Proving Programs Correct Using Plain Old Java Types

Radha Jagadeesan
DePaul University
rjagadeesan@cs.depaul.edu

Alan Jeffrey
Bell Labs, Alcatel-Lucent
ajeffrey@bell-labs.com

Corin Pitcher
DePaul University
cpitcher@cs.depaul.edu

James Riely *
DePaul University
jriely@cs.depaul.edu

Abstract

Tools for constructing proofs of correctness of programs have a long history of development in the research community, but have often faced difficulty in being widely deployed in software development tools. In this paper, we demonstrate that the off-the-shelf Java type system is *already* powerful enough to encode non-trivial proofs of correctness using propositional Hoare preconditions and postconditions.

We illustrate the power of this method by adapting Fähndrich and Leino's work on monotone typestates and Myers and Qi's closely related work on object initialization. Our approach is expressive enough to address phased initialization protocols and the creation of cyclic data structures, thus allowing for the elimination of null and the special status of constructors. To our knowledge, our system is the first that is able to statically validate standard one-pass traversal algorithms for cyclic graphs, such as those that underlie object deserialization. Our proof of correctness is mechanized using the Java type system, without any extensions to the Java language.

1. Introduction

1.1 Eclipse as an interactive proof assistant

Tools for proving the correctness of executable software have a long history of development in the research community. Notable examples include theorem provers such as Coq (Coquand and Huet 1988), Elf (Pfenning 1994), Isabelle (Nipkow et al. 2002) and Twelf (Pfenning and Schürmann 1999) and software model-checkers such as SLAM (Ball and Rajamani 2002). These techniques have been extended to native executables by proof-carrying code (Necula and Lee 1996; Appel 2001) and typed assembly language (Crary and Morrisett 1999). Theorem-proving methods for program correctness are often based on Logical Frameworks (LF) (Harper et al. 1993), which embed logics into program types.

These tools often require extensions to compilers, static analysis systems, run-time systems to support proofs of program correctness. Ideally, standard development environments such as Eclipse or Visual Studio would support proof assistants, and indeed the Spec# (Leino 2006) and Sing# (Fähndrich et al. 2006) languages provide design-by-contract capabilities, and SLAM is available as part of the Windows Drivers Kit.

In this paper, we demonstrate that *with no language or tool extensions*, Java's type system is already powerful enough to encode propositional logic with Hoare (1969) pre- and post-conditions. We do this by adapting the LF encoding of propositional logic to Java's type system. Java types support enough first-order polymorphism to encode zeroth-order logic.

With this in hand, we use the features of the Java type system to represent properties of interest. Roughly, we use one generic parameter per different property of interest in a class. We use generic type parameters to methods to stand for pre-conditions on the callers and the arguments. Similarly, return types are used to yield postconditions on results and represent that the arguments will satisfy a certain property after the call.

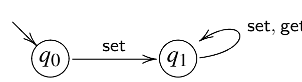
In combination with the observation that phantom types (Leijen and Meijer 1999) can be used to track object properties, our encoding permits us to address interesting protocols such as deserialization.

We have implemented this encoding as a Java API, and have used the Eclipse IDE as an interactive proof assistant to verify the correctness of programs.

There is one important proviso to our results. The soundness of the LF encoding *only* applies to Java with no uses of the null pointer. This is because we are using type inhabitance as formal proof, but (alas!) in Java all object types are inhabited by null. The soundness therefore depends for its correctness on the proof of null-freedom. We resolve this tension by *relying* on the programmer using our system to not using the features of Java that produce null directly (i.e. no explicit null, initialize all fields, don't allow this to escape from constructors, etc.). Given such programmer compliance with these "easy" cases, our methods establish the checks required for the "difficult" cases of null-freedom. Indeed, we can't do much better and stay inside the Java type system.

1.2 Case study: monotone typestates

Traditional types only capture the static signatures of functions and objects; for example, that field f references a file whereas field n references a node of a cyclic graph (when non-null). These types do not capture dynamically variant information: consider the simple example of a logical variable (Lindstrom 1985), say l, that moves from an uninitialized state (that does not have an enabled get method) to an initialized state (with a get method enabled) upon invocation of a set method:



Traditional types provide an upper bound on capabilities available to an object; the type system always allows the calls l.get() and l.set() regardless of the state of the object. The more precise contract on the use of these capabilities is specified informally and

* Supported by NSF Career 0347542.

```
public final class Pair⟨σ,τ⟩ {
  σ x; τ y;
  public void setFst(σ x) { this.x = x; }
  public void setSnd(τ y) { this.y = y; }
  public σ getFst() { return x; }
  public τ getSnd() { return y; }
}
...
Pair⟨Pair⟨Object,Object⟩,Object⟩ u
    = new Pair⟨Pair⟨Object,Object⟩,Object⟩;
Pair⟨Object,Object⟩ v = new Pair⟨Object,Object⟩
u.setFst(v);
u.getFst().setFst(new Object());
v.getFst();
```

Figure 1. Simple Example

tracked manually by programmers, creating many opportunities for error.

Two distinct areas of research address this issue formally:

- Session types for communication centered programming (Takeuchi et al. 1994; Honda et al. 1998) specify the interaction between the sender and receiver on a channel as part of the type; the Singularity operating system (Fähndrich et al. 2006) implements session types as communication abstractions in a general programming context. Static analysis is used to establish that no communication errors occur.
- Typestates for object protocols (Strom 1983; Strom and Yemini 1986) specify the structure of the interaction between the clients and the objects. For example, a typestate can distinguish between the open/closed state of a file or the uninitialized/initialized state of a data structure. Static analysis is used to establish that object protocols are respected.

Linearity and uniqueness ideas play a key technical role in both developments. In the object world, linearity is tantamount to unique references and allows one to sidestep the difficult analysis issues caused by aliasing, e.g., if there are multiple references to a socket object, how can two asynchronously executing clients agree on the typestate of the socket without explicit communication?

Such linearity and aliasing restrictions impede the widespread adoption of typestate ideas in general program development and motivate the consideration of monotone typestates. Monotone typestates (Fähndrich and Leino 2003) are stable under program dynamics, i.e., once a typestate of an object is established, no future interaction with the object (including imperative updates) invalidates this typestate assertion. The logical variable above — crucially, without an unset method — exemplifies monotone typestate. Such monotone typestates are amenable to relaxed aliasing constraints since various aliases to an object can only monotonically advance the typestate of the object (and other shared data structures). Furthermore, since the safe operation of a client on an object does not require awareness of the operations of other clients on shared objects, monotone typestates are also compatible with concurrency.

Our case study shows that the Java type system — as it stands — already addresses the two challenges posed by Fähndrich and Xia (2007) and Fähndrich and Leino (2003): phased initialization and cyclic data structures. We do this by providing a reference API for non-final fields, which uses our coding of pre- and post-conditions to track initialization. For ease of comparison, we elide explicit use of our Reference API throughout the rest of this introduction.

Our techniques facilitate incremental construction of objects, as illustrated in Figure 1, even while ensuring that client access is restricted to properly initialized sections of the object reference graph, e.g. to rule out access to second element of the pair u after the above code. Fähndrich and Leino's (Fähndrich and Leino 2003) monotone typestates system insists that assignments use fully-initialized objects in order to prevent safety issues that are otherwise hard to avoid in the presence of aliasing, thus ruling out the order of object initialization in the program above: u.setFst(v) is not allowed since v is not fully initialized.

We permit the above program, a feature that we share with the state-of-the-art *masked types* system (Qi and Myers 2009). Masked types tracks dependencies, e.g. in the above program initialization of u depends on the initialization of v after the assignment u.setFst(v). Inspired by a bisimulation argument, Qi and Myers (2009) provide an elegant proof rule to eliminate strongly connected components of dependencies. Our system is much simpler (relying on just a propositional Hoare logic) but we show that it can prove correctness of the examples in their paper. (Since our system does not track aliases, we do not have the expressive power of their system, in particular we cannot handle their ! annotation.)

In addition, our proposal is the first solution (to our knowledge) that is also able to statically validate a one-pass graph construction algorithm. Consider Figure 2, where an unsafe graph construction algorithm is given. If used incorrectly, this code can result in a surprising null pointer exception. The bug is caused by the buggy code:

```
this.left = new Vertex(nL);
t = this.left.init(d,t);
```

that fails to maintain an invariant that child nodes are initialized before the parent pointer is set. The correct code should be:

```
Vertex vL = new Vertex(nL);
t = vL.init(d,t);
this.left = vL;
```

The code bug is detected by a deliberately broken Description object:

```
final Vertex v = Vertex.build(d);
Description bad = new Description() {
  String root() { return d.root(); }
  String left() { v.left.left.left; return d.left(); }
  String right() { v.left.left.left; return d.right(); }
}
v.init(bad,Table.build());
```

Our proof system tracks this and hence does not allow the buggy code to typecheck. Masked types cannot capture this example (Qi, personal communication). Similarly Fähndrich and Leino (2003) require cyclic structures to be initialized with dummy nodes, and so do not treat one-pass examples such as this.

In contrast, our proof of correctness of the fixed graph construction algorithm in Figure 2 is mechanized using the Java type system, thus giving a proof of correctness which is ahead of the state of the art, without any extensions to the Java language.

Our contribution is particularly significant because the additional examples that our approach can handle are at the core of object permanence and distribution. Serialization takes an object graph and turns it into stream of bytes that can be stored on disk or transmitted to another machine; deserialization reverses the process. The type that we assign the deserialization algorithm establishes that the returned object graph is fully initialized, and therefore contains no null pointers.

