Property Types: Semantic Programming for Java

Aaron Stump Ian Wehrman
Washington University in St. Louis
{stump,iwehrman}@cse.wustl.edu
http://cl.cse.wustl.edu/

Abstract

Dependent types have been proposed for functional programming languages as a means of expressing semantically rich properties as types. In this paper, we consider the problem of bringing semantic programming based on dependent types to an object-oriented language. A type-theoretically light extension to Java (with generics) is proposed, called *property types*. These are types indexed not by other types (as in Java with generics) but by immutable expressions. We consider programming with two examples of property types: a property type $\mathsf{HasFactors}\langle \mathsf{long}\ x, \mathsf{Set}\ a\rangle$, expressing that the elements of a are the multiplicative prime factors of x; and a property type $\mathsf{Sorted}\langle \mathsf{List}\ l\rangle$, expressing that l is sorted.

Keywords dependent types, Java, program verification, semantic programming

1. Introduction

Correct programming is one of the oldest problems of Computer Science. It also one of the most practically important: a 2002 report issued by the National Institute of Standards and Technology estimates that software errors cost the United States' economy $59.5 billion annually [20]. A vast body of work in several different fields has been devoted to the technical problems of correct programming. On one end of the spectrum, techniques like *theorem proving* are able to achieve full correctness of software, hardware, and other systems, but at tremendous cost. Formalizing a system and proving it correct in a theorem prover is highly labor-intensive and requires extensive special training. Full program verification based on this approach is applied only for truly critical applications [12, 10, 22]. Weaker deductive techniques like *model checking* are automatic, but generally work only for finite state systems, and are computationally expensive [6]. On the other end of the spectrum, static analyses such as *abstract interpretation* and even simple compile-time type checking provide relatively efficient approaches to code validation, albeit for less general problems [18, 16]. Impressive progress has been made towards more correct software using static bug finding techniques (e.g., [28, 14, 2, 8]). Testing and runtime monitoring, even though they cannot demonstrate the complete absence of bugs, remain mainstays of correct program development in practice (e.g., [1, 19, 21]).

In the functional programming community, there is renewed interest in alternative language-based approaches to developing correct programs, developed originally in systems like Martin-Löf type theory [17]. Proofs are reified as data in the programming language, just like ints or strings. The type of a proof is the theorem it proves. To express that a function has precondition ϕ, one requires the function to take a proof of ϕ as an additional argument. Postconditions ψ are expressed similarly, by having the function return a proof of ψ as an additional output. The programmer constructs such proofs by hand; in principle, techniques from automated deduction could be used to help. The benefits of this approach over traditional approaches based on proving extracted verification conditions in a proof assistant include:

- proofs and programs are intertwined in a single artifact;
- constructing a proof has a computational, rather than mathematical, feel, since proofs are data; and
- proofs are available at run-time, which is potentially valuable for applications like proof-carrying code [15]. Note that some systems may allow (computationally irrelevant) proofs to be removed during compilation, so as to avoid incurring runtime costs.

The hope is for these benefits to lead to verification with the expressive power of traditional theorem-proving approaches, but with a much lower burden on the developer. The first author proposes "semantic programming" as a catch phrase for these ideas. They are the main focus of languages like ATS, RSP1, Epigram, and Ωmega [5, 27, 13, 24]. They are also supported in some proof assistants (e.g., Coq) [25]. Ideas related to language-based approaches to program verification will be explored further in an upcoming FLoC workshop, PLPV 2006 ("Programming Languages meets Program Verification"), organized by the first author and Hongwei Xi.

In this paper, we seek to explore possible answers to the following question: what is a minimal, type-theoretically light extension to a mainstream object-oriented programming language like Java that captures the spirit of semantic programming? This paper proposes *property types* as such an extension. Property types are a restricted form of (one version of) dependent types. They are declared like generic classes, but instead of being parametrized by type variables, they are parametrized by expression variables. Property types are type-theoretically light because we severely restrict the form of expressions which can index property types (thus avoiding difficult meta-theoretic issues like strong normalization of compile-time evaluation, which cause great complexity for the meta-theory of other type systems).

Despite the restrictions, property types still enable useful program verification. We demonstrate this with two examples. The first uses a property type $\mathsf{HasFactors}\langle \mathsf{long}\ x, \mathsf{Set}\ a\rangle$ to express that the elements of a are the prime multiplicative factors of x (Section 4). We write code to perform exponentiation by successive squaring

for which we statically verify, using property types, that the set of factors of x^y is equal to the set of factors of x. The second example uses a property type Sorted⟨List l⟩, expressing that l is sorted (Section 5). We implement a routine for merging two non-null sorted lists for which we statically verify that the resulting list is also sorted. We begin by showing how to adapt dependent types from functional programming (Section 2) to obtain property types (Section 3). Note that the work is in progress, due to the lack of an implementation and formal meta-theory.

