A Logical Framework for Software Proof Reuse*

Joshua E. Caplan
caplan@cs.uiuc.edu

Mehdi T. Harandi
harandi@cs.uiuc.edu

Department of Computer Science, University of Illinois at Urbana-Champaign

Abstract

We describe a logical framework *PR* for verification of reusable software components. Within our system, developers can employ the advantages traditionally associated with software reuse to reduce the cost of software verification by reusing abstract proofs and specifications. One can construct an algorithm with parameters, a specification with parameters, and a proof that the algorithm satisfies the specification *provided* the parameters satisfy certain conditions. Proofs in *PR* will themselves contain parameters for subproofs concerning those conditions. In this framework, typing, type checking, and proof checking are decidable.

1 Introduction

A reusable software artifact is most useful when it carries verification information, including assertions, data invariants, and restrictions on subtasks or data structure operations. The aim of our research has been to provide a logical framework which allows one to construct an algorithm with parameters, a specification with parameters, and a proof that the algorithm satisfies the specification *provided* the parameters satisfy certain conditions. We treat the proviso above as a need to allow parameters in the proof as well (ranging over proofs of the requisite conditions).

Once one has built a library of widely-applicable parametric algorithms and their proofs, one can verify each closed instance (program) by plugging appropriate sub-proofs into the abstract proof rather than constructing a complete proof from scratch. *Proof reuse* is just this practice. The redundancy in proving the correctness of similar algorithms is minimized or eliminated in much the same way as in programming similar algorithms. The cost of building the abstract proof for the abstract algorithm can be allocated to the many concrete algorithms and proofs created from it, substantially reducing the cost of program verification. In short, the many benefits of software reuse can also be enjoyed in software verification.

Harandi and Young [5] argue that a standard set of correct abstract algorithms is necessary to reduce the development time and improve the quality of new software. Within each individual domain, specialization modules for use with these abstract algorithms can be developed separately and plugged into them later. Software development becomes a simple, mechanizable process of plugging pre-developed and pre-verified components together. After some research, we have come to know this utopian vision as "Code Reuse Run Amok," for it overlooks the *increase* in complexity arising when modules for specialization have to be contorted to conform to the specifications given in their destined abstract algorithm. In order for an algorithm to be abstract enough to function across vastly different domains, it must be applicable irrespective of the data type and representation of its target, and thus consist solely of control information. Standard algorithms, however, tend to be closely coupled to aspects of the data structures underlying them. The specialization modules end up containing nearly all the interesting code. Extracting the trivial remaining control information into abstract algorithms and breaking up the specialization modules according to that control seems to create more work than using a standard object-oriented or functional approach to development.

The paradigm we suggest as an evolutionary response to [5] lets programmers remain programmers. It is the logical framework for constructing highly abstract programs which must be tractable for developers, so that they can pursue their own reuse strategy and still mathematically verify the correctness of their software, no matter how abstract.

Our framework consists of a formal system *PR*

*This work is partially supported by NASA grant NAG 1-613 (ICLASS).

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
SSR '95, Seattle, WA,USA
© 1995 ACM 0-89791-739-1/95/0004...$3.50

for building programs in an Algol-like language and program proofs in a Hoare-like logic. Typing, type checking, and proof checking are decidable. We chose this programming language because it is a relatively clean integration of imperative programming with higher-order functional programming, so it is expressive enough for most typical kinds of reuse. We chose Hoare Logic as a starting point, for it remains the clearest system for reasoning about basic imperative programs. The system we have developed is interesting in its own right, but the technique used to create it is perhaps of more importance, as it appears to be applicable to the creation of similar systems for a variety of programming languages and logics. By virtue of the technique, there is room for improvement in two orthogonal directions, upon which we will elaborate further in the conclusion.

Programmers cannot be expected to work within a formal system by hand; the system can be implemented on a computer in the form of a proof assistant for developers. Our approach has proved beneficial in this respect, as well, for producing an implementation of *PR* is as simple as entering its signature into any existing implementation of the Edinburgh Logical Framework (LF), of which there are several. We have also built a simple implementation of LF and used it in this way to debug and check the toy example in section 5. Existing heuristics for automatic verification, and the notion of proof *tactics* [4], could also be incorporated to partially automate proof search in a more practical implementation.