1.3 Rest of this paper

In Section 2 we present language extensions to Java to add support for propositional pre- and post-conditions. These extensions are purely for presentation purposes; in Section 4 we show that these extensions are just syntax sugar for existing Java. Before that,

```
interface Description {
  String root();
  String left(String n);
  String right(String n);
}
abstract class Table {
  Table put(String n, Vertex v);
  Vertex get(String n);
  boolean containsKey(String n);
  static Table build() { ... }
}
public final class Vertex {
  final String name;
  Vertex left;
  Vertex right;
  Vertex(String name) { this.name = name; }
  public Table init(Description d, Table t) {
    t.put(this.name,this);
    String nL = d.left(this.name);
    if (t.containsKey(nL)) {
      this.left = t.get(nL);
    } else {
      this.left = new Vertex(nL);
      t = this.left.init(d,t);
    }
    ... ditto right ...
    return t;
  }
  static public Vertex build(Description d) {
    Vertex v = new Vertex(d.root());
    v.init(d,Table.build());
    return v;
  }
}
```

Figure 2. Example of unsafe cyclic graph construction

in Section 3 we demonstrate the use of Java types for proving the correctness of initialization for several cyclic data structures; the syntax sugar makes these examples much more readable. The proofs of correctness are mechanized in the Eclipse IDE with no additional plug-ins. We conclude with a discussion of future work in Section 8. For a fuller version of this paper, with additional examples and definitions, see http://www.depaul.edu/~jriely/papers/2009-pojt.pdf

2. Language Extensions

In this section we give a presentation of our Java language extensions, and give illustrative examples based on ensuring safe access to object references. Our extensions are:

1. Support for inline unpacking of existential types.
2. Propositional logic and proofs in Java types.
3. An API for object references tracking initialization.

The extra layer of indirection introduced by references could be removed by changing the semantics of fields to match that of our reference types; our interest here, however, is to develop techniques that work with the existing Java language. In fact, in Section 4 we will show that these language extensions are, in fact, not extensions at all, but can be expressed in the existing Java type system.

2.1 Unpacking Existential Types

Our first extension to Java is not related to formalizing proof, but is just to increase the readability of programs. We will be making heavy use of existential types (Mitchell and Plotkin 1988) (also known as wildcard types in Java (Torgersen et al. 2004)), but Java provides no mechanism for inline unpacking of existentials, and instead requires unpacking to be performed across method calls.

```
interface Comp⟨σ◁ Comp⟨σ⟩⟩ { public boolean leq(σ x); }
public class BTree⟨σ◁ Comp⟨σ⟩,τ⟩ {
  public void put(σ key, τ value) { ... }
  ...
}
interface GetMap⟨σ⟩ { public Map⟨σ,?⟩ map(); }
public class GetTreeFromMap {
  public⟨σ◁ Comp⟨σ⟩⟩ BTree⟨σ,?⟩ tree(GetMap⟨σ⟩ a) {
    protected class Unpacker {
      protected⟨τ⟩ BTree⟨σ,τ⟩ tree(Map⟨σ,τ⟩ map) {
        BTree⟨σ,τ⟩ result = new BTree⟨σ,τ⟩();
        for (σ x : map.keys()) { result.put(x,map.get(x))); }
        return result;
      }
    }
    new Unpacker().tree(a.map());
  }
}
```

Figure 3. Example of unpacking an existential in Java.

```
public class GetTreeFromMap {
  public⟨σ◁ Comp⟨σ⟩⟩ BTree⟨σ,?⟩ tree(GetMap⟨σ⟩ a) {
    unpack⟨τ⟩ Map⟨σ,τ⟩ map = a.map();
    BTree⟨σ,τ⟩ result = new BTree⟨σ,τ⟩();
    for (σ x : map.keys()) { result.put(x,map.get(x))); }
    return result;
  }
}
```

Figure 4. Example of syntax sugar for unpacking.

$$B \ ::= \ \mathsf{unpack}\langle \vec{\sigma} \lhd \vec{T}\rangle T x = M;\ B \mid \cdots$$

$$\frac{E \ni c\langle \vec{\sigma} \lhd \vec{T}, \vec{\tau} \lhd \vec{U}\rangle \lhd \vec{V} \quad E \vdash M : c\langle \vec{W}, \vec{?}\rangle \quad E, \vec{\upsilon} \lhd \vec{U}\{\vec{W},\vec{\upsilon}/\vec{\sigma},\vec{\tau}\}, x : c\langle \vec{W}, \vec{\upsilon}\rangle \vdash B : T}{E \vdash (\mathsf{unpack}\langle \vec{\upsilon} \lhd \vec{U}\{\vec{W},\vec{\upsilon}/\vec{\sigma},\vec{\tau}\}\rangle c\langle \vec{W}, \vec{\upsilon}\rangle x = M;\ B) : T}$$

Figure 5. Syntax and type rule for unpacking existentials

This often impacts code readability, as seen in Figure 3. Java requires the use of a separate unpacker class to provide access to the map object at type $\mathsf{Map}\langle\sigma,\tau\rangle$ rather than $\mathsf{Map}\langle\sigma,?\rangle$ (an unpacker method in the same class cannot be used, as we would have no way to pass it the σ type parameter). We will increase the readability of our code by introducing an explicit unpack operation which unpacks an existential type, as shown in Figure 4.

The syntax and type rule for unpacking is given in Figure 5, and are an adaption of the usual type rule for unpacking existential types for Java wildcards. We do not present full type rules for a fragment of Java here, but note that we are assuming type judgements $E \vdash B : T$ read as "in typing environment E, block B returns type T", and $E \vdash M : T$ read as "in typing environment E, expression M has type T. A fully formal treatment of types for Java is given in (Igarashi et al. 2001) (for an expression-based rather than block-based language, but we do not expect this to be an important distinction). Typing environments E contain bindings for class declarations $c\langle\vec{\sigma} \lhd \vec{T}\rangle \lhd \vec{U}$, variables $x : T$ and type variables $\sigma \lhd T$. For conciseness, we write $\lhd$ as shorthand for Java's extends keyword.

```
public class Formula⟨α:Ω⟩ { }
public class Proof⟨α:Ω,π:α⟩ { }
```

Figure 6. Classes for formulae and proofs

$$\begin{array}{rcl} \alpha,\beta,\gamma & \in & \text{propositional variables} \\ F,G,H & ::= & \alpha \mid \mathbf{1} \mid \mathbf{0} \mid F \Rightarrow G \mid F \wedge G \mid F \vee G \end{array}$$

$$\frac{E \ni (\alpha:\Omega)}{E \vdash \alpha:\Omega} \quad \frac{}{E \vdash \mathbf{1}:\Omega} \quad \frac{}{E \vdash \mathbf{0}:\Omega} \quad \frac{E \vdash F:\Omega \quad E \vdash G:\Omega}{E \vdash F \Rightarrow G:\Omega}$$

$$\frac{E \vdash F:\Omega \quad E \vdash G:\Omega}{E \vdash F \wedge G:\Omega} \quad \frac{E \vdash F:\Omega \quad E \vdash G:\Omega}{E \vdash F \vee G:\Omega}$$

Figure 7. Syntax and type rules for propositional formulae

$$\begin{array}{rcl} \pi,\rho,\phi,\psi & \in & \text{proof variables} \\ P,Q,R & ::= & \pi \mid \textsc{Term}\langle F\rangle \mid \textsc{Init}\langle F\rangle \mid \textsc{Id}\langle F\rangle \mid \textsc{Comp}\langle P,Q\rangle \\ & \mid & \textsc{Curry}\langle P\rangle \mid \textsc{Uncurry}\langle P\rangle \mid \textsc{Apply}\langle P,Q\rangle \\ & \mid & \textsc{AndM}\langle P,Q\rangle \mid \textsc{Proj1}\langle F,G\rangle \mid \textsc{Proj2}\langle F,G\rangle \\ & \mid & \textsc{OrM}\langle P,Q\rangle \mid \textsc{Inj1}\langle F,G\rangle \mid \textsc{Inj2}\langle F,G\rangle \end{array}$$

$$\frac{E \ni (\pi:F)}{E \vdash \pi:F} \quad \frac{E \vdash F:\Omega}{E \vdash \textsc{Term}\langle F\rangle : F \Rightarrow \mathbf{1}} \quad \frac{E \vdash F:\Omega}{E \vdash \textsc{Init}\langle F\rangle : \mathbf{0} \Rightarrow F}$$

$$\frac{E \vdash F:\Omega}{E \vdash \textsc{Id}\langle F\rangle : F \Rightarrow F} \quad \frac{E \vdash P:F \Rightarrow G \quad E \vdash Q:G \Rightarrow H}{E \vdash \textsc{Comp}\langle P,Q\rangle : F \Rightarrow H}$$

$$\frac{E \vdash P:F \Rightarrow G \quad E \vdash Q:F}{E \vdash \textsc{Apply}\langle P,Q\rangle : G} \quad \frac{E \vdash P:(F \wedge G) \Rightarrow H}{E \vdash \textsc{Curry}\langle P\rangle : F \Rightarrow (G \Rightarrow H)}$$

$$\frac{E \vdash P:F \Rightarrow (G \Rightarrow H)}{E \vdash \textsc{Uncurry}\langle P\rangle : (F \wedge G) \Rightarrow H} \quad \frac{E \vdash P:F \Rightarrow G \quad E \vdash Q:F \Rightarrow H}{E \vdash \textsc{AndM}\langle P,Q\rangle : F \Rightarrow G \wedge H}$$

$$\frac{E \vdash F:\Omega \quad E \vdash G:\Omega}{E \vdash \textsc{Proj1}\langle F,G\rangle : F \wedge G \Rightarrow F} \quad \frac{E \vdash F:\Omega \quad E \vdash G:\Omega}{E \vdash \textsc{Proj2}\langle F,G\rangle : F \wedge G \Rightarrow G}$$

$$\frac{E \vdash P:F \Rightarrow H \quad E \vdash Q:G \Rightarrow H}{E \vdash \textsc{OrM}\langle P,Q\rangle : F \vee G \Rightarrow H}$$

$$\frac{E \vdash F:\Omega \quad E \vdash G:\Omega}{E \vdash \textsc{Inj1}\langle F,G\rangle : F \Rightarrow F \vee G} \quad \frac{E \vdash F:\Omega \quad E \vdash G:\Omega}{E \vdash \textsc{Inj2}\langle F,G\rangle : G \Rightarrow F \vee G}$$

Figure 8. Syntax and type rules for propositional proofs

2.2 Propositional Logic

Our Java extension requires programmers to annotate programs with logical properties and proofs. Figure 7 describes the logic.

