Modules With Proofs

Jason J. Hickey
Department of Computer Science
Cornell University

The ML module system provides proven mechanisms for organizing and maintaining large programs through the use of *structures*, to implement program units, and *signatures*, that give an abstract specification of structures. A signature is a partial specification: it simply lists the components of the structure together with their types. For domains with security or timing requirements, it is important to give stronger guarantees about program behavior.

We address this issue by augmenting program signatures with formal specifications, and structures with proofs. Security specifications are given in terms of a type theoretic interpretation of the program implementation. The type theory required to support this extension is the translucent sum calculus of Harper and Lillibridge extended with equality and constraint types. The Curry-Howard isomorphism provides a correspondence between executable programs and their proofs, allowing control over the degree of security desired.

1. INTRODUCTION

Module systems provide a methodology for dividing a program into units that can be separately defined. The central tenet of the module system is to separate programs into *structures* that *implement* program units, and *signatures* that provide well-defined abstract interfaces to the implementations. Functors provide parameterized structures: functions from structures to structures. Signatures are partial specifications of modules, listing the components of the structure together with their types. This provides enough information to the compiler to catch simple errors. ML programs are *sound*—their execution will never result in a type error.

While modularity is a useful development tool, it imposes an efficiency penalty because run-time safety checks are required to enforce security properties that are not expressible in module signatures. This phenomenon is widespread today—hardware and software checks are required at the kernel/user abstraction boundary; migratory programs on the Internet rely on run-time security enforcement like sandboxing [28]. In general, modular systems exhibit a tension between program *efficiency* and program *security*, suggesting that one way to improve efficiency is by *static* security verification. An example of this is given by Necula and Lee [20; 21], who suggest the use of *Proof Carrying Code* to deliver a proof of secure behavior along with the migrant program, reducing the cost of secure evaluation to the one-time cost of proof checking.

Contact information: Jason J. Hickey, Department of Computer Science, Upson Hall, Cornell University, Ithaca, NY 14853 USA. Tel: 607-255-1372, email: `jyh@cs.cornell.edu`.
Support for this research was provided by the Office of Naval Research through grant N00014-92-J-1764, from the National Science Foundation through grant CCR-9244739, from DARPA through grant 93-11-271, and from AASERT through grant N00014-95-1-0985.

Although Necula verifies security at the assembly level, we believe that general security verification must occur at the source level. Security proofs can be hard; they require a declarative understanding of the program that can be used to interprete its operational behavior. The general security checking problem is undecidable, and sufficiently complex properties (like liveness) may require hand proofs, which are feasible only at the source level.

In this paper, we develop an extension to the ML module system to allow modules and their signatures to be augmented with formal specifications of program behavior. The extension is conservative, allowing security proofs to be added incrementally: as a program is developed, as security and efficiency requirements increase, and as the programmer develops proficiency. To some degree, our purposes align with those of Extended ML [14; 15]. However, unlike Extended ML, we divide the language into two parts: a meta language for formal specifications, and the ML object language to express executable programs. By separating the languages, we provide an expressive meta language embedded in a type theory that can state properties of security, optimization, and even computational complexity [2]. The connection between these languages is expressed through a type theoretic semantics based on the translucent sum calculus of Harper and Lillibridge [7] (hereafter called $F^{\triangleright}$), given a constructive interpretation, and extended with equality types and total types. We use the Curry-Howard isomorphism to allow formal specifications to be interleaved with their standard counterparts.

The main contribution of this paper is this formal extension to the ML module system. We have implemented an initial version of this module system using the Nuprl-Light theorem prover to extend the Objective Caml module system. The contribution can be divided into the following results:

—a semantics of modules based on an extension of the translucent sum calculus of Harper and Lillibridge, augmented with equality and constraint types,

—an integration of programs and proofs, obtained through a constructive interpretation of the calculus, using the Curry-Howard isomorphism to provide a correspondence between propositions and types,

—a system architecture, with an initial implementation.

The outline of this paper is as follows. Before giving the semantics of the extension, we cover a few informal examples that describe the extended module system. Following the semantics, we discuss the architecture of the system and our implementation, and give a discussion of related work.

2. EXTENDED MODULES

One commonly described formal data structure is a module implementing a "set," shown in Figure 1[1]. A set provides three operations: a `singleton` operation to construct a set containing a single element, a `union` operation that combines two sets, and a `mem` operation to test for membership in the set. The set contains a carrier type t for the type of sets, and an `elem` type for the type of elements in the set.

[1]For the examples in this papers, we use the syntax of the Objective Caml module system

```
module type SetSig = sig
     type t
     type elem
     val singleton :  elem -> t
     val union :  t -> t -> t
     val mem :  elem -> t -> bool
     axiom sing_mem :  ∀a, b: elem.(a = b: elem) ⇔ (mem a (singleton b))
     axiom union_mem :  ∀a: elem.∀s1, s2: t.
       (mem a (union s1 s2)) ⇔ ((mem a s1) ∨ (mem a s2))
     axiom ind :  ∀P: (t → Prop).(∀a: elem.P (singleton a)) ⇒
       (∀s: t.(P s) ⇒ (∀a: elem.P (union s (singleton a))) ⇒
       (∀s: t.P s)
end
module Set :  SetSig = struct
     type elem = int
     type t = { l :  elem list | List.length l <> 0 }
     let singleton x = [x]
     let union s1 s2 = merge s1 s2
     let mem = List.mem
     thm sing_mem :  ∀a, b: elem.a = b: elem ⇔ (mem a (singleton b)) = ···
     thm union_mem :  ∀a: elem.∀s1, s2: t.
       (mem a (union s1 s2)) ⇔ ((mem a s1) ∨ (mem a s2)) = ···
     thm ind :  ∀P: (t → Prop).··· = ···
end
```

Fig. 1. Set module

Without the axiom specifications, the signature is a partial specification. For instance, the singleton operation could be implemented to return the empty set. The three additional axioms in the signature complete the specification. The first axiom relates the behavior of the singleton and mem functions, and the second relates the union and mem functions. The final axiom specifies an induction principle for sets: any proposition that is true on singleton sets and that remains true when elements are added is true for all sets. The induction principle closes off the module; the only set constructors are singleton and union.

The Set module implements a set of integers as a nonempty list (the merge function appends and sorts the two arguments and removes duplicate elements). The nonempty constraint is necessary for the proof of the induction principle. The proofs in the implementation follow from the properties of the List module, which also provides an induction principle. In our implementation, the proofs are generated interactively using a theorem prover. The largest proof in this example requires about 30 steps.

