

Prufrock: A Framework for Constructing Polytypic Theorem Provers

Justin Ward
Information and
Telecommunications
Technology Center
University of Kansas
2335 Irving Hill Road
Lawrence, KS 66045-7612
wardj@itc.ku.edu

Garrin Kimmell
Information and
Telecommunications
Technology Center
University of Kansas
2335 Irving Hill Road
Lawrence, KS 66045-7612
kimmell@itc.ku.edu

Perry Alexander
Information and
Telecommunications
Technology Center
University of Kansas
2335 Irving Hill Road
Lawrence, KS 66045-7612
alex@itc.ku.edu

ABSTRACT

Current formal software engineering methodologies provide a vast array of languages for specifying correctness properties, as well as a wide assortment automated tools that aid in the verification of specified properties. Unfortunately, the implementation of each such tool requires an early commitment to a particular methodology and language, in terms of both high-level semantic concerns and the lower-level syntactic representations of properties and proofs. In this paper, we present Prufrock, a novel approach to automated reasoning systems, which abstracts semantic concerns over entire classes of potential implementation languages. Prufrock is a modular prover framework written in the Haskell programming language. It consists of a set of largely independent logic modules, which define the semantics required for proof over entire classes of abstract syntaxes, using polytypic programming techniques. Any given representation language may be used for specifying and verifying properties in Prufrock, so long as it provides a semantics consistent with the logic modules that required for a proof. The implementation details of the reasoning system thus remain independent of the structure specification language, facilitating large-scale reuse in the construction of automated reasoning tools. At the same time, Prufrock aids in closing the gap between actual source-level implementation and the formal specification and verification of correctness properties. This paper provides an overview of the major design philosophy behind Prufrock, as well as a brief description of the polytypic techniques that make its implementation possible.

1. INTRODUCTION

Verification tools, such as theorem provers, perform their tasks by operating directly on an internal abstract syntax. This syntax encodes the theorems, properties, and axioms

used in the verification process. Inference and proof are carried out by manipulating this syntax in a logically sound fashion. Unfortunately, this approach can be a serious hindrance to modularity and reuse.

Users of a given verification tool must either formulate their specification of theorems and properties in a compatible syntax from the very beginning, or convert their existing specifications to a syntax suitable for the tool. The former approach results in serious constraints on the formal specification language that may be used, while the latter brings about the risk that the translation of a specification may not correctly capture all of its information. Even if a specification's soundness is preserved, the user faces the frustrating task of correlating the verification tool's output, presented in terms of its own abstract syntax, with the original specification. Thus, we may perceive an understandable fear of commitment whenever the subject of formal methods is broached. While verifying the correctness of a given software system remains the primary concern throughout the formalized development process, it intersects with a myriad of troubling pragmatic concerns related to the methodology. Developers want a formalized logic that accurately captures the semantics of the software system. This initial choice represents a serious commitment that will reverberate throughout the entire development process. Unfortunately, this choice is dependent not only the suitability of a given specification language, but also on the existence of adequate tools for that language.

These difficulties are only amplified for the designers of specification languages. A given language's utility and overall acceptance is severely hampered by the lack of viable tools to assist in its use. This leaves the designer of a specification language with a choice similar to that its user: either write a new suite of tools compatible with the language, thus bringing new tools to the language, or provide a means to translate the language into a form accepted by existing tools, thus bringing the language to existing tools. Writing tools requires a great deal of work, much of which requires merely reworking existing proof and verification strategies in a form compatible with the new language's basic structure. The converse approach, which involves translating the language into the accepted structure of existing tools, may result in the loss of many features that made the language

novel in the first place.

2. SEPARATING INFERENCE FROM SYNTAX

Tool implementers are not free from this mire of concerns, either. The typical theorem prover is a tightly coupled fusion of a variety of design tasks. A standard abstract syntax is required for representing logical data throughout the proof process. Additionally, the proof process usually requires a fair amount of bookkeeping and storage that must be managed in a consistent fashion. Inference rules are then formulated within the overarching constraints imposed by the proof language and the proof execution system. Even Isabelle[14], which provides an otherwise advanced modular system, requires that proofs are stored in a predetermined abstract syntax.

By comparison, the typical mathematical text on first-order logic is usually maddeningly simplistic. Most likely, the text spends a chapter or so defining the notation and terminology used throughout, and then assumes that the reader will mentally translate displayed formulae into meaningful logical objects. A particular text's notation does not, of course, directly affect the soundness or basic essence of laws and properties governing these operations. All that is strictly necessary for a sound logical system is a symbolic notation that is both consistent and sufficiently detailed to capture and convey the necessary logical semantics being described by laws and properties.

We may thus eschew the requirement that properties and models are written in a language predetermined early in the development of the verification tool, and replace it with a looser, more abstract requirement that the tool be capable of dealing with *any language so long as it provides an appropriate logical semantics*. We need not be concerned with the exact representation of conjunction in abstract syntax, for example, so much as we must know that *some* consistent representation of conjunction *exists*. After specifying the bare representational requirements for a particular logic, we may abstract common inference rules and proof techniques over the *entire class* of abstract syntaxes providing such representations.