- $F:\Omega$, which means "F is a formula", including logical variables $\alpha:\Omega$. We abbreviate $\alpha:\Omega,\beta:\Omega$ as $\alpha,\beta:\Omega$.
- $P:F$, which means "P is a proof of F", including proof variables $\pi:F$.

Figure 8 describes the proofs. We note that these proofs are just a syntactic description of proof trees for derivability in intuitionistic propositional logic, using a first-order fragment of a Logical Framework (Harper et al. 1993).

Example formulae, with their derived proofs, are standardly deduced. These include::

- $\textsc{TrueI}:\mathbf{1}$
- If $P:F$ and $Q:G$ then $\textsc{AndI}\langle P,Q\rangle : F \wedge G$
- If $P:F \wedge G$ then $\textsc{AndE1}\langle P\rangle : F$ and $\textsc{AndE2}\langle P\rangle : G$

We allow propositional formulae and proofs to occur wherever types may occur in the Java syntax, for example in type parameters to generic classes and methods, as wildcards, and in unpack. For example, Figure 6 gives standard classes for atoms, formulae and proofs. Note that since Java implements generics by type erasure (Bracha et al. 1998), the proof annotations on a program are erased along with the type annotations. Any proofs the program carries out are only performed at compile-time, not at run-time. This is an instance of phantom types — "polymorphic types whose type parameter is only used at compile-time but whose values never carry any value of the parameter type" (Leijen and Meijer 1999).

```
public abstract class Ref⟨σ,α:Ω⟩ {
  public abstract Proof⟨α,?⟩ set(σ x);
  public abstract⟨π:α⟩ σ get();
  static public⟨σ⟩ Ref⟨σ,?⟩ build() { return new RefImpl⟨σ⟩(); }
}
```

Figure 9. Reference API: public interface

Note that the soundness of this logic depends on null-freedom of the program, for example we can construct a proof of **0** as:

```
unpack⟨π:0⟩ Proof⟨0,π⟩ p = null;
```

This is an inherent feature of Java: all object types are inhabited by null, and so any attempt to encode proofs as programs will be unsound in the presence of null. Our formal proof of soundness, in Section 5 does not allow null to be typechecked for this reason.

2.3 Reference API

We now have enough infrastructure to statically check propositional proofs. We now show how this can be used to track the initialization state of references. Consider the Ref⟨σ,α⟩ type defined in Figure 9. This gives an API for references of type σ, which are initialized whenever α can be proved true.

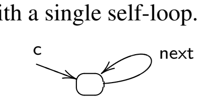
Whenever a reference r is generated, a new atomic proposition α is generated, and is tracked in the type Ref⟨T,α⟩. Initially, no proof of α is provided, and so r.get cannot be called, but when r.set is called, it returns a proof $\pi:\alpha$, which allows r.⟨π⟩get to be called.

Representing preconditions (such as the one to get) as type parameters to generic methods ensures that the caller has to establish them. Postconditions (such as the one from set) are represented as returns of Proof objects of appropriate type.

In this case, since the only way to generate proofs of α are from calling set, we ensure that references are initialized before they are accessed. (Recall that type parameters to a Java class are not in scope in the static methods of the class; thus the parameter σ to build does not hide the class parameter of the same name. Recall also that Java constructors implicitly share the type arguments of the class.)

Some simple example programs are shown in Figure 10, demonstrating a prototypical use of a reference, an unsafe use which is caught by the compiler, and an unsafe use where the programmer uses casting to deliberately bypass the static guarantees.

Cyclic structures are interesting because they require mutability during construction, even if the final product is immutable. In Figures 11 and 12 we give the simplest example of a cyclic data structure: an object with a single self-loop.



The static method C.build creates an instance of C and returns it, packed with type variable α. The method C.init then sets the self loop, satisfying the postcondition α by returning a proof of α. This allows subsequent calls to next, which has precondition α.

The code is mostly routine, but there is one interesing point, which is that when the next reference is created in C.build, the contents are given type C⟨σ⟩ which depends on σ. This cannot be typed using the existing unpack type rule in Figure 5 as the

```
public void ok() {
  unpack⟨α:Ω⟩ Ref⟨String,α⟩ r = Ref.⟨String⟩build();
  unpack⟨π:α⟩ r.set("hello, world");
  System.out.println(r.⟨π⟩get());
}
public void notOK() {
  unpack⟨α:Ω⟩ Ref⟨String,α⟩ r = Ref.⟨String⟩build();
  // r.get() generates a compile-time type bound error
  System.out.println(r.get());
}
public void usesCasts() {
  unpack⟨α:Ω⟩ Ref⟨String,α⟩ r = Ref.⟨String⟩build();
  Proof⟨1,?⟩ p = Proof.⟨TrueI⟩build();
  // Casting bypasses static checking
  @SupressWarnings("unchecked")
  Proof⟨α,?⟩ q = (Proof)p;
  unpack⟨π:α⟩ q;
  System.out.println(s.⟨π⟩get());
}
```

Figure 10. Examples of using references

```
public class C⟨α:Ω⟩ {
  final Ref⟨C⟨α⟩,α⟩ next;
  C(Ref⟨C⟨α⟩,α⟩ next) { this.next = next; }

  public Proof⟨α,?⟩ init() { return this.next.set(this); }
  public⟨π:α⟩ C⟨α⟩ next() { return this.next.⟨π⟩get(); }

  static public C⟨?⟩ build() {
    cyclic unpack⟨α:Ω⟩ Ref⟨C⟨α⟩,α⟩ r = Ref.⟨C⟨α⟩⟩build();
    return new C⟨α⟩(r);
  }
}
```

Figure 11. Example of cyclic references

```
public void run() {
  unpack⟨α:Ω⟩ C⟨α⟩ c = C.build();
  unpack⟨π:α⟩ c.init();
  c.⟨π⟩next().⟨π⟩next();
}
```

Figure 12. Example of using cyclic references

$$B \ ::= \ \mathsf{cyclic\ unpack}\langle\vec{\sigma}\lhd\vec{T}\rangle T x = M;\ B \ \mid\ \cdots$$

$$\frac{E,\alpha:\Omega \vdash U \lhd \mathsf{Object} \qquad E,\alpha:\Omega,x:\mathsf{Ref}\langle U,\alpha\rangle \vdash B:T}{E \vdash (\mathsf{cyclic\ unpack}\langle\alpha:\Omega\rangle \mathsf{Ref}\langle U,\alpha\rangle x = \mathsf{Ref}.\langle U\rangle \mathsf{build}();\ B):T}$$

Figure 13. Syntax and type rule for cyclic unpacking

program

```
unpack⟨α:Ω⟩ Ref⟨T,F⟩ r = Ref.⟨T⟩build();
```

will only typecheck when T does not contain α.

To support cyclic structures such as C, we allow the creation of references which mention their own type using a cyclic unpack operation. In Section 4 we give a general framework for such cyclic unpacking, but for now we will just give the type rule for the specialized instance of unpacking a fresh reference in Figure 13. The difference between this rule and the specialization of the rule in Figure 5 is whether α is in scope when we check $U \lhd$ Object; in Figure 5 α is not in scope, but in Figure 13 it is. One may generalize Figure 13 to support mutual recursion simply by allowing vectors of type variables on the left and vectors of reference declarations on the right. Our implementation supports this generalization.

3. Examples

3.1 Partially Completed Structures

We borrow an example from Fähndrich and Leino (2003) to illustrate the use of our methods to specify pre(post) conditions and data invariants. This example views the AST in a typical compiler front-end as being in one of three states: (1) An initial state "Naked" for the node created after parsing, (2) State "Bound" indicates completion of name resolution, and (3) State "Typed" indicates completion of type checking. The left of Figure 14 describes the Java version of their annotations. For example, the precondition on typeCheck is that the receiver object is in state "Bound" and the postcondition indicates that the method on completion leaves the object in state "Typed".

