Gamma \Vdash \bot[\Delta] \preccurlyeq \Psi \rhd \mathfrak{a} :: \Delta : \mathsf{St}(J)$ and $\Gamma \Vdash \bot[\Delta] \preccurlyeq$ ✠$[\Delta] : \mathsf{St}(J)$.

The difference between *unsuccess* and *failure* is closely related to the difference between the statements "It is not the case that P is true" and "P is false" in constructive mathematics. In the context of proof refinement in the presence of schematic variables, it may be the case that a rule does not apply at first, but following a substitution, it does apply; capturing this case is the purpose of introducing the $\bot$ proof state.

Again, the precise rules for forming proof states are given in Figure 2.

$$\frac{\overline{n} \in \mathbb{N}}{\Gamma \vdash_{\mathbb{A}} \overline{n} : \mathsf{num}} \qquad \frac{\Gamma \vdash_{\mathbb{A}} n : \mathsf{num}}{\Gamma \vdash_{\mathbb{A}} \mathsf{num}[n] : \mathsf{exp}} \qquad \frac{\Gamma \vdash_{\mathbb{A}} e_1 : \mathsf{exp} \quad \Gamma \vdash_{\mathbb{A}} e_2 : \mathsf{exp}}{\Gamma \vdash_{\mathbb{A}} e_1 + e_2 : \mathsf{exp}}$$

$$\boxed{J_{\mathbb{A}} : \mathbb{J}} \qquad \frac{\Gamma \vdash_{\mathbb{A}} e : \mathsf{exp}}{\Gamma \Vdash \mathsf{eval}(e) : J_{\mathbb{A}} \rightsquigarrow x_c : \mathsf{num}, x_v : \mathsf{num}} \qquad \frac{\Gamma \vdash_{\mathbb{A}} m : \mathsf{num} \quad \Gamma \vdash_{\mathbb{A}} n : \mathsf{num}}{\Gamma \Vdash \mathsf{add}(m;n) : J_{\mathbb{A}} \rightsquigarrow \mathsf{num}}$$

Fig. 1. An example theory $\mathbb{A}$ and judgment structure $J_{\mathbb{A}}$.

$$\boxed{\mathsf{Tl} : \mathbb{J} \to \mathbb{J}} \qquad \frac{}{\Gamma \Vdash * : \mathsf{Tl}(J) \rightsquigarrow \cdot} \qquad \frac{\Gamma \Vdash X : J \rightsquigarrow \Delta \quad \Gamma, x : \Delta \Vdash \Psi_x : \mathsf{Tl}(J) \rightsquigarrow \Xi}{\Gamma \Vdash x : X.\Psi_x : \mathsf{Tl}(J) \rightsquigarrow x : \Delta, \Xi} \qquad \text{(Telescopes)}$$

$$\boxed{\mathsf{St} : \mathbb{J} \to \mathbb{J}} \qquad \frac{\Gamma \Vdash \Psi : \mathsf{Tl}(J) \rightsquigarrow \Xi \quad \Gamma \Vdash \mathfrak{a} : (\Xi \vdash \Delta)}{\Gamma \Vdash (\Psi \triangleright \mathfrak{a} :: \Delta) : \mathsf{St}(J) \rightsquigarrow \Delta} \qquad \frac{}{\Gamma \Vdash \maltese[\Delta] : \mathsf{St}(J) \rightsquigarrow \Delta} \qquad \frac{}{\Gamma \Vdash \bot[\Delta] : \mathsf{St}(J) \rightsquigarrow \Delta} \qquad \text{(Proof States)}$$

$$\frac{\Gamma \Vdash S : \mathsf{St}(J) \rightsquigarrow \Delta}{\Gamma \Vdash \bot[\Delta] \preccurlyeq S : \mathsf{St}(J)} \qquad \text{(Approximation)}$$

$$\begin{aligned} \Psi \triangleright \mathfrak{a} &\triangleq \Psi \triangleright \mathfrak{a} :: \Delta \\ \maltese &\triangleq \maltese[\Delta] \\ \bot &\triangleq \bot[\Delta] \end{aligned} \qquad \text{(Notations)}$$

Fig. 2. Definitions of telescopes and proof states. Except where otherwise specified, the discrete order is assumed in all definitions above.

Notation IV.5. We will write $\Psi \ldots \Psi'$ for the concatenation of two telescopes, where Ψ' may have variables from $\pi^{\Gamma}_{\mathsf{Tl}(J)}(\Psi)$ free. Likewise, we will write $\Psi' \ldots S$ to mean $\Psi' \ldots \Psi \triangleright \mathfrak{a}$ when $S \equiv \Psi \triangleright \mathfrak{a}$, and $\maltese$ when $S \equiv \maltese$, and $\bot$ when $S \equiv \bot$.

We can instantiate a monad structure on proof states, which will abstractly implement the composition of refinement rules, and which will play a crucial part in the *identity* and *sequencing* tacticals from LCF.

$$\mathbf{1} \xrightarrow{\eta} \mathsf{St} \xleftarrow{\mu} \mathsf{St} \circ \mathsf{St}$$

The unit and multiplication operators are defined by the following equations:

$$\begin{aligned} \eta_\Gamma(X) &= x : X. * \triangleright x \\ \mu_\Gamma(* \triangleright \mathfrak{a}) &= * \triangleright \mathfrak{a} \\ \mu_\Gamma(x : (\Psi_x \triangleright \mathfrak{a}_x).\underline{\Psi} \triangleright \mathfrak{a}) &= \Psi_x \ldots \mu_{\Gamma, \pi^{\Gamma}_{\mathsf{Tl}(J)}(\Psi_x)}(\underline{\Psi} \triangleright \mathfrak{a})[\mathfrak{a}_x / x] \\ \mu_\Gamma(x : \maltese.\underline{\Psi} \triangleright \mathfrak{a}) &= \maltese \\ \mu_\Gamma(x : \bot.\underline{\Psi} \triangleright \mathfrak{a}) &= \bot \\ \mu_\Gamma(\maltese) &= \maltese \\ \mu_\Gamma(\bot) &= \bot \end{aligned}$$

In the interest of clear notation, we have used Ψ to range over $\mathsf{Tl}(J)$ and $\underline{\Psi}$ to range over $\mathsf{Tl}(\mathsf{St}(J))$.

Theorem IV.6. *Proof states form a monad on $\mathbb{J}$, i.e. the following diagrams commute:*

$$\begin{tikzcd} \mathsf{St} \arrow[r, "\eta"] \arrow[dr, "\mathbf{1}"'] & \mathsf{St} \circ \mathsf{St} \arrow[d, "\mu"] & \mathsf{St} \arrow[l, "\mathsf{St}(\eta)"'] \arrow[dl, "\mathbf{1}"] \\ & \mathsf{St} & \end{tikzcd}$$

$$\begin{tikzcd} \mathsf{St} \circ \mathsf{St} \circ \mathsf{St} \arrow[r, "\mu"] \arrow[d, "\mathsf{St}(\mu)"'] & \mathsf{St} \circ \mathsf{St} \arrow[d, "\mu"] \\ \mathsf{St} \circ \mathsf{St} \arrow[r, "\mu"] & \mathsf{St} \end{tikzcd}$$

Proof. By nested induction; see Appendix B. □

C. Refinement Rules and Lax Naturality

We can now directly define the notions of *refinement rules*, *tactics* and *multitactics* as judgment structure homomorphisms.

Definition IV.7 (Refinement rules)**.** For judgment structures $J_0, J_1 : \mathbb{J}$, a *refinement rule* from J_0 to J_1 is a $\mathbb{J}$-homomorphism $\rho : \mathsf{Rule}(J_0, J_1) \triangleq J_0 \to \mathsf{St}(J_1)$. Unpacking definitions, ρ is a *lax* natural transformation between the underlying presheaves of J and $\mathsf{St}(J_1)$ which preserves the projection π.

Usually, one works with homogeneous refinement rules $\rho : J \to \mathsf{St}(J)$, which can be called J-rules.

The ordered character of the $\mathsf{St}(J)$ judgment structure is crucial in combination with lax naturality; it is this which allows us to define a refinement rule which neither succeeds nor fails when it encounters a schematic variable that is blocking its applicability: that is, it does not commit to failure under all possible instantiations for that variable.

Full naturality would entail that a refinement rule commute with all substitutions from $\mathbb{T}$, whereas lax naturality only requires this square to commute up to approximation—that is, for $P, Q : \widetilde{\mathbb{T}}$, $\phi : P \to Q$ and $\mathfrak{a} : \Gamma \Rightarrow \Delta$, rather than the identity $Q(\mathfrak{a}) \circ \phi_\Delta = \phi_\Gamma \circ P(\mathfrak{a})$, we require only the approximation $Q(\mathfrak{a}) \circ \phi_\Delta \preccurlyeq \phi_\Gamma \circ P(\mathfrak{a})$.

