A Transformational Perspective into the Core of an Abstract Class Loader for the SSP

<table>
<thead>
<tr>
<th>Journal:</th>
<th>Transactions on Embedded Computing Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manuscript ID:</td>
<td>draft</td>
</tr>
<tr>
<td>Manuscript Type:</td>
<td>Paper</td>
</tr>
<tr>
<td>Date Submitted by the Author:</td>
<td>n/a</td>
</tr>
<tr>
<td>Complete List of Authors:</td>
<td>Winter, Victor; University of Nebraska at Omaha, Computer Science Beranek, Jason; University of Nebraska at Omaha, Computer Science Fraij, Fares; University of Texas at El Paso, Computer Science Roach, Steve; University of Texas at El Paso, Computer Science Wickstrom, Greg; Sandia National Laboratories, Department of Surety Electronics and Software</td>
</tr>
<tr>
<td>Computing Classification Systems:</td>
<td>D.3.1, F.3.1 , F.3.2 , C.1.0</td>
</tr>
</tbody>
</table>
A Transformational Perspective into the Core of an Abstract Class Loader for the SSP *

Victor L. Winter†
University of Nebraska at Omaha
Department of Computer Science
vwinter@mail.unomaha.edu

Jason Beranek‡
University of Nebraska at Omaha
Department of Computer Science
jason.beranek@gmail.com

Fares Fraij
University of Texas at El Paso
Department of Computer Science
fares@cs.utep.edu

Steve Roach
University of Texas at El Paso
Department of Computer Science
sroach@cs.utep.edu

Greg Wickstrom
Sandia National Laboratories
Department of Surety Electronics and Software
glwicks@sandia.gov

January 20, 2006

Abstract

The SSP is a hardware implementation of a subset of the JVM for use in high consequence embedded applications. In this context, a majority of the activities belonging to class loading, as it is defined in the specification of the JVM, can be performed statically. Static class loading has the net result of dramatically simplifying the design of the SSP as well as increasing its performance. Due to the high consequence nature of its applications, strong evidence must be provided that all aspects of the SSP have been implemented correctly. This includes the class loader. This article explores the possibility of formally verifying a class loader for the SSP implemented in the strategic programming language TL. Specifically, an implementation of the core activities of an abstract class loader is presented and its verification in ACL2 is considered.

1 Introduction

At Sandia National Laboratories, a subset of the Java Virtual Machine (JVM) has been developed in hardware for use in high-consequence embedded applications. The implementation is called the Sandia Secure Processor (SSP) [12, 22] and supports a subset of Java bytecodes as its native instruction set.

This paper has three objectives: (1) to informally describe the core functionality of the class loader for the SSP, (2) to demonstrate how the abstract functionality of this core class loader can be implemented using higher-order strategic programming techniques, and (3) to discuss how the correctness of the class loader can be formally verified. The paper is organized as follows. We begin with an introduction to the goals of the SSP class loader. Section 2 gives an overview of the Java class file structure and restricts our attention to the subset of class files impacted by the SSP class loader core. This section also gives an informal description of the core activities of the SSP class loader. Section 3 gives an overview of the

---

*This work was in part supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy.
†Victor Winter was also partially supported by NSF grant number CCR-0209187.
‡Jason Beranek was also partially supported by NSF grant number CCR-0209187.
higher-order strategic programming language TL. Section 4 presents and discusses a TL implementation of an abstract class loader core. Section 5 describes preliminary efforts at verifying and validating the transformations.

1.1 The ROM Images Executed by the SSP

An application program for the SSP is called a ROM image and consists of a collection of class file images stored on a Read-Only Memory (ROM). The image of a class file in the ROM contains (1) a ROM constant pool and (2) a method table and methods section. For the purposes of this article, there are two major differences between ROM constant pools and the constant pools found in Java class files. The first major difference is that in ROM constant pools, class, field, and method entries are represented in terms of absolute addresses or offset addresses together with additional data that, broadly speaking, provides type information describing the entry. In contrast, in Java class files such entries are represented as encodings of symbolic references. In this article, we refer to the aggregation of one or more constant pool indexes as an encoding of a symbolic reference. The second major difference is that, unlike the constant pools in Java class files, ROM constant pools do not contain any name_and_type or Utf8 entries. Constant pools in the ROM are limited to the following entries: constant integer, constant long, class, method, and field entries which are separated into two distinct types: (1) static field entries and (2) instance field entries. Among the method entries, in this article, we restrict our consideration only to virtual methods. That is, methods that are invoked within an application program using the invokevirtual bytecode. In this article, we do not consider static or special methods whose invocations are respectively achieved through the invokestatic and invokespecial bytecodes. This restriction to virtual methods is essentially without loss of generality and serves the purpose of simplifying our discussion. The one exception being an anomalous case involving dynamic binding of a method that has been invoked using the invokespecial bytecode [9, 18].

In a ROM image, the methods section of class files have also been modified. In particular, within the bytecode of a method, constant pool indexes (i.e., encodings of symbolic references) have been replaced with offset addresses into the ROM constant pool. These are the major differences between class files as produced by a Java compiler and as they appear on the ROM.

Due to the similarity between the JVM and the SSP, the primary goal of the class loader for the SSP can be simply stated:

---

**The Primary Goal of the SSP Class Loader:**

*Resolve encoded symbolic references (indexes) in class files to absolute addresses or offset addresses.*

---

Though the focus on this paper is on the primary goal as stated above, we would like to mention that there are numerous secondary goals a class loader for the SSP must satisfy. These secondary goals include the construction of various kinds of type data associated with constant pool entries, padding method bodies so methods begin and end on word boundaries, and so on.

Within the SSP, absolute addresses belong to one of two domains. These domains model the address spaces of the ROM and heap respectively. The first domain, $D_{ROM}$, describes the location of bytes within the ROM image. Addresses belonging to $D_{ROM}$ are used to describe the location of immutable structures associated with a class such as the location of a ROM constant pool, the starting address of a method,
<table>
<thead>
<tr>
<th>Context where Index Occurs</th>
<th>Interpretation of Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>within a bytecode in the body of a method</td>
<td>ROM constant pool offset address where the type of the constant pool entry can be inferred from the bytecode</td>
</tr>
<tr>
<td>this_class</td>
<td>absolute address in ROM</td>
</tr>
<tr>
<td>super_class</td>
<td>absolute address in ROM</td>
</tr>
<tr>
<td>class entry in ROM constant pool</td>
<td>absolute address in ROM</td>
</tr>
<tr>
<td>static field entry in ROM constant pool</td>
<td>absolute address in heap</td>
</tr>
<tr>
<td>static field entry in fields section of class file</td>
<td>object offset address</td>
</tr>
<tr>
<td>instance field entry in ROM constant pool</td>
<td>object offset address</td>
</tr>
<tr>
<td>virtual method entry in ROM constant pool</td>
<td>absolute address of method in ROM</td>
</tr>
</tbody>
</table>

Figure 1: Resolution of indexes (aka. encoded symbolic references)

and the method table associated with a class. The second domain, $D_{HEAP}$, describes the location of bytes within the SSP’s heap memory. From the perspective of the class loader, absolute addresses belonging to $D_{HEAP}$ are used solely to describe the location of static fields. All other assignments of absolute addresses in $D_{HEAP}$ occur during runtime. For example, during execution bytecodes such as `new` are responsible for heap memory allocation. These allocations yield absolute addresses in $D_{HEAP}$ belonging to objects (i.e., object references).

In contrast to absolute addresses, offset addresses are used to describe the location of data in a relative fashion. Adding an offset address to an appropriate absolute address (such as the start of a constant pool) yields an absolute address that describes the location of a particular data item within an absolute address space (e.g., the address of a constant pool entry). The design of the SSP makes use of three types of offset addresses: constant pool offsets, method table offsets, and object offsets. Constant pool offset addresses are values belonging to the domain $D_{CP}$ and describe offsets relative to the start of a ROM constant pool. Semantically speaking, constant pool offset addresses are used to describe the location of ROM constant pool entries. Method table offset addresses are values belonging to the domain $D_{MT}$ and describe offsets relative to the start of a method table residing in the ROM. Semantically speaking, method table offset addresses are used to describe the location of entries in a method table. Method table entries contain information needed to execute a particular virtual method (which resides in the ROM). This information includes the starting address of the virtual method, the absolute address of the constant pool associated with this virtual method, and type information defining the number of local variables and parameters used by the method. And lastly, object offset addresses are values belonging to $D_{OBJ}$ and describe offsets relative to the start of an object (i.e., an instance of a class). Semantically speaking, object offset addresses are used to describe the location of data within an object. This data consists primarily of the object’s instance fields. However, the first data entry of an object is an absolute address belonging to $D_{ROM}$ describing the location of the class from which this object is an instance (this is needed when a method for this object is invoked).

Given an index, the type of address to which it should be resolved can be uniquely determined from the context in which it is used. Figure 1 lists the contexts that must be considered.
Within an application program, classes may be ordered to form inheritance hierarchies through the Java 
"extends" directive. The subtype relation resulting from inheritance hierarchies impose constraints on how 
object offsets and method table offsets must be calculated. These constraints as well as others can be 
expressed in the form of properties that a resolved collection of encoded symbolic references must possess 
in order to be correct. A non-exhaustive list of these properties follows.

- **Consistent-Offset:** Within the scope of an inheritance hierarchy, all instance fields must be resolved 
to unique object offsets in a manner that is consistent with upcasting. *Rationale:* This property 
ensures that two instance fields are not mapped to the same offset and that identical instance fields 
reside in the same position in every object in which they occur. *Example:* Let *myA* be an instance of 
class *A* and *myB* be an instance of class *B*. If *B* is a subtype of *A*, then offset consistency requires 
that an instance field *x* in *myB* that is inherited from *A* be given the same offset address that this 
field is given in *myA*. Offset consistency assures that the field *x* declared in class *A* can be uniformly 
accessed in expressions like `((A)myB).x` and `myA.x` using the same offset.

- **Consistent-Address:** Within an application, every static field must consistently be resolved to an 
absolute address that is unique within the heap. *Rationale:* This requirement ensures that two static 
fields are not mapped to the same absolute address within the heap.

- **Consistent-MT:** Virtual method invocations must be referred to indirectly via an offset to a method 
table (mt). Furthermore, within an inheritance hierarchy all method tables must be consistent with 
respect to the positioning of method table entries. *Rationale:* Positional table entry consistency 
among method tables within an inheritance hierarchy ensures that proper method invocation will 
result in the presence of upcasting. *Example:* A method table provides a means to indirectly reference 
a (virtual) method. Each class has an associated method table. Conceptually, one can think of a 
method table entry to contain the starting address of the method that should be executed when the 
entry is referenced. The reason for the method table is that the semantics of Java’s subtype 
system requires that a method declaration destructively overwrite an inherited declaration and that 
this overwrite remain unchanged in the presence of upcasting (which is distinctly different from the 
treatment of inherited instance fields). In contrast to instance fields, where copies of instance fields 
are individually maintained for each object, only a single definition for each method is maintained. 
This is done for reasons of efficiency.