2. Dependent Types

Suppose we are writing in a functional programming language, and have code which requires or guarantees that one natural number be less than or equal to another. Dependently typed programming languages (e.g., RSP1 [27]) can express the "less than or equal to" relation on natural numbers using a datatype indexed by expressions (more specifically, by terms). Suppose we have a datatype of natural numbers:

```
nat : type.
z : nat.
succ : nat => nat.
```

We represent the mathematical statement $x \leq y$ as a type, leq x y. We may declare the following datatype for this term-indexed type:

```
leq : nat => nat => type.
leqax : x:nat => leq x x.
leqs : x:nat => y:nat => leq x y => leq x (s y).
```

The first declaration states that leq is indexed by the two natural numbers in question. The declaration of leqax can be read as an axiom stating that for any natural number x, x is less than or equal to itself. Note the use of the dependent function space, in general of the form x:A => B(x), to give a name (x) to the argument to leqax. This name is then used to index the return type of leqax. Finally, the declaration of leqs is an axiom stating that if x and y are natural numbers, and if x is less than or equal to y, then x is less than or equal to the successor of y. This defines the datatype of proofs that one natural number is less than or equal to another. Note that the datatype is not connected to built-in numeric types or built-in arithmetic operations of the programming language. Instead, these proofs are about data in the user-declared datatype nat. Functions which have preconditions or postconditions stating that one number is less than or equal to another can now require proofs of the appropriate "less than or equal to" statement (we skip examples of this here; the reader may find several examples, e.g., in [27]).

Type systems like those for ownership types (e.g., [4]) also use indexed types in an object-oriented setting. Such systems are designed to implement special-purpose static analyses, and have had promising results conservatively verifying important (but restricted) properties of code. Dependent types of the kind we wish to bring to Java are, in contrast, used for expressing quite general data properties. The price we pay is that it is difficult if not practically impossible to provide a similar level of automation for checking that the expressed properties hold.

3. Property Types

Adding dependent types as just described to a mainstream object-oriented language requires some non-trivial adaptations. Our basic starting idea is to allow class types to be indexed by terms, possibly from some restricted class. So we may simply extend the Java 1.5 syntax for parametrized classes to allow variable declarations in parameter lists. For example, if we wish to represent the property that int i is less than or equal to int j, we can declare a class like this:

```
public class Leq<int i, int j> {
...
}
```

Subsequently, we might have variables, fields, or function parameters with types like Leq$\langle 3, 4\rangle$. We call parametrized classes like Leq *property classes*, and their instantiations, like Leq$\langle 3, 4\rangle$, *property types*.

The reader may immediately wonder why it is necessary to index a class like Leq at all. Why not just have an unindexed class Leq whose constructor takes in the ints i and j, and asserts that i is less than or equal to j? With this approach, no instance of Leq could be successfully created without the corresponding less-than property holding. Whenever one has a non-null instance of Leq with fields i and j, one knows that i must be less than or equal to j. For purposes of static verification, this approach is not sufficient. In general, pre- and post-conditions of functions express relations between inputs and between inputs and outputs. For verification in the style we are exploring, it is not sufficient for a function to require an argument of type Leq to record that, say, the first argument is less than the second. This is because the type Leq does not record *which* two numbers are in the less-than relation. Adding indexes to the type makes that information available to the type checker for static verification.

Once we decide to index types like Leq by terms, several issues are immediately raised. First, how do we represent proof rules? Since in dependently typed functional languages proof rules are written as functions, we naturally think to write them as methods in an object-oriented language. But this presses the issue of what expressions are allowed to index types. For example, in the return type of leqs in the previous Section, we have s y as an index. Would this be modeled as a constructor or other method call in Java? It can be problematic to index types by arbitrary code, so this sounds worrisome.

Even more distressing are the problems caused by mutable state. Suppose we have local variables x and y, and we obtain an object of type Leq⟨x,y⟩. Now suppose we assign to x (or y). The property might only hold for the value of x before the assignment, and hence might be violated by a later assignment. We could imagine having our type checker try to track (conservatively) the values of variables and invalidate proof terms if variables might change. But this places a rather heavy requirement on the type system, of the kind we are trying to avoid. We instead adopt the seemingly severe restriction that class types can be indexed only by expressions whose value cannot change in different memory contexts at run-time. To express this restriction, we can take advantage of the `final` keyword in Java, and require that classes only be indexed by literal values or final expressions. It turns out to be practically necessary to allow indexing by more than just final local variables, final fields, and final method parameters. We must take final expressions to be these, or final fields of final expressions. So if x is a final variable, field, or method parameter of type A, where A has a final field f, we allow types to be indexed by $x.f$.

We are still left with the possibility that while a reference indexing a type cannot be changed, the object pointed to by the reference might be modified. We propose that for now, property types be used just to express properties of immutable objects. The type checker certainly could help by allowing classes to be declared `immutable`. Note that this is quite different, and seemingly simpler than, adding support to allow references to be declared `readonly` [26, 3]. Immutability of a class simply means that all fields must be final references to immutable objects. But we emphasize that we do not require the type checker to support this. It is not entirely unreasonable

to leave immutability up to the programmer to enforce. Naturally, this would only be appropriate when guarantees conditional on unverified properties (like immutability in this case) are still valuable.