The module extension also applies to functors, as shown by the example in Figure 2. The module in this example provides a *choice* function that produces an element of the set passed as an argument. The choose function is not definable from just the ML part of the Set module; it *relies* on the computability of the Set.ind function to provide an induction combinator:

$$\texttt{set\_ind}\colon \forall\alpha.t \to (\texttt{elem} \to \alpha) \to (t \to \texttt{elem} \to \alpha) \to \alpha.$$

The signature for ChoiceSig is too weak to be able to express a formal specifi-

```
module type ChoiceSig = sig
     type t
     type elem
     val choose : t -> elem
end
module MakeChoice (Set : SetSig) : ChoiceSig
          with type t = Set.t
          with type elem = Set.elem
          with axiom spec : ∀s:t. Set.mem (choose s) s
     = struct
          type t = Set.t
          type elem = Set.elem
          let choose s = set_ind s (λa:elem.a) (λs:t.λa:elem.a)
          thm spec : ∀s:t. Set.mem (choose s) s = ···
end
```

Fig. 2. Set choice functor

cation. At the time the signature is defined, there is no notion of membership, or even that the type t is to represent a set. The solution we use in Figure 2 is to use sharing constraints to specify the result of the `MakeChoice` functor. A more general method that allows the *class* of functors to be defined requires a parameterized *signature*:

```
module type ChoiceSig (Set : SetSig) = sig
     type t = Set.t
     type elem = Set.elem
     val choose : t -> elem
     axiom spec : ∀s:t. Set.mem (choose s) s
end
```

This signature specifies that a correct implementation of the `choose` function must compute an element of the set provided as the argument. The `Set` parameter is required in this signature to be able to express the specification. The correct specification for the `MakeChoice` functor is now:

```
module MakeChoice (Set : SetSig) : (ChoiceSig Set).
```

3. STATIC SEMANTICS

As we will show, the type theory that we need for the semantics of the module system is an extension of the translucent sum calculus of Harper and Lillibridge ($F^{\triangleright}$) with the following extensions:

—In order to make use of the Curry-Howard isomorphism, a constructive calculus is required. The calculus $F^{\triangleright}$ is constructive, but the computational content of proofs must be described.

—To express program equivalence, equality types are introduced over programs and over types.

—To further constrain types with propositions, constraint types are introduced.

—To give significant propositional meaning to types, total types (types containing programs that do not diverge) are introduced.

Propositions			Programs		
Kind	K_P	$::= \Omega_P \mid K_P \Rightarrow K'_P$	Kind	K	$::= \Omega \mid K \Rightarrow K'$
Prop	P	$::= A \mid \Pi p{:}P.P' \mid \{D_{P_1},\ldots,D_{P_n}\} \mid \lambda\rho{::}K_P.P \mid P\ P' \mid V_P.\mathbf{c} \mid M = M'{:}A \mid A = A'{::}K$	Const	A	$::= \alpha \mid \Pi x{:}A.A' \mid \{D_1,\ldots,D_n\} \mid \lambda\alpha{::}K.A \mid A\ A' \mid V.\mathbf{b} \mid \{x{:}A \mid P\}$
Decl	D_P	$::= D \mid \mathbf{c} \triangleright \rho{::}K_P \mid \mathbf{p} \triangleright p{:}P$	Decl	D	$::= \mathbf{b} \triangleright \alpha{::}K \mid \mathbf{b} \triangleright \alpha{::}K = A \mid \mathbf{y} \triangleright x{:}A$
Term	M_P	$::= M \mid \lambda p{:}P.M_P \mid M_P\ M'_P \mid M_P{:}P \mid \{B_{P_1},\ldots,B_{P_n}\} \mid M_P.\mathbf{p} \mid \circ$	Term	M	$::= x \mid \lambda x{:}A.M \mid M\ M' \mid M{:}A \mid \{B_1,\ldots,B_n\} \mid M.\mathbf{y}$
Bind	B_P	$::= B \mid \mathbf{c} \triangleright \rho = P \mid \mathbf{p} \triangleright p = M_p$	Bind	B	$::= \mathbf{b} \triangleright \alpha = A \mid \mathbf{y} \triangleright x = M$
Value	V_P	$::= V \mid \lambda p{:}P.M_P \mid M_P\ M'_P \mid M_P{:}P \mid \{B_{VP_1},\ldots,B_{VP_n}\} \mid M_P.\mathbf{p} \mid \circ$	Value	V	$::= x \mid \lambda x{:}A.M \mid \{B_{V_1},\ldots,B_{V_n}\} \mid V.\mathbf{y}$
	B_{VP}	$::= B_V \mid \mathbf{c} \triangleright \rho = P \mid \mathbf{p} \triangleright p = V_P$		B_V	$::= \mathbf{b} \triangleright \alpha = A \mid \mathbf{y} \triangleright x = V$
Context	Γ	$::= \bullet \mid \Gamma, \rho{::}K_P \mid \Gamma, \alpha{::}K = A \mid \Gamma, p{:}P$			

Fig. 3. Syntax rules for the calculus ($n \geq 0$)

The syntax for calculus we use is shown in Figure 3. The calculus $F^{\triangleright}$ extended with constraint types, is shown on the right as the syntax of "programs." The additional levels on the left are for the syntax of "propositions." The main syntactical differences with $F^{\triangleright}$ are in the *proposition* level P, which extends the constructor level with equality types, and the new constraint constructor $\{x{:}A \mid P\}$. The propositional kind level K_P is needed to described the class of propositions. The term class T_P is extended with the term $\circ$, which is a unique term that will be used to inhabit equality types. Propositional terms may contain type quantifications over propositions. Since it is unlikely that the propositions would be extended with effects, the propositional *values* are allowed to use the full collection of propositional term constructors. Contexts are extended with propositional assumptions.

The meta variable ρ is drawn from the collection of proposition variables, α from the constructor variables, p from the propositional term variables, and x is from the collection of term variables. Similarly the meta variable $\mathbf{c}$ is drawn from the collection of proposition labels, $\mathbf{b}$ from the collection of constructor labels, $\mathbf{p}$ from the proposition term labels, and $\mathbf{y}$ from the collection of term labels.

3.1 Equality propositions

The equality proposition expresses equalities between programs $M = M'{:}A$, and equalities between constructors $A = A'{::}K$. An equality type contains the single canonical element $\circ$ if the equality is true, otherwise the equality type is empty. Since arbitrary programs can be compared using equality types, membership in an equality type is undecidable. For this reason, equality types are restricted to the level of propositions, which may be used only in formal specifications (where decidability is not an issue).