Our translation of this interface is described in the right of Figure 14. In the interface AstNode$\langle\alpha,\beta:\Omega\rangle$, α represents status of "Bound" and β represents status of "Typed". Furthermore, following the earlier described pattern for programming pre/post conditions: (a) the precondition $\langle\rho:\alpha\rangle$ of typeCheck ensures that the caller has a proof of xfrm, and (b) the return type Proof$\langle\beta,?\rangle$ ensures that the method establishes the postcondition β.

Figure 15 shows how Fähndrich and Leino (2003) use typestates can be used to capture data invariants using code fragments of Expr subclasses. Our version is presented on the right. In our version, in UnaryExpr$\langle\beta,\gamma,\rangle:\Omega$, γ captures whether the type field is set, and β captures whether the subexpression is typed. Consider the requirements that arise if the unary expression object is in state "Typed".

- Their [InState("Typed", WhenEnclosingState="Typed")] captures the requirement that the subexpression is in state "Typed". The logical implication $\beta\wedge\gamma\Rightarrow\beta$ achieves this in our presentation.
- The requirement that the type field is set is captured by their annotation [NotNull(WhenEnclosingState="Typed")] and in our system by the logical implication $\beta\wedge\gamma\Rightarrow\gamma$.

One notable difference is that their main body requires no annotation, where ours requires explicit annotation with proofs. Thus, we are only considering proof checking, where Fähndrich and Leino consider proof inference.

3.2 Pre/Post conditions on arguments

Preconditions on arguments are also given by parametric proofs; e.g., the requirement that prev be initialized is given as:

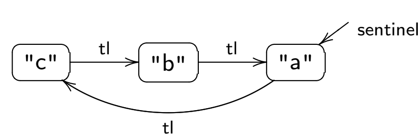
```
public⟨α:Ω,π:α⟩ void foo(Ref⟨String,α⟩ prev) { prev⟨π⟩.get(); }
```

Similarly, to represent that the argument prev will satisfy α after the call:

```
public⟨α:Ω⟩ Proof⟨α,?⟩ bar(Ref⟨String,α⟩ prev) { return prev.set(''hello''); }
```

3.3 Cyclic Structures

In Figures 16 and 17 we give a more realistic example of a cyclic structure. When called with the arguments "a", "b" and "c", the program creates the following circular list.



```
// Fähndrich and Leino's original (ported to Java)
void main(...) {
  AstNode ast = p.parse(file);
  unpack ast.resolveNames(emptyEnv);
  unpack ast.typeCheck(emptyTEnv);
  ast.emit(...);
}
interface Parser {
  [return:Post("Naked")] public AstNode parse(String file);
}
interface AstNode {
  [Pre("Naked"),Post("Bound")] public void resolveNames(Env env);
  [Pre("Bound"),Post("Typed")] public typeCheck(TEnv tenv);
  [Pre("Typed")] public void emit(...);
}
```

```
// Our translation
void main(...) {
  unpack⟨α,β:Ω⟩ AstNode⟨α,β⟩ ast = p.parse(file);
  unpack⟨π:α⟩ ast.resolveNames(emptyEnv);
  unpack⟨ρ:β⟩ ast.⟨π⟩typeCheck(emptyTEnv);
  ast.⟨ρ⟩emit(...);
}
interface Parser {
  public AstNode⟨?,?⟩ parse(String file);
}
interface AstNode⟨α,β:Ω⟩ {
  public Proof⟨α,?⟩ resolveNames(Env env);
  public⟨ρ:α⟩ Proof⟨β,?⟩ typeCheck(TEnv tenv);
  public⟨φ:β⟩ void emit(...);
}
```

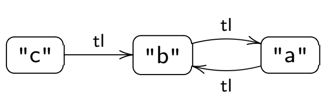
Figure 14. Fähndrich and Leino's AST example

```
// Fähndrich and Leino's original (ported to Java)
interface Expr ◁ AstNode {
  [Pre("Typed")] public Type type();
  ...
}
abstract class ExprImpl ◁ Expr {
  [NotNull(WhenEnclosingState="Typed")] Type type;
  ...
  [Pre("Typed")] public Type type() {
    return this.type;
  }
}
class UnaryExpr ◁ ExprImpl {
  Op op;
  [NotNull(WhenEnclosingState="Naked,Bound,Typed")]
  [InState("Naked", WhenEnclosingState="Naked")]
  [InState("Bound", WhenEnclosingState="Bound")]
  [InState("Typed", WhenEnclosingState="Typed")]
  Expr arg;
  ...
  [Pre("Bound"),Post("Typed")]
  public void typeCheck(TEnv tenv) {
    this.arg.typeCheck(tenv);
    Type t = Type.build(this.op,this.arg.type());
    this.type = t;
  }
}
```

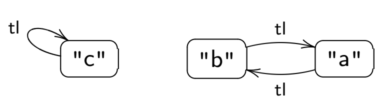
```
// Our translation
interface Expr⟨α,β:Ω⟩ ◁ AstNode⟨α,β⟩ {
  public⟨π:β⟩ Type type();
  ...
}
abstract class ExprImpl⟨α,β,γ:Ω⟩ ◁ Expr⟨α,⟩ {
  final Ref⟨Type,γ⟩ type;
  ...
  public⟨π:β∧γ⟩ Type type() {
    return this.type.⟨ANDE2⟨π⟩⟩.get();
  }
}
class UnaryExpr⟨α,β,γ:Ω⟩ ◁ ExprImpl⟨α,β,γ⟩ {
  final Op op;
  final Expr⟨α,β⟩ arg;
  ...

  public⟨π:α⟩ Proof⟨β∧γ⟩ typeCheck(TEnv tenv) {
    unpack⟨ρ:β⟩ this.arg.typeCheck(tenv);
    Type t = Type.build(this.op,this.arg.⟨ρ⟩type());
    unpack⟨φ:γ⟩ this.type.set(t);
    return new Proof⟨ANDI⟨ρ,φ⟩⟩();
  }
}
```

Figure 15. Fähndrich and Leino's AST implementation

The code builds the cyclic list, then traverses it three times. By changing the highlighted line in Figure 17, one can build the following structure instead.



Subsequently using setTail from Figure 16, one can further create structures such as the following:



Nodes may be created two ways: using the static method S.build to create a sentinel, or using N.cons to create a node which prepends an existing node. Instances of the sentinel class S$\langle\alpha\rangle$, come packed with type variable α, which is precondition to the accessor method tail. Any node created using cons shares the precondition of its sentinel. The tail of any node must share its precondition. In particular, one cannot link two independently created sentinels using setTail, since they will not share the same precondition.

An instance of N$\langle\alpha,\beta:\Omega,\pi:\alpha\Rightarrow\beta\rangle$ has a tail method guarded by α. Clients use nodes at type N$\langle\alpha,?,?\rangle$. The variables β and π are instantiated by S.build and N.cons. β is the guard for underlying reference and π is a proof that β implies α. S.build creates nodes where β is α; therefore π is simply $\textsc{Id}\langle\alpha\rangle$. N.cons creates new nodes and sets their tail reference, establishing the proof π' that the tail is set; therefore π is $\textsc{ImplK}\langle\alpha,\pi'\rangle$.

4. Translation to Java

4.1 Unpacking Existential Types

The translation of unpack into Java is straightforward, but has an impact on code readability and performance. There are two translation schemes we could use: native unpacking and double-dispatch.

The native unpacking approach replaces:

$$\mathsf{unpack}\langle\vec{\tau}\triangleleft\vec{U}\rangle\ c\langle\vec{T},\vec{\tau}\rangle\ x = M;\ B$$

```
public class N⟨α,β:Ω,π:α ⇒ β⟩ {
  String hd;
  final Ref⟨N⟨α,?,?⟩,β⟩ tl;
  N(String hd, Ref⟨N⟨α,?,?⟩,β⟩ tl) {
    this.hd = hd; this.tl = tl;
  }
  public N⟨α,?,?⟩ cons(String hd) {
    unpack⟨β′:Ω⟩ Ref⟨N⟨α,?,?⟩,β′⟩ tl = Ref.⟨N⟨α,?,?⟩⟩build();
    unpack⟨π′:β′⟩ tl.set(this);
    return new N⟨α,β′,IMPLK⟨α,π′⟩⟩(hd,tl);
  }
  public String head() {
    return this.hd;
  }
  public void setHead(String hd) {
    this.hd = hd;
  }
  public⟨ρ:α⟩ N⟨α,?,?⟩ tail() {
    return this.tl.⟨APPLY⟨π,ρ⟩⟩get();
  }
  public Proof⟨β,?⟩ setTail(N⟨α,?,?⟩ tl) {
    return this.tl.set(tl);
  }
}
public class S⟨α:Ω⟩ ◁ N⟨α,α,ID⟨α⟩⟩ {
  S(String hd, Ref⟨N⟨α,?,?⟩,α⟩ tl) {
    super(hd,tl);
  }
  static public S⟨?⟩ build(String hd) {
    cyclic unpack⟨α:Ω⟩ Ref⟨N⟨α,?,?⟩,α⟩ tl = Ref.⟨N⟨α,?,?⟩⟩build();
    return new S⟨α⟩(hd,tl);
  }
}
```

Figure 16. Example of cyclic linked lists

```
public void run(String[] args) {
  unpack⟨α:Ω⟩ S⟨α⟩ sentinel = S.build(args[0]);
  N⟨α,?,?⟩ node = sentinel;
  for(int i=1; i<args.length; i++) {
    node = node.cons(args[i]);
  }
  unpack⟨π:α⟩ sentinel.setTail(node);
  for(int i=0; i<args.length*3; i++) {
    System.out.println(node.head());
    node = node.⟨π⟩tail();
  }
}
```