Let us assume that *A* and *B* are classes and that *B* is a subtype of *A*. Method invocation consistency 
requires that a virtual method *f* that is declared in *A* and redeclared in *B* be given the same offset 
address *i* in the method tables constructed for their respective classes. In this case, a reference to the 
index *i* in the method table associated with class *A* will cause the version of *f* declared in class *A* 
to be executed. In contrast, a reference to the index *i* in the method table associated with the class *B* 
will cause the version of *f* declared in class *B* to be executed. The effect is that the same method 
table index *i* can be used in an expression like `myA.f()` to execute the version of *f* declared in class 
*A* and in `((A)myB).f()` to execute the version of *f* declared in class *B*.

- **Non-Overlapping-Addresses:** The values of primitive types supported by the SSP must be 
mapped to memory regions that are sufficiently large to hold all legal values of that type. For 
example, an integer field must be mapped to a memory region that is at least 32-bits wide. *Ration-
ale:* This property prevents instance fields from being packed so tightly within an object that their 
memory spaces overlap. This property also prevents static fields from being packed so tightly in the 
heap that their memory spaces overlap. Note that simply requiring that instance fields have unique 
offsets is not sufficient to ensure this property.
• **Address Units**

- Constant pool offset addresses must be in word units (i.e., 32-bit quantities) in the range \(0 \leq cp_{offset} < 2^{16}\). The only range exception being that of the `ldc` bytecode which has the range \(0 \leq cp_{offset} < 2^8\). *Rationale:* In the JVM, a number of bytecodes have 8 or 16-bit constant pool indexes as operands. In the SSP, these bytecodes have corresponding 8 or 16-bit constant pool offsets as operands. Class loading for the SSP requires that abstract constant pool indexes used by bytecodes be replaced by corresponding offset addresses and that the number of bits (e.g., 8 or 16) used to denote an index/offset remain constant. Unfortunately, under these constraints, the number of distinct constant pool entries that can be denoted using abstract indexing is larger than the number of entries that can be denoted using offset addresses. The reason for this is that constant pool entries related to methods occupy one (abstract) position in a class file constant pool and occupy two words in the ROM constant pool. In the worst case, a ROM constant pool (consisting entirely of method entries) could contain only half as many addressable entries as its Java counterpart. As a result, one can guarantee that the addressing scheme of the SSP will be able to handle all Java constant pools having \(2^{15}\) elements or less, with \(2^{15}\) being a conservative (guaranteed achievable) lower bound. The class loader must flag as an error all constant pools whose element count exceed this threshold.

- Method table offsets must be in word units in the range \(0 \leq mt_{offset} < 2^{16}\). *Rationale:* This design decision enables the addressing scheme of the SSP to handle Java classes whose total number of virtual methods (both declared and inherited) is less than or equal to \(2^{16}\). The class loader must flag as an error all classes whose virtual method count exceeds this threshold.

- Absolute addresses in the heap must be in byte units in the range \(0 \leq address < 2^{24}\). *Rationale:* This enables a Java constant pool entry containing a symbolic reference to a static field to be resolved to a one-word ROM constant pool entry describing the type (8-bits) and absolute heap address (24-bits) of the static field.

The list given above is not meant to be complete, but rather to give the reader a feeling for the necessary kinds of properties that must be satisfied in order for the resolution performed by a class loader to be considered correct. Taking a more rigorous approach, correctness properties of the kind just described can be formalized and expressed in terms of a formula \(C\) in first-order logic. Abstractly speaking, this formula defines properties and relationships between (1) encoded symbolic references and addresses, and (2) addresses within an address space.

In addition to correctness properties, the resolution of encoded symbolic references must also satisfy a number of efficiency-based constraints. A number of these constraints are related to optimization of memory usage and access. However, fault-tolerance, safety, and security constraints are also possible. A non-exhaustive list of these constraints follows.

• **Spatial-Efficiency:**

- Instance fields should be packed as tightly as possible in objects and static fields should be tightly packed in the heap. *Rationale:* This is a necessary condition to assure that heap memory is efficiently utilized.

- Object offset addresses must be in byte units (i.e., 8-bit quantities). *Rationale:* This design decision enables sequences of short, char, byte, and Boolean fields to be closely packed within an object.

• **Temporal-Efficiency:**
64-bit values (i.e., longs) should not span 32-bit boundaries within the heap or ROM address space. For example, a long may be stored at byte address 0x0000 or 0x0004 but should not be stored at byte address 0x0002 because retrieving a long value stored in this manner would require 3 memory fetches instead of 2. Rationale: This is a necessary condition to assure that the SSP executes efficiently.

32-bit values such as integers and references should not span 32-bit boundaries within the heap or ROM address space. For example, an integer value may be stored at byte address 0x0000 or 0x0004 but should not be stored at byte address 0x0002 because retrieving a value stored in this manner would require 2 memory fetches instead of 1. Rationale: This is a necessary condition to assure that the SSP executes efficiently.

16-bit values such as shorts and chars should not span 16-bit boundaries within the heap or ROM address space. Rationale: This is a necessary condition to assure that the SSP executes efficiently.

This list of hardware constraints is not meant to be exhaustive, but rather it demonstrates the nature of the constraints imposed by the SSP hardware. Taking a more rigorous approach, hardware constraints can be expressed in terms of a formula $H$ in first-order logic. Abstractly speaking, this formula defines properties and relationships between (1) encoded symbolic references and addresses, and (2) addresses within an address space.

Given the formula $CH \equiv C \land H$ an interpretation $I$ is a mapping from encoded symbolic references in $CH$ to the domain $D \equiv D_{HEAP} \cup D_{ROM} \cup D_{CP} \cup D_{MT} \cup D_{OBJ}$ of absolute addresses and offset addresses. In this setting, resolution for the SSP can be defined as a function that constructs an interpretation $I$ over $D$ satisfying the formula $CH$. From an operational perspective, the interpretation $I$ is an assignment of address values to indexes (i.e., encoded symbolic references). In this article, the terms resolution and resolve are used to describe the processes behind the construction of such assignments.

We are now in a position to give a high-level definition of the class loader core that is the focus of this article.

**Definition 1** The core of the class loader for the SSP is an interpretation $I_{core}$ mapping indexes to (1) offset addresses in $D_{MT} \cup D_{OBJ}$, and (2) absolute addresses in the space $D_{HEAP}$ such that $I_{core}$ satisfies $CH$.

Notice that the definition of the core, as we have defined it, excludes the offset address space $D_{CP}$ and the absolute address space $D_{ROM}$. The reason for excluding these address spaces from consideration by the core is mainly for simplicity and space considerations. For example, calculating offsets for the constant pool entries is essentially the same as calculating offsets for method table entries. Since method table construction is included in the core the notion of calculating offsets for tables is already covered in the core. The calculation of absolute addresses for immutable objects in the ROM is also abstractly similar to table offset calculation and is also therefore omitted.

### 2 An Overview of Class Loading for the SSP

Figure 2 gives a top-level view of the components contained in a Java class file. Components such as $cp\_info$, $interfaces$, $field\_info$, and $method\_info$ have highly complex structures and contain a number of subcomponents which are not shown in Figure 2. For more information about the structure of these components as well as class files in general see [9, 18].
For Peer Review

The hex value 0xCAFEBABE indicating that this file is a Java class file.

The minor version of the compiler that produced this class.

The major version of the compiler that produced this class.

The number of entries in the constant pool.

The constant pool.

Modifiers associated with this class or interface (e.g., private, final, abstract, etc.).

A constant pool index that when resolved yields the name of the class.

The value 0 or a constant pool index that when resolved yields the name of the super class.

The number of direct super interfaces of the class or interface.

The interfaces implemented by the class.

The number of fields explicitly declared in the class.

The fields explicitly declared in the class.

The number of methods explicitly declared in the class.

The methods explicitly declared in the class.

The number of attributes of the class.

The attributes of the class.

Figure 2: The components of a Java class file

In this paper we restrict our attention to class loading activities for the SSP pertaining to the subset of Java class files shown in Figure 3. In particular, we will look at (1) index resolution, (2) static field address calculation, (3) offset address calculation, (4) method table construction, and (5) inter-class absolute address and offset address distribution. Collectively, we will refer to this set of class loading activities as the class loader core. In the following sections we informally describe each of these activities in detail.

2.1 Index Resolution

In Java class files, references to field, method, this-class, and super-class information are abstractly encoded as indexes into the class file’s constant pool. Within a constant pool, such indexes directly or indirectly denote information that is ultimately expressed symbolically in terms of Utf8 strings. We will refer to the symbolic information denoted by an index as the abstract meaning otherwise known as the symbolic reference of that index. We use the term index resolution to denote the process of constructing the abstract

<table>
<thead>
<tr>
<th>Java Structure Name</th>
<th>SSP Structure Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp_info</td>
<td>cp</td>
<td>the constant pool</td>
</tr>
<tr>
<td>this_class</td>
<td>this</td>
<td>the name of this class</td>
</tr>
<tr>
<td>super_class</td>
<td>super</td>
<td>the name of the parent class</td>
</tr>
<tr>
<td>field_info</td>
<td>sfields &amp; ifields</td>
<td>fields are separated into a list of static fields sfields and a list of instance fields ifields</td>
</tr>
<tr>
<td>method_info</td>
<td>mt &amp; methods</td>
<td>methods are associated with a method table mt</td>
</tr>
</tbody>
</table>

Figure 3: The subset of the Java class file structure relevant to the class loader core
Consider the constant pool fragment shown in Figure 4. In this example, much detail has been abstracted away from the structure of the constant pool, and only the salient portions remain. For example, Java constant pool entries have tags that indicate the type of the entry, and Utf8 entries are realized in terms of a list of bytes. However, regardless of the presence or absence of such detail the concept of index resolution remains the same.

Given the abstracted constant pool in the Figure 4, we ask “What is the symbolic reference of the index 1 with respect to this constant pool”? To determine the meaning of an index, the chain of index/value pairs are followed until a collection of Utf8 values are reached. The concatenation of these Utf8 values forms the symbolic reference. The value at index 1 in the constant pool contains an entry of type CONSTANT_Fieldref_info, which indicates that the index 1 denotes an encoding of a symbolic reference to a field. When resolved this symbolic reference will consist of the name of the class in which the field is declared, the name of the field, and a descriptor describing the type of the field (e.g., integer, short, long).

The contents of the CONSTANT_Fieldref_info entry at position 1 consists of two indexes. The Java class file specification requires that the first index be interpreted as a class index denoting the class in which the field is declared. The second index should be interpreted as a name_index which provides information about the name and type of the field. Thus the first step in the index resolution process is: $1 \Rightarrow 2 \ 3$. Next, index resolution is individually performed on the indexes 2 and 3. The constant pool entry having index 2 contains a name index 4. Thus, the current state of the resolution is: $1 \Rightarrow 2 \ 3 \Rightarrow 4 \ 3$. Now, the constant pool entry having index 4 contains an entry of type CONSTANT_Utf8_info whose value is “B”. Thus, “B” is the name of the class in which the field denoted by index 1 is declared.

The second part of the value located at index 1 in the constant pool is the index 3. The constant pool entry at index 3 contains a name_index 5 that in this context denotes the name of the field, and a descriptor_index 6 denoting the type of the field. Thus, $3 \Rightarrow 5 \ 6$. The contents of the constant pool at index 5 tells us that the name of the field is the Utf8 value “y”. The contents at index 6 indicates that the field is of type “I”. Thus, the symbolic reference of index 1 with respect to the given constant pool is $B\ y\ I$, that is, index 1 denotes the field $y$ of type integer that is declared in class $B$. The index resolution of 1 can be summarized by the following rewrite sequence:

$$1 \Rightarrow 2 \ 3 \Rightarrow 4 \ 3 \Rightarrow B \ 3 \Rightarrow B \ 5 \ 6 \Rightarrow B \ y \ 6 \Rightarrow B \ y \ I$$

In general, index resolution concerns itself with the replacement of indexes with their symbolic references. The scope of this type of rewriting is limited to individual class files and is based solely on information found in the constant pool for the class. Complete details on the structure of constant pools can be found in the literature [9, 18].