The semantics of equality is intensional, except for functions. Two types are

$$
\begin{array}{rcl}
\forall x\colon P.P' & \equiv & \Pi x\colon P.P' \\
\exists x\colon P.P' & \equiv & \{x \rhd x\colon P, \_ \rhd \_\colon P'\} \\
P \Rightarrow P' & \equiv & \forall \_\colon P.P' \\
P \wedge P' & \equiv & \exists \_\colon P.P' \\
P \vee P' & \equiv & \texttt{type Left of } P \mid \texttt{Right of } P'
\end{array}
$$

Fig. 4. Propositions-as-types

equivalent if they are beta-convertible, and two programs are equivalent if they a beta-convertible, or they are functions that are extensionally equal. The rules for equalities on constructors is unchanged from $F^{\rhd}$, and a similar set of rules is required for equality on terms. For instance, the standard rule for functions requires them to be lambda abstractions:

$$\frac{\Gamma, x\colon A \vdash M = M'\colon A'}{\Gamma \vdash \lambda x\colon A.M = \lambda x\colon A.M'\colon \Pi x\colon A.A'}\ \textit{E-lam}.$$

Extensional equality on functions is provided by the following rule. The two initial assumptions are necessary to show that f_1 and f_2 are indeed functions, and the final assumptions states that the functions evaluate to equal values on equal arguments, or both functions diverge.

$$\frac{\Gamma \vdash f_1 : \Pi x\colon A.A' \quad \Gamma \vdash f_2 : \Pi x\colon A.A' \quad \Gamma, x\colon A \vdash f_1\ x = f_2\ x\colon A'}{\Gamma \vdash f_1 = f_2\colon \Pi x\colon A.A'}\ \textit{E-lam-ext}.$$

Extensional function equality is crucial for optimization to enable algorithmic program transformation. If the functions f and f' are equal, then f can be replaced by f' even if f' is computationally different. For optimization, we also need a substitution rule:[2]

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash x = M\colon A}{\Gamma \vdash M' = [M/x]M'\colon A}\ \textit{SUBST}.$$

3.2 Propositions

The *proposition* level is used to express specifications. The interpretation of proposition relies on the Curry-Howard isomorphism to establish a correspondence between types and propositions. Using this correspondence, the interpretation of types is given in Figure 4 (in this Figure, the _ variable stands for "fresh" or nonbinding variable). Note that the disjunction relies on the introduction of datatypes.

In general, a proposition is *true* if there is a term that has the proposition as its type. For instance, the proposition $\forall \rho\colon\colon \Omega_P.\rho \Rightarrow \rho$ is true because it is inhabited by the program $\lambda \rho\colon\colon \Omega_P.\lambda x\colon \rho.x$. Similarly, the proposition $\lambda \rho\colon\colon \Omega_P.\rho$ is false because there is no function that can compute a value of every type. For convenience, we can define the proposition *true* as any nonempty proposition, say $\{\} = \{\}\colon \{\}$, which is inhabited by the canonical program $\circ$. The proposition *false* can be represented by an empty type, say $\forall \alpha\colon\colon \Omega.\alpha$.

The addition of propositions adds a new judgment form $\Gamma \vdash P$. One method of

[2]In a calculus with effects, this rule will have to be modified. A possible choice is to limit substitution to functional computations.

proof is to show that there is a program that implements the proof:

$$\frac{\Gamma \vdash M : P}{\Gamma \vdash P}\ \textit{F-WIT}.$$

In practice, this is a difficult rule to use because it requires guessing a correct witness M before completing the proof. For this reason we add incremental formation rules for each of the type constructors (no additional rules are need for equality types). Each of these rules follows from the logical interpretation of the proposition. For example, the formation rule for a universal quantifier is as follows:

$$\frac{\Gamma, x\colon A \vdash A'}{\Gamma \vdash \Pi x\colon A.A'}\ \textit{F-LAM}.$$

3.3 Constraint types

In the `Set` implementation of Figure 1, a constraint type was required to restrict the list implementation to non-empty lists. Without this constraint, the induction principle would not hold because of the existence of empty sets. In general, the constraint type $\{x\colon A \mid P\}$ contains the terms in A that also satisfy the proposition P.

The constraint type is a mix of a program and a proposition. As with equality, membership in a constraint type is not decidable. To preserve type checking and computational behavior, constraints are eliminated before being passed to the ML compiler. This has an important consequence—for elements in the constraint type $\{x\colon A \mid P\}$ the computational content of the proof of P is *not* available. For example, if $f\colon \{g\colon \mathit{Int} \rightarrow \mathit{Int} \mid \exists i\colon \mathit{Int}.g\ i = i : \mathit{Int}\}$, a fixpoint $f(i)$ is known to exist, but the particular fixpoint is omitted. In general, the sequent $x\colon \{x'\colon A \mid P\} \vdash [x/x']P$ may not be provable.

However, the truth of the constraint P is known, even its proof is not. For goals that do not require a computational interpretation, such as for membership and equality, the truth of the constraint can be made available. This also holds in the membership in a constraint type. for the sequent Γ, the expose function $|\Gamma|$ replaces every assumption $x\colon \{x'\colon A \mid P\}$ with two assumptions, $x\colon A$ and $[x'/x]P$.

$$\frac{\Gamma \vdash M\colon A \quad |\Gamma| \vdash [M/x]P}{\Gamma \vdash M\colon \{y\colon A \mid P\}}\ \textit{F-CON}$$

3.4 Constructivity

The next step in the development of the calculus is to give computational meaning to the types. It is a well-known property of constructive proof that the computational content can be computed from the proof. We adopt the technique used to Nuprl, which annotates each step of the proof with a schema for constructing the computational content.

It is only necessary to construct the computational content for the propositional judgment $\Gamma \vdash P$ just introduced. We denote the schema for the computational content by labeling each of the premises of the rule with a variable standing for its content. The conclusion is then labeled with the program computed from each of the premises. An example of a rule of this form is the rule for dependent function formation:

$$\frac{\Gamma, x\colon P \vdash P' \quad \texttt{ext}\ x_1}{\Gamma \vdash \Pi x\colon P.P' \quad \texttt{ext}\ \lambda x\colon P.x_1}\ \textit{F-DFUN.}$$

The constructive content of an equality proposition is the trivial $\circ$ program, regardless of the premises of the rule.

$$\frac{\Gamma, \alpha::K \vdash A::K' \quad \Gamma \vdash A'::K}{\Gamma \vdash (\lambda\alpha::K.A)A' = [A'/\alpha]A::K' \quad \texttt{ext}\ \circ}\ \textit{E-BETA}$$

3.5 Partial types

In a calculus with propositions, it is important to differentiate between partial and total terms. In a calculus of partial types, each type contains every nonterminating program. If propositions are interpreted with partial types, their meaning becomes trivial. Consider, for example, the proposition $\forall\rho::\Omega.\texttt{false}$. In the current interpretation, this type is empty, because there is no program that is a proof of `false` (as defined in Section 3.2). In a calculus of partial types, this proposition would be *true* because it would be inhabited by every nonterminating program (such as $\mu x::\Omega.x$).