2 Related Work

Constructive Type Theory in computer science has been an active area of research for the last fifteen years. Most incarnations are built around a typed λ-calculus in which terms and proofs are identified, and likewise types and specifications are identified, so that uniform abstraction is possible at all levels (see, for example, [8, 17]). These systems provide reasonable support of the kind we seek for purely functional languages, but are usually lacking in handling general recursion or imperative features. Adding general recursion has been explored most prominently by R. Constable and S. Smith [3, 14, 15]. Some work, in particular the dissertation of V. Swarup [16], has focused on extending type theory to deal with the addition of assignments to the programming language. The resulting language enjoins a novel continuation-passing style of programming, and an even more unusual form of proof.

Mainstream attempts to extend Hoare Logic directly as a proof calculus for procedures have had limited success, usually restricted to a small portion of the full λ-calculus of Idealized Algol [11], and not supporting abstractions of proofs or dealing with interference. J. Reynolds has embedded Hoare Logic as the atomic formulas of a first-order theory for Algol called Specification Logic [12], which incorporates predicates for directly reasoning about interference between identifiers. Another approach, also spearheaded by Reynolds [10, 13] and further taken up by P. O'Hearn and R. Tennent [9] has been to restrict the programming language using syntactic means to prohibit interference from occurring, which enables a much cleaner form of Specification Logic but limits the set of valid programs in non-trivial ways.

Specifically pertaining to our technique, there have been two published formalizations of Hoare Logic in the Edinburgh Logical Framework, which appear in [1, 7]. In one formalization, new judgment forms denoting non-interference and non-identity of program variables are used to express a correct assignment axiom, as in Specification Logic. The other formalization is a *family* of signatures, one signature for each choice of program variable set; each one turns out to be adequate for a fragment of our calculus QHL_0 (section 4.1.3), but certainly not for full Hoare Logic.

3 Reuse and Abstraction

The essence of reusability is abstraction. In the system we seek, there is a clear need for abstraction at the following levels.

Values: Functions and procedures, which arise from value abstraction, are often reused many times within a single piece of software.

Types: The presence of polymorphism greatly increases the reusability of datatype-independent algorithms.

Programs: Reuse across the development of different systems can be facilitated by allowing meta-programs and higher-order programs.

Specifications: The specification of a sufficiently abstract artifact may depend on the specifications of its parameters.

Proofs: In general, we can only verify a reusable artifact in terms of assumed proofs concerning its parameters.

Systems for software reuse and verification usually include abstraction of only the first three kinds, and systems for formal specification often allow abstraction of

the fourth sort as well (but have no formal notion of proof).

Having identified these categories of need, the tendency might be to design a framework which takes the five kinds of abstraction as separate primitive notions. The result would be both excessively complicated, for with five kinds of lambda one needs five times as large a semantics, and unnecessarily restrictive, for there is no reason to believe the above list is exhaustive. Instead, we have sought a solution within the elegance and simplicity of type theory.

The following examples are intended to demonstrate both the utility of the approach and the necessity for the levels of abstraction listed above.

3.1 Choose

Our domain is arithmetic on the reals and the natural numbers; we intend to implement a program *choose* to compute the binomial coefficient $\binom{n}{k}$ of numbers n and k in terms of the factorial expansion $n!/(k!(n-k)!)$. That is, we will assume the existence of

- a program f for computing the factorial of a number, and
- a proof ϕ that f is correct.

In order to write the desired program c in terms of f and give a proof χ, in terms of f and ϕ, that c is correct, our system must support abstraction over values (so that c can be a procedure), programs (so that c can refer to f), and proofs (so that χ can refer to ϕ). Having built χ, we can implement *choose* using any implementation of f and assure that *choose* is correct just by proving the correctness of f. This example will be taken up further in section 5.

3.2 Flowlines

Our domain is geological map algebra. We desire an algorithm *flow* which takes an elevation map and aspect (direction of water flow) map for the same area and produces a runoff density map and a flow vector map. Parameters include

- the type **coord** of coordinates, where maps consist of finite functions from coordinates to values,
- the type **point** of points,
- a hash function H from points to coordinates,
- a representative function R from coordinates to designated points within them, and
- a routine n which computes the next point on a flow vector given elevation and aspect maps. Different algorithms for n reflect different geological models of flow.

The types **coord** and **point** and the functions H and R properly belong in a standard library for map algebra. As new representations and implementations of the library are developed, the duly updated instances of the flowline algorithm should remain correct without any modifications. We may even want to allow access to various versions of n as options at runtime.