<table>
<thead>
<tr>
<th>Index</th>
<th>Entry Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CONSTANT_Fieldref_info</td>
<td>2 3</td>
</tr>
<tr>
<td>2</td>
<td>CONSTANT_Class_info</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>CONSTANT_NameAndType_info</td>
<td>5 6</td>
</tr>
<tr>
<td>4</td>
<td>CONSTANT_Utf8_info</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>CONSTANT_Utf8_info</td>
<td>y</td>
</tr>
<tr>
<td>6</td>
<td>CONSTANT_UTF8_info</td>
<td>I</td>
</tr>
</tbody>
</table>

Figure 4: A constant pool description of the integer field $B.y$
2.2 Static Field Address Calculation

The goal of static field address calculation is to assign a unique absolute address to each static field within a Java application. Since static fields are associated with a class rather than an object (i.e., an instance of a class), their number remains constant during runtime. Consider the Java application program shown in Figure 5 consisting of the classes B, C, and D.

```
class B { ... static int x, b2; ... }
class C { ... static int c1, x, c3; ... }
class D { ... static int d1; ... }
```

Figure 5: Classes and their static fields

<table>
<thead>
<tr>
<th>Static Field</th>
<th>Absolute Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.x</td>
<td>0x0000</td>
</tr>
<tr>
<td>B.b2</td>
<td>0x0004</td>
</tr>
<tr>
<td>C.c1</td>
<td>0x0008</td>
</tr>
<tr>
<td>C.x</td>
<td>0x000C</td>
</tr>
<tr>
<td>C.c3</td>
<td>0x0010</td>
</tr>
<tr>
<td>D.d1</td>
<td>0x0014</td>
</tr>
</tbody>
</table>

Figure 6: Mapping static fields to absolute heap addresses

If we assume a byte-addressable memory in the range 0x0000...0xFFFF, then the static fields in the application could be assigned to the absolute addresses as shown in Figure 6. From a semantic perspective this class loader activity can be seen as providing an interpretation (i.e., a concrete meaning) for the symbolic references of static fields.

2.3 Instance Field Offset Calculation

Instance fields, in contrast to static fields, are associated with objects rather than classes. Each object contains its own copy of every instance field declared in its corresponding class plus all of the instance fields inherited from its super class. Figure 7 shows the instance field declarations in a number of Java class fragments.

Figures 8 and 9 show possible offset calculations respectively for the instance fields of class C and E. We would like to point out that the class loader does not actually construct objects of the kind shown in Figures 8 and 9. That functionality is entrusted to the microcode implementation of the bytecode `new`. Instead, the purpose of the class loader is to construct an interpretation (i.e., assign a concrete semantics) in the form of a mapping from symbolic references of instance fields to object offset addresses in a manner that is consistent with the formula $C H$ mentioned in Section 1.2. The interpretation provided by the object offset addresses must be sound with respect to all dynamic uses of objects. For example, in Java objects can be dynamically upcast and downcast. Thus all objects within an inheritance hierarchy must have a consistent interpretation for all shared (i.e., inherited) instance fields.

2.4 Method Table Construction

Encoded symbolic references to methods must ultimately be resolvable to the ROM address where the bytecode for the method resides. However, this resolution is complicated by the interplay of two aspects
of Java’s subtype system. First, within an inheritance hierarchy multiple definitions for a single method may occur. Second, Java’s `upcast` operation provides a mechanism by which the type of an object may be cast to that of any ancestor belonging to the inheritance chain.

Within such a fluid inheritance hierarchy, methods must utilize symbolic method references in a consistent fashion. Consistency here means that a symbolic reference to a method such as `B.foo` must have the ability to denote any redefinition of `foo` in every descendent of `B`. This capability is needed because during runtime, an object that is an instance of a descendent of `B` may be upcast to `B` after which the method `foo` could be invoked on the upcast object. In such a case, the constant pool of the class in which `B.foo` is invoked will contain a symbolic reference to `B.foo`. However, because multiple definitions of `foo` may exist throughout the inheritance hierarchy this symbolic reference cannot be directly resolved to a single absolute address. A standard solution to this problem is to construct a method table for each class [9, 18]. This method table forms a layer of indirection that enables methods to be referenced in a consistent fashion. The entries in a method table contain data necessary to execute the bytecodes corresponding to the implementation of a method as seen from the perspective of a particular class. For example, data in a method table may include the address of the first bytecode in the method as well as the address of the method’s corresponding constant pool. Symbolic references to methods such as `B.foo` are now resolved to offsets into the method table. Of course in order for the indirection provided by the method table to solve our problem, all classes that inherit or redefine `foo` must store data related to `foo` in the same relative position (i.e., offset) in their method table.
Figure 10 shows a class B and a class C which extends B. Note that although the method foo is redeclared in C, the method tables for B and C, shown in Figure 11, place the data for foo in the same relative location in their method tables. As a result, the symbolic reference to B.foo can be uniformly resolved to the method table offset 0x0008. Which method table this offset is applied to depends on the class from which an object is derived (e.g., B or C in Figure 10). For example, the expression \((B)(new\ C()).\text{foo()})\) will access the data at offset 0x0008 in C’s method table while the expression \((new\ B()).\text{foo()})\) will access the data at offset 0x0008 in B’s method table.

```
<table>
<thead>
<tr>
<th>Offset</th>
<th>Method Table for B</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>data for B.f1(I)I</td>
<td>declared</td>
</tr>
<tr>
<td>0x0008</td>
<td>data for B.foo()I</td>
<td>declared</td>
</tr>
<tr>
<td>0x0010</td>
<td>data for B.g1()I</td>
<td>declared</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Method Table for C</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>data for B.f1(I)I</td>
<td>inherited</td>
</tr>
<tr>
<td>0x0008</td>
<td>data for C.foo()I</td>
<td>redeclared</td>
</tr>
<tr>
<td>0x0010</td>
<td>data for B.g1()I</td>
<td>inherited</td>
</tr>
<tr>
<td>0x0018</td>
<td>data for C.g2()I</td>
<td>declared</td>
</tr>
</tbody>
</table>
```

Figure 10: Classes and their virtual methods

Figure 11: Method table construction

### 2.5 Inter-class Absolute Address and Offset Address Distribution

Inter-class distribution is concerned with the distribution of absolute addresses and offset addresses between the various class files that make up a Java application. Within a single class file, symbolic references to locally declared fields and methods can be resolved to absolute addresses (for static fields), object offsets (for instance fields), and method table offsets (for methods). However, within a Java application, a class file \(X\) may have a symbolic reference to fields and methods that have been declared in another class file \(Y\). These references, external to \(X\), show up as symbolic references in the constant pool of \(X\) and must be resolved using information originating from the class \(Y\). Specifically, absolute address and offset address information must be distributed from the class in which the declarations occur (e.g., \(Y\)) to all classes referencing these declarations (e.g., \(X\)).

Figure 12 shows two class files B and C. The constant pool of class file B contains symbolic (non-local) references to a field and method declared in (local to) C, and the constant pool of C contains symbolic references to a field and method declared in B.

Figure 13 shows the class files B and C after the inter-class distribution phase. Note that field address and method table offset data has been propagated between the classes.
### Class B

<table>
<thead>
<tr>
<th>Index</th>
<th>Constant Pool Entries</th>
<th>Fields</th>
<th>Method Table Offset</th>
<th>Method Table Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B.x I local</td>
<td>B.w I 0x0000</td>
<td>0x00</td>
<td>B.g (I)I ...</td>
</tr>
<tr>
<td>2</td>
<td>C.x I non-local</td>
<td>B.x I 0x0004</td>
<td>0x01</td>
<td>B.f (I)I ...</td>
</tr>
<tr>
<td>3</td>
<td>B.f (I) local</td>
<td>B.y I 0x0008</td>
<td>0x02</td>
<td>B.foo (I)I ...</td>
</tr>
<tr>
<td>4</td>
<td>C.g (I) non-local</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### Class C

<table>
<thead>
<tr>
<th>Index</th>
<th>Constant Pool Entries</th>
<th>Fields</th>
<th>Method Table Offset</th>
<th>Method Table Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B.x I non-local</td>
<td>C.x I 0x0000</td>
<td>0x00</td>
<td>C.bar (I)I ...</td>
</tr>
<tr>
<td>2</td>
<td>C.x I local</td>
<td>C.y I 0x0004</td>
<td>0x01</td>
<td>C.f (I)I ...</td>
</tr>
<tr>
<td>3</td>
<td>B.f (I) 0x01</td>
<td>B.y I 0x0008</td>
<td>0x02</td>
<td>C.g (I)I ...</td>
</tr>
<tr>
<td>4</td>
<td>C.g (I) local</td>
<td>C.k I 0x000C</td>
<td>0x03</td>
<td>C.h (I)I ...</td>
</tr>
</tbody>
</table>

Figure 12: An examples of two classes having external symbolic references to fields and methods

<table>
<thead>
<tr>
<th>Index</th>
<th>Constant Pool Entries</th>
<th>Fields</th>
<th>Method Table Offset</th>
<th>Method Table Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B.x I 0x0004</td>
<td>B.w I 0x0000</td>
<td>0x00</td>
<td>B.g (I)I ...</td>
</tr>
<tr>
<td>2</td>
<td>C.x I 0x0000</td>
<td>B.x I 0x0004</td>
<td>0x01</td>
<td>B.f (I)I ...</td>
</tr>
<tr>
<td>3</td>
<td>B.f (I) 0x01</td>
<td>B.y I 0x0008</td>
<td>0x02</td>
<td>B.foo (I)I ...</td>
</tr>
<tr>
<td>4</td>
<td>C.g (I) 0x02</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 13: Inter-class distribution of field offset/address and method table offsets

### 3 An overview of TL

The specification of the JVM enables class loading to occur dynamically (e.g., during runtime). However, from the perspective of class loading, the SSP can be considered a *closed system* because all the class files in an application must be stored on the ROM prior to execution. The closed nature of the SSP’s execution environment enables the class loading activities of the SSP to be performed statically, prior to execution. Under these conditions, the functionality of a class loader is well suited to a transformation-oriented implementation [26]. This section gives an overview of the strategic programming language TL which we will use to implement an abstract version of the class loader core. We would like to mention that the design presented here represents an abstraction derived from an actual class loader for the SSP that has been implemented in TL.

In general, strategic programming systems are rewriting systems that have been extended with a variety of constructs enabling explicit control over the application of rewrite rules. Such control is typically necessary when dealing with rule sets that are non-confluent and/or non-terminating. When constructed appropriately, rewriting systems and strategic programming systems have properties that make them well-suited to formal verification. Our research goal into the development of TL has been to provide a framework whereby strategic programs can be written to realize complex functionality in a manner which nevertheless remains within the grasp of formal automated verification.
3.1 The Basic Constructs of TL

TL is a higher-order strategic programming language [25, 24, 23]. In TL, conditional rewrite rules can be combined to form expressions called strategies. Strategies define controlled sequences of rewrites and can be applied to tree structures to produce other tree structures. Thus, a strategy can be viewed as a function that rewrites or transforms one tree into another.