To address this problem, we introduce total types; programs have partial types, and propositions use total types to make statements about partial programs. The complete treatment of partial types is based on the work of Crary [4], which simplifies the results of Smith [26; 27]. This treatment of partial type provides an induction principle expressed with the following rule for a fixpoint combinator $\texttt{fix}(x\colon A)$:

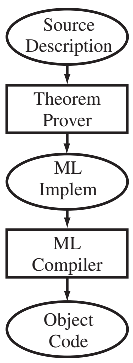
$$\frac{\Gamma \vdash x\colon \overline{A} \to \overline{A}}{\Gamma \vdash \texttt{fix}(x\colon A)\colon \overline{A}}\ \textit{F-FIX}$$

The relation between partial and total types is expressed with rules for halting. A program inhabits a total type $M : A$ if it inhabits the partial type $M : \overline{A}$ and M halts (denoted $M \downarrow A$). Halting is a typed judgment, defined by induction over the structure of programs. For example, a translucent sum halts if all its fields halt.

$$\frac{\Gamma \vdash x\colon A \quad \Gamma, x \downarrow A \vdash \{D_1, \ldots, D_n\} \downarrow \Omega}{\Gamma \vdash \{\mathbf{y} \triangleright x\colon \overline{A}, D_1, \ldots, D_n\} \downarrow \Omega}\ \textit{T-SUM}$$

The introduction of new rules for halting more than doubles the size of the calculus, with half of the calculus devoted to partial programs, and the other half devoted to total propositions.

4. SYSTEM ARCHITECTURE AND IMPLEMENTATION



The system architecture for supporting extended modules includes a theorem prover as a front end to the compiler, as shown in the Figure at the right. The theorem prover preprocesses the input programs to verify that the specifications are well-formed, and that the implementations satisfy their specification. The theorem prover is equipped with the static semantics we have described, as well as a dynamic semantics of the programming language.

The theorem prover is strictly a front end. The output of the theorem prover after static analysis is an intermediate form expressed in the unextended ML module system, containing both the executable contents of the module as well as the proofs of specification.

The theorem prover performs the following tasks:

—proof assistance for generating formal proofs,

—proof checking to show implementations satisfy their specification,

—static code optimization,

—elimination of formal terms, including:
  —type simplification,
  —elimination of parameterized signatures,
  —construction of computational content.

The results we have shown apply to any module system that uses the Harper-Lillibridge module calculus, such as the SML module system, or the KML [3] language, which provides higher-order modules with subtyping and first class polymorphism. Our implementation, called Nuprl-Light, is based on the Objective Caml programming language [25] from INRIA. The module system extension is implemented as a front end to the OCaml compiler using the Camlp4 preprocessor [18]. One of the intents of our system is to enhance the module system to the extent that it can be used to implement *type theories*. An initial summary of the logical framework is described in Hickey [11].

The semantics we use for the calculus is actually based on the Nuprl type theory. The Nuprl type theory is an expressive, predicative type theory with *untyped* terms. One effect this has on the calculus is that instead of a single type universe Ω, there is actually a hierarchy of universes $\Omega_1, \Omega_2, \ldots$, where each universe Ω_i belong to universe Ω_{i+1} but not vice versa. In practice, this introduces few problems, although it does prevent self-application.

Currently, our implementation is not complete. Although the formal reasoning part of the module system is implemented, the OCaml semantics is based on an interpretation of OCaml with limited effects by Kreitz [16], which is being used to verify the Ensemble [9] group membership system. While we do not plan to implement a complete semantics of OCaml, the present semantics can be usefully expanded. In the following sections we discuss the tasks of the implementation in more detail.

4.1 Proof generation and checking

One feature of the propositions-as-types methodology we have used is that the verification semantics *is* the static semantics. Every program that type checks is correct. However, the proof search problem is undecidable, and the proofs needed to satisfy the specifications are often complex. In our experience using the Nuprl theorem prover, we have found the need to combine interactive proof with proof automation.

The Nuprl-Light system is a *tactic*-oriented theorem prover, in the style of LCF [23]. A tactic is an ML program that implements a backward-chaining step of logical reasoning. Tactics are generated using the module mechanism with the `axiom` form. When an axiom is declared in a signature, the theorem prover provides a tactic that can be used to prove instances of the axiom.[3] For example, consider the `MakeChoice` functor in Section 2, which has the following abbreviated form:

```
module type SetSig = sig
    ··· axiom ind :  ∀P:t → Ω.···
end
module MakeChoice (Set :  SetSig) :  ChoiceSig= struct
    ··· thm spec :  ∀s:t.Set.mem (choose s) s
end.
```

The proof of the `spec` specification in the `MakeChoice` functor relies on the induction principle `ind` from the `Set` argument. When `spec` is proved, the theorem prover provides a tactic (also called `ind`) that represents the `ind` declaration in the context.

Each rule in the calculus has a corresponding tactic that applies the rule to refine a goal by backward chaining. The correctness of tactics is enforced by the ML type system; every term of type `tactic` is a tactic that either fails when it is applied, or it produces an correct proof tree. Tactics can be combined to produce proof search algorithms by using tactic combinators. The `andthen : tactic -> tactic list -> tactic` combinator represents proof composition, and the `orelse : tactic -> tactic -> tactic` implements alternation. The tactic (`orelse t1 t2`) tries the tactic `t1`; if it fails in its proof search `t2` is applied instead. A sequent is provable if, and only if, there is a tactic that proves the proposition without assumptions.

4.2 Type simplification

Once the theorem prover has checked the validity of a program, the next step is to produce a program that can be compiled. Since the static semantics have been verified, the program is known to be type sound, and the goal in this phase is to simplify types by erasure. There are three type extensions that must be considered: the constraint type, propositions, and parameterized signatures. As we mentioned in Section 3.3, constraint types are simplified by omitting the constraint. Propositions present more of a problem. The type theory for propositions is more powerful than the ML type system and it is not possible to simplify the type of every proposition, especially propositions with kinds and dependent types.