To understand why this is desirable, let us return to the example of the judgment form P *true*; supposing that our ambient theory $\mathbb{T}$ has a sort for propositions $\mathsf{prop} : \mathbb{T}$ and a sort for program expressions $\mathsf{exp} : \mathbb{T}$, we can define a judgment structure $L : \mathbb{J}$ by specifying a single form of judgment:

$$\frac{\Gamma \vdash P : \mathsf{prop}}{\Gamma \Vdash \boxed{P\ \mathit{true}} : L \rightsquigarrow \mathsf{exp}}$$

Then, our task is to code the inference rules of our logic, such as the following,

$$\frac{P\ \mathit{true}}{P \vee Q\ \mathit{true}}\ \vee I_1$$

as *refinement rules* for the judgment structure L. Such a rule is a judgment structure homomorphism $\wedge I : L \to \mathsf{St}(L)$ in $\mathbb{J}$; at first, we might try and write the following:

$$\vee I_1^\Gamma \left| \begin{array}{ll} P \vee Q\ \mathit{true} & \Mapsto x : P\ \mathit{true}\,.\, * \triangleright \mathsf{inl}(x) \\ \_ & \Mapsto \maltese \end{array}\right. \qquad (*)$$

This, however, is not a well-formed definition, because it does not commute in any sense with substitutions: for instance $\vee I_1^{\Gamma, x:\mathsf{prop}}(x\ \mathit{true})[P \vee Q/x] = \maltese$, whereas $\vee I_1^\Gamma(x\ \mathit{true}[P \vee Q/x]) = x : P\ \mathit{true}\,.\, * \triangleright \mathsf{inl}(x)$. However, recall that refinement rules are subject only to lax naturality, i.e. naturality up to approximation; with a small adjustment to our definition, we can make it commute with substitutions up to approximation:

$$\vee I_1^\Gamma \left| \begin{array}{ll} P \vee Q\ \mathit{true} & \Mapsto x : P\ \mathit{true}\,.\, * \triangleright \mathsf{inl}(x) \\ x & \Mapsto \bot \\ \_ & \Mapsto \maltese \end{array}\right.$$

Indeed, the above definition is well-formed (because we have $\Gamma \Vdash \bot \preccurlyeq x : P\ \mathit{true}\,.\, * \triangleright \mathsf{inl}(x) : \mathsf{St}(L)$). This example reflects the difference between *unsuccess* and *failure*: namely, the introduction rule above does not *yet* apply to the goal x *true*, but supposing x were substituted for by some $P \vee Q$, it would then apply. On the other hand, the rule does not apply at all to the goal $P \wedge Q$ *true*.

Example IV.8 (Refinement rules for cost dynamics)**.** It will be instructive to consider a more sophisticated example. Resuming what we started in Example IV.3, we are now equipped to encode formal refinement rules for the judgment structure $J_{\mathbb{A}} : \mathbb{J}$ defined in Figure 1.

We will implement the cost dynamics using two evaluation rules and one rule to implement the addition of numerals:

$$\begin{aligned} \mathbf{eval}_{\mathsf{num}} &: J_{\mathbb{A}} \to \mathsf{St}(J_{\mathbb{A}}) \\ \mathbf{eval}_{+} &: J_{\mathbb{A}} \to \mathsf{St}(J_{\mathbb{A}}) \\ \mathbf{add} &: J_{\mathbb{A}} \to \mathsf{St}(J_{\mathbb{A}}) \end{aligned}$$

First, let's consider what these rules would look like informally on paper, writing $e \Downarrow^k n$ for the statement that the judgment $\mathsf{eval}(e)$ obtains, synthesizing cost k and numeral n, and writing $m + n \equiv o$ for the statement that the judgment $\mathsf{add}(m; n)$ obtains, synthesizing numeral o:

$$\frac{}{\mathsf{num}[n] \Downarrow^0 n}\ \mathbf{eval}_{\mathsf{num}}$$

$$\frac{\begin{array}{ccc} e_1 \Downarrow^{k_1} n_1 & k_1 + k_2 \equiv k_{12} & \\ e_2 \Downarrow^{k_2} n_2 & \overline{1} + k_{12} \equiv k & n_1 + n_2 \equiv n \end{array}}{e_1 + e_2 \Downarrow^k n}\ \mathbf{eval}_{+}$$

$$\frac{}{\overline{m} + \overline{n} \equiv \overline{m + n}}\ \mathbf{add}$$

In keeping with standard practice and notation, in the informal definition of a refinement rule, clauses for failure and unsuccess are elided. When we code these rules as judgment structure homomorphisms, we add these clauses in the appropriate places, as can be seen from the formal definitions of $\mathbf{eval}_{\mathsf{num}}$, $\mathbf{eval}_{+}$ and $\mathbf{add}$ in Figure 3.

D. Combinators for Derived Refinement Rules

We can develop a menagerie of combinators for refinement rules which allow the development of *derived* rules. More generally, given a basis of refinement rules, it is possible to characterize the space of *derivable rules* by the closure of this basis under certain combinators (see Appendix D).

First, we will begin by defining an auxiliary judgment structure $J_{\times\mathbb{N}} : \mathbb{J}$, which tags J-judgments with an index:

$$\boxed{J_{\times\mathbb{N}} : \mathbb{J}} \qquad \frac{\Gamma \Vdash X : J \rightsquigarrow \Delta \quad i \in \mathbb{N}}{\Gamma \Vdash \langle X, i\rangle : J_{\times\mathbb{N}} \rightsquigarrow \Delta}$$

Next, define an operation to label the subgoals of a proof state with their index:

$$\mathbf{lbl} : \mathsf{St}(J) \to \mathsf{St}(J_{\times\mathbb{N}})$$

$$\mathbf{lbl}_\Gamma \left| \begin{array}{ll} \Psi \triangleright \mathfrak{a} & \Mapsto \mathbf{lbl}_0^\Gamma(\Psi) \triangleright \mathfrak{a} \\ \bot & \Mapsto \bot \\ \maltese & \Mapsto \maltese \end{array}\right.$$

where

$$\mathbf{lbl}_i : \mathsf{Tl}(J) \to \mathsf{Tl}(J_{\times\mathbb{N}})$$

$$\mathbf{lbl}_i^\Gamma \left| \begin{array}{ll} * & \Mapsto * \\ x : X.\,\Psi & \Mapsto x : \langle X, i\rangle.\,\mathbf{lbl}_{i+1}^{\Gamma, x:\pi_J^\Gamma(X)}(\Psi) \end{array}\right.$$

Let $\vec{\rho}$ range over a list of rules $\rho_i : \mathsf{Rule}(J_0, J_1)$; now we define a derived rule which applies the appropriate rule to a labeled judgment:

$$\mathbf{proj}(\vec{\rho}) : \mathsf{Rule}(J_{\times\mathbb{N}}, J)$$

$$\mathbf{proj}(\vec{\rho})_\Gamma \langle X, i\rangle \triangleq \begin{cases} \rho_i^\Gamma(X) & \textbf{if } i < |\vec{\rho}| \\ \eta_\Gamma(X) & \textbf{otherwise} \end{cases}$$

$$
\begin{array}{l|lcl}
\mathbf{eval}^{\Gamma}_{\mathsf{num}} & \mathsf{eval}(\mathsf{num}[m]) & \Mapsto & * \triangleright [\overline{0}, m] \\
 & \mathsf{eval}(x) & \Mapsto & \bot \\
 & \_ & \Mapsto & \maltese \\
\mathbf{eval}^{\Gamma}_{+} & \mathsf{eval}(e_1 + e_2) & \Mapsto & \left[\begin{array}{l} [x_c, x_v] : \mathsf{eval}(e_1).\ [y_c, y_v] : \mathsf{eval}(e_2). \\ z_c : \mathsf{add}(x_c; y_c).\ z'_c : \mathsf{add}(\overline{1}; z_c).\ z_v : \mathsf{add}(x_v; y_v) \end{array}\right] \triangleright [z'_c, z_v] \\
 & \mathsf{eval}(x) & \Mapsto & \bot \\
 & \_ & \Mapsto & \maltese \\
\mathbf{add}_{\Gamma} & \mathsf{add}(\overline{m}; \overline{n}) & \Mapsto & * \triangleright \overline{m+n} \\
 & \mathsf{add}(\_;\_) & \Mapsto & \bot \\
 & \_ & \Mapsto & \maltese
\end{array}
$$

Fig. 3. Defining refinement rules for $J_{\mathbb{A}}$, the judgment structure of cost dynamics for arithmetic expressions.

Now, given rules $\rho : \mathsf{Rule}(J_0, J_1)$ and $\vec{\rho} : \mathbf{List}(\mathsf{Rule}(J_1, J_2))$, we can define the composition $\rho; \vec{\rho} : \mathsf{Rule}(J_0, J_2)$ using the multiplication operator of the St monad as follows:

$$
\begin{array}{l}
\rho; \vec{\rho} : \mathsf{Rule}(J_0, J_2) \\
\rho; \vec{\rho} \triangleq \mu \circ \mathsf{St}(\mathbf{proj}(\vec{\rho})) \circ \mathbf{lbl} \circ \rho
\end{array}
$$

In Appendix D, we develop a very sophisticated fibered categorical notion of closed refinement logic ("refiner") together with a characterization of derivability relative to refiners.

E. From Refinement Rules to Tactics

Tactics are distinguished from refinement rules in that they are not subject to the lax naturality condition; this is because in tactic-based proof refinement, it is necessary to support tactics which do not commute with substitutions, not even up to approximation—for example, **orelse** and **try** will result in completely proof states before and after a substitution. For this reason, we define a new category $\mathbb{J}_\circ$ of *discrete judgment structures*, for which tactics will be certain homomorphisms.

Definition IV.9 (Discrete Judgment Structures)**.** We now define the category of *discrete judgment structures*, $\mathbb{J}_\circ \triangleq \widehat{|\mathbb{T}|}/\mathsf{Ctx}$, where $|\mathbb{T}|$ is the subcategory of $\mathbb{T}$ which contains only identity arrows, and $\widehat{|\mathbb{T}|} \triangleq |\mathbb{T}|^{\mathsf{op}} \to \mathbb{SET}$.

Then $\mathbb{J}_\circ$ homomorphisms are the same as those of $\mathbb{J}$, except their naturality condition is trivially satisfied for any collection of components; this is because only identity maps occur $|\mathbb{T}|$, so the naturality squares are degenerate. Furthermore, because the components of such homomorphisms are defined in $\mathbb{SET}$, there is no requirement of monotonicity.