The primary constructs and abstractions in a first-order strategic programming language typically include:

1. patterns – A pattern is a notation for describing the tree structures that are being manipulated. This notation typically includes variables, potentially typed, that are quantified over a variety of tree structures.

2. rewrite rules – A rewrite rule is a construct for specifying that one pattern is to be replaced by another pattern. In a strategic system, rewrite rules are also considered to be a degenerative form of strategy. In other words, a rewrite rule is always considered to be a strategy, but a strategy is not always (just) a rewrite rule.

3. conditions – A condition is a construct associated with a rewrite rule that restricts its application.

4. combinators – A combinator is an operator (generally unary or binary) that can be used to compose one or more strategies into a new strategy.

5. generic traversals – A generic traversal can be thought of as a curried function parameterized on a strategy \( s \) and a tree \( t \). As the name suggests, a generic traversal will traverse its input tree structure \( t \) and apply its input strategy \( s \) at one or more points along the traversal. Two common traversals are a top-down left-to-right traversal which TL denotes by the symbol \( TDL \), and a bottom-up left-to-right traversal which TL denotes by the symbol \( BUL \). From a computational perspective, a \( TDL \) traversal can be understood as corresponding roughly to non-strict (outside-in) evaluation while the traversal \( BUL \) corresponds roughly to strict (inside-out) evaluation.

Another traversal that is also very useful is \( FIX_{TDL} \) which recursively traverses a tree in a \( TDL \) fashion until a fixed point is reached.

6. strategies – In its purest sense, a first-order strategy can be characterized as any function that transforms one tree into another tree. Structurally speaking however, a strategy is an expression composed of rewrite rules, combinators, and generic traversals.

7. labels – A strategy can be bound to a label for the purposes of abstraction.

In addition to the first-order constructs mentioned above, TL also supports the following higher-order constructs:

1. higher-order strategies – A higher-order strategy is a strategy that when applied to a tree will return a strategy rather than a tree. For example, when applied to a tree, a second-order strategy \( s^2 \) will yield a first-order strategy \( s^1 \). In general, the application of a strategy of order \( n \) to a term \( t \) will yield a strategy of order \( n - 1 \). For the purposes of uniformity, in this framework, a tree is considered to be a strategy of order 0.

2. higher-order generic traversals – A higher-order generic traversal can be thought of as a curried function that is parameterized on a higher-order strategy \( s^n \) (where \( n \) denotes the order of the strategy) and a tree \( t \). Its application to \( t \) yields a strategy of order \( n - 1 \). Higher-order generic traversals are
very useful for creating specific strategies that are “tuned” the data occurring within a particular
tree. In other words, they are a mechanism for dynamically generating strategies customized for a
given input tree.

In the following sections we briefly describe each of the constructs mentioned above.

### 3.2 Tree/Term Notation

Let \( G = (N, T, P, S) \) denote a context-free grammar where \( N \) is the set of nonterminals, \( T \) is the set of terminals, \( P \) is the set of productions, and \( S \) is the start symbol. Given an arbitrary symbol \( B \in N \) and a string of symbols \( \alpha = X_1X_2...X_m \) where for all \( 1 \leq i \leq m : X_i \in N \cup T \), we say \( B \) derives \( \alpha \) if the productions in \( P \) can be used to expand \( B \) to \( \alpha \). Traditionally, the expression \( B \xrightarrow{*} \alpha \) is used to denote that \( B \) can derive \( \alpha \) in zero or more expansion steps. Similarly, one can write \( B \xrightarrow{+} \alpha \) to denote a derivation consisting of one or more expansion steps.

In TL, we write \( B[\alpha'] \) to denote an instance of the derivation \( B \xrightarrow{*} \alpha \) whose resulting value is a parse tree having \( B \) as its root symbol. In TL, expressions of the form \( B[\alpha'] \) are referred to as parse expressions. In the parse expression \( B[\alpha'] \) the string \( \alpha' \) is an instance of \( \alpha \) because nonterminal symbols in \( \alpha' \) are constrained through the use of subscripts. Subscripted nonterminal symbols are referred to as schema variables or simply variables for short. TL also considers a schema variable (e.g., \( B_i \)) to be a parse expression in its own right. Within a rewrite rule all occurrences of schema variables having the same subscript denote the same variable.

Figure 14 shows a BNF grammar fragment describing a small portion of an imperative language. In the context of this grammar, the parse expressions \( stmt[\ id_1 = 5 \] and \( stmt[\ id_2 = 5 \] both describe instances of the derivation \( stmt \xrightarrow{*} id = 5 \).

```plaintext
prog ::= stmt_list
stmt_list ::= stmt ";" stmt_list | stmt
stmt ::= assign | cond | ...
assign ::= lvalue "=" expr
...
lvalue ::= id
expr ::= int | int + int
...
```

Figure 14: A concrete syntax fragment

When the specific structure of a parse expression is unimportant the parse expression will be denoted by variables of the form \( t, t_1, ... \) or variables of the form \( tree, tree_1, tree_2, \) and so on. Parse expressions containing no schema variables are called ground and parse expressions containing one or more schema variables are called non-ground. And finally, within the context of rewriting or strategic programming, trees as described here can and generally are viewed as terms. When the distinction is unimportant, we will refer to trees and terms interchangeably.

### 3.3 Conditional Rewrite Rules

Figure 15 defines the structure of TL rules in terms of an extended-BNF. The meta-symbols of this BNF are \([,]\) and \(::=\). Symbols enclosed in square brackets denote optional portions of a production. For example,
rewriting rules in TL have the form \( \text{lhs} \rightarrow \text{rhs} \) and may have optional labels and conditions associated with them.

Based on the tree grammar given in Figure 14, we can write the following condition-less rewrite rule:

\[
\text{stmt}_{1} \rightarrow \text{stmt}_{1}
\]

This rule states that an assignment statement having the expression 4 + 1 as its right-hand side should be rewritten to an assignment statement having the constant 5 as its right-hand side.

### 3.3.1 Conditions

The conditional portion of a rule is a match expression consisting of one or more match equations. The symbol \( \ll \), adapted from the \( \rho \)-calculus [5], is used to denote first-order matching modulo an empty equational theory. Let \( t_2 \) denote a ground tree and let \( t_1 \) denote a parse expression which may contain one or more schema variables. A match equation is denoted \( t_1 \ll t_2 \) or equivalently \( t_2 \gg t_1 \). A match equation is a Boolean valued operation that produces a substitution \( \sigma \) as a by-product. A substitution \( \sigma \) binding schema variables to ground parse expressions is a solution to a match equation consisting of one or more match equations. Match expressions may be constructed using the standard Boolean operators: \( \wedge, \vee, \neg \). A substitution \( \sigma \) is a solution to a match expression \( m \) iff \( \sigma(m) \) evaluates to true using the standard semantics for Boolean operators.

Based on the tree grammar given in Figure 14, we can write the following conditional rewrite rule:

\[
r_2 : \text{stmt}_1 \rightarrow \text{stmt}_1 [id_1 = 5] \text{ if } \text{stmt}_1 \gg \text{stmt}_1 [id_1 = 4 + 1]
\]

The conditional rule \( r_2 \) says that a statement \( \text{stmt}_1 \) should be rewritten to the statement \( \text{stmt}_1 [id_1 = 5] \) only if the condition \( \text{stmt}_1 \gg \text{stmt}_1 [id_1 = 4 + 1] \) is satisfied. In other words, if \( \text{stmt}_1 \) is an assignment statement whose right-hand side is the expression 4 + 1. Note that the application of \( r_2 \) to a term \( t \) will implicitly bind \( \text{stmt}_1 \) thereby making it a ground term.

The rule \( r_3 \) shown below gives a trivial example of a rule condition consisting of two match equations.

\[
r_3 : \text{stmt}_1 \rightarrow \text{stmt}_1 [id_1 = 5] \text{ if } \text{stmt}_1 \gg \text{stmt}_1 [id_1 = expr_1] \wedge expr_1 \gg expr_1 [4 + 1]
\]

The conditional rule \( r_3 \) says that a statement \( \text{stmt}_1 \) should be rewritten to the statement \( \text{stmt}_1 [id_1 = 5] \) only if \( \text{stmt}_1 \) is an assignment statement whose right-hand side is \( expr_1 \) and \( expr_1 \) is 4 + 1.

### 3.3.2 Rule Application

The application of a conditional rewrite rule \( r \) to a tree \( t \) is expressed as \( r(t) \) where \( r \) is either an abstraction of a rewrite rule (i.e., a label) or an anonymous rule value e.g., \( \text{lhs} \rightarrow \text{rhs} \). We adopt a curried notation in the style of ML where application is a left-associative implicit operator and parentheses are used to
3.4 Combinators

TL provides three binary combinators enabling (1) the sequential composition of strategies which is denoted by the semi-colon symbol, (2) the left-biased composition of strategies which is denoted by the symbol $\leftarrow\rightarrow$, and (3) the right-biased composition of strategies which is denoted by the symbol $\rightarrow\leftarrow$.

Let $s_1$ and $s_2$ denote two first-order strategies. The strategy $s_1; s_2$ denotes the sequential composition of $s_1$ and $s_2$. When applied to a term $t$ the strategy $s_1; s_2$ will first apply $s_1$ to $t$ yielding $t'$ and then apply $s_2$ to $t'$ yielding $t''$.

The strategy $s_1 \leftarrow\rightarrow s_2$ denotes the left-biased composition of $s_1$ and $s_2$. When applied to a term $t$ the strategy $s_1 \leftarrow\rightarrow s_2$ will first try to apply $s_1$ to $t$. If the application $s_1 t$ succeeds, and produces $t'$ as its result, then $t'$ is returned as the final result and $s_2$ is not used. However, if the application $s_1 t$ fails, then the result of $s_2 t$ is returned as the final result.

The strategy $s_1 \rightarrow\leftarrow s_2$ denotes the right-biased composition of $s_1$ and $s_2$. Semantically speaking, $s_1 \rightarrow\leftarrow s_2$ is equivalent to $s_2 \leftarrow\rightarrow s_1$.

In addition to the three binary combinators previously mentioned, TL also provides the unary combinators transient and hide. The semantics of these combinators are described in Section 4.2.

3.5 Generic First-Order Traversals

TL supports a number of generic first-order traversals including: TDL, FIX_TDL, and TDL_B. The traversal TDL accepts a first-order strategy $s$ and a tree $t$ as input and performs a single top-down left-to-right traversal over $t$ applying $s$ to every sub-tree encountered. The traversal FIX_TDL accepts a first-order strategy $s$ and a tree $t_0$ and performs the evaluation TDL $s t_0$ yielding $t_1$. If one or more rewrites occurred during the evaluation of TDL $s t_0$, then FIX_TDL will perform the evaluation TDL $s t_1$ yielding $t_2$. Additional evaluations of the form TDL $s t_i$ will continue until a tree $t_j$ is reached such that the evaluation TDL $s t_j$ completes without a single rewrite being performed on $t_j$. If such a tree $t_j$ is found, then this is the value of the evaluation of FIX_TDL $s t_0$. Otherwise the evaluation of FIX_TDL $s t_0$ does not terminate.

The generic first-order traversal TDL_B is described in Section 4.4.