As an example, consider one of DeMorgan's laws, $\neg(a \wedge b) \iff \neg a \vee \neg b$. While this law could be understood (and put to good use in a traditional automated reasoning system) merely as a rule for rearranging the symbols \neg , \vee , and \wedge , and the expressions a and b , given that they satisfy an appropriate initial configuration, most students of logic will interpret this statement in more abstract, semantic terms, such as: "the negation of a conjunction of two terms is equivalent to the disjunction of the negations those terms."

3. POLYTPIC PROGRAMMING

Discussing this separation of syntax and inference is easy from a convenient theoretical perspective, but we need an actual implementation of the system we have briefly described. To do this, we require a means for abstracting operations on structures from the particular structures themselves. This means is succinctly provided through polytypic program-

ming. Polytypic programming[8] is an approach to implementation that separates common operations, such as the familiar *map* and *fold* functions, from the exact structure of the data type over which they operate. While *map* serves as hylomorphism for lists, a polytypic *map* serves as hylomorphism for an entire class of structures. This is made possible by defining *map* over a more general set of functions, which may then be instantiated for each type over which we wish to use *map*. Standard polytypic approaches can derive these functions automatically through induction on the structure of datatypes. At a basic conceptual level, we may say that polytypic programming is concerned primarily with a type's *shape* than its exact *structure*.

The Haskell programming language[9] supports a variety of extensions that aid in polytypic programming and similar techniques. Jansson and Jeuring's PolyP language extension facilitates the construction of generic functions through analysis of datatypes' shapes[5, 13]. A similar effect is achieved in Generic Haskell, which facilitates generic functions via type casts[4]. While most polytypic programming systems utilize datatypes' structural information to automate the formulation of polytypic functions, another common approach involves defining a set of type classes that provide a common interface. Polytypic functions then simply program to this interface. This technique has been used by Jansson and Jeuring in the development of polytypic unification[6] and rewriting systems[7].

Our prover construction system is based upon type classes that provide functionality similar to those already utilized for unification and rewriting. An important difference exists, however, between the techniques that we shall present, and those commonly considered polytypic programming. As we have mentioned, most extant polytypic programming frameworks are built upon abstraction over shapes of datatypes. This allows many of the requisite instances may be derived automatically by induction over datatypes. In our case, the class-centered programming techniques remain the same, but the functions we are defining provide a logical *semantics* for a given class of type. Such semantics are not generally apparent from the structure of a type alone, but depend rather upon the interpretation of some structure. Thus, the user must manually specify instances for most type classes.

To illustrate the techniques that are used in our prover framework, we shall consider our example of DeMorgan's law again. A naive, monotypic function for rewriting the left side of our rule to the right side might look similar to figure 1, given the associated abstract syntax for representing propositions. The syntactic manipulations may be removed, and replaced with a set of abstract functions, which are defined for the datatype and then passed as parameters to the new rule, as shown in figure 2. The resulting function, *deMorgans2* is clumsy at best. While it is significantly more polymorphic than *deMorgans*, accepting any type a as input, it also requires six function arguments. We can simplify things by using Haskell's type class system, which allows us to associate a set of functions with a class of types, and provide specific instances for a given type. The correct functions for a type will be automatically selected at compile time by the Haskell type system. The resulting type class and rule is shown in figure 3. Note that the type sig-

```

deMorgans2 :: (a → Bool) → (a → Bool) → (a → a) →
  (a → a) → (a → a) → (a → a) a → a
deMorgans2 isNeg isAnd mkOr mkNeg fstArg sndArg expr =
  if (isNeg expr)
  then let inner = (fstArg expr)
        in if (isAnd inner)
            then mkOr (mkNeg (fstArg inner))
                      (mkNeg (sndArg inner))
            else expr
  else expr

```

Figure 2: A (clumsy) reformulation of DeMorgan’s law

```

data PropA = AndA PropA PropA |
  OrA PropA PropA |
  NegA PropA |
  VarA String

deMorgans :: PropA → PropA
deMorgans (NegA p) =
  case p of
    (AndA p1 p2) → (OrA (NotA p1) (NotA p2))
    otherwise → (NegA p)
deMorgans p = p

```

Figure 1: A naive implementation of DeMorgan’s law

nature of *deMorgans3* specifies that the polymorphic type *a* must be a member of the *DeMorgan* class, thus “carrying” the necessary functions with it. We can then use this

```

class DeMorgan a where
  isNeg :: a → Bool
  isAnd :: a → Bool
  mkOr :: a → a → a
  mkNeg :: a → a
  fstArg :: a → a
  sndArg :: a → a

deMorgans3 :: (DeMorgan a) ⇒ a → a
deMorgans3 expr =
  if (isNeg expr)
  then let inner = (fstArg expr)
        in if (isAnd inner)
            then mkOr (mkNeg (fstArg inner))
                      (mkNeg (sndArg inner))
            else expr
  else expr