Figure 17. Example of using cyclic linked lists

(where M has type $c\langle\vec{T},\vec{?}\rangle$ and B returns type T) by:

```
class Unpacker {
  public⟨τ⃗◁U⃗⟩ T unpacked(c⟨T⃗,τ⃗⟩ x) { B }
};
return new Unpacker().unpacked(M);
```

This approach has the benefit of working with every existential type, but has a problem with code readability: in the original, it is clear that M is evaluated before B, but in the translation, M has been placed after B. Moreover, B is often quite long, and (especially when multiple unpacks are translated) M is separated from x by many lines of code. To improve readability, where possible we adopt a double-dispatch translation, given in Figure 24. For example, the straight-line code:

```
public void run() {
  unpack⟨σ⟩ Foo⟨σ⟩ foo = Foo.build();
  unpack⟨τ⟩ Bar⟨σ,τ⟩ bar = foo.bar();
```

```
interface List⟨α:Ω⟩ {
  public⟨ρ:α⟩ PList⟨?,?⟩ next();
  public⟨ρ:α⟩ PList⟨?,?⟩ prev();
}
class PList⟨α:Ω,π:α⟩ {
  final List⟨α⟩ list;
  PList( List⟨α⟩ list) { this.list = list; }
  public List⟨α⟩ list() { return this.list; }
}
class Root⟨α,β:Ω,π:β⟩ ◁ List⟨α⟩ {
  final Node⟨1,α,β,?,?⟩ next;
  Root(Node⟨1,α,β,?,?⟩ next) { this.next = next; }
  public⟨ρ:α⟩ PList⟨?,?⟩ next() {
    return new PList⟨1∧α∧β,ANDI⟨TRUEI,ANDI⟨ρ,π⟩⟩⟩
      (this.next);
  }
  public⟨ρ:α⟩ PList⟨?,?⟩ prev() {
    return new PList⟨α,ρ⟩(this);
  }
}
abstract class Node⟨α,β,γ,δ:Ω,π:δ⟩ ◁ List⟨α∧β∧γ⟩ {
  Ref⟨List⟨α∧β⟩,β⟩ prev;
  Node(Ref⟨List⟨α∧β⟩,β⟩ prev) { this.prev = prev; }
  abstract Proof⟨γ,?⟩ init();
  public⟨ρ:α∧β∧γ⟩ PList⟨?,?⟩ prev() {
    return new PList⟨α∧β,ANDE1⟨ρ⟩⟩
      (prev.⟨ANDE2⟨ANDE1⟨ρ⟩⟩⟩get());
  }
}
class Nil⟨α,β:Ω⟩ ◁ Node⟨α,β,1,1,TrueI⟩ {
  Nil(Ref⟨List⟨α∧β⟩,β⟩ prev) { super(prev); }
  Proof⟨1,?⟩ init() { return Proof.⟨TrueI⟩build(); }
  public⟨ρ:α∧β∧1⟩ PList⟨?,?⟩ next() {
    return new PList⟨α∧β∧1,ρ⟩(this);
  }
}
class Cons⟨α,β,γ,δ:Ω,π:δ⟩ ◁ Node⟨α,β,γ,δ,π⟩ {
  final Node⟨α∧β,γ,δ,?,?⟩ next;
  Cons(Ref⟨List⟨α∧β⟩,β⟩ prev, Node⟨α∧β,γ,δ,?,?⟩ next) {
    super(prev); this.next = next;
  }
  Proof⟨γ,?⟩ init() { return this.next.prev.set(this); }
  public⟨ρ:α∧β∧γ⟩ PList⟨?,?⟩ next() {
    return new PList⟨α∧β∧γ∧δ,ANDI⟨ρ,π⟩⟩(this.next);
  }
}
```

Figure 18. Example of doubly linked lists

```
    System.out.println(bar);
  }
```

becomes a tangle of double-dispatch methods:

```
public Void run() {
  return Foo.build().unpack(new UnpackFoo⟨Void⟩() {
    public⟨σ⟩ Void unpacked(final Foo⟨σ⟩ foo) {
      return foo.bar().unpack(new UnpackBar⟨Void,σ⟩() {
        public ⟨τ⟩ Void unpacked(final Bar⟨σ,τ⟩ bar) {
          System.out.println(bar); return new Void();
}}); }}); }
```

The resulting code has all the problems of any use of double-dispatch in Java (e.g. the visitor pattern or attaching listeners to GUI components): it requires variables to be declared final, does not play well with imperative features such as for loops, has problems with exception tracking, and has issues with non-linear control flow. Moreover, its execution requires a double-dispatch, when all it is doing at run-time is assigning a value to a variable.

This translation is quite frustrating, and is caused by a design decision in the Java language: unpacking wildcards has been

```
class Test {
  static public⟨α:Ω⟩ Node⟨α,?,?,?,?⟩ build(int i) {
    cyclic unpack⟨β:Ω⟩ Ref⟨List⟨α∧β⟩,β⟩ prev
      = Ref⟨List⟨α∧β⟩⟩.build();
    if (i<0) {
      return new Nil(prev);
    } else {
      unpack⟨γ,δ:Ω⟩ Node⟨α∧β,γ,δ,?,?⟩ next
        = Test.⟨α∧β⟩build(i-1);
      unpack⟨π:δ⟩ next.init();
      return new Cons⟨α,β,γ,δ,π⟩(next,prev);
    }
  }
  static public PList⟨?,?⟩ buildRoot(int i) {
    unpack⟨α,β:Ω⟩ Node⟨1,α,β,?,?⟩ next = Test.⟨1⟩build(i);
    unpack⟨π:β⟩ next.init();
    Root⟨α,β,π⟩ root = new Root⟨α,β,π⟩(next);
    unpack⟨ρ:α⟩ next.prev.set(root);
    return new PList⟨α,ρ⟩(root);
  }
  static public void main(int i) {
    unpack⟨α:Ω,π:α⟩ List⟨α⟩ x = buildRoot(i).list();
    unpack⟨β:Ω,ρ:β⟩ List⟨β⟩ y = x.⟨π⟩next().list();
    unpack⟨γ:Ω,ϕ:γ⟩ List⟨γ⟩ z = y.⟨ρ⟩prev().list();
    ...
  }
}
```

Figure 19. Example of using doubly linked lists

Replace:

```
unpack⟨τ⃗◁U⃗⟩ c⟨T⃗,τ⃗⟩ x = M; B
```

where $M : c\langle\vec{T},\vec{?}\rangle$ and $B : T$ by:

```
return M.unpack(new Unpackc⟨T⃗,T⟩() {
  public⟨τ⃗◁U⃗⟩ T unpacked(c⟨T⃗,τ⃗⟩ x) { B }
});
```

where:

```
class c⟨σ⃗◁V⃗,τ⃗◁W⃗⟩ { ...
  public⟨σ⟩ σ unpack(Unpackc⟨σ⃗,σ⟩ u) {
    return u.⟨τ⃗⟩unpacked(this);
  }
}
interface Unpackc⟨σ⃗◁V⃗,σ⟩ {
  public⟨τ⃗◁W⃗⟩ σ unpacked(c⟨σ⃗,τ⃗⟩ x);
}
```

Figure 20. Translating unpack into Java

merged with method call, causing a big impact on readability of unpacking code. Due to this design decision, method inlining in Java is non-type preserving.

The lack of type-preserving method inlining is not just a problem for mechanized proof assistants, but also impacts source-to-source Java tools. For example, method inlining of static or final methods is not a valid optimization of a Java compiler based on a typed intermediate language (Morrisset 1995): method inlining can only be performed after type erasure, which is problematic.

We strongly support extending the Java language to make method inlining type-preserving.

4.2 Propositional Logic

The translation of propositional formulae and proofs into Java types is inspired by the translation into LF (Harper et al. 1993). The translation is direct, using the Java types declared in Figure 25.

```
void main(...) {
  unpack⟨α ◁ Formula⟨α⟩,β ◁ Formula⟨β⟩⟩
    AstNode⟨α,β⟩ ast = p.parse(file);
  unpack⟨π ◁ Proof⟨α,π⟩⟩ ast.resolveNames(emptyEnv);
  unpack⟨ρ ◁ Proof⟨β,ρ⟩⟩ ast.⟨π⟩typeCheck(emptyTEnv);
  ast.⟨ρ⟩emit(...);
}
interface Parser {
  public AstNode⟨?,?⟩ parse(String file);
}
interface AstNode⟨α ◁ Formula⟨α⟩,β ◁ Formula⟨β⟩⟩ {
  public Proof⟨α,?⟩ resolveNames(Env env);
  public⟨ρ ◁ Proof⟨α,ρ⟩⟩ Proof⟨β,?⟩ typeCheck(TEnv tenv);
  public⟨ϕ ◁ Proof⟨β,ϕ⟩⟩ void emit(...);
}
```

Figure 22. Fähndrich and Leino's AST example after translating propositional logic into Java

For each formula $F : \Omega$, we build a type F satisfying the equation $F \lhd \mathsf{Formula}\langle F\rangle$, for example:

$$\alpha : \Omega \vdash \alpha \wedge \mathbf{1} : \Omega$$

is translated to:

$$\begin{array}{l}\alpha \lhd \mathsf{Formula}\langle\alpha\rangle \\ \quad\vdash \mathsf{And}\langle\alpha,\mathsf{True}\rangle \\ \qquad\lhd \mathsf{Formula}\langle\mathsf{And}\langle\alpha,\mathsf{True}\rangle\rangle\end{array}$$

Similarly, for each proof $P : F$, we build a type P such that $P \lhd \mathsf{Proof}\langle F,P\rangle$, for example:

$$\alpha : \Omega,\ \pi : \alpha \vdash \textsc{AndI}\langle\pi,\textsc{TrueI}\rangle : \alpha \wedge \mathbf{1}$$

is translated to:

$$\begin{array}{l}\alpha \lhd \mathsf{Formula}\langle\alpha\rangle,\ \pi \lhd \mathsf{Proof}\langle\alpha,\pi\rangle \\ \quad\vdash \mathsf{AndI}\langle\alpha,\mathsf{True},\pi,\mathsf{TrueI}\rangle \\ \qquad\lhd \mathsf{Proof}\langle\mathsf{And}\langle\alpha,\mathsf{True}\rangle,\mathsf{AndI}\langle\alpha,\mathsf{True},\pi,\mathsf{TrueI}\rangle\rangle\end{array}$$

In Figure 26, we apply this translation to the AST example given in Figure 14.