In the specification, we insist that for any input maps E and A, *flow* produces output maps D and F, where each coordinate c in D contains the number of coordinates in F with flow vectors passing through c, and each coordinate d in F contains a flow vector such that each point follows from the preceding via n. Because n is by nature an approximation, we cannot constrain it (aside from giving its type), but we still use it as a parameter in the abstract specification of *flow*.

The algorithm itself runs as expected:

```
for each coordinate c
        l := ⟨⟩ ; p := R(c) ;
        repeat
                add p to l ;
                c := n(p)(E)(A) ;
                case c of SOME p' ⟹
                        if H(p') ∉ {H(q) | q ∈ l}
                          increment D(H(p')) ;
                        p := p'
        until   c = NONE ;
        F(c) := l
```

In the proof of the algorithm, we will rely on proof parameters validating the correctness of the library functions, demonstrating things like $(\forall x)[H(R(x)) = x]$.

3.3 Divide and Conquer

Our domain is completely generic. The parameters of our algorithm *divconq* will be the input type **D**, output type **A**, operations s, d, c, and m, specification S, and proofs σ and λ. Intuitively,

- s decides if an input object is small enough to be directly solved by c,
- σ is a proof that c is correct with respect to S for small input,
- d divides a large input object into a list of smaller input objects,

- m merges a list of output objects into a single output object, and
- λ is a proof that for large input k, if l is a list of correct (with respect to S) outputs for the members of $d(k)$, then $m(l)$ is a correct output for k.

The algorithm looks something like

$$\begin{aligned} divconq\,(i) \stackrel{\text{dfn}}{=} & \text{ if } s\,(i) \text{ then } c\,(i) \\ & \text{ else } m\,(\text{map}\,(divconq)\,(d\,(i))) \end{aligned}$$

and the specification indicates that given satisfactory parameters as above, for any input i, $S(i, divconq\,(i))$. To formally construct artifacts and the proof of correctness δ, we must support all five types of abstraction. The resulting proof δ will perform an abstract induction; to confidently use *divconq* in an actual program one must merely instantiate its parameters and build the two smaller proofs σ and λ.

With an artifact as abstract as *divconq*, however, we can do more. For example, certain special cases can be derived from it by instantiating only some of the parameters; the resulting artifacts are slightly more concrete and have simpler proof obligations. If abstraction is supported on all levels in a uniform way, as envisioned, then the degree of abstraction and the potential for reuse is truly up to the developer.

4 The Framework

The development proceeds in two stages. First, we give a syntactic modification of Hoare Logic which repairs its non-standard behavior with respect to substitution. Second, we embed the expressions and rules of the modified logic in a typed λ-calculus, affording a means for formal abstraction over programs and proofs.

4.1 Quantified Hoare Logic

4.1.1 Standard Hoare Logic

The usual natural-deduction formulation of Hoare Logic, defined relative to an underlying first-order language L and first-order L-theory of expressions T, is given in table 1. There is only one judgment form, asserting the validity of a Hoare triple. The other judgment form which appears in the rule of consequence (Con) is that of T, so we consider Con to be schematic not only in the meta-variables P', P, Q, Q', c but also in the derivations of the L-formulas $P' \Rightarrow P$ and $Q \Rightarrow Q'$. Each rule in table 1 should be read as: "If we instantiate the meta-variables, and we can find proof trees to match each judgment above the line, then we get a new proof tree by attaching them as children to the judgment below the line." As presented, there are no *closed* Hoare triples, because every program must contain at least one assignment (with a free identifier).

(Assign)	$\dfrac{}{\{P[t/x]\}\, x := t\, \{P\}}$
(Seq)	$\dfrac{\{P\}\, c\, \{Q\} \quad \{Q\}\, d\, \{R\}}{\{P\}\, c\ ;\ d\, \{R\}}$
(If)	$\dfrac{\{P \wedge b\}\, c\, \{Q\} \quad \{P \wedge \neg b\}\, d\, \{Q\}}{\{P\}\, \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ d\, \{Q\}}$
(While)	$\dfrac{\{P \wedge b\}\, c\, \{P\}}{\{P\}\, \texttt{while}\ b\ \texttt{do}\ c\, \{P \wedge \neg b\}}$
(New)*	$\dfrac{\{P \wedge y = e\}\, c[y/x]\, \{Q\}}{\{P\}\, \texttt{new}\ x := e\ ;\ c\, \{Q\}}$
(Con)	$\dfrac{P' \Rightarrow P \quad \{P\}\, c\, \{Q\} \quad Q \Rightarrow Q'}{\{P'\}\, c\, \{Q'\}}$

*y not free in P, e, c, or Q

Table 1: Hoare Logic.