```

Figure 3: deMorgan’s law with type classes

rule on our initial abstract syntax from figure 1 by declaring an appropriate instance of the *DeMorgan* class for the *PropA* datatype, as in figure 4. While we have significantly increased the amount of code required to implement a very simple rule, we have also completely separated the structure of the datatype representing terms from all necessary operations on terms. If we declare a new datatype, *PropB* and provide an instance for it, as in figure 5, we can our rule

```

instance deMorgan PropA where
  isNeg (NegA _) = True
  isNeg _ = False
  isAnd (AndA _ _) = True
  isAnd _ = False
  mkOr = OrA
  mkNeg = NegA
  fstArg (AndA a b) = a
  fstArg (OrA a b) = a
  fstArg (NegA a) = a
  sndArg (AndA a b) = b
  sndArg (OrA a b) = b

```

Figure 4: Class instance for *PropA*

for DeMorgan’s law “for free.” In this fashion, we may write each inference rule only once, without regard to specific term representations, and reuse it for any abstract syntax providing a sufficient logical semantics to satisfy the *DeMorgan* class. We have separated the mechanics of inference from the details of implementation.

```

data PropB = ConnB ConnectiveB PropB PropB |
  NegB PropB |
  VarB Int

data ConnectiveB = AndB | OrB

instance DeMorgan PropB where
  isNeg (NegB _) = True
  isNeg _ = False
  isAnd (ConnB AndB _ _) = True
  isAnd _ = False
  mkOr = ConnB OrB
  mkNeg = NegB
  fstArg (ConnB _ a b) = a
  fstArg (NegB a) = a
  sndArg (ConnB _ a b) = b

```

Figure 5: The *PropB* datatype

4. PRUFROCK

We have utilized these polytypic techniques in the development of Prufrock, a framework for generic theorem provers. Prufrock is a collection of composable logic modules, which provide support for assorted inference systems. Prufrock is

built upon a standard sequent calculus, extended by adding substitution restrictions. This extension facilitates the use of metavariables in the processing of quantifier rules, which significantly aids the proof search process by postponing instantiation decisions until near the end of the proof. A discussion of the underlying proof system used by Prufrock raises another important concern. Thus far, we have only considered the relation between syntax and inference. There is another similar dependency relating the actual mechanics of inference to the inference rules themselves. For instance, Prufrock’s sequent calculus requires a persistent state that tracks active subgoal sequents, as well as backtracking search. The actual implementation that satisfies these requirements may be considered orthogonal to the declaration of inference rules, in much the same way that syntax and inference rules may be deemed separate concerns. Thus, each logic module in Prufrock utilizes a pair of type classes: one that specifies a logical semantics that the proof-carrying syntax must provide, and another that specifies mechanical manipulations of the prover state. These manipulations are encapsulated in a monad class, which declares an interface to global storage maintained by a state monad. The structure of a typical logic module is depicted in figure 6. Inference

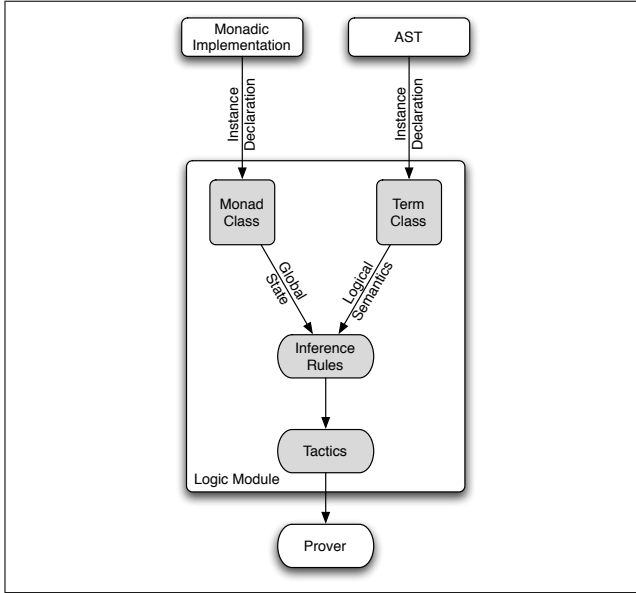


Figure 6: Structure of a typical Prufrock logic module

rules are written in terms of the logic module’s logical semantics, provided by a term class that describes syntaxes, and its state access, provided by a term class that describes monadic state operations. These inference rules are refined into tactics, using a uniform system, then exported for use in an actual theorem prover. The framework consisting of all exported proof tactics is then instantiated with a particular abstract syntax for representing proofs and a particular monadic implementation that carries out the proof process.

Because inference rules are defined over type classes, their type signatures reveal exactly what logical and monadic functionality they may utilize. This eases the task of ensuring the correctness of logic modules, by facilitating conser-

vative extension. A module containing rules for quantifiers may be combined with a module containing a complete set of rules for propositions to obtain first-order logic. However, the rules for quantifiers may be defined without reference to the type classes necessary for propositions, and thus the implementer may rest assured that these rules are a conservative extension that will not impact the correctness of the already defined propositional rules. Composition of modules is standardized within the Prufrock framework, drastically simplifying module reuse and verification.