By composing with the inclusion $i : |\mathbb{T}| \to \mathbb{T}$ and the forgetful functor $U : \mathbb{POS} \to \mathbb{SET}$, any judgment structure $J : \mathbb{J}$ can be reindexed to a discrete judgment structure $|J| : \mathbb{J}_\circ \triangleq U \circ J \circ i$, with $\pi^{\Gamma}_{|J|} \triangleq \pi^{\Gamma}_{J}$:

$$
\begin{tikzcd}
{|\mathbb{T}|^{\mathsf{op}}} \arrow[r, "i"] \arrow[rrr, bend left, dashed, "|J|"] & \mathbb{T}^{\mathsf{op}} \arrow[r, "J"] & \mathbb{POS} \arrow[r, "U"] & \mathbb{SET}
\end{tikzcd}
$$

Remark IV.10*.* If we intended to develop a theory of tactics which do not commute with substitution, why did we bother with the presheaf apparatus in the first place? In essence, the reason is that we are building a semantics which accounts for both *refinement rules* and *tactics*, and refinement rules are distinguished from other proof refinement strategies precisely by the characteristic of lax naturality.

The purpose of tactics, on the other hand, is to subvert naturality in order to formalize modular "proof sketches" which are applicable to a broad class of goals. Indeed, this subversion of naturality by tactics is simultaneously the source of their unparalleled practicality as well as the cause of their notoriously brittle character. The uniformity of action induced by lax naturality lies in stark opposition to the modularity required of tactics; we will neutralize this contradiction by passing through discretization to $\mathbb{J}_\circ$.

F. Tactics and Recursion

In keeping with standard usage, a *proof tactic* is a potentially diverging program that computes a proof on the basis of some collection of *refinement rules*. In order to define tactics precisely, we will first have to specify how we intend to interpret recursion.

A very lightweight way to interpret recursion is suggested by Capretta's *delay monad* [38] (the completely iterative monad on the identity functor), a coinductive representation of a process which *may eventually* return a value. We can define a variation on Capretta's construction as a monad $\infty : \mathbb{J}_\circ \to \mathbb{J}_\circ$ on discrete judgment structures, defined as the *greatest* judgment structure closed under the rules in Figure 4.

To summarize, for a judgment structure $J : \mathbb{J}_\circ$, there are two ways to construct a ∞J-judgment:

1) $\lfloor X \rfloor$ is an ∞J-judgment when X is a J-judgment.
2) $\blacktriangleright X$ is an ∞J-judgment when X is an ∞J-judgment.

Lemma IV.11 (Delay monad)**.** *$\infty : \mathbb{J}_\circ \to \mathbb{J}_\circ$ forms a monad on $\mathbb{J}_\circ$.*

We can repeat the same construction as above to acquire a monad $\infty : \mathbb{J} \to \mathbb{J}$, which adds to the previous construction the appropriate action for substitutions. Abusing notation, we will use the same symbol for both monads when it is clear from

$$\boxed{\infty : \mathbb{J}_\circ \to \mathbb{J}_\circ} \qquad \frac{\Gamma \Vdash X : J \rightsquigarrow \Delta}{\Gamma \Vdash \lfloor X \rfloor : \infty J \rightsquigarrow \Delta} \qquad \frac{\Gamma \Vdash X : \infty J \rightsquigarrow \Delta}{\Gamma \Vdash \blacktriangleright_\Delta X : \infty J \rightsquigarrow \Delta} \tag{Delay Monad}$$

$$\blacktriangleright X \triangleq \blacktriangleright_\Delta X \tag{Notation}$$

Fig. 4. Capretta's delay monad on discrete judgment structures.

context what is meant; this is justified in practice, because the assignment of objects is the same for the two monads.

Notation IV.12. We will write $\eta_\infty : \mathbf{1}_{\mathbb{J}_\circ} \to \infty$ and $\mu_\infty : \infty \circ \infty \to \infty$ for the unit and multiplication operators respectively. We will also employ the following notational convention, inspired by the "do-notation" used in the Haskell programming language for monads:

$$x \leftarrow M; N(x) \triangleq \mu_\infty(\infty(x \mapsto N(x))(M))$$

Definition IV.13 (Tactics and multitactics)**.** A *tactic* for judgment structures $J_0, J_1 : \mathbb{J}$ is a $\mathbb{J}_\circ$-homomorphism:

$$\mathsf{Tactic}(J_0, J_1) \triangleq |J_0| \to \infty|\mathsf{St}(J_1)|$$

Usually one works with homogeneous tactics $\phi : \mathsf{Tactic}(J, J)$, which are called J-tactics. A J-*multitactic* is a tactic for the judgment structure $\mathsf{St}(J)$.

G. Tacticals as Tactic Combinators

At this point we are equipped to begin defining a collection of standard "tacticals", or *tactic combinators*.

1) Tactics from Rules: Every rule $\rho : J_0 \to \mathsf{St}(J_1)$ can be made into a tactic $\boxed{\lfloor \rho \rfloor : \mathsf{Tactic}(J_0, J_1)} \triangleq \eta_\infty \circ |\rho|$.

2) Conditional Tacticals: To begin with, we can define the join of two tactics $\phi, \psi : \mathsf{Tactic}(J_0, J_1)$, which implements **orelse** from LCF:

$$\begin{aligned} &\phi \oplus \psi : \mathsf{Tactic}(J_0, J_1) \\ &(\phi \oplus \psi)_\Gamma(X) \triangleq \; S \leftarrow \phi_\Gamma(X); \\ &\qquad \begin{cases} \lfloor S \rfloor & \textbf{if } S = \Psi \rhd \mathfrak{a} \\ \psi_\Gamma(X) & \textbf{otherwise} \end{cases} \end{aligned}$$

In $\phi \oplus \psi$ we have an example of a natural transformation which does *not* commute with substitutions; this is fine, because tactics are defined as *discrete* judgment structure homomorphisms, and are therefore subject to only trivial naturality conditions.

In combination with the identity tactic $\boxed{\mathbf{id} : \mathsf{Tactic}(J, J)} \triangleq \lfloor \eta \rfloor$, we can define the **try** tactical which replaces a failure or unsuccess with the identity:

$$\begin{aligned} &\mathbf{try}(\phi) : \mathsf{Tactic}(J, J) \\ &\mathbf{try}(\phi) \triangleq \phi \oplus \mathbf{id} \end{aligned}$$

3) Multitacticals: We will factor the LCF sequencing tacticals **then** and **thenl** into a combination of a "multitactical" (a tactic that operates on $\mathsf{St}(J)$ instead of J) and a generic sequencing operation.

These multitacticals will be factored through tacticals that are sensitive to the *position* of a goal within a proof state, namely **const** and **proj**.

Let ϕ range over tactics $\mathsf{Tactic}(J, J)$, and let $\vec{\phi}$ range over a list of such tactics. We will now define some further tactics which work over labeled judgments (following Section IV-D):

$$\begin{aligned} &\mathbf{const}(\phi) : \mathsf{Tactic}(J_{\times\mathbb{N}}, J) \\ &\mathbf{const}(\phi)_\Gamma\langle X, i\rangle \triangleq \phi_\Gamma(X) \end{aligned}$$

$$\begin{aligned} &\mathbf{proj}(\vec{\phi}) : \mathsf{Tactic}(J_{\times\mathbb{N}}, J) \\ &\mathbf{proj}(\vec{\phi})_\Gamma\langle X, i\rangle \triangleq \begin{cases} \phi_i^\Gamma(X) & \textbf{if } i < |\vec{\phi}| \\ \lfloor \eta_\Gamma(X) \rfloor & \textbf{otherwise} \end{cases} \end{aligned}$$

Now, we need to show how to turn transform an operation on labeled judgments into a *multitactic*. We will need an operation to turn a telescope of potentially diverging computations into a potentially diverging computation of a telescope:

$$\begin{aligned} &\mathsf{await} : \mathsf{Tl}(\infty J) \to \infty\mathsf{Tl}(J) \\ &\mathsf{await}_\Gamma \left| \begin{array}{lll} * & \Rightarrow & \lfloor * \rfloor \\ x : X^\infty.\, \Psi^\infty & \Rightarrow & X \leftarrow X^\infty; \\ & & \Psi \leftarrow \mathsf{await}_{\Gamma, x:\pi_J^\Gamma(X)}(\Psi^\infty); \\ & & \lfloor x : X.\, \Psi \rfloor \end{array} \right. \end{aligned}$$

Using the above, we can transform $\chi : \mathsf{Tactic}(J_{\times\mathbb{N}}, J)$ into a multitactic $\mathsf{St}\{\chi\} : \mathsf{Tactic}(\mathsf{St}(J), \mathsf{St}(J))$:

$$\begin{aligned} &\mathsf{St}\{\chi\} : \mathsf{Tactic}(\mathsf{St}(J), \mathsf{St}(J)) \\ &\mathsf{St}\{\chi\}_\Gamma \left| \begin{array}{lll} \bot & \Rightarrow & \lfloor \bot \rfloor \\ \maltese & \Rightarrow & \lfloor \maltese \rfloor \\ \Psi \rhd \mathfrak{a} & \Rightarrow & \Psi' \leftarrow \mathsf{await}_\Gamma(\mathsf{Tl}_\Gamma(\chi)(\mathbf{lbl}_0^\Gamma(\Psi))); \\ & & \lfloor \Psi' \rhd \mathfrak{a} \rfloor \end{array} \right. \end{aligned}$$

defining the auxiliary function Tl_Γ as follows:[1]

$$\begin{aligned} &\mathsf{Tl}_\Gamma : \mathsf{Tactic}(J_0, J_1) \times \mathsf{Tl}(J_0)(\Gamma) \to \mathsf{Tl}(\infty\mathsf{St}(J_1))(\Gamma) \\ &\mathsf{Tl}_\Gamma(\chi) \left| \begin{array}{lll} * & \Rightarrow & * \\ x : X.\, \Psi & \Rightarrow & x : \chi_\Gamma(X).\, \mathsf{Tl}_{\Gamma, \pi_J^\Gamma(X)}(\chi, \Psi) \end{array} \right. \end{aligned}$$

Finally, we can define two multitacticals: **all** which applies a single tactic to all goals, and **each** which applies a list of tactics pointwise to the subgoals:

$$\begin{aligned} &\mathbf{all}, \mathbf{each} : \mathsf{Tactic}(J, J) \to \mathsf{Tactic}(\mathsf{St}J, \mathsf{St}J) \\ &\mathbf{all}(\phi) \triangleq \mathsf{St}\{\mathbf{const}(\phi)\} \\ &\mathbf{each}(\phi) \triangleq \mathsf{St}\{\mathbf{proj}(\phi)\} \end{aligned}$$

[1]Note that this does not follow immediately from the functoriality of $\mathsf{Tl} : \mathbb{J} \to \mathbb{J}$, because χ only a map in $\mathbb{J}_\circ$.