Let us return to the issue of proof rules. We propose that inferences not be modeled in a one-to-one way by objects (so, a complex proof is not modeled as a complex object of some kind). Instead, we propose that trusted code, which we call an *authority*, simply give out instances of the property type itself to assert that the property holds. We call such instances proofs, even though they do not directly correspond to mathematical proofs. For example, the authority for Leq is a class called LeqAuthority, with static methods like this one corresponding to the function leqax we had above:

```
public static Leq<i,i> leqAx(final int i) {
  return new Leq<i,i>();
}
```

This approach requires organizing the code so that only the trusted parts can instantiate property types. Fortunately, this can easily be done in Java either by making the property class a static inner class of the authority class and making the property class's constructor private; or by putting the authority class and the property class in the same trusted package, and making the property class's constructor have package-level access protection.

Note that runtime checks can always be performed by an authority. For example, LeqAuthority could supply a method assertLeq which takes in ints x and y and if $x \leq y$, returns a proof of Leq⟨x,y⟩; and otherwise assert-fails. For the program verification examples below, we will write untrusted code in such a way that it never needs to call such possibly failing methods.

Another point to consider is whether or not to allow `null` to have a property type. Since allowing `null` to be supplied where a proof is required fundamentally undermines the requirement to supply a proof, we wish to disallow `null` at property types. This requires an extension to the type checker, but one which has already been studied by others [7]. The only change here is that it is preferable to allow a class to be declared as not tolerating null instances, instead of just allowing a reference be declared to be non-null. This could be implemented using the techniques in the cited work by viewing a declaration of a class as non-nullable as implicitly marking every variable of that class type as non-null.

A related issue is whether or not to allow property types to be indexed by expressions which could evaluate to null. We adopt the position that this is acceptable, since non-nullness could be enforced by a separate mechanism. Also, developers have more flexibility with this approach to declare that properties hold of expressions, without worrying about whether or not related expressions are null. Insisting on non-nullness here would impose additional burdens on programmers, which we choose to let them take on at their own discretion.

Now we can return to the issue of how to model the constructor term s y in the return type of the functional leqs constructor. The proposal is that whenever a return type's index is the result of computation, that index and the proof about it must be passed around explicitly in an bundling object. For the case of leqs, we can declare a class Helper to hold a value y and a proof that x is less than or equal to y:

```
public class Helper<int x> {
  public final int y;
  public final Leq<x,y> p;
  public Helper(final int y, Leq<x,y> p) {
    this.y = y;
    this.p = p;
  }
}
```

Expression-indexed helper classes like this one may be distinguished informally from property classes by noting that they need not be trusted (since they do not need to instantiate property types), and they need not be required to be non-nullable. But note here that we do need a modest bit of help from the type checker. To type check the second assignment in Helper's constructor, we need p to have type Leq⟨x, *this*.y⟩. But it has inequivalent type Leq⟨x,y⟩. In this one situation of assignment of final expressions to final fields in constructors, we need the type checker to take into account the assignment $\mathit{this}.y = y$. We do not need the type checker to take such assignments into account elsewhere in code, which is good, since that would require a static analysis (which we are trying to avoid wherever possible).

Now we can write the leqs proof rule as a method of the Leq authority class LeqAuthority:

```
public static Helper<x> LeqS(final int x,
                             final int y,
                             Leq<x,y> p) {
  return new Helper<x>(y+1,p);
}
```

Requirements on Type Checking. In summary, to implement property types as informally described above, a Java type checker would need to be extended in the following ways:

- Allow class declarations to be parametrized not just by type variables but also by expression variables (and type check the bodies of those declarations appropriately).
- Restrict the kind of expressions allowed to index types to just literal values and final expressions.
- Take into account assignments of final expressions to final fields in a constructor when type checking the rest of the constructor.
- Allow classes to be declared as non-nullable (`null` cannot inhabit them).
- When type checking method calls, the dependencies between inputs and outputs must be checked.

Support for immutable classes (not to be confused with support for readonly references, which appears to be more complex [3, 26]) would be useful, but is not crucial for reasonable usage of property types.

4. Example: Exponentiation by Successive Squaring

Our first example is exponentiation by successive squaring. The property statically verified is that the set of prime factors of the base x is equal to the set of prime factors of the result, x^y, for a positive integral exponent y. Trusted code provides basic functionality for factoring and multiplication and untrusted client code provides a proof-producing exponentiation method. Declarations for the property classes and method signatures are shown in Fig. 1 and Fig. 2.

There are two trusted classes in this example. The first is the SetAuthority class, which manages properties about equality between sets. The property type Eq⟨Set a, Set b⟩ expresses equality between sets a and b. The property IsUnion⟨Set a, Set b, Set c⟩ states that the set c is union of sets a and b. These properties are internal to the SetAuthority class, which has exclusive ability to construct them and is trusted to do so in a safe way.

For example, the method symm(Set a, Set b, Eq⟨a, b⟩ p) returns a proof object of type Eq⟨b, a⟩. Similarly, the methods refl(Set a) and trans(Set a, Set b, Set c, Eq⟨a, b⟩ p_1, Eq⟨b, c⟩ p_2) produce objects of type Eq⟨a, a⟩ resp. Eq⟨a, c⟩.