5. PROPOSITIONAL LOGIC

We shall now present a series of simple logic modules, which illustrate by example the power and flexibility of the Prufrock framework. Before we present our sample logic modules, however, we must define a few basic types used throughout the prover. The *Sequent* type is merely a pair of lists joined with the infix constructor \vdash . Inference rules are represented as functions from a sequent to a list of new sequents resulting from the application of the rule. Similarly, decision procedures are functions from a sequent to a Boolean value indicating whether or not the sequent may be marked as closed and discharged. Note that the *Sequent* type is parametrized over a type variable t that represents the type describing the syntax used in proofs. Inference rules and decision procedures are abstracted over a syntax type t , as well as a monad type m . These types correspond to the syntax type and monad type used to finally construct the prover. By abstracting over them, we may ensure that all inference rules and decision procedures do not depend on particular syntax or monad implementations. The return type of an inference rule is wrapped in the monad m to facilitate monadic actions (such as state access or backtracking) within the inference rule. We can now begin our set of

```

data Sequent t = [t] :⊢ [t]

type InferenceRule m t =
  Sequent t → m [Sequent t]

type DecisionProcedure m t =
  Sequent t → m Bool
  
```

Figure 7: Basic types for sequents and inference rules

examples by defining a module for propositional logic in the cut-free Gentzen-type variant of the sequent calculus. The inference rules associated with this system are shown in their traditional formulation in figure 8. In order to implement the rules for propositional logic, we first define a term class that allows us to recognize, extract, and create the necessary logical operations. In particular, we need to recognize logical negation (\neg), and the four logical connectives (\wedge , \vee , \rightarrow , and \leftrightarrow). We declare the type class *PropLogic* to define the necessary logical semantics (Figure 9). *PropLogic* provides a set of recognizer and constructor functions for the literal Boolean values true and false, logical negation, conjunction, disjunction, implication, and bi-implication. Boolean values are recognized with the function *isBool* and interpreted using the *getBool* function, which returns a Boolean value’s truth value. The other recognizers and constructors are

```

connL :: (MonadNondet m) =>
  (t -> Bool) -> SequentChange t -> InferenceRule m t
connL p f seq@(hs :- cs) =
  let mkNewSeq (t, ts) = f t (ts :- cs)
  in maybe mzero (\lambda s -> return (mkNewSeq s)) (findRemove p hs)

connR :: (MonadNondet m) =>
  (t -> Bool) -> SequentChange t -> InferenceRule m t
connR p f seq@(hs :- cs) =
  let mkNewSeq (t, ts) = f t (hs :- ts) s
  in maybe mzero (\lambda s -> return (mkNewSeq s)) (findRemove p cs)

type SequentChange t = t -> Sequent t -> [Sequent t]

findRemove :: (t -> Bool) -> [t] -> Maybe (t, [t])
findRemove p xs = findRemove' p xs []
findRemove' p [] ns = Nothing
findRemove' p (y : ys) ns | p y = Just (y, ns ++ ys)
                        | otherwise = findRemove p ys (y : ns)

```

Figure 10: Utility functions for propositional logic

$$\begin{array}{c}
\text{basic}_\phi \frac{\phi, \Gamma \vdash \phi, \Delta}{\Gamma \vdash \phi, \Delta} \quad \text{basic}_\top \frac{\perp, \Gamma \vdash \Delta}{\Gamma \vdash \top, \Delta} \\
\text{basic}_\perp \frac{\Gamma \vdash \top, \Delta}{\Gamma \vdash \perp, \Delta} \\
\neg \vdash \frac{\Gamma \vdash \phi, \Delta}{\neg \phi, \Gamma \vdash \Delta} \quad \vdash \neg \frac{\phi, \Gamma \vdash \Delta}{\Gamma \vdash \neg \phi, \Delta} \\
\wedge \vdash \frac{\phi, \psi, \Gamma \vdash \Delta}{\phi \wedge \psi, \Gamma \vdash \Delta} \quad \vdash \wedge \frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta} \\
\vee \vdash \frac{\phi, \Gamma \vdash \Delta \quad \psi, \Gamma \vdash \Delta}{\phi \vee \psi, \Gamma \vdash \Delta} \quad \vdash \vee \frac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \vee \psi, \Delta} \\
\rightarrow \vdash \frac{\Gamma \vdash \phi, \Delta \quad \psi, \Gamma \vdash \Delta}{\phi \rightarrow \psi, \Gamma \vdash \Delta} \quad \vdash \rightarrow \frac{\phi, \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \rightarrow \psi, \Delta} \\
\iff \vdash \frac{\Gamma \vdash \phi, \Delta \quad \psi, \Gamma \vdash \Delta}{\phi \iff \psi, \Gamma \vdash \Delta} \\
\vdash \iff \frac{\phi, \Gamma \vdash \psi, \Delta \quad \psi, \Gamma \vdash \phi, \Delta}{\Gamma \vdash \phi \iff \psi, \Delta}
\end{array}$$