4) Generic Sequencing: Fixing a tactic $\phi : \mathsf{Tactic}(\mathcal{J}_0, \mathcal{J}_1)$ and $\underline{\psi} : \mathsf{Tactic}(\mathsf{St}(\mathcal{J}_1), \mathsf{St}(\mathcal{J}_2))$ as above, we can define the *sequencing* of $\underline{\psi}$ after ϕ as the following composite, also displayed in Figure 5:

$$\begin{aligned}&\mathbf{seq}(\phi, \underline{\psi}) : \mathsf{Tactic}(\mathcal{J}_0, \mathcal{J}_2)\\&\mathbf{seq}(\phi, \underline{\psi}) \triangleq \infty|\mu| \circ \mu_\infty \circ \infty(\underline{\psi}) \circ \phi\end{aligned}$$

Now observe that the LCF sequencing tacticals can be defined in terms of the above combinators:

$$\begin{aligned}\mathbf{then}(\phi, \psi) &\triangleq \mathbf{seq}(\phi, \mathbf{all}(\psi))\\ \mathbf{thenl}(\phi, \vec{\psi}) &\triangleq \mathbf{seq}(\phi, \mathbf{each}(\vec{\psi}))\end{aligned}$$

5) Recursive Tacticals: Capretta's delay monad (Figure 4) allows us to develop a fixed point combinator for tacticals; in particular, given a tactical $T : \mathsf{Tactic}(\mathcal{J}, \mathcal{J}) \to \mathsf{Tactic}(\mathcal{J}, \mathcal{J})$, we have a fixed point $\mathbf{fix}(T) : \mathsf{Tactic}(\mathcal{J}, \mathcal{J})$. For the full construction of the fixed point $\mathbf{fix}(T)$, see Appendix C.

Using this, we can develop the standard **repeat** tactical from LCF, which is in practice the most commonly used recursive tactical:

$$\begin{aligned}&\mathbf{repeat}(\phi) : \mathsf{Tactic}(\mathcal{J}, \mathcal{J})\\&\mathbf{repeat}(\phi) \triangleq \mathbf{fix}(\psi \mapsto \mathbf{try}(\mathbf{then}(\phi, \psi)))\end{aligned}$$

Other recursive tacticals are possible, including recursive multitacticals.

Example IV.14 (Tactic for cost dynamics)**.** Returning to our running example (Examples IV.3, IV.8), we can now define a useful tactic to discharge all $\mathcal{J}_{\mathbb{A}}$-judgments; our first cut can be defined in the following way:

$$\begin{aligned}&\mathbf{auto}_{aux} : \mathsf{Tactic}(\mathcal{J}_{\mathbb{A}}, \mathcal{J}_{\mathbb{A}})\\&\mathbf{auto}_{aux} \triangleq \lfloor \mathbf{eval}_{\mathsf{num}} \rfloor \oplus \lfloor \mathbf{eval}_{+} \rfloor \oplus \lfloor \mathbf{add} \rfloor\end{aligned}$$

$$\begin{aligned}&\mathbf{auto} : \mathsf{Tactic}(\mathcal{J}_{\mathbb{A}}, \mathcal{J}_{\mathbb{A}})\\&\mathbf{auto} \triangleq \mathbf{repeat}(\mathbf{auto}_{aux}) \qquad (*)\end{aligned}$$

This is not quite, however, what we want: the force of this tactic is to run all our rules repeatedly (until failure or completion) on each subgoal. This is fine, but because these processes are taking place independently on each subgoal, the instantiations induced in one subgoal will not propagate to an adjacent subgoal until the entire process has quiesced.

The practical result of this approach is that the **auto** tactic will terminate with unresolved subgoals, and must be run again; our intention was, however, for the tactic to discharge all subgoals through repetition.

What we defined above can be described as *depth-first repetition*; what we want is *breadth-first repetition*, in which we run all the rules once on each subgoal, repeatedly. Then, substitutions propagate along the subgoals telescope with *every* application of $\mathbf{auto}_{aux}$, instead of propagating only after *all* applications of $\mathbf{auto}_{aux}$.

The way to achieve this is to apply our repetition at the level of multitactics, instantiating the tactical as $\mathbf{repeat} : \mathsf{Tactic}(\mathsf{St}(\mathcal{J}_{\mathbb{A}}), \mathsf{St}(\mathcal{J}_{\mathbb{A}})) \to \mathsf{Tactic}(\mathsf{St}(\mathcal{J}_{\mathbb{A}}), \mathsf{St}(\mathcal{J}_{\mathbb{A}}))$ instead of $\mathbf{repeat} : \mathsf{Tactic}(\mathcal{J}_{\mathbb{A}}, \mathcal{J}_{\mathbb{A}}) \to \mathsf{Tactic}(\mathcal{J}_{\mathbb{A}}, \mathcal{J}_{\mathbb{A}})$. This we can accomplish as follows:

$$\begin{aligned}&\mathbf{auto}_{multi} : \mathsf{Tactic}(\mathsf{St}(\mathcal{J}_{\mathbb{A}}), \mathsf{St}(\mathcal{J}_{\mathbb{A}}))\\&\mathbf{auto}_{multi} \triangleq \mathbf{repeat}(\mathbf{all}(\mathbf{auto}_{aux}))\end{aligned}$$

$$\begin{aligned}&\mathbf{auto} : \mathsf{Tactic}(\mathcal{J}_{\mathbb{A}}, \mathcal{J}_{\mathbb{A}})\\&\mathbf{auto} \triangleq \mathbf{seq}(\mathbf{id}, \mathbf{auto}_{multi})\end{aligned}$$

V. Concrete Implementation in Standard ML

As part of the **RedPRL** project [6], we have built a practical implementation of the apparatus described above in the Standard ML programming language [39].[2]

RedPRL is an interactive proof assistant in the Nuprl tradition for computational cubical type theory [30], a higher dimensional variant of Martin-Löf's extensional type theory [10]. Replacing LCF with Dependent LCF has enabled us to eliminate every last disruption to the proof refinement process in **RedPRL**'s refinement logic, including the introduction rule for dependent sums (as described in Section I-B), and dually, the elimination rule for dependent products.

Dependent LCF has also sufficed for us as a matrix in which to develop sophisticated type synthesis rules à la bidirectional typing, which has greatly simplified the proof obligations routinely incurred in an elaborator or refiner for extensional type theory, without needing to develop brittle and complex tactics for this purpose as was required in the Nuprl System.

Our experience suggests that, contrary to popularly-accepted folk wisdom, practical and usable implementations of extensional type theory are eminently possible, assuming that a powerful enough form of proof refinement apparatus is adopted.

Acknowledgments

The first author would like to thank David Christiansen for many hours of discussion on dependent proof refinement, and Sam Tobin-Hochstadt for graciously funding a visit to Indiana University during which the seeds for this paper were planted. Thanks also to Arnaud Spiwack, Adrien Guatto, Danny Gratzer, Brandon Bohrer and Darin Morrison for helpful conversations about proof refinement. The authors gratefully acknowledge the support of the Air Force Office of Scientific Research through MURI grant FA9550-15-1-0053. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR.

References

[1] R. Milner, "Logic for computable functions: Description of a machine implementation." Stanford University, Tech. Rep., 1972. 1

[2] M. Gordon, R. Milner, and C. Wadsworth, "Edinburgh LCF: A mechanized logic of computation," in *Lecture Notes in Computer Science*, vol. 78. Springer-Verlag, 1979. 1

[3] M. Gordon, "From LCF to HOL: A short history," in *Proof, Language, and Interaction*, G. Plotkin, C. Stirling, and M. Tofte, Eds. Cambridge, MA, USA: MIT Press, 2000, pp. 169–185. 1

[2] Our implementation is open-source and available at http://www.github.com/RedPRL/sml-dependent-lcf.

$$\begin{tikzcd}
{|J_0|} \arrow[r, "\phi"] \arrow[rrrr, bend left, dashed, "\mathbf{seq}(\phi, \underline{\psi})"] & \infty|\mathsf{St}(J_1)| \arrow[r, "\infty(\underline{\psi})"] & \infty\infty|\mathsf{St}(\mathsf{St}(J_2))| \arrow[r, "\mu_\infty"] & \infty|\mathsf{St}(\mathsf{St}(J_2))| \arrow[r, "\infty|\mu|"] & \infty|\mathsf{St}(J_2)|
\end{tikzcd}$$

Fig. 5. The generic sequencing tactical displayed as a composite.