To preserve the shape of programs, type simplification is specified as a *partial* function in the meta language. Functions are modeled using the function type, and quantification over types is modeled as polymorphism (with the restriction that the result type must be expressible in prenex form).

$$\begin{aligned} \|\forall p{:}\, P.P'\| &= \|P\| \to \|P'\| \\ \|\forall \alpha{:}{:}\, \Omega.A\| &= \forall \alpha.\|A\| \end{aligned}$$

For the equality type, we choose a type `unit` containing a single element. Rather than giving a complete simplification for translucent sums, we restrict the algorithm

[3] As described in Hickey [11], the `axiom` mechanism is also used to define inference rules. In our implementation the module system is used both to define the type theory and the semantics of the programming language.

to existential types, which are modeled as tuples.

$$\begin{aligned} \|M = M' \colon A\| &= \texttt{unit} \\ \|\exists x \colon A.A'\| &= \|A\| * \|A'\| \end{aligned}$$

The translation of terms simplifies all occurrences of type constraints, and modules are translated by simplifying all occurrences of terms, types, and propositions. As a practical matter, axioms that have no translation are dropped from their module at compile time.

The final construct to consider is the parameterized signatures. Since parameterized signatures do not exist in ML, we introduce another phase to the phase distinction and require that all parameterized signatures be fully applied at compile time. Parameterized signatures are reduced to extended signatures by substitution, and the simplification algorithm is applied to produce an ML signature.

4.3 Verification and optimization

Once a program has been verified, the theorem prover produces a proof in several forms: as the tactic tree for the high level proof, as a primitive proof tree of rule instances, and as a program extracted from the computational content of the proof. In our implementation, the tactic tree for each `thm` form is saved in the module as a value of type `proof`, which is a proof at the source level. To produce a proof at the target level, the proof must be transformed by the compiler. Although we do not address this issue here, a promising path is suggested by Morrisett *et al.* [19], where type information is propagated by the compiler all the way to the assembly level.

Another use of the theorem prover is for *optimization* at the source level, based on program equivalence. Since the theorem prover is equipped to reason about program equivalence, strategies for program optimization are expressed as tactics that replace programs with equivalent, perhaps more efficient, programs.

As we mentioned in the introduction, one significant optimization is the elimination of safety checks that is enabled by static safety analysis. Another class of optimizations relies on *ad-hoc* transformations, where optimizations are implemented as compile-time *tactics*. If a proof search reveals that a safety check never fails, the check can be eliminated. Another example is function inlining, which can be implemented as beta-reduction if the theorem prover is allowed to cross module boundaries. The power of theorem prover optimization relies on the fact that it is safe tool that can be used by the programmer. Given the semantics of the programming language, it is not possible to construct an incorrect optimization (although some "optimizations" may result in a slower program).

Program transformation is also a class of *ad-hoc* optimization. An example of this is shown by the `MakeChoice` functor in Figure 2, which defines the `choose` function using the induction combinator on sets:

$$\texttt{let choose } s \texttt{ = set\_ind } s\ (\lambda a \colon \texttt{elem}.a)\ (\lambda s \colon t.\lambda a \colon \texttt{elem}.a)\texttt{.}$$

For the particular functor application `MakeChoiceSet` it is known that the `Set` is implemented as a list, and a more efficient implementation for `choose` is to pick the first element of the list.

```
let choose = List.hd
```

It would be difficult for a compiler to deduce an optimization of this form—one reason is that without the constraint that sets are nonempty, the `List.hd` function may raise an exception, while the `set_ind` implementation is pure. With the additional constraint specification, it can be shown that both implementations are purely functional, and the implementations can be interchanged. We are currently using this type of optimization for the Ensemble [9; 16] group communications system.

5. RELATED WORK

As we have mentioned before, we are extending the calculus of Harper and Lillibridge [7]. The module theorem prover is also related to the class of *logical frameworks*, and bears some resemblance to the Isabelle generic theorem prover [24; 22]. While both systems provide a generic framework for constructing type theories, logics, and deductions systems, the Nuprl-Light systems places more emphasis on integrating the theorem prover with the module system. Another influential logical framework is the Edinburgh Logical Framework (LF), which can be found in Harper [6]. Harper and Pfenning [8] have proposed a module system for the Elf programming language.

Module systems with formal specifications have a long history. For instance, in AUTOMATH, DeBruijn's *telescopes* [5] provide first class contexts. The closest relation to the system we present is the Extended ML systems implemented by Sannella et.al. [14; 15]. While both systems add specifications to modules, the approach is quite different. The Extended ML system is intended to preserve the feel of Standard ML, and the specification language is closely modeled on the Standard ML language. In contrast, our specification language is modeled on a type theoretic interpretation of the programming language, and we make a formal distinction between the meta and object languages. This reflects the difference in our intent—the Nuprl-Light system is also designed to provide a logical framework. There is a tradeoff: the Nuprl-Light programmer can express specifications in a powerful type theory, at the expense of having to *learn* the type theory.

The extension we propose to ML modules also apply to objects and classes. Hofmann et.al. [13] have implemented a system in LEGO [17] for adding proofs to classes. An interpretation of objects with proofs in the Nuprl type theory is presented in Hickey [10; 12].

6. CONCLUSION

The main contribution of this paper is an extension of the $F^{\triangleright}$ calculus to provide ML modules containing formal specifications and proofs. The Curry-Howard isomorphism serves to relate proofs with programs, providing complete control over the level of formality desired by the programmer. In addition, we present a new system architecture, joining a theorem prover with the compiler to provide an integrated environment for verification and optimization.

There are many areas for future work:

—Develop a more complete treatment of effects, partial functions, exceptions, and datatypes.

—Determine a methodology for compiling *proofs* so that they apply to the generated machine code.

—Investigate a formal calculus for parameterized signatures.

—Develop better strategies for proof, including strategies for verification and optimization.

REFERENCES

[1] Stuart F. Allen. A non-type-theoretic definition of martin-lof's types. In *Proceedings of the Second Conference on Logic in Computer Science*, pages 215–224, June 1987.

[2] Robert L. Constable. Expressing computational complexity in constructive type theory. In *LNCS/LNAI Proceedings*, pages 131–144. Springer, 1994.

[3] Karl Crary. Foundations for the implementation of higher-order subtyping. In *ACM SIGPLAN International Conference on Functional Programming*, pages 125–135, Amsterdam, 1997.

[4] Karl Crary. Recursive computation in foundational type theory. Technical report, Department of Computer Science, Cornell University, Forthcoming.