Figure 8: Propositional inference rules for the sequent calculus

declared in a straightforward manner. Finally we declare the functions, *getFirstArg* and *getSecondArg*, that extract the first and second arguments of an expression made up from one of the four logical connectives, and the function *getNegArg*, which extracts the argument from a negated term. Propositional logic does not require additional requirements for global storage, but we will use the existing *MonadNondet* class, taken from Hinze’s work [3] on backtracking monad transformers, to implement backtracking. A monad that is a member of *MonadNondet* provides, at a

```

class PropLogic t where
  isBool :: t -> Bool
  getBool :: t -> Bool
  mkBool :: Bool -> t
  isNot :: t -> Bool
  isAnd :: t -> Bool
  isOr :: t -> Bool
  isImp :: t -> Bool
  isBicond :: t -> Bool
  mkNot :: t -> t
  mkAnd :: t -> t -> t
  mkOr :: t -> t -> t
  mkImp :: t -> t -> t
  mkBicond :: t -> t -> t
  getFirstArg :: t -> t
  getSecondArg :: t -> t
  getNegaArg :: t -> t

```

Figure 9: A term class for propositional logic

minimum, functions *msum*, *mcommit*, and *mzero*. These are used, respectively, to specify non-determinism (backtracking), choice (“cuts” in the backtracking tree), and failure. We will only show the implementation for the *basic*, \neg , and \wedge rules. The implementation of the other rules is straightforward, proceeding in the same fashion. We use a set of utility functions, shown in figure 10, to handle repeated tasks in the inference rules and thus make the interesting parts more apparent. *ConnL* and *ConnR* take a predicate over terms, and a function that takes a term and a sequent and returns a list of sequents. They then produce an inference rule that attempts to find a term satisfying the given predicate on the appropriate side of sequent, then applies the *SequentChange* function to the sequent and the selected term, returning a list of new sequents. If no term can be found to satisfy the predicate, the resulting rule returns *mzero* to indicate failure. The utility functions are used to define the inference rules presented in figure 11. Note that most can be derived

in a straightforward fashion from their abstract representations in Figure 8. The *basic* axioms are represented as a single decision procedure that encompasses all three axioms. The other rules are implemented in a straightforward manner.

```

basicRule :: (PropLogic t) =>
    DecisionProcedure m t
basicRule (hs :+ cs) = return decision
  where
    decision = (or (map isFalse hs)) ∨
               (or (map isTrue cs)) ∨
               (or [h ≡ c | h ← hs, c ← cs])
    isTrue x = (isBool x) ∧ (getBool x)
    isFalse x = (isBool x) ∧ (¬ (getBool x))

notLeftRule :: (MonadNondet m, PropLogic t) =>
    InferenceRule m t
notLeftRule = connL isNot notLeftRule'
  where
    notLeftRule' t (hs :+ cs) =
      [hs :+ ((getFirstArg t) : cs)]

andRightRule :: (MonadNondet m, PropLogic t) =>
    InferenceRule m t
andRightRule = connR isAnd andRightRule'
  where
    andRightRule' t (hs :+ cs) =
      [hs :+ ((getFirstArg t) : cs),
       hs :+ ((getSecondArg t) : cs)]

```

Figure 11: Encoding of inference rules for propositional logic

6. PREDICATE CALCULUS

Now that we turn to an implementation of the quantifier rules of predicate calculus. Our implementation of the predicate calculus relies on the notion of variables and meta-variables, which allow decisions about the universal-strength instantiation rules $\vdash \forall$ and $\exists \vdash$ to be postponed until further proof information is available, in a fashion analogous to lazy evaluation. Rather than choosing a specific instantiation for the constructs associated with these rules, we replace the quantified variable with a meta-variable. Quantified variables of existential-strength (i.e. the ones manipulated by $\vdash \exists$ and $\forall \vdash$) are replaced by parameters. In order to prevent variable capture, we manage a list of forbidden substitutions, in which every parameter introduced after a meta-variable is a forbidden substitution for that meta-variable. This satisfies the provisos that are usually associated with universal-strength quantifiers[15]. The resulting inference rules operate over the restricted sequent calculus [11]. A set of forbidden substitutions, denoted by \mathcal{R} , are carried with each sequent in this variant of the sequent calculus, which appears in figure 12. In Figure 12 we use $addMeta_m$ and $addParam_y$ to denote the addition of the meta-variable m or the parameter y to the restrictions \mathcal{R} in the fashion described.