The method union(Set a, Set b) returns a helper object[1] consisting of a new set c (the union of a and b) and an object of type IsUnion$\langle a, b, c\rangle$. The SetAuthority class also provides a method unionEqSets1( ) which returns a proof that the union of sets a and b is equal to the set a, provided $a = b$. Another required method (not shown) returns a proof that the union of any set a with the empty set $\emptyset$ is equal to a.

The second trusted class is FactorAuthority, shown in Fig. 3, which manages the property HasFactors$\langle$long x, Set $a\rangle$, denoting that integer x has as prime factors the set of integers a. The method factor(long x) returns a helper object whose only members are a set a and an object of type HasFactors$\langle x, a\rangle$. The method multiply( ) returns a helper object containing an integer y (the result of multiplying some x_1 and x_2), a set b, a proof of IsUnion$\langle a_1, a_2, b\rangle$ (with a_1, a_2 the factors of x_1 resp. x_2) and a proof of HasFactors$\langle y, b\rangle$. While the factor( ) method is in general quite slow (viz. pseudo-polynomial time), the time required for the multiply( ) method to update the HasFactors$\langle\rangle$ proof is fast — it is dominated by the linear-time union operation on the two incoming sets.

In this example, an untrusted ExpClient class provides an exponentiation method that returns correctness proofs along with the result of the main computation. Specifically, the method exp(long x, int y), shown in Figure 5, returns a helper object with the result of the exponentiation z, a set a of prime factors of the output z, a set b of prime factors of the base x, and finally a proof that the sets are equal (an object of type Eq$\langle a, b\rangle$.

The correctness of exponentiation by successive squaring is more difficult to show than that of a naïve implementation that requires time linear in the size of the exponent. This is because as computation proceeds the value of the base x changes — if $x = 2m$, then we continue in the next step with $x := m$ and $y := y/2$. Consequently, a recursive helper method expRec( ), shown in Figure 4, is defined that requires extra proofs as preconditions (i.e., with a strengthened inductive hypothesis); specifically a proof that the factors of the original base x are equal to the those of the current base x', and a proof that the factors of the previous computation are equal to those of the current base x'.

There are two cases[2] to expRec( ): either the exponent is exactly 1 or the exponent is at least 2. In the first case, we simply return the result of the previous computation and a proof that its factors are equal to those of the original base (by transitivity of the precondition equality proofs, the factors of the original base are equal to the factors of the new base, and the factors of the new base are equal to the factors of the previous computation).

In the second case, we start by computing the current step of exponentiation by multiplying the previous result with the current base. We then handle two subcases based on whether or not the exponent is odd or even. In the odd subcase, we decrement the exponent and recurse with the current exponentiation value just computed. In the even subcase, we want to recurse with a new base x'' that is the square of the previous base x'. Before we do that, though, we must supply a proof to the recursive call that the factors of the new base are equal to those of the original base and to the factors of the current exponentiation. To do so, we use transitivity of the incoming proofs (the inductive hypothesis) as well as the judgment provided by SetManager that the union of two equal sets is itself equal to each of the individual sets. Note that when expRec( ) is called in the top-level definition of exp(long x, int y) (Fig. 5), the arguments denoting the current base x' and the current computation are just the same as the original base x.

[1] Definitions of Helper classes are not given here, but are entirely straightforward.

[2] Note that we do not handle the case when the exponent is 0, for then it would not be true that the prime factors of the exponentiation are equal to those of the base.

```
public static final class Eq<Set,Set> {}

public static Eq<a,a>
    refl(final Set a) { /* ... */ }

public static Eq<b,a>
    symm(final Set a,
         final Set b,
         Eq<a,b> proof) { /* ... */ }

public static Eq<a,c>
    trans(final Set a,
          final Set b,
          final Set c,
          Eq<a,b> abProof,
          Eq<b,c> bcProof) { /* ... */ }
```

Figure 1. Exponentiation: SetAuthority methods for equality.

```
public static final class IsUnion<Set,Set,Set> {}

public static UnionHelper<a,b>
    union(final Set a, final Set b) { /* ... */ }

public static Eq<a,c>
    unionEqSets1(final Set a, final Set b,
                 final Set c,
                 IsUnion unionProof<a,b,c>,
                 Eq eqProof<a,b>) { /* ... */ }
```

Figure 2. Exponentiation: SetAuthority methods for union.