We now discuss some design decisions in the translation of propositional logic into Java types.

No user-defined axioms. One question in the design of a machine-assisted proof system is whether to close the system from additional user-defined axioms. In our translation, we do this, for a benefit of soundness with the trade-off of not supporting user-defined extensions to the logic. We do this by making the constructor for Proof package-protected, which stops users from defining their own subclasses of Proof.

As a future extension, we may investigate allowing user-defined logical extensions while still maintaining soundness. A related extension is to support user-defined theorems: currently proofs must be given in full every time they are used, which can become tedious. A theorem mechanism would allow existing proofs to be named and saved for later re-use. The challenge with such a mechanism is that Java does not support typedef or other non-recursive type definitions, and so some additional mechanism is required to ensure that theorems are non-recursive. We leave this for future work.

Proof types are F-bounded. In this translation, proof types are F-bounded (Cardelli et al. 1994) $P \lhd \mathsf{Proof}\langle F,P\rangle$ rather than just bounded $P \lhd \mathsf{Proof}\langle F\rangle$. This is to ensure that users cannot build unsound proof types, for example $\mathsf{Proof}\langle\mathsf{False}\rangle$. An attempt to do so in the current system results in trying to build a type T such that $T \lhd \mathsf{Proof}\langle\mathsf{False},T\rangle$, and the only way to solve such a recursive type equation in Java is by introducing a new class. But since we have package-protected the constructor for Proof, the user cannot define such a new class and must use one of the existing

Replace $F : \Omega$ by $F \lhd$ Formula$\langle F\rangle$, where:

```
public class Formula⟨α ◁ Formula⟨α⟩⟩ {
  Formula() {}
  static public⟨α ◁ Formula⟨α⟩⟩ Formula⟨α⟩ build() { return new Formula⟨α⟩(); }
}
public abstract class True ◁ Formula⟨True⟩ { }
public abstract class False ◁ Formula⟨False⟩ { }
public abstract class And⟨α ◁ Formula⟨α⟩, β ◁ Formula⟨β⟩⟩ ◁ Formula⟨And⟨α,β⟩⟩ { }
public abstract class Or⟨α ◁ Formula⟨α⟩, β ◁ Formula⟨β⟩⟩ ◁ Formula⟨Or⟨α,β⟩⟩ { }
public abstract class Impl⟨α ◁ Formula⟨α⟩, β ◁ Formula⟨β⟩⟩ ◁ Formula⟨Impl⟨α,β⟩⟩ { }
```

Replace $P : F$ by $P \lhd$ Proof$\langle F, P\rangle$, where:

```
public class Proof⟨α ◁ Formula⟨α⟩, π ◁ Proof⟨α,π⟩⟩ {
  Proof() {}
  static public⟨α ◁ Formula⟨α⟩, π ◁ Proof⟨α,π⟩⟩ Proof⟨α,π⟩ build() { return new Proof⟨α,π⟩(); }
}
abstract class Term⟨α ◁ Formula⟨α⟩⟩ ◁ Proof⟨Impl⟨α,True⟩,Term⟨α⟩⟩ { }
abstract class Init⟨α ◁ Formula⟨α⟩⟩ ◁ Proof⟨Impl⟨False,α⟩,Init⟨α⟩⟩ { }
abstract class Id⟨α ◁ Formula⟨α⟩⟩ ◁ Proof⟨Impl⟨α,α⟩,Id⟨α⟩⟩ { }
abstract class Comp⟨α ◁ Formula⟨α⟩, β ◁ Formula⟨β⟩, γ ◁ Formula⟨γ⟩, π ◁ Proof⟨Impl⟨α,β⟩⟩, φ ◁ Proof⟨Impl⟨β,γ⟩⟩⟩
  ◁ Proof⟨Impl⟨α,γ⟩,Comp⟨α,β,γ,π,ρ⟩⟩ { }
⋮
abstract class TrueI
  ◁ Proof⟨True,TrueI⟩ { }
abstract class ImplK⟨α ◁ Formula⟨α⟩, β ◁ Formula⟨β⟩, π ◁ Proof⟨β,π⟩⟩
  ◁ Proof⟨Impl⟨α,β⟩,ImplK⟨α,β,π⟩⟩ { }
abstract class AndI⟨α ◁ Formula⟨α⟩, β ◁ Formula⟨β⟩, π ◁ Proof⟨α,π⟩, ρ ◁ Proof⟨β,ρ⟩⟩
  ◁ Proof⟨And⟨α,β⟩,AndI⟨α,β,π,ρ⟩⟩ { }
⋮
```

Figure 21. Translating propositional logic into Java

subclasses of Proof, none of which allow a construction of a type $T \lhd$ Proof$\langle$False$, T\rangle$.

No run-time penalty. Since Java supports generics through type erasure (Bracha et al. 1998), there is no run-time penalty for carrying proof types at compile time. (There is a run-time penalty caused by the double-dispatch implementation of unpack, but this is a problem with unpacking wildcards, not a problem with proof types.)

An alternative strategy, adopted by the .NET Common Language Runtime and C# generics (Kennedy and Syme 2004) is to build class objects at run-time for each generic instantiation. This would result in a run-time cost for program proof, as the class objects for proof would be constructed. However, there would be benefits to having the proof objects at run-time, in particular they could be serialized across a network, thus providing C# with a built-in proof-carrying code (Necula and Lee 1996) mechanism. We leave the investigation of PCC as future work.

Only propositional logic. Java's type system does not support higher-order types, and so we cannot translate predicate logics or induction schemes using this technique. Thus, we cannot encode any proof of correctness which relies on a proof by induction separate from the recursion carried out by the program itself.

If Java were to be extended with higher-order generic types, then we could make use of them in translating predicate logic and induction, but such an extension is non-trivial (c.f. higher-order modules in ML (Dreyer et al. 2003) and higher-order unification (Huet 2002)).

4.3 Reference API

In the presence of propositional logic, implementing the Ref API is straightforward, and is given in Figure 27. The API is implemented by the class RefImpl$\langle T\rangle$, which implements Ref$\langle T, \mathbf{1}\rangle$, that is the

```
class RefImpl⟨σ⟩ ◁ Ref⟨σ,1⟩ {
  σ x;
  public Proof⟨1,?⟩ set(σ x) {
    this.x = x; return Proof.⟨1,TrueI⟩build();
  }
  public get() {
    return this.x;
  }
}
```

Figure 23. Implementing the Ref API

flag tracking the state of the reference is secretly true all along. We use type abstraction to hide this from any client classes, until set is called, at which point we reveal the proof TrueI which proves $\mathbf{1}$. This is a classic instance of the use of shadow types (Leijen and Meijer 1999).

The only tricky part of the Ref API to implement is cyclic unpack, which is given in Figure 28. This is based on the double-dispatch implementation of acyclic unpack, but uses quadruple-dispatch to allow α to be in scope in T when a RefImpl$\langle T\rangle$ is constructed.

In Appendix 5, we give a proof of the safety of the Ref API in a language based on System $F_{\leq}$ (Cardelli et al. 1994), a small functional language with F-bounded polymorphism. We expect that the object-oriented features of Java do not impact the safety of Ref. The tricky part of the proof is showing that the implementation of cyclic unpack is correct, everything else is relatively straightforward.

4.4 Using Eclipse as an interactive proof assistant

We have used Eclipse to prove correct the examples given in this paper. Our experience is that most of the time and complexity

Replace:

```
cyclic unpack⟨α:Ω⟩ Ref⟨T,α⟩ x = Ref.⟨T⟩build(); B
```

where B returns type U by:

```
return Ref.⟨U⟩build(new PackedRefFunction⟨U⟩() {
  public⟨α:Ω⟩ RefFunction⟨?,U,α⟩ function() {
    return new RefFunction⟨T,U,α⟩() {
      public U apply(Ref⟨T,α⟩ x) { B }
} } });
```

where:

```
public abstract class Ref⟨σ,α:Ω⟩ {
  ...
  static public⟨τ⟩ τ build(PackedRefFunction⟨τ⟩ p) {
    return building(p.⟨1⟩function());
  }
  static private⟨σ,τ⟩ τ building(RefFunction⟨σ,τ,1⟩ p) {
    return f.apply(new RefImpl⟨σ⟩());
  }
}
interface PackedRefFunction⟨τ⟩ {
  public⟨α:Ω⟩ RefFunction⟨?,τ,α⟩ function();
}
interface RefFunction⟨σ,τ,α:Ω⟩ {
  public τ apply(Ref⟨σ,α⟩ ref);
}
```

Figure 24. Translating cyclic unpack into Java

is spent fighting Java's mechanism for unpacking wildcard types, and that the proofs (once they have been generated by hand) are relatively straightforward to mechanize.