[4] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith, *Implementing Mathematics with the Nuprl Proof Development System*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986. 1

[5] J. J. Hickey, "The MetaPRL logical programming environment," Ph.D. dissertation, Cornell University, Ithaca, NY, January 2001. 1

[6] J. Sterling, D. Gratzer, D. Christiansen, D. Morrison, E. Akentyev, and J. Wilcox, "RedPRL – the People's Refinement Logic," 2016, http://www.redprl.org/. 1, 4, 11

[7] The Coq Development Team, "The Coq Proof Assistant Reference Manual," 2016. 1

[8] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, *The Lean Theorem Prover (System Description)*. Cham: Springer International Publishing, 2015, pp. 378–388. 1

[9] L. C. Paulson, *Logic and computation : interactive proof with Cambridge LCF*, ser. Cambridge tracts in theoretical computer science. Cambridge, New York, Port Chester: Cambridge University Press, 1987, autre tirage : 1990 (br.). 1

[10] P. Martin-Löf, "Constructive mathematics and computer programming," in *6th International Congress for Logic, Methodology and Philosophy of Science*, Hanover, August 1979, pp. 153–175, published by North Holland, Amsterdam. 1982. 1, 11

[11] V. Rahli and M. Bickford, "A nominal exploration of Intuitionism," in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, ser. CPP 2016. New York, NY, USA: ACM, 2016, pp. 130–141. 1

[12] R. L. Constable, "The semantics of evidence," Cornell University, Tech. Rep. TR85-684, 1985. 1

[13] ——, "Evidence semantics and refinement rules for first-order logics: Minimal, intuitionistic, and classical," 2012, course notes. 1

[14] P. Martin-Löf, "Truth of a proposition, evidence of a judgement, validity of a proof," *Synthese*, vol. 73, no. 3, pp. 407–420, 1987. 2

[15] D. Prawitz, "Truth and proof in intuitionism," in *Epistemology versus Ontology: Essays on the Philosophy and Foundations of Mathematics in Honour of Per Martin-Löf*, P. Dybjer, S. Lindström, E. Palmgren, and G. Sundholm, Eds. Dordrecht: Springer Netherlands, 2012, pp. 45–67. 2

[16] R. W. Harper, "Aspects of the implementation of type theory," Ph.D. dissertation, Cornell University, 1985. 2

[17] U. Norell, "Dependently typed programming in agda," in *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, ser. TLDI '09. New York, NY, USA: ACM, 2009, pp. 1–2. 2

[18] J. Cockx, D. Devriese, and F. Piessens, "Unifiers as equivalences: Proof-relevant unification of dependently typed data," in *ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, September 2016. 2

[19] T. B. Knoblock, "Metamathematical extensibility in type theory," Ph.D. dissertation, Cornell University, Ithaca, NY, USA, 1988. 3

[20] M. Wenzel, F. Haftmann, and L. Paulson, "The Isabelle/Isar Implementation," 2016. 3

[21] P. Martin-Löf, *Analytic and Synthetic Judgements in Type Theory*. Dordrecht: Springer Netherlands, 1994, pp. 87–99. 3

[22] C. McBride, "Dependently typed programs and their proofs," Ph.D. dissertation, University of Edinburgh, 1999. 3

[23] ——, "Epigram: Practical programming with dependent types," in *Proceedings of the 5th International Conference on Advanced Functional Programming*, ser. AFP'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 130–170. 3

[24] E. Brady, "Practical Implementation of a Dependently Typed Functional Programming Language," Ph.D. dissertation, Durham University, 2005. 3

[25] ——, "Idris, a general-purpose dependently typed programming language: Design and implementation," *Journal of Functional Programming*, vol. 23, no. 5, p. 552593, Sep 2013. 3

[26] D. Christiansen and E. Brady, "Elaborator reflection: Extending Idris in Idris," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2016. New York, NY, USA: ACM, 2016, pp. 284–297. 3

[27] A. Gundry, C. McBride, and J. McKinna, "Type inference in context," in *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming*, ser. MSFP '10. New York, NY, USA: ACM, 2010, pp. 43–54. 3

[28] B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis, "Mtac: A monad for typed tactic programming in coq," in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '13. New York, NY, USA: ACM, 2013, pp. 87–100. 4

[29] A. Spiwack, "Verified computing in homological algebra, a journey exploring the power and limits of dependent type theory," Ph.D. dissertation, École Polytechnique, 2011. 4

[30] C. Angiuli, R. Harper, and T. Wilson, "Computational higher-dimensional type theory," in *43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, Jan. 2016. 4, 11

[31] M. Fiore and O. Mahmoud, "Second-order algebraic theories," in *Mathematical Foundations of Computer Science 2010: 35th International Symposium, MFCS 2010, Brno, Czech Republic, August 23-27, 2010. Proceedings*, P. Hliněný and A. Kučera, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 368–380. 4

[32] F. Bonchi and F. Zanasi, *Saturated Semantics for Coalgebraic Logic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 80–94. 5

[33] E. Komendantskaya and J. Power, "Category theoretic semantics for theorem proving in logic programming: embracing the laxness," in *Proceedings of Coalgebraic Methods in Computer Science*, ser. Lecture Notes in Computer Science, I. Hasuo, Ed. Springer, June 2016, pp. 94–113. [Online]. Available: http://opus.bath.ac.uk/49979/ 5

[34] S. Mac Lane and I. Moerdijk, *Sheaves in geometry and logic : a first introduction to topos theory*, ser. Universitext. New York: Springer, 1992. 5

[35] P. Martin-Löf, "On the meanings of the logical constants and the justifications of the logical laws," *Nordic Journal of Philosophical Logic*, vol. 1, no. 1, pp. 11–60, 1996. 5

[36] R. Harper, *Practical Foundations for Programming Languages*. New York, NY, USA: Cambridge University Press, 2016. 6

[37] N. de Bruijn, "Telescopic mappings in typed lambda calculus," *Information and Computation*, vol. 91, no. 2, pp. 189 – 204, 1991. 6

[38] V. Capretta, "General recursion via coinductive types," *Logical Methods in Computer Science*, vol. 1, no. 2, pp. 1–18, 2005. 9, 15

[39] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)*. MIT Press, 1997. 11

Appendix

A. Formal Definitions

We have used a convenient syntactic notation based on variable binding in the above. The definitions of telescopes and proof states can be given more precisely in category-theoretic notation, at the cost of some bureaucracy. In return, we can tell that these objects are indeed judgment structures according to our definition purely on the basis of how they are formed.

Definition A.1 (Yoneda embedding)**.** Write $\mathcal{H}(\Gamma) : \widehat{\mathbb{T}}$ for the representable presheaf $- \Rightarrow \Gamma$ of Γ-terms in the "current" context.

The collection of terms-in-context can be internalized into our presheaf category using the Yoneda embedding and the exponential. Define the presheaf $(\Gamma \vdash \Delta) : \widehat{\mathbb{T}}$ as the exponential $\mathcal{H}(\Delta)^{\mathcal{H}(\Gamma)}$; recall the definition of the exponential in a functor category:

$$\begin{aligned}
(\Gamma \vdash \Delta)(\Xi) &= \mathcal{H}(\Delta)^{\mathcal{H}(\Gamma)}(\Xi) \\
&\cong [\mathcal{H}(\Xi) \times \mathcal{H}(\Gamma), \mathcal{H}(\Delta)] && \text{(definition)} \\
&\cong [\mathcal{H}(\Xi \times \Gamma), \mathcal{H}(\Delta)] && \text{(limit preservation)} \\
&\cong \mathcal{H}(\Delta)(\Xi \times \Gamma) && \text{(Yoneda lemma)} \\
&\cong \Xi \times \Gamma \Rightarrow \Delta && \text{(definition)}
\end{aligned}$$

In what follows, we will frequently exponentiate by a representable functor; as above, naturality guarantees that the resulting object depends in no essential way on its input, justifying a purely syntactic notation based on variable binding.

Our definitions of telescopes and proof states as judgment structures can be restated in categorical form below; let p be the evident variable projection map in $\mathbb{T}$.

$$\begin{aligned}
\mathsf{Tl}(\mathbb{J}) &\triangleq \mu T.\ \mathbf{1} + \textstyle\sum_{X:\mathbb{J}} T^{\mathcal{H}(\pi_{\mathbb{J}}(X))} \\
\pi_{\mathsf{Tl}(\mathbb{J})}(\mathsf{roll}(\mathsf{inl}(*))) &\triangleq \mathbf{1} \\
\pi_{\mathsf{Tl}(\mathbb{J})}(\mathsf{roll}(\mathsf{inr}(X, \Psi))) &\triangleq \pi_{\mathbb{J}}(X) \times \pi_{\mathsf{Tl}(\mathbb{J})}(\Psi(\mathsf{p}))
\end{aligned}$$

$$\begin{aligned}
\mathsf{St}(\mathbb{J}) &\triangleq \textstyle\sum_{\Delta:\mathsf{Ctx}} \left(\{\maltese, \bot\} + \sum_{\Psi:\mathsf{Tl}(\mathbb{J})} \pi_{\mathsf{Tl}(\mathbb{J})}(\Psi) \vdash \Delta\right) \\
\pi_{\mathsf{St}(\mathbb{J})}(\Delta, \ldots) &\triangleq \Delta
\end{aligned}$$

B. Proofs of Theorems

Lemma A.2. *We have the following identity for* $S : \mathsf{St}(\mathbb{J})$, $\Psi : \mathsf{Tl}(\mathbb{J})$, $\underline{\Psi} : \mathsf{Tl}(\mathsf{St}(\mathbb{J}))$ *and* $\mathfrak{a} : \pi_{\mathsf{Tl}(\mathsf{St}(\mathbb{J}))}(x : \Psi \ldots S.\underline{\Psi}) \vdash \Delta$*:*

$$\mu(x : (\Psi \ldots S).\underline{\Psi} \rhd \mathfrak{a}) = \Psi \ldots \mu(x : S.\underline{\Psi} \rhd \mathfrak{a})$$

Proof. We proceed by case on S.

Case $S \equiv \maltese$. By calculation.

$$\begin{aligned}
\mu(x : (\Psi \ldots S).\underline{\Psi} \rhd \mathfrak{a}) &= \mu(x : (\Psi \ldots \maltese).\underline{\Psi} \rhd \mathfrak{a}) \\
&= \mu(x : \maltese.\underline{\Psi} \rhd \mathfrak{a}) \\
&= \maltese
\end{aligned}$$

$$\begin{aligned}
\Psi \ldots \mu(x : S.\underline{\Psi} \rhd \mathfrak{a}) &= \Psi \ldots \mu(x : \maltese.\underline{\Psi} \rhd \mathfrak{a}) \\
&= \Psi \ldots \maltese \\
&= \maltese
\end{aligned}$$

Case $S \equiv \bot$. Analogous to the above.