Notice that free identifiers in first-order formulas are implicitly universally quantified; a formula's validity is immune to substitution because universal elimination is a sound rule in first-order logic. Free identifiers in Hoare triples do not enjoy the same genericity. For example, the triple

$$\phi \stackrel{\text{dfn}}{=} \{\mathbf{true}\}\, x := y + 1\, \{y < x\}$$

is valid, but

$$\phi[z/x, z/y] = \{\mathbf{true}\}\, z := z + 1\, \{z < z\}$$

is not. *Validity is not preserved by substitution in Hoare Logic.*

In the simple programming language of Hoare Logic, substitution never arises, and all distinct identifiers implicitly denote *non-interfering* program variables; that is, assigning to one will not affect the value of any other. In a language with procedures, such as Algol, the assignment axiom breaks down because substitutions occur with each procedure call. The typed λ-calculus into which we wish to embed Hoare Logic also permits substitution by nature, but the inhabitability of any type is always preserved by substitution; thus we cannot use the form of Hoare Logic given in Table 1. Instead, we give an essentially equivalent formulation of Hoare Logic, the set of whose valid formulas *is* closed under substitution.

⟨command⟩	::=	$V := E$
	|	(⟨command⟩ ; ⟨command⟩)
	|	(**if** Q **then** ⟨command⟩ **else** ⟨command⟩)
	|	(**while** Q **do** ⟨command⟩)
	|	(**new** $V := E$; ⟨command⟩)
⟨spec⟩	::=	$\{F\}$ ⟨command⟩ $\{F\}$
	|	$\forall V$ ⟨spec⟩

Table 2: Quantified Hoare Logic: Syntax.

$$(\text{Assign}_{n,m})\quad \frac{}{\forall\vec{x}\,\{P[t/x_m]\}\,x_m := t\,\{P\}}$$

$$(\text{Seq}_n)\quad \frac{\forall\vec{x}\,\{P\}\,c\,\{Q\} \quad \forall\vec{x}\,\{Q\}\,d\,\{R\}}{\forall\vec{x}\,\{P\}\,c\ ;\ d\,\{R\}}$$

$$(\text{If}_n)\quad \frac{\forall\vec{x}\,\{P\wedge b\}\,c\,\{Q\} \quad \forall\vec{x}\,\{P\wedge\neg b\}\,d\,\{Q\}}{\forall\vec{x}\,\{P\}\,\textbf{if}\ b\ \textbf{then}\ c\ \textbf{else}\ d\,\{Q\}}$$

$$(\text{While}_n)\quad \frac{\forall\vec{x}\,\{P\wedge b\}\,c\,\{P\}}{\forall\vec{x}\,\{P\}\,\textbf{while}\ b\ \textbf{do}\ c\,\{P\wedge\neg b\}}$$

$$(\text{New}_{n,m})^*\quad \frac{\forall\vec{x}\,\{P\wedge x_m = e\}\,c\,\{Q\}}{\forall\vec{x}^{-m}\,\{P\}\,\textbf{new}\ x_m := e\ ;\ c\,\{Q\}}$$

$$(\text{Con}_n)\quad \frac{P' \Rightarrow P \quad \forall\vec{x}\,\{P\}\,c\,\{Q\} \quad Q \Rightarrow Q'}{\forall\vec{x}\,\{P'\}\,c\,\{Q'\}}$$

$m, n \in \omega, 0 \leq n, 1 \leq m \leq n, \vec{x} \equiv x_1 \ldots x_n$ in all rules
$\forall\vec{x}^{-m} \equiv \forall x_1 \ldots \forall x_{m-1} \forall x_{m+1} \ldots \forall x_n$
*Provided x_m not free in P, e, or Q

Table 3: Calculus QHL_0 for Quantified Hoare Logic.