3.6 Higher-Order Rules and Strategies

In TL, a conditional rewrite rule of order $n + 1$ has the form:

$$
\text{label} : \text{lhs} \rightarrow s^n \text{ if condition}
$$

where $s^n$ is a strategic expression whose evaluation yields a strategy of order $n$. As was the case with first-order rules, the label and conditional portion of higher-order rules are optional.
The combinators in TL can be applied to first-order as well as higher-order strategies. For example, let $s^n_1$ and $s^n_2$ denote two order $n$ strategies. The expressions $s^n_1; s^n_2$ and $s^n_1 \leftarrow s^n_2$ respectively denote the sequential and left-biased conditional composition of $s^n_1$ and $s^n_2$. In this article, we will restrict our attention to the composition of strategies having the same order (e.g., a composition of the form $s^n_i \leftarrow s^n_j$ where $i \neq j$ is prohibited).

### 3.7 Higher-Order Generic Traversals

TL supports a number of generic higher-order traversals including: `seq_tdl` and `lcond_tdl`. The traversal `seq_tdl` accepts a higher-order strategy $s^n$ and a tree $t_1$ as its input and performs a top-down left-to-right traversal of $t_1$ applying the strategy $s^n$ to each (sub)tree encountered. Let $t_1, t_2, ..., t_m$ denote the trees encountered during the traversal of $t_1$. Let $s^{n-1}_i$ denote the strategy obtained from applying $s^n$ to the tree $t_i$. Given these assumptions, the evaluation of $\text{seq} \_\text{tdl} \ s^n \ t_1$ will produce the strategy

$$s^{n-1}_1; s^{n-1}_2; ...; s^{n-1}_m$$

Similarly, the evaluation of $\text{lcond} \_\text{tdl} \ s^n \ t_1$ will produce the strategy

$$s^{n-1}_1 \leftarrow s^{n-1}_2 \leftarrow ... \leftarrow s^{n-1}_m$$

Having described the basics of TL we are now in a position to discuss a strategic implementation of the class loader core.

### 4 A Strategic Implementation of the Class Loader Core

In this section we look at a strategic solution, written in TL, to an abstract version of the class loader core as defined in Section 1.2. In this abstract example, the structure of a class file has been greatly simplified and in several places its structure has even been altered. In spite of these changes, we nevertheless make the claim that, when seen from a strategic perspective, the abstractions below still contain the essence of the class loader core. This claim is based on our experience in successfully developing a strategic-based implementation of an actual class loader for the SSP. This implementation has been developed using the HATS system, an IDE for strategic programming supporting a dialect of TL. The HATS system is freely available [6].

Noteworthy characteristics of the class file structure described in Figure 16 include:

- Additional terminal symbols have been added to enable the class file structure to be described by a context-free grammar.
  - Constant pool entries have been enclosed in parenthesis.
  - The `@` symbol is used to tag absolute addresses belonging to $D_{HEAP}$.
  - The `:` symbol is used to tag object offset addresses belonging to $D_{OBJ}$.
  - The `#` symbol is used to tag method table offset addresses belonging to $D_{MT}$.
  - The `−` symbol is used to denote the value of an uninitialized address.
  - The terminal `ident` is a token whose regular expression defines the set of identifiers.
  - The terminal `integer` is a token whose regular expression defines the set of integers.

- Explicit indexes have been given to the entries in the constant pool.
- Index resolution will take place within parse tree structures whose root symbol is the nonterminal \( d \). This is possible because the grammar has been designed in such a way that \( d \) can derive both an index and an id (i.e., a Utf8). Thus, this grammar design allows parse trees to be constructed where the data component of a constant pool entry can be either an index (i.e., an encoding of a symbolic reference) or a sequence of identifiers (i.e., a symbolic reference). For a more detailed discussion of this see Section 4.1.

- The structure of a class file has been extended with a children list which is initially empty.

- Fields have been partitioned into a list of static fields and a list of instance fields.

- The methods section consists of a method table and a list of methods.

- Methods consist only of a method name. In particular, they do not contain a descriptor and they do not contain any bytecodes.

- All fields are tacitly assumed to be of type integer.

- Fields are denoted by a single index whose resolution yields the name of the field and the class in which it is declared.

- Address and offset units are words.

In the context of these constraints and alterations we present a strategic implementation of the class loader core. Figure 17 shows three abstract class files before class loading, and Figure 18 shows the same three class files after class loading.

### 4.1 Index Resolution in TL

As described in Section 2.1, the goal of index resolution is to rewrite constant pool indexes into their symbolic references. Conceptually speaking, our strategic approach to index resolution is as follows. For every constant pool entry whose structure matches the parse expression \( c_{\text{entry}}[(\text{index}_1, d_1)] \) we create a rewrite rule of the form \( \text{index}_1 \rightarrow d_1 \). Note that not all constant pool entries will have this form. Once these rules have been generated all that remains to be done is to exhaustively apply them to all data indexes within the class file – where an index is considered to be a data index if it is derived from the nonterminal \( d \). Thus, it is only in this structural context that index resolution should be performed. This constraint is captured by the parse expression \( d[\text{index}_1] \), yielding a rewrite rule whose form is \( d[\text{index}_1] \rightarrow d_1 \).

We would like to point out that the structure of the constant pool entries \( c_{\text{entry}}[(\text{index}_1, d_1)] \) has been designed in such a way that \( \text{index}_1 \), the positional denotation of the entry, is not a data index. Data indexes occur in a variety of places throughout a class file such as within the description of fields, within the description of methods including their bytecodes, and within the data portion of constant pool entries.

Figure 19 gives an implementation of index resolution in TL. The behavior of the strategy \( \text{index resolution} \) is as follows. When applied to a class file structure \( \text{class}_0 \) the strategy \( \text{index resolution} \) will first evaluate the strategic expression \( \text{seq}_\text{tdl cp normalize} \text{class}_0 \). Within this expression, the strategy \( \text{seq}_\text{tdl} \) is a higher-order generic traversal that will traverse a term in a top-down left-to-right (tdl) fashion. In this case, the term being traversed is \( \text{class}_0 \). The fact that \( \text{seq}_\text{tdl} \) is higher-order means that it expects to apply a higher-order strategy to the sub-terms of the term it is traversing. In this case, the higher-order strategy being applied is \( \text{cp normalize} \), a second-order strategy that converts a constant pool entry of the form \( c_{\text{entry}}[(\text{index}_1, d_1)] \) into a first-order rewrite rule of the form: \( d[\text{index}_1] \rightarrow d_1 \). When applied to the entries of the constant pool of \( \text{class}_0 \) a number of instances of the rule \( d[\text{index}_1] \rightarrow d_1 \) will be generated. These rule instances are then composed by \( \text{seq}_\text{tdl} \) using TL’s sequential composition operator.
app ::= app class | ε
class ::= { extends info children }
extends ::= class_id parent_id
children ::= children class | ε
info ::= cp fields methods
class_id ::= id
parent_id ::= id
cp ::= cp c_entry | ε
c_entry ::= ( index , data )
fields ::= statics instance
statics ::= statics sfield | ε
sfield ::= data @ addr
instance ::= instance ifield | ε
ifield ::= data : addr
methods ::= mt , method_list
mt ::= mt_entry mt | ε
mt_entry ::= key # addr
method_list ::= m_entry method_list | ε
m_entry ::= data ( )
data ::= key | d | key address_type addr
key ::= d . d
d ::= id | index
address_type ::= @ | # | :
index ::= integer
addr ::= integer
id ::= ident

Figure 16: An extended-BNF grammar describing a simplified application in terms of a list of class files
Recall that this composition is part of the semantics of seq_tdl – which sequentially composes the results generated from its tdl traversal. The resulting first-order strategy is of the form \( r_1; r_2; ... r_n \) where \( r_i \) is the instance of \( d_i[ index_1 ] \rightarrow d_1 \) corresponding to the \( i^{th} \) constant pool entry whose structure can be matched by \( c.entry[index_1, d_1] \). This first-order strategy is then exhaustively applied to class0 in a top-down left-to-right fashion by the first order generic traversal FIX_TDL. The result is that all data indexes are rewritten to their symbolic references.

### 4.2 Static Field Address Calculation in TL

As described in Section 2.2, the goal of static field address calculation is to assign each static field in an application a unique absolute address taken from the address space \( D_{HEAP} \). In TL, this can be accomplished with a strategy that makes use of the strategic combinators transient [25, 24] and hide [23]. Both of these combinators are unique to TL.

The transient combinator is a unary combinator used to create a strategy that can be applied \textit{at most once}. This “at most once” property is the hallmark of the transient combinator. The application of the transient combinator to a strategy \( s \) is written \( \text{transient}(s) \) and the resulting strategy is referred to as a \textit{transient strategy}. Formally stated, the application of a transient strategy is restricted by reducing it to SKIP – a strategic constant in TL whose application always fails. However, in practical terms, it is often easier to think of a transient strategy as simply being removed once it has applied successfully.

For example, let \( s_1 \) denote the strategy \( \text{transient}(addr[ 0 ] \rightarrow addr[ 1 ] \) ). The strategic expression \( TDL \ s_1 \ class_0 \) states that the term \( class_0 \) should be traversed in a top-down left-to-right fashion and that the current value of strategy \( s_1 \) should be applied to every term encountered during this traversal. Due to the use of the \textit{transient} combinator in \( s_1 \), the strategic expression \( TDL \ s_1 \ class_0 \) will have the effect of

| This class | 1 |
| Super class | 19 |
| CP | (1, 1.3)(3, x1)(4, 1.5)(5, x2)(6, 1.8)(7, 1.9)(8, a1)(9, a2)(10, 11.3) |
| Static fields | 2@- 4@- |
| Instance fields | 6@- 7@- |
| MT | 17@- |
| Methods | 18@- |

| This class | 3 |
| Super class | 23 |
| CP | (1, x1)(2, 3.1)(3, C)(4, x2)(5, 3.4)(6, x3)(7, 3.6)(8, b1)(9, 3.8)(10, c2) |
| Static fields | 2@- 5@- 7@- |
| Instance fields | 9@- 11@- |
| MT | 13@- |
| Methods | 15@- |

| This class | 3 |
| Super class | 19 |
| CP | (1, x1)(2, 3.1)(3, B)(4, x2)(5, 3.4)(6, x3)(7,3.6)(8, b1)(9, 3.8)(10, b2) |
| Static fields | 2@- 5@- 7@- |
| Instance fields | 9@- 11@- |
| MT | 13@- |
| Methods | 15@- |

Figure 17: Three abstract class files prior to class loading
Figure 18: The three abstract class files shown in Figure 17 after class loading

| This class | A |
| Super class | Obj |
| CP | (1, A)(2, A.x1@0)(3, x1)(4, A.x2@1)(5, x2)(6, A.a1:0)(7, A.a2:1)(8, a1) |
| Static fields | A.x1@0 A.x2@1 A.x3@2 |
| Instance fields | A.a1:0 A.a2:1 |
| MT | A.foo#0 A.bar#1 |
| Methods | A.foo() A.bar() |

| This class | B |
| Super class | A |
| CP | (1, x1)(2, B.x1@3)(3, B)(4, x2)(5, B.x2@4)(6, x3)(7, B.x3@5)(8, b1) |
| Static fields | B.x1@3 B.x2@4 B.x3@5 |
| Instance fields | B.b1:2 B.b2:3 |
| MT | B.foo#0 A.bar#1 B.f#2 |
| Methods | B.foo() B.f() |

| This class | C |
| Super class | A |
| CP | (1, x1)(2, C.x1@6)(3, C)(4, x2)(5, C.x2@7)(6, x3)(7, C.x3@8)(8, c1) |
| Static fields | C.x1@6 C.x2@7 C.x3@8 |
| Instance fields | C.c1:2 C.x2:3 |
| MT | A.foo#0 C.bar#1 C.f#2 |
| Methods | C.bar() C.f() |

Figure 19: Index Resolution

\[
\text{index\_resolution} : \text{class}_0 \rightarrow \text{FIX\_TDL (seq\_tdl cp\_normalize class}_0) \text{ class}_0 \\
\text{cp\_normalize} : c\_entry[[ index1, d1 ]] \rightarrow d[[ index1 ]] \rightarrow d1
\]
rewriting the first term matching $addr[0]$ to $addr[1]$ and will leave unchanged all other terms.