5. Example: Merging Sort Lists

For a second example, we consider code which merges sorted immutable lists of ints (for simplicity, implemented as a class List with public final fields data and next) to obtain a sorted result list. We do not seek to verify that the result list stores all and only the elements of the input lists, but merely that it is indeed sorted. Thus we see that property types, like some other approaches to program verification, can be used to verify properties on a continuum between very weak properties like type correctness (for example) and very strong ones like the full specification of the code. We also restrict our attention to non-null lists. We express this by taking in proofs of property type NonNull$\langle$x$\rangle$, which has little force unless the type checker enforces non-nullability of such proofs for us (as described in Section 3 above). The method signature we want (in an untrusted class holding the merge routine) is the following:

```
public static MergeInv<l1,l2>
MergeNonNull(final List l1,
             final List l2,
             Sorted<l1> p1,
             Sorted<l2> p2,
             NonNull<l1> p3,
             NonNull<l2> p4);
```

Here, MergeInv is the following helper class, expressing the crucial invariant of MergeNonNull for our purposes, that the first element of the resulting list is the minimum of the first elements of the input lists:

```
public static final class HasFactors<long,Set> {}

public static FactorHelper
    factor(final long n) { /* ... */ }

public static FactorHelper
    multiply(final long n1, final Set f1,
             final HasFactors<n1,f1> p1,
             final long n2, final Set f2,
             final HasFactors<n2,f2> p2) {
    if (1 == n1) {
        return new FactorHelper(n2, f2,
            union(NIL, f2), p2);
    } else if (1 == n2) {
        return new FactorHelper(n1, f1,
            union(f1, NIL), p1);
    } else {
        final long n = n1 * n2;
        final UnionHelper<f1,f2>
            ub = union(f1, f2);
        return new FactorHelper(n, ub.c, ub,
            new HasFactors(n, ub.c)); } }
```

Figure 3. Exponentiation: FactorAuthority methods.

```
public static
class MergeInv<List l1, List l2> {
  public final List l;
  public final Sorted<l> p1;
  public final Min<l.data, l1.data, l2.data> p2;

  public MergeInv(...) { ... }
}
```

Here, Min⟨x,y,z⟩ is an additional property type (supported by its own authority class) expressing that int x is the minimum of ints y and z.

One problem that immediately comes up in writing our merge method is that we take different actions depending on whether *l1.data* is less than or equal to *l2.data*, or vice versa. In either case, we will certainly need a proof of the corresponding Leq type. We could compare those ints and then call a method like LeqAuthority.assertLeq, mentioned above. But this requires a run-time check. How can we get the proofs we need without run-time checks? Intuitively, LeqAuthority should provide a method compare, which given ints x and y returns the appropriate Leq proof depending on whether $x \leq y$ or $y < x$. To handle this disjunctive situation we need a digression on implementing something analogous to the *sum types* of functional programming in Java.

Expressing Disjunction in Java. Functional programming uses sum types to express that a piece of data has either one given type or another given type. When expressing specifications as types, some languages rely on the Curry-Howard isomorphism to express logical disjunctions as sum types [11]. So the question becomes, how does the Curry-Howard isomorphism for disjunction work in Java? Certainly, we could simply add sum types to Java, but this would be another modification to the language, of which we are trying to make as few as possible. It turns out that we can meet our verification needs without sum types. It is natural to think instead of having a base class or interface Or$\langle X, Y\rangle$, with a subclass Or1$\langle X, Y\rangle$ for when we have X, and another subclass Or2$\langle X, Y\rangle$ for when we have Y. This is fine, but the question then becomes, how do we implement a `case` construct (logically, or-elimination) in Java? The cute answer is to use the visitor design pattern, as follows (cf. [9]).

```
private static ExpHelper
    expRec(final FactorHelper base0,
           final FactorHelper base1,
           final Eq<base0.factors,
                    base1.factors> eq01,
           final FactorHelper prev,
           final Eq<base1.factors,
                    prev.factors> eq1p,
           int power) {
    if (power == 1) {
        final Eq<base0.factors,
                 prev.factors> eq0p =
            trans(base0.factors, base1.factors,
                  prev.factors, eq01, eq1p);
        return new ExpHelper(base0, prev, eq0p);

    } else { /* power > 1 */
        final FactorHelper current =
            multiply(base1.n, base1.factors,
                     base1.proof,
                     prev.n, prev.factors,
                     prev.proof);
        final Eq<base1.factors,
                 current.factors> eq1c =
            unionEqSets1(base1.factors,
                         prev.factors,
                         current.union.c,
                         current.union.proof,
                         eq1p);
        if ((power % 2) == 1) {
            return expRec(base0, base1, eq01,
                          current, eq1c,
                          --power);
        } else { /* power is even */
            final FactorHelper
                base2 = square(base1.n,
                               base1.factors,
                               base1.proof);
            final Eq<base1.factors,
                     base2.union.c> eq12 =
                unionEqSets1(base1.factors,
                             base1.factors,
                             base2.union.c,
                             base2.union.proof,
                             refl(base1.factors);
            final Eq<base0.factors,
                     base1.factors> eq02 =
                trans(base0.factors,
                      base1.factors,
                      base2.factors,
                      eq01, eq12);
            final Eq<base2.factors,
                     current.factors> eq2c =
                trans(base2.factors,
                      base1.factors,
                      current.factors,
                      symm(base1.factors,
                           base2.factors, eq12),
                      eq1c);
            return expRec(base0, base2,
                          eq02, current,
                          eq2c, power/2); } }
```

Figure 4. Exponentiation: recursive helper method.

```
public static ExpHelper
    exp(final long base,
        final int power) {
    if (1 > power) {
        throw new IllegalArgumentException
                    ("Non-positive exponent");
    } else {
        final FactorHelper<base>
            fb0 = factor(base);
        final Eq<fb0.factors,
                 fb0.factors>
            f0eqf1 = refl(fb0.factors);
        return expRec(fb0, fb0, f0eqf1, fb0,
                      f0eqfl, power); } }
```