4.1.2 Syntax

Let L be a first-order language, V the set of variables, E the set of L-terms, F the set of L-formulas, and Q the set of quantifier-free L-formulas. The set of formulas ⟨spec⟩ of Quantified Hoare Logic (over L) is given by the grammar in table 2. We call the formulas of Quantified Hoare Logic *specs*. Substitution is defined for Quantified Hoare Logic just as for Hoare Logic, except that $\forall$ is a binding operator.

4.1.3 Proof Calculus QHL_0

A proof calculus for Quantified Hoare Logic, called QHL_0, is given by the axioms and inference rules of Table 3. We say $\vdash_0 \phi$ just in case ϕ is derivable via the rules.

In this calculus there are no "active open" derivable specs, i.e., the active identifiers are all bound by $\forall$ or **new**. Observe that the quantifier $\forall$ does not act as a medium for instantiation, as does the universal quantifier of first-order logic. There is no analogous $\forall$-elimination rule in QHL_0. Rather, through the assignment axiom, we force identifiers appearing on the left-hand sides of assignments in the command portion of derivable specs to be bound by $\forall$. The prefix acts as a context keeping track of actively-occurring identifiers (and protecting them from substitutions). It is possible to show that the provable specs of QHL_0 are just the provable triples of standard Hoare Logic with (at least) all their active identifiers bound by $\forall$ or **new**.

$$(\text{Exch})\quad \frac{\forall\vec{w}\forall x\forall y\forall\vec{z}\phi}{\forall\vec{w}\forall y\forall x\forall\vec{z}\phi}$$

$$(\forall\text{Intro}_n)^\dagger\quad \frac{\begin{matrix}[\psi_1,\ldots,\psi_n]\\ \phi\end{matrix} \quad \forall x\psi_1 \quad \ldots \quad \forall x\psi_n}{\forall x\phi}$$

†Provided x is free only in discharged hypotheses.

Table 4: Structural Rules of Calculus QHL_ω for Quantified Hoare Logic.

4.1.4 Proof Calculus QHL_ω

Each rule of QHL_0 either preserves exactly or decreases by one binding the prefix(es) of its antecedent(s). This feature makes it difficult to combine proofs of different size prefixes. We therefore add the so-called *structural* rules of table 4 to QHL_0, calling the resulting calculus QHL_ω. It is possible to show that the two calculi QHL_0 and QHL_ω are equivalent with respect to theorems, and that the valid formulas of both calculi remain valid under substitution.

4.2 Embedding Into λ-Calculus: The System *PR*

To turn QHL_ω into a viable programming logic, we can formally represent it in a meta-logic like the Edinburgh Logical Framework (LF) [1, 6]. Readers familiar with the LF type theory are referred to [2] for a rigorous and detailed description of the following system.

In *PR* there are two kinds of entities, *terms* and *types*. Types include syntactic categories, like the set of commands or the set of assertions, and judgments, like the set of first-order natural deduction proofs of a certain assertion. Terms include syntactic objects, like commands and assertions, and representations of proof trees. We allow variables of any valid type to occur in expressions, in accordance with our requirement for abstraction from section 3, but in order to do so soundly we introduce *contexts* which assign types

to free variables in term and type expressions. We allow the formation of λ-abstractions from any term, which let us formally construct abstract programs and proofs, and applications of terms of the appropriate types, which permit us to formally specialize these abstract programs and proofs.

We use the notation $\Gamma \vdash A$ to mean that A is a type in context Γ and $\Gamma \vdash M : A$ to mean that M is a term of type A in context Γ. If the context is irrelevant we will simply write $\vdash A$ and $M : A$.

The following three subsections constitute a simultaneous inductive definition of contexts, types, and terms.

4.2.1 Contexts

- $\emptyset$ is a context;
- if Γ is a context, x is a variable not appearing in Γ, and $\Gamma \vdash A$, then $\Gamma, x : A$ is a context.

4.2.2 Types

Let Γ be any context.

- $\vdash \mathbf{exp}$, $\vdash \mathbf{assert}$, $\vdash \mathbf{var}$, $\vdash \mathbf{comm}$, $\vdash \mathbf{spec}$,
- if $\Gamma \vdash a : \mathbf{assert}$, then $\Gamma \vdash \mathbf{T}(a)$;
- if $\Gamma \vdash s : \mathbf{spec}$, then $\Gamma \vdash \mathbf{H}(s)$;
- if $\Gamma \vdash A$ and $\Gamma, x : A \vdash B$, then $\Gamma \vdash \prod_{x:A} B$. If x does not occur in B, we abbreviate $\prod_{x:A} B$ to $A \rightarrow B$.