Eclipse is required for this effort, as Sun's Java compiler has a bug (Sun Bug ID #6729401) with F-bounded polymorphism, due to be fixed in Java 7.0. The presence of this bug suggests that the F-bounded feature of Java generics has not been heavily used.

5. Safety of the Ref API

We now address the safety of the reference API. The language System F_{Ref} is given in Figure 30. It includes the core of the Ref API, including the implementation of cyclic unpack. It also includes an explicit treatment of existential types: we note that this is necessary, as the implementation of existentials using De Morgan dualized universals is unsafe. If we define:

$$\exists\langle\sigma\lhd T\rangle\, U \stackrel{\mathsf{def}}{=} \forall\langle\tau\rangle\,(\forall\langle\sigma\lhd T\rangle\,(U\to\tau))\to\tau$$

then we speculate that a malicious user can write unsafe code. This source of unsafety is caused by the continuation-passing style (CPS) used to implement existentials: if we could limit continuations to be used linearly, then the translation would be safe, but there is no such linear restriction in either System $F_{\leq}$ or Java. This problem would be a potential source of vulnerabilities in a system such as C# which does not support existential types natively.

Write $E(H)$ for the typing environment extracted from H:

$$E(\varepsilon)=\varepsilon$$
$$E(H,\sigma\lhd T)=E(H),\sigma\lhd T$$
$$E(H,x\!:\!T=v)=E(H),x\!:\!T$$

Define a configuration $(H;M)$ to be *well-formed* whenever: (a) Any uses of $\mathsf{building}\langle U\rangle N$ in M are in evaluation contexts. (b) Any uses of $\mathsf{building}\langle U\rangle N$ in M are of the form $\mathsf{building}\langle\sigma\rangle N$ where $\sigma\lhd\mathsf{Formula}\langle\sigma\rangle$ in H. (c) Any uses of $\mathsf{building}\langle U\rangle N$ in M have no $x\!:\!\mathsf{Ref}\langle T,U\rangle=v$ in H. (d) Any uses of $\tau\lhd\mathsf{Proof}\langle U,\tau\rangle$ in H have $x\!:\!\mathsf{Ref}\langle T,U\rangle=v$ in H and $E(H)\vdash v\!:\!T$. (e) Any uses of $x\!:\!\mathsf{Ref}\langle T,U\rangle=v$ in H are of the form $x\!:\!\mathsf{Ref}\langle T,\sigma\rangle=v$, where $\sigma\lhd\mathsf{Formula}\langle\sigma\rangle$ in H.

The only surprising condition here is a, which requires that $\mathsf{building}\langle T\rangle N$ only be used in evaluation contexts. This property is trivially true of any code that does not contain $\mathsf{building}\langle T\rangle N$, and we note that in the Java implementation, this method is made private to the Ref class. Moreover, this property is preserved by reduction, as there are no reduction rules which move code from an evaluation context into a non-evaluation context. This invariant would be broken if callcc were added to Java, and indeed callcc causes the implementation of cyclic unpack to be unsafe. We can now state a subject reduction property, from which it is direct to show safety.

Proposition 1. *If $E(H)\vdash M\!:\!T$ and $(H;M)$ is well-formed and $(H;M)\to(H';M')$, then $E(H')\vdash M'\!:\!T$ and $(H';M')$ is well-formed.*

PROOF. Follows the proof of subject reduction for System $F_{\leq}$ (Cardelli et al. 1994), except the cases for the Ref API, which are straightforward applications of the well-formedness conditions. □

Corollary 2. *If $\vdash M\!:\!T$ and $(\varepsilon;M)\to^{*}(H;N)$ then N is* null*-free.*□

6. Related work

The notion that logics can be coded in types is a key observation of the Logical Frameworks (Harper et al. 1993) approach. The observation that phantom types (Leijen and Meijer 1999) can be used to track object properties is well-known in the Haskell community, e.g., see the Haskell wiki, although its use in Java has not been as wide-spread. Duerig (2008) used phantom types in Java to provide a type-safe builder but did not provide a full coding of propositional logic, or address cyclic structures.

There has been extensive research on typestates in object-oriented languages, e.g.,Strom (1983), Strom and Yemini (1986), Chambers (1993), Kuncak et al. (2002), Fickle (Drossopoulou et al. 2002) and Vault (DeLine and Fähndrich 2001, 2004) and Gay et al. (2009). We have already discussed the relationship to Fähndrich and Leino (2003), Fähndrich and Xia (2007) and Qi and Myers (2009).

In terms of tool-based specification and validation of contracts, the Singularity operating system (Fähndrich et al. 2006) statically validates session-types for first-class linear channels in dynamic communication networks. Plural (Bierhoff et al. 2009) uses dataflow analysis to statically validate usage protocols. Plural interfaces can be annotated with specifications that combine typestates with access permissions. In comparison to Plural, our methods are arguably more general and minimal; on the other hand, much much more work is needed on our infrastructure to build up to the scale and effectiveness of Plural.

7. Conclusion

We have shown that the off-the-shelf Java type system can usefully encode non-trivial proofs of correctness using propositional Hoare pre- and post-conditions. We have demonstrated the power of the method by providing an implementation of object deserialization whose correctness is proven by construction using Java types. To our knowledge, our system is the first that can treat this important example.

The patterns that arise in our code suggest changes to the Java language to facilitate easy expression of our programming idioms, notably an in-line facility for unpacking objects with wildcard type. Such a change to the language would have other benefits, notably it would allow static method calls in Java to be inlined, an important optimization for source-to-source Java tools.

Programs:

$$\begin{array}{l} x,y,z \in \text{program variables} \\ v,w ::= x \mid \mathsf{null} \mid \lambda(x{:}T)M \mid \Lambda\langle\sigma \triangleleft U\rangle M \mid \mathsf{pack}\langle T\rangle M \\ M,N ::= v \mid M\,N \mid M\langle T\rangle \mid \mathsf{unpack}\,M\,N \mid \mathsf{build}\,M \mid \mathsf{building}\langle T\rangle M \mid \mathsf{set}\,M\,N \mid \mathsf{get}\langle T\rangle M \end{array}$$

Types:

$$\begin{array}{l} \sigma,\tau,\upsilon \in \text{type variables} \\ T,U ::= \sigma \mid \top \mid T \to U \mid \forall\langle\sigma \triangleleft U\rangle T \mid \exists\langle\sigma \triangleleft U\rangle T \mid \mathsf{Formula}\langle T\rangle \mid \mathsf{Proof}\langle T,U\rangle \mid \mathsf{Ref}\langle T,U\rangle \end{array}$$

Typing environments:

$$E ::= \varepsilon \mid E,\sigma \triangleleft T \mid E,x{:}T$$

Typing rules:

$$\frac{E \ni (x{:}\sigma)}{E \vdash x{:}T} \qquad \frac{E,x{:}T \vdash M{:}U}{E \vdash \lambda(x{:}T)M{:}T \to U} \qquad \frac{\sigma \notin dom(E), fv(T) \subseteq dom(E) \cup \{\sigma\} \quad E,\sigma \triangleleft T \vdash M{:}U}{E \vdash \Lambda\langle\sigma \triangleleft T\rangle M{:}\forall\langle\sigma \triangleleft T\rangle U} \qquad \frac{E \vdash V \triangleleft T\{^V/_\sigma\} \quad E \vdash M{:}U\{^V/_\sigma\}}{E \vdash \mathsf{pack}\langle V\rangle M{:}\exists\langle\sigma \triangleleft T\rangle U}$$

$$\frac{E \vdash M{:}T \to U \quad E \vdash N{:}T}{E \vdash M\,N{:}U} \qquad \frac{E \vdash M{:}\forall\langle\sigma \triangleleft T\rangle U \quad E \vdash V \triangleleft T\{^V/_\sigma\}}{E \vdash M\langle V\rangle{:}U} \qquad \frac{E \vdash M{:}\exists\langle\sigma \triangleleft T\rangle U \quad E \vdash N{:}\forall\langle\sigma \triangleleft T\rangle U \to V}{E \vdash \mathsf{unpack}\,M\,N{:}V}$$

$$\frac{E \vdash M{:}\forall\langle\sigma \triangleleft \mathsf{Formula}\langle\sigma\rangle\rangle \exists\langle\tau\rangle \mathsf{Ref}\langle\tau,\sigma\rangle \to T}{E \vdash \mathsf{build}\,M{:}T} \qquad \frac{E \vdash U \triangleleft \mathsf{Formula}\langle U\rangle \quad E \vdash M{:}\exists\langle\tau\rangle \mathsf{Ref}\langle\tau,U\rangle \to T}{E \vdash \mathsf{building}\langle U\rangle M{:}T}$$

$$\frac{E \vdash M{:}\mathsf{Ref}\langle T,U\rangle \quad E \vdash N{:}T}{E \vdash \mathsf{set}\,M\,N{:}\exists\langle\sigma \triangleleft \mathsf{Proof}\langle U,\sigma\rangle\rangle \top} \qquad \frac{E \vdash V \triangleleft \mathsf{Proof}\langle U,V\rangle \quad E \vdash M{:}\mathsf{Ref}\langle T,U\rangle}{E \vdash \mathsf{get}\langle V\rangle M{:}T}$$