Now that we have described the basic design strategy for predicate calculus, we will define the necessary term and monad classes. We begin by describing the necessary rec-

$$\begin{array}{c}
 \forall \vdash \frac{[\phi]_x^m, \forall x.\phi, \Gamma \vdash \Delta \parallel addMeta_m(\mathcal{R})}{\forall x.\phi, \Gamma \vdash \Delta \parallel \mathcal{R}} \\
 \vdash \forall \frac{\Gamma \vdash [\phi]_x^y, \Delta \parallel addParam_y(\mathcal{R})}{\Gamma \vdash \forall x.\phi, \Delta \parallel \mathcal{R}} \\
 \exists \vdash \frac{[\phi]_x^y, \Gamma \vdash \Delta \parallel addParam_y(\mathcal{R})}{\exists x.\phi, \Gamma \vdash \Delta \parallel \mathcal{R}} \\
 \vdash \exists \frac{\Gamma \vdash \exists x.\phi, [\phi]_x^m, \Delta \parallel addMeta_m(\mathcal{R})}{\Gamma \vdash \exists x.\phi, \Delta \parallel \mathcal{R}}
 \end{array}$$

Figure 12: Inference rules for predicates in the restricted sequent calculus

ognizers and constructors for predicates, parameters, and meta-variables, and augment these with functions that extract the quantified variable and the body from a quantified expression. The resulting class is shown in figure 13. Next, we define a monad class for predicate logic. We need

```

class PropLogic t => PredLogic t where
  isForall :: t -> Bool
  isExists :: t -> Bool
  isParam  :: t -> Bool
  isMeta   :: t -> Bool
  mkForall :: t -> t -> t
  mkExists :: t -> t -> t
  mkParam  :: String -> t
  mkMeta   :: String -> t
  getVar   :: t -> t
  getBody  :: t -> t

```

Figure 13: A term class for predicate calculus

to manage a list of restrictions, as well as generate fresh meta-variables and parameters during instantiation. The *PredState* class, shown in figure 14, provides this functionality. The *getNextVar* and *getNextParam* functions return current copies of fresh meta-variables and parameters. The *incNextVar* and *incNextParam* freshen the currently stored meta-variable or parameter to ensure that the next call to the corresponding *get* functions will result in a new meta-variable or parameter. *getForbiddens* and *setForbiddens* provide a way to access and alter the a current set of restrictions, stored in a *ForbiddenList*. *ForbiddenList* manages forbidden substitutions as an association list of terms and lists of terms. The first element of each pair is a meta-variable, while the second element is a list of parameters that cannot be substituted for this meta-variable. As an example, the *ForbiddenList*: $[(m1, [p1, p2]), (m2, [p1])]$ indicates that $m1 \vdash p1$, $m1 \vdash p2$, and $m2 \vdash p3$ are forbidden substitutions. Now we may define the actual inference rules for the predicate calculus, again we will use a helper functions, shown in figure 16 to perform “uninteresting” tasks related to managing forbidden lists. *addParam* function adds a new parameter to the restriction list of every existing meta-variable, and the *addMeta* function adds a new meta-variable to the *ForbiddenList* with an empty set of restrictions. This functionality preserves the neces-

```

class (PredLogic t, BasicState m t) => PredState m t where
  getNextVar :: m t
  incNextVar :: m ()
  getNextParam :: m t
  incNextParam :: m ()
  getForbiddens :: m (ForbiddenList t)
  setForbiddens :: ForbiddenList t -> m ()

type ForbiddenList t = [(t, [t])]

```

Figure 14: A monad class for predicate calculus

sary list of restrictions for the restricted sequent calculus, as described above. We will also use $mConnL$ and $mConnR$, which are simply alternate versions of $connL$ and $connR$ from the propositional logic module that allow the sequent transformer function to be monadic. We show the implementations of the \forall rules only. The \exists rules follow the same general form. The sample rule definitions are shown in figure 15. Note that we will eventually need a unifier to remove metavariables from a proof. Prufrock’s unifier is based heavily on Jansson and Jeuring’s existing work[6] on polytypic unification, and thus is not presented in this paper. To avoid a detailed discussion of the unification system, we shall just assume that the $applySubst$ function substitutes its second argument for its first throughout the term specified by its third argument.

```

forallLeftRule :: (PredLogic t, PredState m t,
  MonadNondet m) -> InferenceRule m t
forallLeftRule = mConnL isForall transformer
  where transformer t (hs :- cs) =
    do v <- getNextVar
       fs <- getForbiddens
       incNextVar
       setForbiddens (addMeta v fs)
    let t' = applySubst (getVar t) v (getBody t)
    return [(t' : hs) :- cs]

forallRightRule :: (PredLogic t, PredState m t,
  MonadNondet m) -> InferenceRule m t
forallRightRule = mConnR isForall transformer
  where transformer t (hs :- cs) =
    do p <- getNextParam
       fs <- getForbiddens
       incNextParam
       setForbiddens (addParam p fs)
    let t' = applySubst (getVar t) p (getBody t)
    return [(hs :- (t' : cs))]