Generalizing this example, let $s_i$ denote a strategy of the form $\text{transient}(addr[i] \rightarrow addr[i+1])$, and let $s_{1..n}$ denote a strategy of the form $s_1; s_2; \ldots; s_n$. The strategic expression $TDL \ s_{1..n} \ \text{class} = 0$ will rewrite the first occurrence of $addr[0]$ to $addr[1]$, the second occurrence of $addr[0]$ to $addr[2]$, and the $n^{th}$ occurrence of $addr[0]$ to $addr[n]$. Suppose we are given a Java application whose class files collectively contain $m$ static fields, all of which are of type integer and whose initial default address has been set to 0. When controlled properly, the application of a strategy of the form $s_{1..m}$ can be used to correctly assign a unique address to each static field in the application. In spirit, this is the transformational effect that we want to accomplish. However, for a variety of reasons, in practice it is difficult to attempt to construct such a strategy directly. Thus, we will use the $\text{hide}$ combinator to construct a slightly different strategy whose net effect is equivalent to $s_{1..m}$.

In strategic programming, the left-biased choice combinator $\ll$ is used to specify the conditional application of two or more strategies. Recall that, when applied to a term $t$, the strategy $s_1 \ll s_2$ specifies that first the application of $s_1$ to $t$ should be attempted. If this application succeeds and yields $t'$, then the result of $(s_1 \ll s_2)$ to $t$ is $t'$. On the other hand, if the application of $s_1$ to $t$ fails, then the left-biased choice combinator indicates that the application of $s_2$ to $t$ should be tried next. In light of this, let us consider the strategy $\text{hide}(s_1) \ll s_2$. The $\text{hide}$ combinator is a unary combinator that restricts the ability of the left-biased (or right-biased) choice combinator to observe whether or not the application of $s_1$ has succeeded or failed. More specifically, the $\text{hide}$ combinator always gives the left/right-biased choice combinator the impression that the application of $s_1$ has failed. Thus, the strategy $\text{hide}(s_1) \ll s_2$ is equivalent to $s_1; s_2$.

When considered in isolation, the $\text{hide}$ combinator is not very interesting. However, when combined with the $\text{transient}$ combinator it becomes possible to construct strategies having interesting behaviors, such as a strategy that implements a sum. For example, consider the following strategy:

$$
\text{sum} = \begin{cases} 
\text{hide}(addr[i] \rightarrow addr[i+1]) \ll \text{transient}(addr_1 \rightarrow addr_1) \ll + \\
\text{hide}(addr[i] \rightarrow addr[i+1]) \ll \text{transient}(addr_2 \rightarrow addr_2) \ll + \\
\text{hide}(addr[i] \rightarrow addr[i+1]) \ll \text{transient}(addr_3 \rightarrow addr_3) \ll + 
\end{cases}
$$

where the symbol $\oplus$ performs something we will call an increment and has the following semantics:

$$
addr_1 \oplus y \overset{def}{=} \begin{cases} 
addr[z] & \text{if } \exists x : addr_1 = addr[x] \text{ and } x \text{ is of type integer and } z = x + y \\
addr[0] & \text{if } addr_1 = addr[-] \text{ that is, } addr_1 \text{ is an uninitialized address}
\end{cases}
$$

The advantage of a special purpose increment operator is that the nature of the increment is easier to control. For example, the base value of an field may be some value other than 0, or the addition may be in hex using a numeric representation limited to a specific width.

When applied to a term $t$ using a traversal like $TDL$, the strategy $\text{sum}$ will increment the first $addr$ it encounters once, increment the second $addr$ it encounters twice (e.g., $i \oplus 1 \oplus 1$, and increment the third $addr$ it encounters three times. All remaining instances of $addr$ encountered will be incremented three times. The informal explanation of this is as follows. Within $t$, when applied to the first occurrence of $addr$, the first $\text{hide}$ encapsulated rule in $\text{sum}$ will apply, incrementing the address once. However, due to the encapsulation of the $\text{hide}$ combinator, the conditional operator $\ll$ immediately to the right of the $\text{hide}$ strategy is led to believe that the application of $\text{hide}(addr[i] \rightarrow addr[i+1])$ failed. Thus, the application of the next strategy within $\text{sum}$ is attempted, which in this case is $\text{transient}(addr_1 \rightarrow addr_1)$. The rule $addr_1 \rightarrow addr_1$ is an identity that rewrites the term $addr_1$ to itself. From the perspective of $\ll$ the application of the $\text{transient}$ encapsulated rule succeeds, completing the application of $\text{sum}$ to this particular instance of $addr$. However, since transient strategies can only be applied once, the transient
4.3 Instance Field Offset Calculation in TL

Instance field offset calculation (see Section 2.3) is similar to static field address calculation. The primary difference is that the assignment of offset addresses is constrained by the subtype (i.e., inheritance) relationships between the class files within an application. In static field address calculation, the static fields of all classes in the application could be collected (in any order) and aggregated into a sum, which could then be applied to the entire application (in the same order as collected). In instance field offset calculation, the instance fields of each class in an inheritance chain must be collected according to the order/position of the class in the inheritance chain. The resulting sum must then be applied only to the instance fields of that chain.

In TL, there are several ways to construct strategies whose application is restricted to individual inheritance chains. The approach taken in this article requires that an application first be restructured creating a new structure that explicitly reflects the inheritance hierarchy of the class files. This is accomplished by adding a children element to each class file (see Figure 16). This children structure denotes a list of class files and creates the possibility of restructuring the class files from an application (which are initially in list form) into an inheritance tree whose root is Obj. After this has been accomplished an iterative process can be specified whereby a strategy realizing a partial sum is created using only the instance fields belonging to a given class file class1 and the resulting partial sum strategy is applied to all the classes belonging to the inheritance tree having class1 as its root (i.e., class1 and all its descendants). During the course of a top-down traversal, each class file will in turn become the root of its own inheritance (sub)tree and have a partial sum created for it. The cumulative result is that each instance field will eventually be assigned a proper offset address (i.e., a fully totaled sum).

Figure 21 gives a TL strategy that implements the approach to instance offset calculation just described. When applied to app1, the strategy instance_offsets will first restructure app1 into an inheritance tree. The
For Peer Review

Figure 21: Offset address calculation

evaluation of the strategic expression seq_tdl create_hierarchy app1 will perform a top-down left-to-right traversal on app1 and apply the higher-order strategy create_hierarchy to each class file encountered. The application of create_hierarchy to class1 will produce a first-order strategy that places class1 into the children list of its parent class. Notice that in the strategy create_hierarchy, the variables id1 and id2 respectively denote the class name and parent class name of class1. By definition, the parent class of class1 is that class which has id2 as its class name. Thus, in create_hierarchy the strategy

class[ {id2 id3 info2 children2} ] → class[ {id2 id3 info2 children2 class1} ]

is a rewrite rule that places class1 into the children list of its parent class.

For each class in app1 such a first-order strategy is created and the resulting strategies are sequentially composed. The resulting composition is then applied in a top-down left-to-right fashion to an application structure consisting of the single class Obj, which is initially empty (i.e., Obj contains no constant pool, no fields, no methods, and no children). This application has the effect of growing an inheritance tree rooted at Obj. For example, first the children of Obj will be inserted into the children list of Obj. In turn, the children of Obj will have their children inserted into their children list, and so on. The resulting structure is then bound to app2 via a match equation.

After the application has been restructured into an inheritance tree, the calculation of instance field offsets can begin. In the strategy instance_offsets, the evaluation of the strategic expression TDL sum_offsets app2 will traverse the inheritance tree app2 in a top-down left-to-right fashion and apply the strategy sum_offsets to each class file encountered. In turn, the strategy sum_offsets will traverse the instance fields of the class file class1 to which it is applied and create an instance of local_ifield_sum for each field encountered. The instances of local_ifield_sum are then conditionally composed and the resulting strategy is applied in a top-down left-to-right fashion to the inheritance tree rooted at class1. This has the effect of (1) assigning proper (i.e., completed) offsets for the instance fields in class1, and (2) assigning partially completed offsets for the instance fields of all the classes which are descendants of class1. In general, the partially completed offsets for instance fields local to a given class are completed when the strategy sum_offsets is applied to class1 (i.e., it is treated as the root of an inheritance tree).

Figure 22 graphically depicts the behavior of the strategic expression TDL sum_offsets app2 which is used by the instance_offsets strategy to calculate instance field offsets within a class hierarchy. On the left side of the figure, the strategy depicted in the upper oval represents the strategy created from the evaluation of lcond_tdl partial_sum instance1. The strategy depicted in the lower oval represents the strategy that will be applied to all the children of the class that is currently being processed (the current class has instance offsets : app1 ↠ app3
if app2 ≪ TDL (seq_tdl create_hierarchy app1) app[ {Obj Obj} ]
∧ app3 ≪ TDL sum_offsets app2

create_hierarchy : class1 → class[ {id2 id3 info2 children2} ] → class[ {id2 id3 info2 children2 class1} ]
if class1 → class[ {id1 id2 info1 children1} ]

sum_offsets : class0 → TDL (lcond_tdl partial_sum instance1) class0
if class0 → class[ {class_id parent_id1 cp1 statics1 instance1 methods1 children1} ]

partial_sum : instance0 → TDL( lcond_tdl local_ifield_sum instance0) instance0
local_ifield_sum : ifield0 →
hide( ifield[ key1 : addr1 ] → ifield[ key1 : addr1 ⊕ 1 ] ) ≪ transient(ifield1 → ifield1)
fields $a_1$, $a_2$, and $a_3$). Note that, by the time the strategy has been applied to the last instance field of the current class, all the transient strategies have been “used” and the only strategies that remain are enclosed in the hide combinator.

The right side of the figure shows a later point in the traversal driven by $TDL$ sum_offsets app$_2$ where the current class has the instance fields $c_1$, $c_2$ and $c_3$.

Figure 22: An abstract diagrammatic trace of offset address calculation

4.4 Method Table Construction in TL

Method table construction (see Section 2.4) is similar to instance field offset calculation in the sense that strategies are applied along inheritance chains. In the case of method table construction the goal is to construct a method table for each class within an application. The general algorithm implemented is as follows. One begins at the class Obj with an empty method table. This table is then propagated to all the descendants of Obj after which, method table construction begins for the children of class Obj. Let class$_1$ denote an arbitrary child of class Obj. The methods declared in class$_1$ are “added” to the method table of class$_1$ as well as to the method tables of all the descendants of class$_1$. This distribution represents the methods that are initially inherited by the descendants of class$_1$. In general, the methods declared in class$_i$ are added to the method table of class$_1$ and all of its descendants. The “addition” of a method $m$ to the method table of class$_1$ can take one of the following three forms:

1. **Redeclaration.** An existing method table entry $m'$ is encountered corresponding to the method $m$. This means that $m'$ has been redefined by $m$ in class$_1$. In this case, $m$ replaces (overwrites) the entry
for \( m' \).