[5] N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91(2):189–204, April 1991.

[6] Rober Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1), January 1993.

[7] Robert Harper and Mark Lillibridge. A type–theoretic approach to higher–order modules with sharing. In *21st Annual ACM Symposium on Principles of Programming Languages*, pages 123–137. ACM, January 1994.

[8] Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*, To appear. A preliminary version is available as Technical Report CMU-CS-92-191.

[9] Mark Hayden. Ensemble tutorial. Available at the Ensemble home page through `www.cs.cornell.edu`, 1997.

[10] Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996. Available electronically through the FOOL 3 home page at Williams College.

[11] Jason J. Hickey. Nuprl-Light: An implementation framework for higher–order logics. In *14th International Conference on Automated Deduction*. Springer, 1997.

[12] Jason J. Hickey. A semantics of objects in type theory. Available through the Cornell home page `www.cs.cornell.edu`, 1997.

[13] M. Hofmann, W. Naraschewski, M. Steffen, and T. Stroup. Inheritance of proofs. In *TAPOS*. Wiley, forthcoming.

[14] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML. Technical Report ECS-LFCS-94-300, Laboratory for Foundations of Computer Science, University of Edinburgh, 1994.

[15] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.

[16] Christoph Kreitz. Formal reasoning about communication systems i: Embedding ML into type theory. Technical report, Department of Computer Science, Cornell University, Forthcoming.

[17] Zhaohui Luo and Randy Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-202, University of Edinburgh, 1992.

[18] Michael Mauny and Daniel de Ranglaudre. A complete and realistic implementation of quotations for ML. Camlp4 is available online `ftp.inria.fr`.

[19] G. Morrisett, D. Walker, and K. Crary. From System-F to typed assembly language. 1997.

[20] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, 1996.

[21] George C. Necula. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[22] Tobias Nipkow and Lawrence C. Paulson. Isabelle-91. In *Proceedings of the 11th International Conference on Automated Deduction*, pages 673–676, 1992.

[23] Lawrence C. Paulson. *Logic and Computation: Interactive proof with Cambridge LCF*. Cambridge Univ. Press, 1987.

[24] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer LNCS 828, 1994.

[25] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object–oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 40–53, 1997.

[26] Scott F. Smith and Robert L. Constable. Partial objects in constructive type theory. In *Proceedings of Second Symposium on Logic in Computer Science*, pages 183–193. IEEE, 1987.

[27] Scott Fraser Smith. *Partial Objects in Type Theory*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1989.

[28] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating Systems Principles*, 1993.

APPENDIX

A. RULES FOR TYPING

For brevity we collapse the propositions P with the type constructors A. The type constructors are extended with equality and constraint types, and the terms are extended with the canonical element $\circ$.

We give a *functionality* semantics to sequents [1], which allows extensions of the calculus to refine semantic equality. Technically, this proof theory has equal power to the calculus in Harper-Lillibridge. The main syntactic difference is in the reduction of judgment forms; contexts are always valid.

In our calculus, the proposition $M = M'\colon P$ is overloaded to mean that P is a proposition (propositions *are* types), and M and M' are equal programs in P (so $M\colon P$ is equivalent to $M = M\colon P$). The judgment $\Gamma \vdash P \; ext \; M$ means that P is a proposition in context Γ, and that it is inhabited by the program M. If M is trivial (as for an equality judgment $P = P'\colon\colon K$, then it may be omitted). Sequents are considered equal up to renaming.

Judgments

$$\begin{array}{ll}
\Gamma \vdash P \; \mathit{ext} \; M & \\
\Gamma \vdash P \leq P' & \textit{subtyping} \\
\Gamma \vdash D \leq D' & \textit{subfield}
\end{array}$$

Field name stripping

$$\begin{array}{lcl}
\langle \mathbf{c} \rhd \rho\colon\colon K \rangle & = & \rho\colon\colon K \\
\langle \mathbf{c} \rhd \rho\colon\colon K = P \rangle & = & \rho\colon\colon K = P \\
\langle \mathbf{p} \rhd p\colon P \rangle & = & p\colon P
\end{array}$$

Constraint expansion

$$\begin{array}{lcll}
|\bullet| & = & \bullet & \\
|\Gamma, \{p\colon P \mid P'\}| & = & |\Gamma|, p\colon P, p'\colon P' & (\textit{new } p') \\
|\Gamma, H| & = & |\Gamma|, H & (H \textit{ not a constraint type})
\end{array}$$

Constructor equality

$$\frac{\rho\!:\!:K \in \Gamma}{\Gamma \vdash \rho = \rho\!:\!:K}\ \textit{E-VAR-O} \quad \frac{\rho\!:\!:K = P \in \Gamma}{\Gamma \vdash \rho = \rho\!:\!:K}\ \textit{E-VAR-T}$$

$$\frac{\Gamma \vdash P = P'\!:\!:K}{\Gamma \vdash P' = P\!:\!:K}\ \textit{E-SYM} \quad \frac{\Gamma \vdash P = P'\!:\!:K \quad \Gamma \vdash P' = P''\!:\!:K}{\Gamma \vdash P = P''\!:\!:K}\ \textit{E-TRANS}$$

$$\frac{\Gamma \vdash V = V'\!:\!:\{\mathbf{c} \rhd \rho\!:\!:K\}}{\Gamma \vdash V.\mathbf{c} = V'.\mathbf{c}\!:\!:K}\ \textit{E-EXT-O} \quad \frac{\Gamma \vdash P_1 = P_2\!:\!:\Omega \quad \Gamma, p\!:P_1 \vdash P_1' = P_2'\!:\!:\Omega}{\Gamma \vdash \Pi p\!:P_1.P_1' = \Pi p\!:P_2.P_2'\!:\!:\Omega}\ \textit{E-DFUN}$$

$$\frac{\Gamma \vdash D = D' \quad \Gamma, \langle D\rangle \vdash \{D_1, \ldots, D_n\} = \{D_1', \ldots, D_n'\}\!:\!:\Omega}{\Gamma \vdash \{D, D_1, \ldots, D_n\} = \{D', D_1', \ldots, D_n'\}\!:\!:\Omega}\ \textit{E-TSUM}$$

$$\frac{\Gamma, \rho\!:\!:K \vdash P = P'\!:\!:K'}{\Gamma \vdash \lambda\rho\!:\!:K.P = \lambda\rho\!:\!:K.P'\!:\!:K \Rightarrow K'}\ \textit{E-LAM} \quad \frac{\Gamma \vdash P_1 = P_1'\!:\!:K \Rightarrow K' \quad \Gamma \vdash P_2 = P_2'\!:\!:K}{\Gamma \vdash P_1\ P_2 = P_1'\ P_2'\!:\!:K'}\ \textit{E-APP}$$