```

Figure 15: Encoding of the inference rules for predicate calculus

7. FROM INFERENCE RULES TO TACTICS

We now illustrate how the Prufrock framework converts abstract inference rules from various logic modules into uniform, composable tactics. We must begin with a discussion of the basic state that Prufrock uses to store ongoing proof data, as even the simplest proofs require some global state. This state stores and manage several elements: the main

```

addParam :: (PredLogic t) =>
  t -> ForbiddenList t -> ForbiddenList t
addParam y fs = map (\f@(m, fs') -> (m, y : fs')) fs

addMeta :: (PredLogic t) =>
  t -> ForbiddenList t -> ForbiddenList t
addMeta m fs = (m, []) : fs

mConnR :: (MonadNondet m) =>
  (t -> Bool) -> (t -> InferenceRule m t) ->
  InferenceRule m t
mConnR p f seq@(hs :- cs) =
  let mkNewSeq (t, ts) = f t (hs :- ts)
  in maybe mzero mkNewSeq (findRemove p cs [])

mConnL :: (MonadNondet m) =>
  (t -> Bool) -> (t -> InferenceRule m t) ->
  InferenceRule m t
mConnL p f seq@(hs :- cs) =
  let mkNewSeq (t, ts) = f t (ts :- cs)
  in maybe mzero mkNewSeq (findRemove p hs [])

```

Figure 16: Utility functions for predicate calculus rules

goal for the overall proof, and a set of current subgoals. For both of these elements we provide a pair monadic functions: an accessor that returns the state’s currently stored value, and a mutator that stores a new value in the state. Note

```

class (Term t) => BasicState m t where
  getMainGoal :: m (Sequent t)
  setMainGoal :: (Sequent t) -> m ()
  getSubGoals :: m (RTree (Sequent t))
  setSubGoals :: RTree (Sequent t) -> m ()

```

Figure 17: The BasicState class

that the $RTree$ type is simply an implementation of Rose trees, indexed by type Key , which are used to store the proof’s subgoals. The $applyRule$ and $applyDecProc$ functions, shown in figure 18 provide the basic interface between inference rules and tactics. These functions guarantee a uniform access of the prover’s basic state, freeing logic module implementers from low-level details relating to the prover’s actual manipulation of goals. The $tryLookup$ and $trySplice$ functions simply manipulate the rose tree of subgoals, attempting to look up a given key, or splice a set of subgoals into the tree at a given node¹. All of the rule application functions return tactics, which are defined as monadic actions with type $m ()$.

Now we can define tacticals, shown in figure 19, which combine tactics using monadic operations. Performing one tactic after another is already provided by the standard $Monad$ class’s sequence operator, and may be done in a straightforward manner, using Haskell’s standard **do** notation for sequencing monads. Choosing between two possible tactics is accomplished using the $orRule$ combinator, which joins the tactics with $msum$ to indicate non-determinism, and then

¹Note that splicing $[]$ is equivalent to deleting a node.

```

applyRule :: (BasicState m t, MonadNondet m,
             MonadIO m) =>
             InferenceRule m t -> Key -> m ()
applyRule rule k =
  do subs <- getSubGoals
     sub <- tryLookup subs k
     newSubs <- rule sub
     subs' <- trySplice subs k newSubs
     setSubGoals subs'
     'catchError' \e -> liftIO (putStrLn (show e))

applyDecProc :: (BasicState m t, MonadNondet m,
                MonadIO m) =>
                DecisionProcedure m t -> Key -> m ()
applyDecProc dp k =
  do subs <- getSubGoals
     sub <- tryLookup subs k
     dec <- (dp sub)
     if dec
       then do subs' <- trySplice subs k []
              setSubGoals subs'
              liftIO (putStrLn
                    ("QED (" ++ (show k) ++ ")"))
       else mzero
     'catchError' \e -> liftIO (putStrLn (show e))

tryLookup t k =
  maybe (throwError
        (strError "Subgoal not found"))
        return (lookupEltRT t k)

trySplice t k es =
  maybe (throwError
        (strError "Subgoal not found"))
        return (spliceRT t k es)

```

Figure 18: The *apply functions**

applies *mcommit* to the result, forcing a cut in the backtracking tree at this point. Removing *mcommit* and thus allowing backtracking to occur amongst the possible choices results in the *eitherRule* combinator, which provides a non-deterministic, backtracking selection between two tactics. We may generalize these two combinators using *foldr* to obtain *firstF* and *anyF*, which operate on lists of tactics.

Using these basic tacticals, we may now define more complex search strategies. We may wish, for instance to repeat a given tactic until it is no longer applicable. The *repeatRule* tactical accomplishes this. A similar tactical is depth first search. The function *depthFirst* applies the given tactic until the specified search-termination predicate is satisfied. If the given tactic is made up of several tactics joined by an *eitherRule*, then this tactical will produce a depth-first search function with each tactic as a possible expansion for each proof state.