Figure 5. Exponentiation: top-level method.

```
public interface Or<X,Y> {
    public <C> C visit(OrVisitor<X,Y,C> v);
}
public interface OrVisitor<X,Y,C> {
    public C visitOr1(X x);
    public C visitOr2(Y y);
}
```

Figure 6. Interfaces for disjunctions and visitors of disjunctions.

```
public class Or1<X,Y> implements Or<X,Y> {
    X x;
    public Or1(X x) {
        this.x = x;
    }
    public <C> C visit(OrVisitor<X,Y,C> v) {
        return v.visitOr1(x);
    }
}

public class Or2<X,Y> implements Or<X,Y> {
    Y y;
    public Or2(Y y) {
        this.y = y;
    }
    public <C> C visit(OrVisitor<X,Y,C> v) {
        return v.visitOr2(y);
    }
}
```

Figure 7. The classes implementing Or.

We make the declarations of Figures 6 and 7 for disjunctions. Suppose we want to perform a case analysis on an object p of type $\mathsf{Or}\langle\mathsf{Leq}\langle x, y\rangle, \mathsf{Leq}\langle y, x\rangle\rangle$, returning an object of type C in both cases. We simply call p.visit with a class implementing the interface $\mathsf{OrVisitor}\langle X, Y, C\rangle$, whose visitOr1 and visitOr2 methods handle the cases $\mathsf{Leq}\langle x, y\rangle$ and $\mathsf{Leq}\langle y, x\rangle$ respectively.

We now return to our MergeNonNull example. We equip our trusted LeqAuthority class with the following method for comparing ints:

```
public static Or<Leq<i,j>,Leq<j,i>>
compare(final int i, final int j) {
  if (i <= j)
    return new Or1<Leq<i,j>,Leq<j,i>>
                          (new Leq<i,j>());
  return new Or2<Leq<i,j>,Leq<j,i>>
                          (new Leq<j,i>());
}
```

Then the body of MergeNonNull just compares the first elements of the two sorted input lists, and returns the result of using an (anonymous) OrVisitor to handle whichever situation of *l1.data* $\leq$ *l2.data* or *l2.data* $\leq$ *l1.data* occurs:

```
return
 LeqAuthority.compare(l1.data, l2.data).visit
   (new OrVisitor<Leq<i,j>,Leq<j,i>,
                  MergeInv<l1,l2>>()
    { ... });
```

That OrVisitor works as follows. For the second case, it takes the following shortcut, where MinAuthority.minSymm takes a proof that $x = \mathit{min}(y, z)$ and returns a proof that $x = \mathit{min}(z, y)$:

```
public MergeInv<l1,l2> visitOr2(Leq<j,i> c) {
  MergeInv<l2,l1> r =
    MergeNonNull(l2,l1,p2,p1,p4,p3);
  return new MergeInv(r.l, r.p1,
              MinAuthority.minSymm(r.l.data,
                                   l2.data,
                                   l1.data,
                                   r.p2));
}
```

So the main action of MergeNonNull happens in the case where *l1.data* $\leq$ *l2.data*. Code for the visitor method for this case is shown in Figure 8. The basic idea is that first, if we know that non-null List *l1* is sorted, then surely we know that the immediate sublist *l1.next* is sorted. Our trusted SortedAuthority provides a method sublistSorted implementing this fact. Using the proof obtained from that method, we can call a helper method called Merge2NonNull, which is just like MergeNonNull, except that it insists only that the second input list is NonNull. The first input list must still be sorted, but it might be null (we treat null Lists as sorted). The method Merge2NonNull returns a proof that *l1.next* is Null in the latter case. In the former, it returns an object of type $\mathsf{MergeInv}\langle \mathit{l1.next}, \mathit{l2}\rangle$. We explain how the code of Figure 8 handles these two situations next, starting with the second, slightly easier case.

5.1 Case: *l1.next* is null

When *l1.next* is Null, the visitOr2 method in Figure 8 is called (listed beneath the comment saying "SECOND CASE"). This method just prepends *l1.data* to *l2* using a method buildSorted1 of SortedAuthority, of the following type:

```
public static Helper<i>
buildSorted1(int i, List l,
             Sorted<l> p1, Leq<i,l.data> p2);
```

Here, Helper is the following helper class:

```
public static class Helper<int i> {
  public final List l;
  public final Sorted<l> p1;
  public final IntEq<l.data,i> p2; ... }
```

This relies on a property class IntEq expressing that two ints are equal. For simplicity, we do not pursue here the natural idea of having a single property class Eq parametrized by the type of the equal terms as well as the terms themselves. The idea with buildSorted1 is that Helper returns the result *l* of prepending *l1.data* onto *l2*, a proof that *l* is sorted, and a proof that its first element is equal to *i*.