4.2.3 Terms

Let Γ be any context.

- if $x : A \in \Gamma$ then $\Gamma \vdash x : A$;
- if $\Gamma \vdash v : \mathbf{var}$, then $\Gamma \vdash v\uparrow : \mathbf{exp}$;
- for each first-order term t of T, $t : \mathbf{exp}$;
- for each first-order formula a in T, $a : \mathbf{assert}$;
- for each command c of the simple programming language of Hoare Logic, $c : \mathbf{comm}$;
- for each spec s of Quantified Hoare Logic, $s : \mathbf{spec}$;
- axiom and inference rule schemas of first-order natural deduction are constants, like

$$\wedge\text{-Intro} : \prod_{a:\mathbf{assert}} \prod_{b:\mathbf{assert}} \mathbf{T}(a) \rightarrow \mathbf{T}(b) \rightarrow \mathbf{T}(a \wedge b);$$

- axiom and inference rule schemas of QHL_ω are constants, like

$$\mathrm{AsgnAx}_{1,1} : \prod_{P:\mathbf{exp}\rightarrow\mathbf{assert}} \prod_{t:\mathbf{var}\rightarrow\mathbf{exp}} \mathbf{H}(\forall x\,\{P(t\,x)\}\; x := t\,x\,\{P\,(x\uparrow)\});$$

- if $\Gamma, x : A \vdash M : B$, then $\Gamma \vdash \lambda x : A\,.\,M : \prod_{x:A} B$;
- if $\Gamma \vdash M : \prod_{x:A} B$ and $\Gamma \vdash N : A$, then $\Gamma \vdash M(N) : B[N/x]$.

It can be demonstrated that the terms of type $\mathbf{T}(a)$ correspond bijectively to the proofs in T of a, and similarly that the terms of type $\mathbf{H}(s)$ correspond bijectively to the proofs in QHL_ω of s. In particular, if a happens *not* to be a theorem of T then it will be impossible in our system to construct a term of type $\mathbf{T}(a)$ (in the empty context).

4.2.4 Reduction

A term which looks like $(\lambda x : A\,.\,M)(N)$ is called a *β-redex* and reduces to $M[N/x]$, its *contractum*. If any subterm K of a term (resp., type) M is a β-redex, then M reduces to a new term (resp., type) M' which arises from M by replacing K with its contractum.

A term which looks like $\lambda x : A\,.\,(M(x))$, where x does not occur free in M, is called an *η-redex* and reduces to M. As above, subterms can be reduced within larger terms or types.

A normal term is one containing no redexes. It can be shown that for any term, the process of repeatedly reducing every redex within it in any order eventually terminates in a normal term. Two terms are called $\beta\eta$-equivalent if they both reduce to the same normal term, modulo the names of bound variables. For purposes of assigning types to terms, we consider terms and types only up to $\beta\eta$-equivalence.

5 Example

Returning to the toy example of section 3.1, we wish to demonstrate proof reuse in system *PR*. Suppose $\Gamma \stackrel{\text{dfn}}{=} f : \mathbf{exp} \rightarrow \mathbf{var} \rightarrow \mathbf{comm}$. The following simple program implements *choose* in terms of procedure f which is assumed to compute the factorial of its first argument and assign the result to its second argument:

$$\begin{aligned} \textit{choose} \stackrel{\text{dfn}}{=} \lambda m : \mathbf{exp}\,.\,\lambda k : \mathbf{exp}\,.\,\lambda c : \mathbf{var}\,. \\ \mathbf{new}\, a := 0\ ;\ f\,(k)\,(c)\ ; \\ f\,(m-k)\,(a)\ ;\ c := c\uparrow \times a\uparrow\ ; \\ f\,(m)\,(a)\ ;\ c := a\uparrow/c\uparrow. \end{aligned}$$

$$\Gamma \vdash \textit{choose}\ :\ \mathbf{exp} \rightarrow \mathbf{exp} \rightarrow \mathbf{var} \rightarrow \mathbf{comm}$$