2. **Declaration.** The last entry of a (non-empty) method table is reached without encountering an entry corresponding to \( m \). This means that, \( m \) is a new (i.e., previously unseen) method declared in \( class_i \). In this case, a new entry is added to the end of the method table and its offset is assigned the value of the previous offset plus one.

3. **Basis.** An empty method table is encountered. In this case, an entry for \( m \) is added to the method table and given an offset of zero.

Figure 23 shows how the method table construction strategy described can be implemented in TL. When applied to a method declaration, the strategy \texttt{insert\_method} creates a transient strategy that, through the use of the left-biased choice combinator, captures the three ways a method can be added to a method table. The rule

\[
\text{mt}[\text{id}_3.\text{id}_2 \# \text{addr}_2 \text{mt}_2] \rightarrow \text{mt}[\text{id}_1.\text{id}_2 \# \text{addr}_2 \text{mt}_2]
\]

accounts for the case where a local method definition overwrites an inherited method definition. In this case, \( \text{id}_3 \) denotes the most recent ancestor where the method \( \text{id}_2 \) has been declared and \( \text{id}_1 \) denotes the name of the current class in which the method is being redeclared. The rule

\[
\text{mt}[\text{id}_3.\text{id}_4 \# \text{addr}_2] \rightarrow \text{mt}[\text{id}_3.\text{id}_4 \# \text{addr}_2 \text{id}_1.\text{id}_2 \# \text{addr}_2 \oplus 1]
\]

accounts for the case where a previously unseen method is declared and must therefore be added to the end of the method table with an offset of \( \text{addr}_2 \oplus 1 \). And finally, the rule

\[
\text{mt}[\text{id}_2] \rightarrow \text{mt}[\text{id}_0.\text{id}_1 \# 0]
\]

accounts for the base case where a method is added to an empty method table. In this case the offset address is set to 0. Notice that the aggregation of the above rules needs to be encapsulated within a \texttt{transient} in order to assure that, regardless of which case applies, a method \( m \) will only be added to a given method table once. For example, it would be incorrect to overwrite an existing method and also add a new (i.e., duplicate) entry to the end of the same method table.

Within the strategy \texttt{add\_methods}, the evaluation of the strategic expression \texttt{seq\_tdl insert\_method method\_list_1} will result in the creation of a method table insertion strategy for each method in \texttt{method\_list_1}, which is the list containing the methods that are declared in \( class_1 \). The insertion strategies are sequentially composed by \texttt{seq\_tdl} and the resulting strategy is ready to be applied to a method table. Let \( s \) denote

<table>
<thead>
<tr>
<th>Method Description</th>
<th>Rule</th>
</tr>
</thead>
</table>
| \texttt{mt\_construction}              | \begin{align} \text{app}_1 & \rightarrow \text{TDL add\_methods app}_1 \\ 
| \texttt{add\_methods}                  | \begin{align} \text{class}_1 & \rightarrow \text{TDL}_B (\text{seq\_tdl insert\_method method\_list}_1) \text{ class}_1 \end{align} \\ 
| \texttt{insert\_method}                | \begin{align} \text{id}_1.\text{id}_2 ( ) & \rightarrow \\ & \text{transient} \left( \\ & \text{mt}[\text{id}_3.\text{id}_2 \# \text{addr}_2 \text{mt}_2] \rightarrow \text{mt}[\text{id}_1.\text{id}_2 \# \text{addr}_2 \text{mt}_2] \\ & <\text{mt}[\text{id}_3.\text{id}_4 \# \text{addr}_2] \rightarrow \text{mt}[\text{id}_3.\text{id}_4 \# \text{addr}_2 \text{id}_1.\text{id}_2 \# \text{addr}_2 \oplus 1] \\ & <\text{mt}[\text{id}_1.\text{id}_2 \# 0] \right) \end{align} \\ 

Figure 23: Method table construction
this strategy. The trick that needs to be worked out now is how to apply this value of $s$ to the method table in $\text{class}_1$ as well as the method tables of every class which is a descendant of $\text{class}_1$. The problem is that $s$ contains transient strategies that can only be applied once. For example, adding an entry to the method table of $\text{class}_1$ will change $s$ so that this element cannot in the future be added to the method tables of any of the descendants of $\text{class}_1$. To solve this problem what we need is some way of making a copy of the current value of $s$ (i.e., the value before any transient strategies have been applied). The first-order generic traversal $\text{TDL}_B$ does just that! In general, the evaluation of a strategic expression of the form $\text{TDL}_B \ s \ t$ will perform a top-down left-to-right traversal over the term $t$ and do the following: First, $s$ is applied to the current term $t$ producing a (possibly) new term $t'$ and a (possibly) new strategy $s'$. Next, a copy of the strategy $s'$ is applied to each of the children of $t'$, at which point the process repeats until the entire tree is traversed. As a result the evaluation of the strategic expression

$$\text{TDL}_B(\text{seq}_{\text{tdl}} \ \text{insert\_method} \ \text{method\_list}_1) \ \text{class}_1$$

will correctly insert the methods declared in $\text{method\_list}_1$ into the method table of $\text{class}_1$ as well as in the method tables of all the descendants of $\text{class}_1$. Note that the trees corresponding to the method table and descendants of a class are proper subtrees of the class tree. Therefore, under the $\text{TDL}_B$ will each receive their own copy of the method table insertion strategy resulting from the evaluation of $\text{seq}_{\text{tdl}} \ \text{insert\_method} \ \text{method\_list}_1$).

And finally, the strategy $\text{mt\_construction}$, when applied to an application $\text{app}_1$ that is in the form of an inheritance tree, will create the proper method tables for each class in $\text{app}_1$. It accomplishes this by traversing $\text{app}_1$ in a top-down left-to-right fashion and applying the strategy $\text{add\_methods}$ to every inheritance tree encountered.

### 4.5 Inter-class Absolute Address and Offset Address Distribution in TL

At this point, all data indexes within the application have been resolved to symbolic references, all static fields in the application have been assigned absolute addresses in $D_{\text{HEAP}}$, all instance fields in the application have been assigned offset addresses in $D_{\text{OBJ}}$, and all methods in the application have been assigned method table offset addresses in $D_{\text{MT}}$. Given these preconditions, inter-class absolute address and offset address distribution concerns itself with the distribution of the aforementioned address values to their corresponding symbolic references in constant pool entries. Note that such references are not particularly constrained by inheritance chains. That is, a class file belonging to one inheritance chain may reference static fields, instance fields, and methods declared in a class file belonging to another inheritance chain.

Figure 24 shows a TL implementation of inter-class absolute address and offset address distribution. The evaluation of the strategic expression $\text{lc\_cond\_tdl} \ \text{collect\_fields} \ \text{app}_1$ will create an instance of the rule

$$\text{c\_entry}[ (\text{index}_1, \text{key}_1) ] \rightarrow \text{c\_entry}[ (\text{index}_1, \text{key}_1 : \text{addr}_1) ]$$

for each instance field in the application $\text{app}_1$. This rule adds the offset associated with the instance field $\text{key}_1$ to a constant pool entry containing the symbolic reference $\text{key}_1$. These rule instances are composed using the left-biased choice combinator and the resulting strategy is then applied to $\text{app}_1$ using the generic traversal $\text{TDL}$. The effect is that all constant pool entries containing symbolic references to instance fields will be updated so they also contain the corresponding offset address for that instance field. The result of this transformation is then bound to $\text{app}_2$ via a match equation. Next the absolute addresses for all static fields in $\text{app}_2$ is distributed by the same mechanism that was used to distributed instance field offsets. The result is then bound to $\text{app}_3$ via a match equation. And finally, method table offsets are distributed, completing the class loader core as defined in Section 1.2.

The strategies defined in the previous section can now be composed to form an abstract class loader as shown in Figure 25.
4.6 Merits of a Transformation-based Approach

We believe that the abstractions and primitives used to implement the SSP class loader core discussed in this article have a computational nature that is distinctly different from an implementation written in, say, an imperative or object oriented language. Some strategies, like the implementation of index resolution, have transformational ideas that may seem elegant and intuitive to the reader. Other strategies, like the implementation of method table construction, have transformational ideas that may seem awkward or opaque to the reader. A fair question to raise at this time is: “Why is a transformation-based approach to the class loader problem being explored?” This type of question is of a fundamental nature and our answer is one having philosophical overtones.

Interest in program transformation is driven by the idea that, through their repeated application, a set of “simple” rewrite rules can affect a major change in a software artifact. From the perspective of dependability, the explicit nature of transformation exposes the software development process to various forms of analysis that would otherwise not be possible.

From a theoretical standpoint, virtually all programming languages are Turing complete, so the choice of language is not constrained by computational limitations. Ultimately, the choice of which language paradigm to use is based on several key factors including: (1) the efficiency of the resulting implementation, (2) the ability to scale problem solutions to larger similar problems, and (3) the ability to subject the implementation to various forms of analysis (e.g., code inspections, type checking, formal verification, etc.).

Since class loading for the SSP can be performed off-line, the efficiency of the implementation is a desirable but not critical factor. Furthermore, the concepts of the abstract class loader core described in this article have been scaled to an actual SSP class loader in a relatively straightforward manner (e.g., generic traversals are insensitive to the structure and complexity of the tree to which they are applied).

On the other hand, since the application domain for the SSP includes high-consequence applications, it is essential that high-assurance be provided in the correctness of the functionality of the class loader. To date, a number of validation techniques have been used to provide confidence in the operation of the SSP and its class loader [22]. From the perspective of assurance, formal verification is an attractive technique because it offers the promise of mathematical certainty that a system (such as the class loader) possesses some set of formally specified properties.
Transformation is a paradigm that has its roots in equational reasoning, an area of research for which there exists a tremendous body of knowledge. As a result, transformation-based programming would seem to be a natural candidate that stands to benefit from research in equational reasoning. Offsetting this somewhat are advances in strategic programming that have introduced abstractions (e.g., combinators and traversals) enabling transformation-based paradigms to be effectively used to solve problems using rule sets that are neither confluent nor terminating. Our research in the area of transformation is motivated by the belief that, in the long term, a logic can be developed extending over these abstractions thereby enabling reasoning about strategic programs to more fully take advantage of advances in equational reasoning.

5 Verification and Validation

One motivating factor in the development of TL in general and the SSP specifically is the attainment of strong evidence of correctness. In this section we present current efforts to construct a framework for proving the correctness of TL transformations using the automated theorem prover ACL2.

The goal with respect to the SSP class loader is to prove formally and automatically that the TL implementation of the class loader preserves properties of the kind identified in Section 1.2. To this end, we are beginning to model the behaviors of strategies, traversals, combinators, and conditional rewrites in ACL2. The work described here is only the initial steps towards proving the correctness of the TL implementation of the class loader core. The long-range goal of this work is to reason not only about specific transformations, but also about the internal mechanisms of TL and transformation systems in general. ACL2 has the capability to reason about combinators and traversal strategies, and this reasoning can be reused in other applications that utilize transformation-oriented programming.