$$\frac{\Gamma, \rho\!:\!:K \vdash P = P\!:\!:K' \quad \Gamma \vdash P' = P'\!:\!:K}{\Gamma \vdash (\lambda\rho\!:\!:K.P)\ P' = [P'/\rho]P\!:\!:K'}\ \textit{E-BETA} \quad \frac{\Gamma \vdash P = P\!:\!:K \Rightarrow K'}{\Gamma \vdash (\lambda\rho\!:\!:K.P\ \rho) = P\!:\!:K \Rightarrow K'}\ \textit{E-ETA}$$

$$\frac{\Gamma \vdash V = V\!:\!:\{\mathbf{c} \rhd \rho\!:\!:K = P\}}{\Gamma \vdash V.\mathbf{c} = P\!:\!:K}\ \textit{ABBREV} \quad \frac{\rho\!:\!:K = P \in \Gamma}{\Gamma \vdash \rho = P\!:\!:K}\ \textit{ABBREV}'$$

Declaration equality

$$\frac{\Gamma \vdash P = P'\!:\!:K}{\Gamma \vdash (\mathbf{c} \rhd \rho\!:\!:K = P) = (\mathbf{c} \rhd \rho\!:\!:K = P')}\ \textit{EQ-T} \quad \frac{\Gamma \vdash P = P'\!:\!:\Omega}{\Gamma \vdash \mathbf{p} \rhd p\!:P = \mathbf{p} \rhd p\!:P'}\ \textit{EQ-V}$$

Term equality

$$\frac{p\!:P \in \Gamma}{\Gamma \vdash p = p\!:P}\ \textit{E-var} \quad \frac{p\!:\{p'\!:P \mid P'\} \in \Gamma}{\Gamma \vdash p = p\!:P}\ \textit{E-con} \quad \frac{\Gamma \vdash P = P\!:\!:\Omega \quad \Gamma, p\!:P \vdash M = M'\!:P'}{\Gamma \vdash (\lambda p\!:P.M) = (\lambda p\!:P.M')\!:\Pi p\!:P.P'}\ \textit{E-lam}$$

$$\frac{\Gamma \vdash M = M\!:\Pi p\!:P.P' \quad \Gamma \vdash M' = M'\!:\Pi p\!:P.P' \quad \Gamma, p\!:P \vdash M\ p = M'\ p\!:P'}{\Gamma \vdash M = M'\!:\Pi p\!:P.P'}\ \textit{E-lam-ext}$$

$$\frac{\Gamma \vdash M_1 = M_1'\!:P \to P' \quad \Gamma \vdash M_2 = M_2'\!:P}{\Gamma \vdash M_1\ M_2 = M_1'\ M_2'\!:P'}\ \textit{E-app} \quad \frac{\Gamma, p\!:P \vdash M = M\!:P' \quad \Gamma \vdash M' = M'\!:P}{\Gamma \vdash (\lambda p\!:P.M)\ M' = [M'/p]M\!:P'}\ \textit{E-beta}$$

$$\frac{\Gamma \vdash B = B'\!:D \quad \Gamma, \langle D\rangle \vdash \{B_1, \ldots, B_n\} = \{B_1', \ldots, B_n'\}\!:\{D_1, \ldots, D_n\}}{\Gamma \vdash \{B, B_1, \ldots, B_n\} = \{B', B_1', \ldots, B_n'\}\!:\{D, D_1, \ldots, D_n\}}\ \textit{E-tsum}$$

$$\frac{}{\Gamma \vdash \{\} = \{\}\!:\{\}}\ \textit{E-empty} \quad \frac{\Gamma \vdash M = M'\!:P \quad \Gamma, p\!:P \vdash P' = P'\!:\!:\Omega \quad |\Gamma| \vdash [M/p]P'}{\Gamma \vdash M = M'\!:\{p\!:P \mid P'\}}\ \textit{E-cons}$$

$$\frac{\Gamma \vdash M = M'\!:P}{\Gamma \vdash \circ = \circ\!:(M = M'\!:P)}\ \textit{E-equal-t} \quad \frac{\Gamma \vdash P = P'\!:\!:K}{\Gamma \vdash \circ = \circ\!:(P = P'\!:\!:K)}\ \textit{E-equal-c}$$

$$\frac{\Gamma \vdash V = V'\!:\{\mathbf{c} \rhd \rho\!:\!:K, D_1, \ldots, D_n\}}{\Gamma \vdash V = V'\!:\{\mathbf{c} \rhd \rho\!:\!:K = P, D_1, \ldots, D_n\}}\ \textit{VALUE-O}$$

$$\frac{\Gamma \vdash V.\mathbf{p} = V'.\mathbf{p}\colon P' \quad \Gamma \vdash V = V'\colon \{\mathbf{p} \rhd p\colon A, D_1, \ldots, D_n\}}{\Gamma \vdash V = V'\colon \{\mathbf{p} \rhd p\colon P', D_1, \ldots, D_n\}}\ \textit{VALUE-V}$$

$$\frac{\Gamma \vdash M = M'\colon P' \quad \Gamma \vdash P' \leq P}{\Gamma \vdash M = M'\colon P}\ \textit{SUBS} \quad \frac{\Gamma \vdash M : P \quad \Gamma \vdash p = M\colon P}{\Gamma \vdash M' = [M/p]M'\colon P}\ \textit{SUBST}$$

Binding equality

$$\frac{\Gamma \vdash P = P'\colon\colon K}{\Gamma \vdash (\mathbf{c} \rhd \rho = P) = (\mathbf{c} \rhd \rho = P')\colon \mathbf{c} \rhd \rho\colon\colon K}\ \textit{EQ-BIND-O}$$

$$\frac{\Gamma \vdash P = P'\colon\colon K \quad \Gamma \vdash P = P''\colon\colon K}{\Gamma \vdash (\mathbf{c} \rhd \rho = P) = (\mathbf{c} \rhd \rho = P')\colon \mathbf{c} \rhd \rho\colon\colon K = P''}\ \textit{EQ-BIND-T}$$

$$\frac{\Gamma \vdash M = M'\colon P}{\Gamma \vdash (\mathbf{p} \rhd p = M) = (\mathbf{p} \rhd p = M')\colon \mathbf{p} \rhd p\colon P}\ \textit{EQ-BIND-V}$$