To prove this abstract program correct, we must create a proof term for it which has parameters f for the factorial and ϕ for a proof that f is correct. So suppose we have a larger context

$$\Gamma' \stackrel{\text{dfn}}{=} \begin{bmatrix} f : \mathbf{exp} \to \mathbf{var} \to \mathbf{comm} \\ \phi : \prod_{R:\mathbf{assert}} \prod_{i:\mathbf{exp}} \\ \qquad \mathbf{H}(\forall b\,\{R\}\, f\,(i)\,(b)\,\{R \wedge b\uparrow = i!\}) \end{bmatrix}.$$

We note that two active identifiers, c and a, appear in the inner command of choose, and that f is used once with c and twice with a as the second (active) parameter. The supposed proof ϕ mentions only one active identifier, b. We must use structural rules to derive from ϕ proofs for the two different uses of f which occur:

$$\begin{array}{lll} \phi_{2,1} & : & \prod_{R:\mathbf{assert}} \prod_{i:\mathbf{exp}} \\ & & \quad \mathbf{H}(\forall c \forall a\,\{R\}\, f\,(i)\,(c)\,\{R \wedge c\uparrow = i!\}) \\ \phi_{2,2} & : & \prod_{R:\mathbf{assert}} \prod_{i:\mathbf{exp}} \\ & & \quad \mathbf{H}(\forall c \forall a\,\{R\}\, f\,(i)\,(a)\,\{R \wedge a\uparrow = i!\}). \end{array}$$

From there it is relatively straightforward to construct a term Choose with typing

$$\begin{array}{l} \Gamma' \vdash \text{Choose} : \\ \quad \prod_{m,k:\mathbf{exp}} \\ \qquad \mathbf{H}(\forall c\,\{k \leq m\}\, \mathit{choose}\; m\; k\; c\,\{c\uparrow = \binom{m}{k}\}). \end{array}$$

This example illustrates two things: the gain in expressiveness over standard Hoare Logic via the type theory of the underlying λ-calculus, and the manner in which proof reuse in our system is moderated by the need to prevent interference of active identifiers. Had we been able to bind all the identifiers used in the types of ϕ and Choose using $\prod$, we could have used λ-abstraction and application to produce $\phi_{2,1}$ and $\phi_{2,2}$. But we could also have produced a proof of an invalid spec, because we could cause b and i to be substituted with the same variable. In the type of ϕ, the $\forall$-binding of b insures that b and i cannot interfere, and the structural rules let us maintain just enough flexibility in our locally-bound environment to prevent the incorrect proofs of a more naive encoding.

It is even simpler to produce an actual implementation of the factorial and to prove its correctness with terms

$$\begin{array}{lll} \mathit{fac} & \stackrel{\text{dfn}}{=} & \lambda i : \mathbf{exp}\,.\,\lambda b : \mathbf{var}\,. \\ & & \quad \mathtt{new}\; n := i\;;\; b := 1\;; \\ & & \qquad \mathtt{while}\,(\neg n\uparrow = 0) \\ & & \qquad\quad (b := b\uparrow \times n\uparrow\;;\; n := n\uparrow - 1) \\ \mathit{fac} & : & \mathbf{exp} \to \mathbf{var} \to \mathbf{comm} \\ \text{Fac} & : & \prod_{R:\mathbf{assert}} \prod_{i:\mathbf{exp}} \\ & & \quad \mathbf{H}(\forall b\,\{R\}\, \mathit{fac}\,(i)\,(b)\,\{R \wedge b\uparrow = i!\}). \end{array}$$

Then the term

$$\begin{array}{l} \text{Choose}[\mathit{fac}/f, \text{Fac}/\phi] : \\ \quad \prod_{m,k:\mathbf{exp}} \\ \qquad \mathbf{H}(\forall c\,\{k \leq m\}\, \mathit{choose}[\mathit{fac}/f]\; m\; k\; c\,\{c\uparrow = \binom{m}{k}\}) \end{array}$$

is a proof that instantiating the abstract *choose* procedure with *fac* yields a correct concrete *choose* procedure.

6 Conclusion

We have presented a system *PR* for abstract reasoning about abstract programs, allowing proofs to be reused in the same way as the programs they prove correct. We have achieved the system via a straightforward embedding of a modification of Hoare Logic into a typed λ-calculus, thereby treating abstraction at all levels uniformly. The important issue of controlling interference of active identifiers is addressed via the syntax and structural rules of the calculus. Typing, and thus proof checking, is decidable. A prototype proof checker has been built and employed by simply entering a signature for a finitary variant of QHL_ω into an existing implementation of the Edinburgh Logical Framework.