Case $S \equiv \Psi_S \rhd \mathfrak{a}_S$. By calculation.

$$\begin{aligned}
\mu(x : (\Psi \ldots S).\underline{\Psi} \rhd \mathfrak{a}) &= \mu(x : (\Psi \ldots (\Psi_S \rhd \mathfrak{a}_S)).\underline{\Psi} \rhd \mathfrak{a}) \\
&= \mu(x : ((\Psi \ldots \Psi_S) \rhd \mathfrak{a}_S).\underline{\Psi} \rhd \mathfrak{a}) \\
&= (\Psi \ldots \Psi_S) \ldots \mu(\underline{\Psi} \rhd \mathfrak{a})[\mathfrak{a}_S / x]
\end{aligned}$$

$$\begin{aligned}
\Psi \ldots \mu(x : S.\underline{\Psi} \rhd \mathfrak{a}) &= \Psi \ldots \mu(x : (\Psi_S \rhd \mathfrak{a}_S).\underline{\Psi} \rhd \mathfrak{a}) \\
&= \Psi \ldots (\Psi_S \ldots \mu(\underline{\Psi} \rhd \mathfrak{a})[\mathfrak{a}_S / x]) \\
&= (\Psi \ldots \Psi_S) \ldots \mu(\underline{\Psi} \rhd \mathfrak{a})[\mathfrak{a}_S / x]
\end{aligned}$$

□

Theorem IV.6. *Proof states form a monad on* $\mathbb{J}$*, i.e. the following diagrams commute:*

$$\begin{tikzcd}
\mathsf{St} \arrow[r, "\eta"] \arrow[dr, "\mathbf{1}"'] & \mathsf{St} \circ \mathsf{St} \arrow[d, "\mu"] & \mathsf{St} \arrow[l, "\mathsf{St}(\eta)"'] \arrow[dl, "\mathbf{1}"] \\
& \mathsf{St} &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathsf{St} \circ \mathsf{St} \circ \mathsf{St} \arrow[r, "\mu"] \arrow[d, "\mathsf{St}(\mu)"'] & \mathsf{St} \circ \mathsf{St} \arrow[d, "\mu"] \\
\mathsf{St} \circ \mathsf{St} \arrow[r, "\mu"] & \mathsf{St}
\end{tikzcd}$$

Proof. First, we have to show that the left triangle of the unit identity commutes for any $S \equiv \Psi \rhd \mathfrak{a}$:

$$\begin{aligned}
\mu(\eta(S)) &= \mu(x : \Psi \rhd \mathfrak{a}. * \rhd x) \\
&= \Psi \rhd x[\mathfrak{a}/x] \\
&= \Psi \rhd \mathfrak{a} \\
&= S
\end{aligned}$$

Next, we must show that the right unit triangle commutes. This, we will do by induction on Ψ:

Case $\Psi \equiv *$.

$$\begin{aligned}
\mu(\mathsf{St}(\eta)(S)) &= \mu(\mathsf{St}(\eta)(* \rhd \mathfrak{a})) \\
&= \mu(* \rhd \mathfrak{a}) \\
&= * \rhd \mathfrak{a} \\
&= S
\end{aligned}$$

Case $\Psi \equiv x : X.\, \Psi'$.

$$\begin{aligned}
\mu(\mathsf{St}(\eta)(S)) &= \mu(\mathsf{St}(\eta)(x : X.\, \Psi' \rhd \mathfrak{a})) \\
&= \mu(\mathsf{Tl}(\eta)(x : X.\, \Psi') \rhd \mathfrak{a}) \\
&= \mu(x : \eta(X).\, \mathsf{Tl}(\eta)(\Psi') \rhd \mathfrak{a}) \\
&= \mu(x : (y : X.\, * \rhd y).\, \mathsf{Tl}(\eta)(\Psi') \rhd \mathfrak{a}) \\
&= (y : X.\, *) \ldots \mu(\mathsf{Tl}(\eta)(\Psi') \rhd \mathfrak{a})[y/x] \\
&= (x : X.\, *) \ldots \mu(\mathsf{Tl}(\eta)(\Psi') \rhd \mathfrak{a}) && \text{(ren.)} \\
&= (x : X.\, *) \ldots \mu(\mathsf{St}(\eta)(\Psi' \rhd \mathfrak{a})) \\
&= (x : X.\, *) \ldots \Psi' \rhd \mathfrak{a} && \text{(i.h.)} \\
&= x : X.\, \Psi' \rhd \mathfrak{a} \\
&= S
\end{aligned}$$

Lastly, we need to show that the monad multiplication square commutes. Fix $\underline{\underline{S}} : \mathsf{St}(\mathsf{St}(\mathsf{St}(J)))$ and proceed by case.
Case $\underline{\underline{S}} \equiv \maltese$.

$$\begin{aligned}
\mu(\mu(\underline{\underline{S}})) &= \mu(\maltese) \\
\mu(\mathsf{St}(\mu)(\underline{\underline{S}})) &= \mu(\maltese)
\end{aligned}$$

Case $\underline{\underline{S}} \equiv \bot$. Analogous to the above.

Case $\underline{\underline{S}} \equiv \underline{\underline{\Psi}} \rhd \mathfrak{a}$. Proceed by induction on $\underline{\underline{\Psi}}$.
Case $\underline{\underline{\Psi}} \equiv *$.

$$\begin{aligned}
\mu(\mu(\underline{\underline{S}})) &= \mu(\mu(* \rhd \mathfrak{a})) \\
&= \mu(* \rhd \mathfrak{a})
\end{aligned}$$

$$\begin{aligned}
\mu(\mathsf{St}(\mu)(\underline{\underline{S}})) &= \mu(\mathsf{St}(\mu)(* \rhd \mathfrak{a})) \\
&= \mu(* \rhd \mathfrak{a})
\end{aligned}$$

Case $\underline{\underline{\Psi}} \equiv x : \underline{S}.\, \underline{\underline{\Psi'}}$ with $\underline{S} \equiv \maltese$.

$$\begin{aligned}
\mu(\mu(\underline{\underline{S}})) &= \mu(\mu(x : \underline{S}.\, \underline{\underline{\Psi'}} \rhd \mathfrak{a})) \\
&= \mu(\maltese) \\
&= \maltese
\end{aligned}$$

$$\begin{aligned}
\mu(\mathsf{St}(\mu)(\underline{\underline{S}})) &= \mu(\mathsf{St}(\mu)(x : \underline{S}.\, \underline{\underline{\Psi'}} \rhd \mathfrak{a})) \\
&= \mu(x : \maltese.\, \mathsf{Tl}(\mu)(\underline{\underline{\Psi'}}) \rhd \mathfrak{a}) \\
&= \maltese
\end{aligned}$$

Case $\underline{\underline{\Psi}} \equiv x : \underline{S}.\, \underline{\underline{\Psi'}}$ with $\underline{S} \equiv \bot$. Analogous to the above.

Case $\underline{\underline{\Psi}} \equiv x : \underline{S}.\, \underline{\underline{\Psi'}}$ with $\underline{S} \equiv \underline{\Psi} \rhd \mathfrak{b}$.

$$\begin{aligned}
\mu(\mu(\underline{\underline{S}})) &= \mu(\mu(x : \underline{S}.\, \underline{\underline{\Psi'}} \rhd \mathfrak{a})) \\
&= \mu(\mu(x : (\underline{\Psi} \rhd \mathfrak{b}).\, \underline{\underline{\Psi'}} \rhd \mathfrak{a})) \\
&= \mu(\underline{\Psi} \ldots \mu(\underline{\underline{\Psi'}} \rhd \mathfrak{a})[\mathfrak{b}/x])
\end{aligned}$$

$$\begin{aligned}
\mu(\mathsf{St}(\mu)(\underline{\underline{S}})) &= \mu(\mathsf{St}(\mu)(x : \underline{S}.\, \underline{\underline{\Psi'}} \rhd \mathfrak{a})) \\
&= \mu(x : \mu(\underline{S}).\, \mathsf{Tl}(\mu)(\underline{\underline{\Psi'}}) \rhd \mathfrak{a}) \\
&= \mu(x : \mu(\underline{\Psi} \rhd \mathfrak{b}).\, \mathsf{Tl}(\mu)(\underline{\underline{\Psi'}}) \rhd \mathfrak{a})
\end{aligned}$$

To proceed further, we must perform a second induction on $\underline{\Psi}$.

Case $\underline{\Psi} \equiv *$.

$$\mu(\mu(\underline{\underline{S}})) = \mu(\mu(\underline{\underline{\Psi'}} \rhd \mathfrak{a}))[\mathfrak{b}/x]$$

$$\begin{aligned}
\mu(\mathsf{St}(\mu)(\underline{\underline{S}})) &= \mu(x : * \rhd \mathfrak{b}.\, \mathsf{Tl}(\mu)(\underline{\underline{\Psi'}}) \rhd \mathfrak{a}) \\
&= \mu(\mathsf{Tl}(\mu)(\underline{\underline{\Psi'}}) \rhd \mathfrak{a})[\mathfrak{b}/x] \\
&= \mu(\mathsf{St}(\mu)(\underline{\underline{\Psi'}} \rhd \mathfrak{a}))[\mathfrak{b}/x]
\end{aligned}$$

By the outer inductive hypothesis, we have $\mu(\mu(\underline{\underline{\Psi'}} \rhd \mathfrak{a})) = \mu(\mathsf{St}(\mu)(\underline{\underline{\Psi'}} \rhd \mathfrak{a}))$.