Subtyping

$$\frac{\Gamma \vdash P = P'\colon\colon \Omega}{\Gamma \vdash P \leq P'}\ \textit{S-REF} \quad \frac{\Gamma \vdash P \leq P' \quad \Gamma \vdash P' \leq P''}{\Gamma \vdash P \leq P''}\ \textit{S-TRAN}$$

$$\frac{\Gamma \vdash P = P\colon\colon \Omega \quad \Gamma, p\colon P \vdash p = p\colon P'}{\Gamma \vdash P \leq P'}\ \textit{S-SUB}$$

$$\frac{\Gamma \vdash D \leq D' \quad \Gamma, \langle D \rangle \vdash \{D_1, \ldots, D_n\} \leq \{D'_1, \ldots, D'_n\}}{\Gamma \vdash \{D, D_1, \ldots, D_n\} \leq \{D', D'_1, \ldots, D'_n\}}\ \textit{S-TSUM}$$

$$\frac{\Gamma \vdash \{D_1, \ldots, D_n, D\} = \{D_1, \ldots, D_n, D\}\colon\colon \Omega}{\Gamma \vdash \{D_1, \ldots, D_n, D\} \leq \{D_1, \ldots, D_n\}}\ \textit{S-THIN}$$

Subfielding

$$\frac{\Gamma \vdash D = D'}{\Gamma \vdash D \leq D'}\ \textit{SUB-REF} \quad \frac{\Gamma \vdash P \leq P'}{\Gamma \vdash \mathbf{p} \rhd p\colon P \leq \mathbf{p} \rhd p\colon P'}\ \textit{SUB-VALUE}$$

$$\frac{\Gamma \vdash P = P\colon\colon \Omega}{\Gamma \vdash \mathbf{c} \rhd \rho\colon\colon K = P \leq \mathbf{c} \rhd \rho\colon\colon K}\ \textit{SUB-FORGET}$$

Proposition formation

$$\frac{|\Gamma| \vdash P = P\colon\colon \Omega \quad \Gamma, p\colon P \vdash P'\ \mathit{ext}\ M}{\Gamma \vdash \Pi p\colon P.P'\ \mathit{ext}\ \lambda p\colon P.M}\ \textit{F-LAM} \quad \frac{|\Gamma| \vdash M = M\colon P}{\Gamma \vdash P\ \mathit{ext}\ M}\ \textit{F-WIT}$$

$$\frac{\Gamma \vdash P = P\colon\colon \Omega \quad \Gamma, \rho\colon\colon K = P \vdash \{D_1, \ldots, D_n\}\ \mathit{ext}\ \{B_1, \ldots, B_n\}}{\Gamma \vdash \{\mathbf{b} \rhd \rho\colon\colon K = P, D_1, \ldots, D_n\}\ \mathit{ext}\ \{\mathbf{b} \rhd \rho = P, B_1, \ldots, B_n\}}\ \textit{F-TSUM-T}$$

$$\frac{\Gamma \vdash P\ \mathit{ext}\ M \quad \Gamma, p\colon P \vdash \{D_1, \ldots, D_n\}\ \mathit{ext}\ \{B_1, \ldots, B_n\}}{\Gamma \vdash \{\mathbf{p} \rhd p\colon P, D_1, \ldots, D_n\}\ \mathit{ext}\ \{\mathbf{p} \rhd p = M, B_1, \ldots, B_n\}}\ \textit{F-TSUM-V}$$

B. TERMINATION

To give propositions nontrivial meaning, propositions use total types containing only total programs. The following rules describe the extension of the calculus with partial types (denoted $\overline{P}$). We give a sample of the extension here.

Termination judgment

$$\Gamma \vdash M \downarrow P \quad \textit{evaluation of } M \textit{ halts, and } M\colon P$$

Partiality

$$\frac{\Gamma \vdash \mathtt{fix}(M) = \mathtt{fix}(M')\colon \overline{P}}{\Gamma \vdash M = M'\colon \overline{P} \to \overline{P}}\ \textit{P-FIX} \qquad \frac{\Gamma \vdash M = M'\colon \overline{P} \quad \Gamma \vdash M \downarrow P'}{\Gamma \vdash M = M'\colon P}\ \textit{P-TERM}$$

Termination

$$\frac{\Gamma \vdash P = P' \colon\colon \Omega \quad \Gamma \vdash M = M'\colon \overline{P'}}{\Gamma \vdash \lambda p\colon P.M \downarrow P \to P'}\ \textit{T-LAM} \qquad \frac{\Gamma \vdash M \downarrow P}{\Gamma \vdash \mathbf{p} \rhd p = M \downarrow \mathbf{p} \rhd p\colon P}\ \textit{T-BIND-V}$$

$$\frac{\Gamma \vdash B \downarrow D \quad \Gamma \vdash \{B_1, \ldots, B_n\} \downarrow \{D_1, \ldots, D_n\}}{\Gamma \vdash \{B, B_1, \ldots, B_n\} \downarrow \{D, D_1, \ldots, D_n\}}\ \textit{T-SUB} \qquad \frac{\Gamma \vdash \{B_1, \ldots, B_n\} \downarrow \{D_1, \ldots, D_n\}}{\Gamma \vdash B_i \downarrow D_i}\ \textit{T-SUP}$$

C. PROGRAM SIMPLIFICATION

In lieu of extending the ML compiler, program simplification is used to give ML types and implementations to the proof component of modules. The proofs are already type correct; the goal is to find a well-formed ML type. The simplification is partial because some formal types have no corresponding ML approximation.

$$\begin{array}{rcl@{\qquad}rcl}
\|\alpha\| & = & \alpha & \|p\| & = & p \\
\|\forall p\colon P.P'\| & = & \|P\| \to \|P'\| & \|\lambda p\colon P.M\| & = & \lambda p\colon \|P\|.\|M\| \\
\|\forall p\colon \Omega.P\| & = & \forall \alpha.\|P\| \quad (\text{new } \alpha) & \|M\ M'\| & = & \|M\|\ \|M'\| \\
\|\exists p\colon P.P'\| & = & \|P\| * \|P'\| & \|M\colon P\| & = & \|M\|\colon \|P\| \\
\|\{p\colon A \mid P'\}\| & = & \|A\| & \|\{B_1, \ldots, B_n\}\| & = & \{\|B_1\|, \ldots, \|B_n\|\} \\
\|A = A' \colon\colon K\| & = & \mathtt{unit} & \|M.\mathbf{p}\| & = & \|M\|.\mathbf{p} \\
\|M = M'\colon A\| & = & \mathtt{unit} & \|\circ\| & = & \circ
\end{array}$$