By virtue of the simple definition of *PR*, we are continuing the work begun here in two orthogonal directions. Currently, the major weaknesses of the system are the absences of complex data structures, abstraction over types (polymorphism), and recursive functions. It is easy to allow the theory T of expressions to be multi-sorted, which slightly improves the datatype capabilities. We are currently investigating a more extensive base system than QHL_ω which internalizes the set of phrase types of Idealized Algol [11] and the notion of reduction in order to obtain a system for the full language via the embedding.

However, we are also exploring various extensions to the type theory of the λ-calculus itself, which is extremely minimal in the current paper. The embedding given here exploits primitive type contructors of LF to represent the procedural types of Algol. Embedding QHL_ω into a more expressive type theory, provided it remains decidable, will accordingly improve the power of the resulting *PR*. In particular, we are examining the incorporation of certain induction principles which would at least allow reasoning about *primitive*-recursive functions at all types, and of certain standard type constructors, such as cartesian products and disjoint sum (union) types. Thus the basic phrase type constructors of Algol would be available in the embedding via the same signature of section 4.2.

Our goal, by either route, is to fully incorporate all the kinds of abstract algorithms and proofs sampled in section 3.

7 Acknowledgements

We wish to thank Uday Reddy for many helpful discussions.

References

[1] A. Avron, F. Honsell, I. A. Mason, and R. Pollack, *Using typed λ-calculus to implement formal systems on a machine*, Journal of Automated Reasoning, 9 (1992), pp. 309–354.

[2] J. E. Caplan, *Formalizing Hoare Logic*, in Second ACM SIGPLAN Workshop on State in Programming Languages, San Francisco, Jan. 1995, Association for Computing Machinery, pp. 1–18. Available as University of Illinois at Urbana-Champaign tech. rep. UIUCDCS-R-95-1900.

[3] R. L. Constable and S. F. Smith, *Partial objects in constructive type theory*, in Symposium on Logic in Computer Science, Ithaca, New York, 1987, pp. 183–193.

[4] M. J. C. Gordon, R. Milner, and C. P. Wadsworth, *Edinburgh LCF : a mechanised logic of computation*, vol. 78 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1979.

[5] M. T. Harandi and F. H. Young, *Software design using reusable algorithm abstractions*, in Software Engineering 88, University of Liverpool, July 1988, Institution of Electrical Engineers/British Computer Society, The Institution, pp. 93–97.

[6] R. Harper, F. Honsell, and G. Plotkin, *A framework for defining logics*, Journal of the Association for Computing Machinery, 40 (1993), pp. 143–184.

[7] I. A. Mason, *Hoare's Logic in the LF*, Tech. Rep. ECS-LFCS-87-32, Laboratory for the Foundations of Computer Science, Edinburgh, Scotland, 1987.

[8] B. Nordström, K. Petersson, and J. M. Smith, *Programming in Martin-Löf's Type Theory: An Introduction*, vol. 7 of International Series of Monographs on Computer Science, Clarendon Press, Oxford, 1990.

[9] P. W. O'Hearn and R. D. Tennent, *Syntactic control of interference revisited*, May 1994.

[10] J. C. Reynolds, *Syntactic control of interference*, in 5th Annual ACM Symposium on Principles of Programming Languages, Tuscon, Arizona, 1978, Association for Computing Machinery, pp. 39–46.

[11] ——, *The essence of ALGOL*, in Algorithmic Languages, J. W. de Bakker and J. C. van Vliet, eds., North-Holland, 1981, pp. 345–372.

[12] ——, *Idealized ALGOL and its specification logic*, in Tools and Notions for Program Construction: An Advanced Course, D. Néel, ed., Cambridge University Press, 1982, pp. 121–161.

[13] ——, *Syntactic control of interference, part 2*, in International Colloquium on Automata, Languages, and Programming, G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, eds., vol. 372 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1989, pp. 704–722.

[14] S. F. Smith, *Partial Objects in Type Theory*, PhD thesis, Cornell University, Ithaca, New York, Aug. 1988.

[15] ——, *Partial computations in constructive type theory*, tech. rep., The Johns Hopkins University, Baltimore, Maryland, Feb. 1991.

[16] V. Swarup, *Type Theoretic Properties of Assignments*, PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1992.

[17] S. Thompson, *Type Theory and Functional Programming*, International Computer Science Series, Addison-Wesley, Wokingham, England, 1991.