5.2 Case: *l1.next* is non-null

The case listed beneath the comment saying "FIRST CASE" of Figure 8 is for when *l1.next* is non-null, and we have a proof of MergeInv$\langle$*l1.next*, *l2*$\rangle$. It is easy to extend this to to a proof of MergeInv$\langle$*l1*, *l2*$\rangle$. To do so, we rely on additional trusted helper methods, given in Figure 9. For example, MinAuthority.minEq implements a congruence principle: if $i = i'$ and i is the minimum of j and k, then i' is the minimum of j and k. The method SortedAuthority.step says that if l is sorted, then the first element in l is less than or equal to the second. Nothing here enforces that l has a first or second element. So we are making use here of the principle, described in Section 3 above, that property types can be indexed by expressions which would not evaluate if a mentioned reference (here l or $l.next$) were null.

5.3 Finishing

The code of Figure 8 finishes by using additional helper methods to build the required proof of MergeInv$\langle$*l1*, *l2*$\rangle$. The only code left to give is that for Merge2NonNull, which allows the first sorted input list to be possibly null. The code for this method is given in Figure 10, which relies on a trusted helper method to distinguish between the case where *l1* is Null and where it is NonNull. The actions in either case are simple, and we omit further explanation.

6. Runtime Verification

Even in the absence of support for property types, the above ideas can support more correct programming via runtime verification. To illustrate this, we recast the property class IsUnion$\langle$Set, Set, Set$\rangle$ as a simple Java class, as shown in Fig. 11. Here, IsUnion is a public static final inner class (of SetAuthority) with a private constructor that takes in the three sets to which it refers. The method check( ) is then used at run-time to verify the validity of the proof object. This runtime check may be called in any context that requires valid proofs about data, whether in trusted or untrusted code.

The proof is valid with respect to some data if the data objects are pointer-equal to those contained within the proof itself. Checking the pointer-equality of the objects is sufficient here because we trust the SetAuthority to perform the union operation correctly, only it can create new IsUnion objects, and we require that the objects to which the proof refers be immutable. This is important because, in this scheme, each runtime check requires only constant time. This is clearly preferable to asserting the correctness of the data at crucial steps, which in this case, would amount to asserting, for each element $\gamma \in c$, that either $\gamma \in a$ or $\gamma \in b$; for each $\alpha \in a$ that $\alpha \in c$; and for each $\beta \in b$ that $\beta \in c$. This would require linear time for each check. Note that if we choose *not* to completely trust the authority classes, or if we simply wish to provide further assurance of their correctness, a full (expensive) check of the property fits neatly in the check( ) method with the other runtime assertions.

Using these ideas, we implemented the exponentiation and list merging examples given above in Java 1.5 (without our proposed extensions), and ran some small tests without assert failures. This gives some confirmation, in the absence of a type checker for Java with property types, that the above examples work as intended.

7. Conclusion

We have proposed type-theoretically light extensions to the Java language that allow programmers to express semantically rich properties of their programs as types, and then verify those properties with respect to trusted authority code. Using dependent types and other facilities to ensure that objects are not null (or immutable), proofs are built and manipulated by trusted code as objects with term-indexed property types. Type correctness of such

```
public MergeInv<l1,l2>
visitOr1(final Leq<l1.data,l2.data> c) {
    Merge2NonNull
       (l1.next,
        l2,
        SortedAuthority.sublistSorted(l1,p1),
        p2, p4).visit
  final SortedAuthority.Helper<l1.data> n =
    (new OrVisitor<MergeInv<l1.next,l2>,
                   Null<l1.next>,
                   MergeInv<l1,l2>>() {
    //
    // FIRST CASE: l1.next is non-null.
    //
    public MergeInv<l1,l2>
    visitOr1(MergeInv<l1.next,l2> I) {
      Leq<l1.data,I.l.data> pp =
        LeqAuthority.leqMin
           (l1.data,
            I.l.data,
            l1.next.data,
            l2.data,
            SortedAuthority.step(l1,p1),
            c,I.p2);

      return SortedAuthority.buildSorted1
                  (l1.data, I.l, I.p1, pp);
    }

    //
    // SECOND CASE: l1.next is null.
    //
    public MergeInv<l1,l2>
    visitOr2(Null<l1.next> p) {
      return
        SortedAuthority.buildSorted1
          (l1.data, l2, p2, c);
    }

  });

  return new MergeInv<l1,l2>
        (n.l, n.p1,
         MinAuthority.minEq
           (l1.data,
            l1.data,
            l2.data,
            n.l.data,
            MinAuthority.min(l1.data,
                             l2.data, c),
            n.p2));
}
```

Figure 8. Visitor code used to handle the main case in MergeNonNull.

```
public static Min<ip,j,k>
MinAuthority.minEq(final int i,
                   final int j,
                   final int k,
                   final int ip,
                   Min<i,j,k> p1,
                   IntEq<i,ip> p2);

public static Min<i,j>
MinAuthority.min(int i, int j, Leq<i,j> p);

public static Leq<pi,qi>
LeqAuthority.leqMin(final int pi,
                    final int qi,
                    final int qj,
                    final int qk,
                    Leq<pi,qj> p1,
                    Leq<pi,qk> p2,
                    Min<qi,qj,qk> q);

public static Leq<l.data, l.next.data>
SortedAuthority.step(List l, Sorted<l> p1);
```