Case $\underline{\Psi} \equiv y : S.\, \underline{\Psi'}$ with $S \equiv \maltese$.

$$\begin{aligned}
\mu(\mu(\underline{\underline{S}})) &= \mu(y : \maltese.\, \underline{\Psi'} \ldots \mu(\underline{\underline{\Psi'}} \rhd \mathfrak{a})[\mathfrak{b}/x]) \\
&= \maltese
\end{aligned}$$

$$\begin{aligned}
\mu(\mathsf{St}(\mu)(\underline{\underline{S}})) &= \mu(x : \mu(y : \maltese.\, \underline{\Psi'} \rhd \mathfrak{b}).\, \mathsf{Tl}(\mu)(\underline{\underline{\Psi'}}) \rhd \mathfrak{a}) \\
&= \mu(x : \maltese.\, \mathsf{Tl}(\mu)(\underline{\underline{\Psi'}}) \rhd \mathfrak{a}) \\
&= \maltese
\end{aligned}$$

Case $\underline{\Psi} \equiv y : S.\, \underline{\Psi'}$ with $S \equiv \bot$. Analogous to the above.

Case $\underline{\Psi} \equiv y : S.\, \underline{\Psi'}$ with $S \equiv \Psi \rhd \mathfrak{c}$.

$$\begin{aligned}
\mu(\mu(\underline{\underline{S}})) &= \mu(y : (\Psi \rhd \mathfrak{c}).\, \underline{\Psi'} \ldots \mu(\underline{\underline{\Psi'}} \rhd \mathfrak{a})[\mathfrak{b}/x]) \\
&= \Psi \ldots \mu((\underline{\Psi'} \ldots \mu(\underline{\underline{\Psi'}} \rhd \mathfrak{a})[\mathfrak{b}/x]))[\mathfrak{c}/y]
\end{aligned}$$

$$\begin{aligned}
\mu(\mathsf{St}(\mu)(\underline{\underline{S}})) &= \mu(x : \mu(y : (\Psi \rhd \mathfrak{c}).\, \underline{\Psi'} \rhd \mathfrak{b}).\, \mathsf{Tl}(\mu)(\underline{\underline{\Psi'}}) \rhd \mathfrak{a}) \\
&= \mu(x : (\Psi \ldots \mu(\underline{\Psi'} \rhd \mathfrak{b})[\mathfrak{c}/y]).\, \mathsf{Tl}(\mu)(\underline{\underline{\Psi'}}) \rhd \mathfrak{a}) \\
&= \Psi \ldots \mu(x : \mu(\underline{\Psi'} \rhd \mathfrak{b})[\mathfrak{c}/y].\, \mathsf{Tl}(\mu)(\underline{\underline{\Psi'}}) \rhd \mathfrak{a}) \\
& && \text{(Lemma A.2)}
\end{aligned}$$

Now, we need to show the following:

$$\begin{aligned}
&\mu((\underline{\Psi'} \ldots \mu(\underline{\underline{\Psi'}} \rhd \mathfrak{a})[\mathfrak{b}/x]))[\mathfrak{c}/y] \\
&\quad = \mu(x : \mu(\underline{\Psi'} \rhd \mathfrak{b})[\mathfrak{c}/y].\, \mathsf{Tl}(\mu)(\underline{\underline{\Psi'}}) \rhd \mathfrak{a})
\end{aligned}$$

It suffices to show:

$$\begin{aligned}
&\mu((\underline{\Psi'} \ldots \mu(\underline{\underline{\Psi'}} \rhd \mathfrak{a})[\mathfrak{b}/x]))[\mathfrak{c}/y] \\
&\quad = \mu(x : \mu(\underline{\Psi'} \rhd \mathfrak{b}).\, \mathsf{Tl}(\mu)(\underline{\underline{\Psi'}}) \rhd \mathfrak{a})[\mathfrak{c}/y]
\end{aligned}$$

Cancelling substitutions, we need only show the following:

$$\begin{aligned}
&\mu((\underline{\Psi'} \ldots \mu(\underline{\underline{\Psi'}} \rhd \mathfrak{a})[\mathfrak{b}/x])) \\
&\quad = \mu(x : \mu(\underline{\Psi'} \rhd \mathfrak{b}).\, \mathsf{Tl}(\mu)(\underline{\underline{\Psi'}}) \rhd \mathfrak{a})
\end{aligned}$$

But this holds by our inner inductive hypothesis.

□

C. Fixed Points of Tacticals

We will now demonstrate how to take the fixed point of a tactical using Capretta's delay monad (Figure 4), and use it to construct the commonly-used **repeat** tactical from LCF. To begin with, we will need to define fiberwise products and ω-sequences as operations on judgment structures. The fiberwise product of two judgment structures is a judgment structure whose judgments consist in pairs of judgments that synthesize the same sort Δ:

$$\boxed{\otimes : \mathbb{J}_\circ \times \mathbb{J}_\circ \longrightarrow \mathbb{J}_\circ} \qquad \frac{\Gamma \Vdash X : J_0 \rightsquigarrow \Delta \quad \Gamma \Vdash Y : J_1 \rightsquigarrow \Delta}{\Gamma \Vdash \langle X, Y \rangle : J_0 \otimes J_1 \rightsquigarrow \Delta}$$
(Fiberwise Product)

This is just the pullback of the two judgment structures when viewed as objects in the slice category $\widetilde{|\mathbb{T}|}/\mathsf{Ctx}$:

$$\begin{tikzcd}
J_0 \otimes J_1 \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & J_0 \arrow[d, "\pi_{J_0}"] \\
J_1 \arrow[r, "\pi_{J_1}"] & \mathsf{Ctx}
\end{tikzcd}$$

Next, we define an endofunctor on judgment structures that takes infinite sequences of judgments:

$$\boxed{-^{\omega} : \mathbb{J}_\circ \longrightarrow \mathbb{J}_\circ} \qquad \frac{\Gamma \Vdash X : J_n \rightsquigarrow \Delta \quad (n \in \mathbb{N})}{\Gamma \Vdash \langle X_n \mid n \rangle : J^{\omega} \rightsquigarrow \Delta}$$
(ω-Sequence)

This too can be presented as the limit $\varprojlim_{n \in \mathbb{N}} J^n$ where J^n is the n-fold fiberwise product of J with itself.

Now that we have defined the objects we require, we will begin to define the operations by which we can take the fixed point of a tactical, following [38]. First, we have a way to "race" two delayed judgments, returning the one which resolves first:

$$\begin{array}{l}
\mathbf{race} : \infty J \otimes \infty J \longrightarrow \infty J \\
\mathbf{race}_{\Gamma} \left| \begin{array}{lcl}
\langle \lfloor X \rfloor, Y_\infty \rangle & \Mapsto & \lfloor X \rfloor \\
\langle \blacktriangleright X_\infty, \lfloor Y \rfloor \rangle & \Mapsto & \lfloor Y \rfloor \\
\langle \blacktriangleright X_\infty, \blacktriangleright Y_\infty \rangle & \Mapsto & \mathbf{race}_{\Gamma} \langle X_\infty, Y_\infty \rangle
\end{array} \right.
\end{array}$$

This construction can be lifted to an ω-sequence of judgments as follows:

$$\begin{array}{l}
\mathbf{search}_n : (\infty J)^{\omega} \otimes \infty J \longrightarrow \infty J \\
\mathbf{search}_n^{\Gamma} \left| \begin{array}{lcl}
\langle F, \lfloor X \rfloor \rangle & \Mapsto & \lfloor X \rfloor \\
\langle F, \blacktriangleright X_\infty \rangle & \Mapsto & \blacktriangleright \mathbf{search}_{n+1}^{\Gamma} \langle F, \mathbf{race}_{\Gamma} \langle X_\infty, F_n \rangle \rangle
\end{array} \right.
\end{array}$$

In the delay monad, it is possible to define an object which never resolves:

$$\begin{array}{l}
\mathbf{never}_{\Gamma} : \infty J(\Gamma) \\
\mathbf{never}_{\Gamma} = \blacktriangleright \mathbf{never}_{\Gamma}
\end{array}$$

Now, using this and our unbounded search operator, we can take the least upper bound of an ω-sequence of judgments:

$$\begin{array}{c}
\bigsqcup : (\infty J)^{\omega} \longrightarrow \infty J \\
\bigsqcup_{\Gamma}(F) \triangleq \mathbf{search}_0^{\Gamma} \langle F, \mathbf{never}_{\Gamma} \rangle
\end{array}$$

Now fix a tactical $T : \mathsf{Tactic}(J, J) \longrightarrow \mathsf{Tactic}(J, J)$; it is now easy to get the fixed point of T (if it exists) by taking the least upper bound of a sequence of increasingly many applications of T to itself:

$$\begin{array}{c}
\mathbf{fix}(T) : \mathsf{Tactic}(J, J) \\
\mathbf{fix}(T)_{\Gamma}(X) \triangleq \bigsqcup \langle T_{\Gamma}^{n}(X) \mid n \rangle
\end{array}$$

where

$$\begin{array}{rl}
T^n : & \mathsf{Tactic}(J, J) \\
T_{\Gamma}^{0} = & X \mapsto \mathbf{never}_{\Gamma} \\
T_{\Gamma}^{n+1} = & T_{\Gamma}(T_{\Gamma}^{n})
\end{array}$$

D. Defining Refinement Logics

So far, we have built up a sophisticated apparatus for deterministic refinement proof, but have not shown how to instantiate it to a closed logic. In what follows, we will define a category of *refiners*, which can be thought of as implementations of a logic.

Definition A.3 (Heterogeneous refiners)**.** A *heterogeneous refiner* for judgment structures $J_0 : \mathbb{J}^{\mathsf{op}}$ and $J_1 : \mathbb{J}$ is a signature of rule names $\Sigma : \mathbb{POS}$ equipped with an interpretation $\mathcal{R} : \Sigma \longrightarrow \mathsf{Rule}(J_0, J_1)$. More generally, the *category of heterogeneous refiners* $\mathbf{HRef}(J_0, J_1)$ is the lax slice category $\mathbb{POS} /\!\!/ \mathsf{Rule}(J_0, J_1)$.