Subtyping rules:

$$\frac{E \ni (\sigma \triangleleft T)}{E \vdash \sigma \triangleleft \sigma} \qquad \frac{E \ni (\sigma \triangleleft T) \quad E \vdash T \triangleleft U}{E \vdash \sigma \triangleleft U} \qquad \frac{}{E \vdash \top \triangleleft \top} \qquad \frac{E \vdash T \triangleleft T}{E \vdash T \triangleleft \top} \qquad \frac{E \vdash T \triangleleft T' \quad E \vdash U \triangleleft U'}{E \vdash (T' \to U) \triangleleft (T \to U')}$$

$$\frac{E,\sigma \triangleleft T \vdash T \triangleleft T \quad E,\sigma \triangleleft T \vdash U \triangleleft U'}{E \vdash (\forall\langle\sigma \triangleleft T\rangle U) \triangleleft (\forall\langle\sigma \triangleleft T\rangle U')} \qquad \frac{E,\sigma \triangleleft T \vdash T \triangleleft T \quad E,\sigma \triangleleft T \vdash U \triangleleft U'}{E \vdash (\exists\langle\sigma \triangleleft T\rangle U) \triangleleft (\exists\langle\sigma \triangleleft T\rangle U')}$$

$$\frac{E \vdash T \triangleleft \mathsf{Formula}\langle T\rangle}{E \vdash (\mathsf{Formula}\langle T\rangle) \triangleleft (\mathsf{Formula}\langle T\rangle)} \qquad \frac{E \vdash T \triangleleft \mathsf{Formula}\langle T\rangle \quad E \vdash U \triangleleft \mathsf{Proof}\langle T,U\rangle}{E \vdash (\mathsf{Proof}\langle T,U\rangle) \triangleleft (\mathsf{Proof}\langle T,U\rangle)} \qquad \frac{E \vdash T \triangleleft T \quad E \vdash U \triangleleft \mathsf{Formula}\langle U\rangle}{E \vdash (\mathsf{Ref}\langle T,U\rangle) \triangleleft (\mathsf{Ref}\langle T,U\rangle)}$$

Subsumption:

$$\frac{E \vdash M{:}T \quad E \vdash T \triangleleft U}{E \vdash M{:}U}$$

Evaluation contexts:

$$\mathscr{E} ::= \cdot \mid \mathscr{E}\,N \mid v\,\mathscr{E} \mid \mathscr{E}\langle T\rangle \mid \mathsf{unpack}\,\mathscr{E}\,N \mid \mathsf{unpack}\,v\,\mathscr{E} \mid \mathsf{build}\,\mathscr{E} \mid \mathsf{building}\langle T\rangle\,\mathscr{E} \mid \mathsf{set}\,\mathscr{E}\,N \mid \mathsf{set}\,v\,\mathscr{E} \mid \mathsf{get}\langle T\rangle\,\mathscr{E}$$

Heaps:

$$H ::= \varepsilon \mid H,\sigma \triangleleft \mathsf{Formula}\langle\sigma\rangle \mid H,\sigma \triangleleft \mathsf{Proof}\langle T,\sigma\rangle \mid H,x{:}\mathsf{Ref}\langle T,U\rangle = v$$

Operational semantics:

$$\begin{array}{rl} (H;\mathscr{E}[(\lambda(x{:}T)M)\,w]) & \to (H;\mathscr{E}[M\{^w/_x\}]) \\ (H;\mathscr{E}[(\Lambda\langle\sigma \triangleleft T\rangle M)\langle U\rangle]) & \to (H;\mathscr{E}[M\{^U/_\sigma\}]) \\ (H;\mathscr{E}[\mathsf{unpack}\,(\mathsf{pack}\langle T\rangle M)\,w]) & \to (H;\mathscr{E}[w\langle T\rangle M]) \\ (H;\mathscr{E}[\mathsf{build}\,v]) & \to (H,\sigma \triangleleft \mathsf{Formula}\langle\sigma\rangle;\mathscr{E}[\mathsf{building}\langle\sigma\rangle\,(v\langle\sigma\rangle)]) \\ (H;\mathscr{E}[\mathsf{building}\langle U\rangle\,(\mathsf{pack}\langle T\rangle M)]) & \to (H,x{:}\mathsf{Ref}\langle T,U\rangle = \mathsf{null};\mathscr{E}[M\,x]) \\ (H,x{:}\mathsf{Ref}\langle T,U\rangle = v,H';\mathscr{E}[\mathsf{set}\,x\,w]) & \to (H,x{:}\mathsf{Ref}\langle T,U\rangle = w,H',\sigma \triangleleft \mathsf{Proof}\langle U,\sigma\rangle;\mathscr{E}[\mathsf{pack}\langle\sigma\rangle\,w]) \\ (H,x{:}\mathsf{Ref}\langle T,U\rangle = v,H';\mathscr{E}[\mathsf{get}\langle V\rangle\,x]) & \to (H,x{:}\mathsf{Ref}\langle T,U\rangle = v,H';\mathscr{E}[v]) \end{array}$$

Figure 25. System F_{Ref}

References

A. W. Appel. Foundational proof-carrying code. In *Proc. IEEE Logic in Computer Science*, 2001.

T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. ACM Principals of Programming Languages*, pages 1–3, 2002.

K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *Proc. European Conf. Object-Oriented Programming*, 2009. To appear.

G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. Int. Conf. Object Oriented Programming, Systems, Languages and Applications*, pages 183–200, 1998.

L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of system f with subtyping. *Inf. Comput.*, 109(1-2):4–56, 1994.

C. Chambers. Predicate classes. In *Proc. European Conf. Object-Oriented Programming*, LNCS, pages 268–296, 1993.

T. Coquand and G. P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

K. Crary and J. G. Morrisett. Type structure for low-level programming languages. In *Proc. Int. Colloq. Automata, Languages and Programming*, LNCS, pages 40–54, 1999.

R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM Programming Language Design and Implementation*, pages 59–69, 2001.

R. DeLine and M. Fähndrich. Typestates for objects. In *Proc. European Conf. Object-Oriented Programming*, LNCS, pages 465–490, 2004.

D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *Proc. ACM Principals of Programming Languages*, pages 236–249, New York, NY, USA, 2003.

S. Drossopoulou, F. Damiani, D. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle II. *ACM Trans. Programming Languages and Systems*, pages 153–191, 2002.

M. Duerig. Type-safe builder pattern in Java, 2008. http://michid.wordpress.com/.

M. Fähndrich and K. R. M. Leino. Heap monotonic typestates. In *Proc. Int. Workshop on Alias Confinement and Ownership*, 2003.

M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *Proc. Int. Conf. Object Oriented Programming, Systems, Languages and Applications*, pages 337–350, 2007.

M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, pages 177–190, 2006.

S. J. Gay, A. Ravara, and V. T. Vasconcelos. Dynamic interfaces. In *Proc. Foundations Object-Oriented Languages*, 2009.

R. Harper, F. Honsell, and G. D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

C. A. R. Hoare. An axiomatic basis for computer programming. *C. ACM*, 12(10):576–580, 1969.

K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. European Symp. Programming Languages and Systems*, LNCS, pages 122–138, 1998.

G. P. Huet. Higher order unification 30 years later. In *TPHOLs*, LNCS, pages 3–12, 2002.

A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for Java and GJ. *ACM Trans. Programming Languages and Systems*, 23(3):396–450, 2001.

A. Kennedy and D. Syme. Transposing F to C#: expressivity of parametric polymorphism in an object-oriented language. *Concurrency - Practice and Experience*, 16(7):707–733, 2004.

V. Kuncak, P. Lam, and M. C. Rinard. Role analysis. In *Proc. ACM Principals of Programming Languages*, pages 17–32, 2002.

D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proc. USENIX Conf. Domain-Specific Languages*, pages 109–122, 1999.

K. R. M. Leino. Specifying and verifying programs in Spec#. In *Ershov Memorial Conference*, LNCS, page 20, 2006.

G. Lindstrom. Functional programming and the logical variable. In *Proc. ACM Principals of Programming Languages*, pages 266–280, 1985.

J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. *ACM Trans. Programming Languages and Systems*, 10 (3):470–502, 1988.

G. Morrisset. *Compiling with types*. PhD thesis, Carnegie Mellon University, 1995.

G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI*, pages 229–243, 1996.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS. Springer, 2002.

F. Pfenning. Elf: A meta-language for deductive systems (system descrition). In *Proc. Conf. Automated Deduction*, LNCS, pages 811–815, 1994.

F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proc. Conf. Automated Deduction*, LNCS, pages 202–206, 1999.

X. Qi and A. C. Myers. Masked types for sound object initialization. In *Proc. ACM Principals of Programming Languages*, pages 53–65, 2009.

R. E. Strom. Mechanisms for compile-time enforcement of security. In *Proc. ACM Principals of Programming Languages*, pages 276–284, 1983.

R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proc. Int. Conf. Parallel Architectures and Languages*, LNCS, pages 398–413, 1994.

M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. In *Proc. ACM Symp. Applied Computing*, pages 1289–1296, 2004.