Figure 9. Signatures for trusted helper methods used in the code for MergeNonNull (other signatures discussed in the text).

```
protected static Or<MergeInv<l1,l2>,Null<l1>>
Merge2NonNull(final List l1, final List l2,
      final Sorted<l1> p1,
      final Sorted<l2> p2,
      final NonNull<l2> p3) {
  return NullA.isNull(l1).visit
    (new OrVisitor<Null<l1>,NonNull<l1>,
                   Or<MergeInv<l1,l2>,
                      Null<l1>>>()
    {
      public Or<MergeInv<l1,l2>,Null<l1>>
      visitOr1(Null<l1> p) {
        return new Or2<MergeInv<l1,l2>,
                       Null<l1>>(p);
      }
      public Or<MergeInv<l1,l2>,Null<l1>>
      visitOr2(NonNull<l1> p) {
        MergeInv<l1,l2> r =
          MergeNonNull(l1,l2,p1,p2,p,p3);
        return new Or1<MergeInv<l1,l2>,
                       Null<l1>>(r);
      }
    });
}
```

Figure 10. Helper method Merge2NonNull, allowing the first input list to be Null.

```
public static final class IsUnion {
    final Set a, b, c;
    private IsUnion(Set a, Set b, Set aub) {
        this.a = a; this.b = b;
        this.c = aub;
    }
    public void check(Set a, Set b, Set c) {
        if (this.a != a || this.b != b ||
            this.c != c) {
            throw new RuntimeException
                ("proof-object mismatch");
        } } }
```

Figure 11. Runtime verification of union.

manipulations is statically verified by the type checker. Our proposed method is capable of verifying non-trivial properties of programs, as demonstrated by two examples. Finally, we have discussed a pattern for runtime verification that emerged as a result of combining the programming-with-proofs paradigm with Java.

Future work includes a formal presentation of Java with property types, and its meta-theory. We also wish to augment a Java implementation to support property types, and to continue to develop and verify correct programs in this setting.

Acknowledgments

Thanks to the four anonymous reviewers for helpful feedback on an earlier version of this paper. This work is partially supported by NSF grant CCF-0448275 titled “CAREER: Semantic Programming”.

References

[1] J. Andrews. General Test Result Checking with Log File Analysis. *IEEE Transactions on Software Engineering*, 29(7), July 2003.

[2] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, 2002.

[3] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 35–49, Vancouver, BC, Canada, October 26–28, 2004.

[4] C. Boyapati, B. Liskov, and L. Shrira. Ownership Types for Object Encapsulation. In G. Morrisett, editor, *ACM Symposium on Principles of Programming Languages*, 2003.

[5] Chiyan Chen and Hongwei Xi. Combining Programming with Theorem Proving. In *Proceedings of the 10th International Conference on Functional Programming (ICFP05)*, Tallinn, Estonia, September 2005.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[7] M. Fahndrich, K. Rustan, and M. Leino. Declaring and checking non-null types in an object-oriented language, 2003.

[8] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended Static Checking for Java. *SIGPLAN Notices*, 37, 2002.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] J. Harrison. Formal Verification of IA-64 Division Algorithms. In *13th International Conference on Theorem Proving in Higher Order Logics*, 2000.

[11] W. Howard. *The formulae-as-types notion of construction*, pages 479–490. In Seldin and Hindley [23], 1980.

[12] G. Klein and T. Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298(3):583–626, 2003.

[13] C. McBride and J. McKinna. The View from the Left. *Journal of Functional Programming*, 14(1), 2004.

[14] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *5th Symposium on Operating Systems Design and Implementation*, December 2002.

[15] G. Necula. Proof-Carrying Code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[16] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[17] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[18] B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[19] M. Pol, R. Teunissen, and E. van Veenendaal. *Software Testing : a Guide to the TMap Approach*. Addison-Wesley, 2002.

[20] Research Triangle Institute. The Economic Impacts of Inadequate Infrastructure for Software Testing, 2002. Sponsored by the Department of Commerce's National Institute of Standards and Technology.

[21] D. Rosenblum. A Practical Approach to Programming With Assertions. *IEEE Transactions on Software Engineering*, 21(1), January 1995.

[22] H. Rueß N. Shankar, and M. Srivas. Modular Verification of SRT Division. *Formal Methods in System Design*, 14(1), 1999.

[23] J. Seldin and J. Hindley, editors. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[24] T. Sheard. Languages of the future. In D. Schmidt, editor, *Proceedings of 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[25] The Coq Development Team. The Coq Proof Assistant Reference Manual, Version V8.0, 2004. http://coq.inria.fr.

[26] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, San Diego, CA, USA, October 16–20, 2005.

[27] E. Westbrook, A. Stump, and I. Wehrman. A Language-based Approach to Functionally Correct Imperative Programming. In *Proceedings of the 10th International Conference on Functional Programming (ICFP05)*, 2005.

[28] Y. Xie and A. Aiken. Scalable Error Detection using Boolean Satisfiability. In M. Abadi, editor, *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, 2005.