That is to say, a *refiner homomorphism* is a renaming of rules which preserves behavior up to approximation:

$$\begin{tikzcd}
\Sigma_0 \arrow[r, "\phi"] \arrow[dr, "\mathcal{R}_0"'] \arrow[r, phantom, shift right=4ex, "\preccurlyeq"] & \Sigma_1 \arrow[d, "\mathcal{R}_1"] \\
& \mathsf{Rule}(J_0, J_1)
\end{tikzcd}$$

Because this definition gives rise to a functor $\mathbf{HRef} : \mathbb{J}^{\mathsf{op}} \times \mathbb{J} \longrightarrow \mathbb{CAT}$, via the Grothendieck construction we can view refiners in general as forming a fibered category $\begin{array}{c}\mathbf{HRef}\\ \downarrow p\\ \mathbb{J} \times \mathbb{J}^{\mathsf{op}}\end{array}$, defining $\mathbf{HRef} \triangleq \oint^{\mathbb{J} \times \mathbb{J}^{\mathsf{op}}} \mathbf{HRef}$. The fibrational version has the advantage of specifying the notion of a refiner without fixing a particular judgment structure, thereby enabling a direct characterization of homomorphisms between refiners over different judgment structures.

We will usually restrict our attention to refiners that transform goals into subgoals of the same judgment structure; therefore, we must define a notion of *homogeneous refiner*.

First, let $\mathbb{J}_{\cong}$ be the *groupoid core* of $\mathbb{J}$, i.e. the largest subcategory of $\mathbb{J}$ whose arrows are all isomorphisms; we have the evident diagonal functor $\delta : \mathbb{J}_{\cong} \longrightarrow \mathbb{J} \times \mathbb{J}^{\mathsf{op}}$. Then, the category **Ref** of *homogeneous refiners* is easily described as the following pullback of the refiner fibration along the diagonal:

$$\begin{tikzcd}
\mathbf{Ref} \arrow[r] \arrow[d, "p_{\cong}"'] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{HRef} \arrow[d, "p"] \\
\mathbb{J}_{\cong} \arrow[r, "\delta"'] & \mathbb{J} \times \mathbb{J}^{\mathsf{op}}
\end{tikzcd}$$

Because it is the pullback of a fibration, $\begin{array}{c}\mathbf{Ref}\\ \downarrow p_{\cong}\\ \mathbb{J}_{\cong}\end{array}$ is also a fibered category.

Definition A.4 (Derivability Closure)**.** We can define the derivability closure of a homogeneous refiner $\mathfrak{R} \equiv \langle \mathbb{J}, \Sigma, \mathcal{R}\rangle$: **Ref**. First define a new rule signature which contains all derived rules, and extend the interpreter $\mathcal{R}$ appropriately:

$$\frac{r \in \Sigma}{\mathsf{leaf}(r) \in \Sigma^\star} \qquad \frac{r \in \Sigma^\star \quad \vec{r} \in \mathbf{List}(\Sigma^\star)}{\mathsf{branch}(r;\ \vec{r}) \in \Sigma^\star}$$

$$\frac{r_0 \preccurlyeq r_1}{\mathsf{leaf}(r_0) \preccurlyeq \mathsf{leaf}(r_1)} \qquad \frac{r_0 \preccurlyeq r_1 \quad \vec{r}_0 \preccurlyeq \vec{r}_1}{\mathsf{branch}(r_0;\ \vec{r}_0) \preccurlyeq \mathsf{branch}(r_1;\ \vec{r}_1)}$$

$$\mathcal{R}^\star : \Sigma^\star \longrightarrow \mathsf{Rule}(\mathbb{J}, \mathbb{J})$$

$$\mathcal{R}^\star \left| \begin{array}{ll} \mathsf{leaf}(r) & \Mapsto \mathcal{R}(r) \\ \mathsf{branch}(r;\ \vec{r}) & \Mapsto \mathcal{R}^*(r);\ \mathbf{List}(\mathcal{R}^*)(\vec{r}) \end{array}\right.$$

Then the derivability closure of $\mathfrak{R}$ is $\mathfrak{R}^\star \triangleq \langle \mathbb{J}, \Sigma^\star, \mathcal{R}^\star\rangle$.

Theorem A.5. *We have a refiner homomorphism* $i : \mathfrak{R} \longrightarrow \mathfrak{R}^\star$.

Proof. Define the action on rule names as $i(r) \triangleq \mathsf{leaf}(r)$; this is clearly monotone. It suffices to show that the following diagram commutes:

$$\begin{tikzcd} \Sigma \arrow[r, "i"] \arrow[dr, "\mathcal{R}"'] & \Sigma^\star \arrow[d, "\mathcal{R}^\star"] \\ & \mathsf{Rule}(\mathbb{J}, \mathbb{J}) \end{tikzcd} \qquad \begin{tikzcd} r \arrow[r, "i"] \arrow[dr, "\mathcal{R}"'] & \mathsf{leaf}(r) \arrow[d, "\mathcal{R}^\star"] \\ & \mathcal{R}(r) = \mathcal{R}(r) \end{tikzcd}$$

□

Definition A.6 (Presentation)**.** A *presentation* of a refiner $\mathfrak{R} \triangleq \langle \mathbb{J}, \Sigma, \mathcal{R}\rangle$: **Ref** is another refiner $\mathfrak{R}_p \triangleq \langle \mathbb{J}_p, \Sigma_p, \mathcal{R}_p\rangle$ together with a refiner homomorphism $p : \mathfrak{R}_p \longrightarrow \mathfrak{R}$ such that the induced judgment structure homomorphism $p_0 : \mathbb{J}_p \longrightarrow \mathbb{J}$ is equipped with a section; that is, we have the following:

$$\begin{tikzcd} \mathbb{J} \arrow[r, dashed, "s"] \arrow[dr, "\mathbf{1}_{\mathbb{J}}"'] & \mathbb{J}_p \arrow[d, "p_0"] \\ & \mathbb{J} \end{tikzcd}$$

Definition A.7 (Category of Presentations)**.** We can capture Definition A.6 in a *category of presentations* as the following pullback situation, letting $\mathfrak{R} \triangleq \langle \mathbb{J}, \Sigma, \mathcal{R}\rangle$ and $\mathbf{Pt}\,\mathbb{J}$ be the category of split epimorphisms in $\mathbb{J}$:

$$\begin{tikzcd} \mathbf{Pres}(\mathfrak{R}) \arrow[r, dashed] \arrow[d, dashed] & \mathbf{Ref}/\mathfrak{R} \arrow[d, "\pi_1"] \\ \mathbf{Pt}\,\mathbb{J}/\mathbb{J} \arrow[r, tail, "i"'] & \mathbb{J}/\mathbb{J} \end{tikzcd}$$

Definition A.8 (Canonizing Presentation)**.** A presentation $p : \mathfrak{R}_p \longrightarrow \mathfrak{R}$ is called *canonizing* when any construction that can be effected in $\mathfrak{R}$ can be effected by a unique rule in $\mathfrak{R}_p$. That is to say, for any $\Gamma \Vdash X : \mathbb{J} \rightsquigarrow \Delta$ and $\mathfrak{a} : \Gamma \Rightarrow \Delta$, we have:

$$(\exists r : \Sigma.\ \mathcal{R}(r)(X) \equiv * \rhd \mathfrak{a}) \Rightarrow (\exists! r : \Sigma_p.\ \mathcal{R}_p(r)(p^{-1}(X)) \equiv * \rhd \mathfrak{a})$$

E. Tactic Scripts and their Dynamics

Fixing a refiner $\mathfrak{R} \equiv \langle \mathbb{J}, \Sigma, \mathcal{R}\rangle$: **Ref**, we can now define a language of tactic scripts for $\mathfrak{R}$, letting r range over Σ:

$$\begin{array}{rcl} t & ::= & r \mid 1 \mid t \oplus t \mid t\star \mid t; m \\ m & ::= & \Box t \mid [t, \ldots, t] \mid m\star \end{array}$$

In the above, the classic **orelse** tactical from LCF is implemented by $t_1 \oplus t_2$; as in Section IV-G we have decomposed the standard LCF tacticals **then** and **thenl** into a combination of multitacticals and the sequencing tactical: respectively $t_0; \Box t_1$ and $t; [t_0, \ldots, t_n]$.

$$\begin{array}{r} \mathcal{T}[\![t]\!] : \mathsf{Tactic}(\mathbb{J}, \mathbb{J}) \\ \mathcal{M}[\![m]\!] : \mathsf{Tactic}(\mathsf{St}(\mathbb{J}), \mathsf{St}(\mathbb{J})) \end{array}$$

$$\begin{aligned} \mathcal{T}[\![r]\!] &= \lfloor \mathcal{R}(r) \rfloor \\ \mathcal{T}[\![1]\!] &= \mathbf{id} \\ \mathcal{T}[\![t_1 \oplus t_2]\!] &= \mathcal{T}[\![t_1]\!] \oplus \mathcal{T}[\![t_2]\!] \\ \mathcal{T}[\![t\star]\!] &= \mathbf{repeat}(\mathcal{T}[\![t]\!]) \\ \mathcal{T}[\![t; m]\!] &= \mathbf{seq}(\mathcal{T}[\![t]\!], \mathcal{T}[\![m]\!]) \end{aligned}$$

$$\begin{aligned} \mathcal{M}[\![\Box t]\!] &= \mathsf{St}\{\mathbf{all}(\mathcal{T}[\![t]\!])\} \\ \mathcal{M}[\![[t_0, \ldots, t_n]]\!] &= \mathsf{St}\{\mathbf{each}(\langle \mathcal{T}[\![t_0]\!], \ldots, \mathcal{T}[\![t_n]\!]\rangle)\} \\ \mathcal{M}[\![m\star]\!] &= \mathbf{repeat}(\mathcal{M}[\![m]\!]) \end{aligned}$$