8. CONCLUSION AND FUTURE WORK

Prufrock provides a robust set of modules and features extending well beyond the scope of this paper. In addition to full implementations of propositional logic and predicate

```

orRule :: (MonadNondet m) => m a -> m a -> m a
orRule r1 r2 = mcommit (r1 'mplus' r2)

eitherRule :: (MonadNondet m) => m a -> m a -> m a
eitherRule r1 r2 = r1 'mplus' r2

firstF :: (MonadNondet m) => [m a] -> m a
firstF rules = mcommit (foldr orRule mzero rules)

anyF :: (MonadNondet m) => [m a] -> m a
anyF rules = foldr orRule mzero rules

repeatRule :: (MonadNondet m) => m () -> m ()
repeatRule r = do r
              ((repeatRule r) 'mplus' return ())

depthFirst :: (MonadNondet m) =>
              m Bool -> m a -> m ()
depthFirst pred tac = do predResult <- pred
                        if predResult
                          then return ()
                          else do tac
                                depthFirst pred tac

```

Figure 19: Tacticals

calculus, it features a unification system, a polytypic term rewriting system, which can be used to rewrite equalities in proofs, and simplification of arithmetic expressions and conditional expressions. The prover framework also provides many useful interactive features, including a detailed logging system that tracks proof steps (even inside of large tactics such as depth-first search) and allows the user to save and replay proofs. The user may also pretty-print proofs to \TeX files. This functionality is implemented polytypically, so that any language's proofs may be printed with no additional overhead. Prufrock's full logic modules by interactive commands that may be composed along with the modules to create an interactive shell supporting predefined logics.

The framework has been successfully used in several contexts, primarily in the development of the Rosetta language for system-level design[2, 1]. In particular, Prufrock has been used for rapid-prototyping of type systems for the Rosetta evaluator [10], and for verifying Rosetta models [17]. In the case of type-checking, Prufrock has proved useful due to the simplicity with which type rules may be encoded and executed in the prover environments. A set of class instances for the Rosetta expression language has already been implemented, and work to extend these instances to facets, which may represent entire specifications, is underway. This makes it possible for an entire Rosetta specification to appear directly in proofs as a first-class object that may be manipulated according to its logical semantics. In addition to the Rosetta-project, Prufrock has been tested on problems from the TPTP library[16], which it processes natively in the TPTP syntax.

Prufrock also implements an interface into the ZChaff SAT-solver[12], which may be used as a decision procedure. Because the logical semantics required to classify sequents is specified entirely by the first-order logic modules, any lan-

guage that provides an interface to Prufrock’s first-order logic capabilities may be used in ZChaff without any additional work. This is accomplished by defining the clausifier and translation functions polytypically. We are currently exploring the possibility of using the Prufrock framework as an intermediary between various existing tools, such as ZChaff, and arbitrary abstract syntaxes. In addition, several tools can be used on the same abstract syntax, provided that the required translation functions were defined polytypically. Unfortunately, however linking to external tools requires syntactic translation, a problem that Prufrock was designed to avoid.

The primary feature of the Prufrock framework, however, remains its flexibility. Prufrock allows implementers to design separate modules independent of prover internals and syntactic concerns, then compose these modules into a working prover for any appropriate abstract syntax. This ameliorates the gap between tools and specification languages, by allowing a language’s proof tools to evolve with the language itself. Furthermore, once a logic module has been refined and finalized, its functionality remains the same, even as the language representing proofs and the execution details of the prover evolve.

9. REFERENCES

- [1] P. Alexander, D. Barton, and C. Kong. *Rosetta Usage Guide*. The University of Kansas Information and Telecommunications Technology Center, 2335 Irving Hill Rd, Lawrence, KS.
- [2] P. Alexander, D. Barton, C. Kong, and C. Menon. The rosetta strawman. Technical report, The University of Kansas, 2002.
- [3] R. Hinze. Deriving backtracking monad transformers. *ACM SIGPLAN Notices*, 35(9):186–197, 2000.
- [4] R. Hinze and J. Jeuring. Generic haskell: practice and theory, 2003.
- [5] P. Jansson and J. Jeuring. PolyP—A polytypic programming language extension. In *Conf. Record 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL’97, Paris, France, 15–17 Jan 1997*, pages 470–482. ACM Press, New York, 1997.
- [6] P. Jansson and J. Jeuring. Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, 1998.
- [7] P. Jansson and J. Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In *WGP*. Utrecht University, 2000. UU-CS-2000-19.
- [8] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text 2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, volume 1129, pages 68–114. Springer-Verlag, Berlin, 1996.
- [9] S. P. e. Jones. Haskell 98 language and libraries: The revised report, December 2002.
- [10] E. Komp, G. Kimmell, J. Ward, and P. Alexander. The Raskell Evaluation Environment. Technical report, The University of Kansas Information and Telecommunications Technology Center, 2335 Irving Hill Rd, Lawrence, KS, USA, November 2003.
- [11] R. Kumar, T. Kropf, and K. Schneider. Integrating a First-Order Automatic Prover in the HOL Environment. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 170–176, Davis, California, 1991. IEEE Computer Society Press.
- [12] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC’01)*, 2001.
- [13] U. Norell and P. Jansson. Polytypic programming in haskell, 2004.
- [14] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.
- [15] L. C. Paulson. *ML for the Working Programmer*, chapter 10, pages 397–444. Cambridge UP, second edition, 2000.
- [16] G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
- [17] J. Ward and G. Kimmell. Rosetta Theorem Prover. Technical report, The University of Kansas Information and Telecommunications Technology Center, 2335 Irving Hill Rd, Lawrence, KS, USA, June 2003.