5.1 The ACL2 Theorem Prover

ACL2 [7, 8] is a first-order, quantifier-free mathematical logic based on recursively defined total functions. It is also a programming language based on the applicative subset of Common Lisp in which users can build executable models of software systems and prove that these models have certain properties. To use ACL2, a user first builds an executable model by writing functions in the ACL2 language. Before a function definition is accepted, ACL2 must prove that the function is total, i.e., it eventually terminates for any input. Once a function is accepted, a user may execute the function to test it or prove theorems about the function.

Theorems are specified using the construct defthm. While ACL2’s theorem prover is fully automatic, it is usually necessary for a user to supply lemmas to guide the proof. Once a theorem has been proved, it can be used in subsequent proofs. To prove theorems, ACL2 uses six proof techniques: simplification, elimination of destructors, use of equivalences, generalization, elimination of irrelevance, and induction. The two most important techniques are simplification and induction. After proving a theorem, ACL2 will store the result as a rewrite and use it during simplification.

To prove a theorem, \( T(n) \), by classical induction, we have to show as a base case that \( T(0) \) holds, then as the induction step that \( T(n) \rightarrow T(n + 1) \). The induction used in ACL2 restates the classical one: to prove \( T(n) \) we have to show that \( T(0) \) holds, then as the induction step we have to show that \( (n \neq 0) \) and \( T(n - 1) \rightarrow T(n) \). Thus, ACL2 attempts to prove a theorem \( T(n) \) by induction from smaller instances of the theorem, \( T(n - 1) \).

5.2 Verifying the Correctness of the SSP Class Loader: A Roadmap

In the limit, we would like formal verification to provide us with a rigorous argument that the SSP class loader constructs a suitable ROM image for every Java application that the SSP is capable of executing,
For Peer Review and to reject those applications that lie outside of the input domain of the SSP. For the purposes of stating our verification goal we will assume that the SSP itself has been implemented correctly, and that a Java Virtual Machine JVM exists that has also been implemented correctly. We assume that SSP\textsubscript{Subset} is a predicate that, when given a Java application \texttt{app}, evaluates to \texttt{true} if \texttt{app} belongs to the input domain of the SSP and \texttt{false} otherwise. Ideally, we would like to use formal verification to demonstrate the following:

\[
\forall \texttt{app} \in \texttt{Java} : \texttt{SSP}_{\text{Subset}}(\texttt{app}) \Rightarrow \text{JVM}(\texttt{app}) \equiv \text{SSP}(\texttt{load}(\texttt{app}))
\]

where \texttt{load}(\texttt{app}) denotes the ROM image resulting from applying the TL strategy \texttt{load} to \texttt{app}, and \texttt{JVM}(\texttt{app}) \equiv \text{SSP}(\texttt{load}(\texttt{app})) means that the observable behavior of the JVM and SSP are equivalent for the given input. Formally verifying the above stated equivalence directly is a daunting task. For this reason we are currently focusing our efforts on verifying a weaker property. The rationale for our current focus is based on the following:

**Conjecture 1** \[
\forall \texttt{app} \in \texttt{Java} : \texttt{SSP}_{\text{Subset}}(\texttt{app}) \land \text{CH}(\mathcal{I}_{\text{app}}) \Rightarrow \text{JVM}(\texttt{app}) \equiv \text{SSP}(\texttt{load}(\texttt{app}))
\]

This conjecture claims that if a Java application \texttt{app} belongs to the input domain of the SSP and the interpretation \mathcal{I}_{\text{app}} constructed by the SSP class loader satisfies a set of requirements denoted by \text{CH} (see Section 1.2), then the observable behavior of the JVM and SSP will be equivalent.

To date, roughly 190 properties have been identified \cite{22} and can be collectively seen to form \text{CH}. In practice, these properties have shown themselves to be sensitive enough to detect single bit-flips in ROM images for a number of test applications. At the level of abstraction discussed in this article however, many of the properties cannot be expressed because they refer to portions of the class file that have been abstracted away. Nevertheless, several core properties still remain, some of which have been described in Section 1.2. Figure 26 gives a formal definition for some of the correctness properties \mathcal{C} originally mentioned in Section 1.2.

Under the assumption that Conjecture 1 holds, the correctness of the SSP class loader can be established by proving the following theorem:

**Theorem 1** \[
\forall \texttt{app} \in \texttt{Java} : \texttt{SSP}_{\text{Subset}}(\texttt{app}) \Rightarrow \text{CH}(\mathcal{I}_{\text{app}})
\]

An important thing to note is that the soundness of Theorem 1 as well the definitions given in Figure 26 rests on the assumption that index resolution has been performed correctly. Thus, in order to increase our confidence in the correctness of index resolution, we now discuss how we have used ACL2 to model and reason about the behavior of the TL strategy \texttt{index\_resolution}. In particular, we will show the equivalence between two definitions of index resolution. The first definition, denoted \texttt{M\textsubscript{dref}}, represents a direct formalization of index resolution, in the style of big step semantics. This definition is denoted \texttt{M\textsubscript{dref}} and is modelled in ACL2 by the function \texttt{dref}. In contrast, the TL strategy \texttt{index\_resolution} implements index resolution in the style of small step semantics. The remainder of this section is devoted to a discussion of how ACL2 was used to prove the equivalence of these two definitions.

Assuming that sufficient evidence has been provided that the \texttt{index\_resolution} strategy is correct, it is then a trivial task to show that the remaining strategies in \texttt{load} preserve the correctness of index resolution, thus justifying our conclusion that index resolution is correct for \mathcal{I}_{\text{app}}.
For Peer Review

Figure 25: Formulation of the correctness properties described in Section 1.2

5.3 Exception-free Class Loading

The JVM has precise rules concerning when exceptions can be thrown. In particular, it must appear to the external observer that any exceptions thrown as a result of class loading are thrown after the computation initiates a first use of the offending class (regardless of when the class was actually loaded). Examples of exceptions that can be thrown during class loading include: `IllegalAccessError`, `NoSuchFieldError`, `IncompatibleClassChangeError` and so on.\(^1\)

In this paper, we restrict our attention to Java applications (i.e., collections of class files) in which class loading is free from the kinds of exceptions mentioned in the previous paragraph. We will use the term `exception-free` to characterize this form of class loading. For the SSP, `exception-free` class loading is assured by a variety of checks and development constraints.

---

\(^1\)Initialization issues (e.g., exceptions) resulting from the execution of `<clinit>` are beyond the scope of this discussion.
5.4 The Basic Correctness Theorem for Index Resolution

Our approach to proving the correctness of \textit{index\_resolution} in ACL2 consists of modelling and verification phases and is summarized in Figure 27. The proof of Theorem 2, stated in Figure 27, forms the basis of our claim that \textit{index\_resolution} is correctness-preserving.

1. Modelling Phase:
   (a) The general class file structure \(c\) is modelled in a manner that ACL2 understands.
   (b) The TL transformation mechanism defined by \textit{index\_resolution} is modelled as the ACL2 function \textit{index\_res\_ACL2}.
   (c) The semantic function, \(M_{dref}\), is implemented in ACL2 by a function called \textit{dref}. The function \textit{dref} represents the transitive closure of the discrete reduction steps that form the basis of \textit{index\_resolution}.

2. Termination Phase:
   (a) A proof is constructed showing that \textit{index\_res\_ACL2} has a decreasing measure and thus represents a terminating computation. (Proof of termination is a prerequisite for reasoning about a function in ACL2).
   (b) A proof is also constructed (not discussed in this paper) showing that \textit{dref} terminates.

3. Verification Phase:
   (a) A proof is constructed showing that the application of the \textit{index\_res\_ACL2} to an input class file will preserve the semantics of \(c\), i.e., the transformation rules, when applied to the \(c\) in a certain order, will preserve correctness. This last goal is achieved by proving the following:

   \[
   \text{Theorem 2 } \forall c : dref(c) \equiv dref(index\_res\_ACL2(c))
   \]

Figure 27: Overview of verification approach

It is important to note that the transfer of our correctness result from ACL2 back to the realm of TL (i.e., the \textit{index\_resolution} strategy is correctness-preserving) is dependent upon a number of factors. Factors pertinent to this discussion are that all artifacts, produced under the modelling phase of our approach, have been constructed correctly and faithfully capture all salient features of TL transformation. Although the modelling phase is relatively straightforward, by its very nature, it must fall outside the of the scope of ACL2 verification. Therefore, we have used other techniques such as code inspections and rigorous testing to validate the correctness of our initial models.

5.5 Modelling Phase

In this section we give an overview of the ACL2 models for \textit{cp}, \textit{index\_resolution}, and \(M_{dref}\).

5.5.1 The Constant Pool

In our ACL2 model, a Java class file is modelled by a list of structures consistent with our description in Section 2. The \textit{constant pool}, which represents the first structure in the list, is modelled by a list whose elements model constant pool entries. A constant pool \textit{entry} is modelled as a tuple consisting of two
5.5.2 Modelling the Mechanism of index\_resolution

\[
\begin{align*}
\text{index\_res\_ACL2}(c) & \triangleq \text{index\_res}(\text{get-cp}(c)) \\
\text{get-cp}(c) & \triangleq \text{car}(c) \\
\text{index\_res}(cp) & \triangleq \text{fix\_tdl}(\text{gr}(cp), cp) \\
\text{fix\_tdl}(rs, cp) & \triangleq \begin{cases} cp & \text{if } \text{acount-cp}(\text{once\_tdl}(rs, cp, cp)) \geq \text{acount-cp}(cp) \\
\text{fix\_tdl}(rs, \text{once\_tdl}(rs, cp, cp)) & \text{otherwise} \end{cases} \\
\text{once\_tdl}(rs, tail, cp) & \triangleq \ldots \\
\text{acount-cp}(cp) & \triangleq \text{acount-cp1}(cp, cp) \\
\text{acount-cp1}(tail, cp) & \triangleq \begin{cases} 0 & \text{if } \text{endp}(tail) \\
\text{add}(\text{acount}(\text{car}(\text{car}(tail))), cp), \text{acount-cp1}(\text{cdr}(tail), cp)) & \text{otherwise} \end{cases} \\
\ldots
\end{align*}
\]

Figure 28: A fragment of the ACL2 model of index\_resolution

The transformation mechanism defined by index\_resolution is modelled in the framework of ACL2 by the function index\_res\_ACL2 – a function which takes as input an ACL2 class file model (c) and returns a class file model in which all indexes has been resolved. The function index\_res\_ACL2 first extracts the constant pool (cp) of the class file (c), by applying the function get-cp to the c, and applies the function index\_res to the cp. The function index\_res first generates a list of first-order rewrite rules, and then applies these rules exhaustively to the input cp. The main portion of the ACL2 model of index\_resolution is shown in Figure 28.

In the body of index\_res, the rule generation process gr(cp) simulates the behavior of the strategic expression (seq\_tdl cp\_normalize cp0) that occurs in the index resolution strategy shown in Figure 19. The evaluation of gr(cp) yields a list of rewrite rules rs. The rules in this list are treated as being (implicitly) composed using the sequential combinator (;). Each rewrite rule in the rs has the form: \((\text{LHS}, \text{RHS})\) where LHS and RHS denote the ACL2 models of \([d[\text{index}1]]\) and \(d_1\) respectively.

After generating the appropriate strategy rs, the function index\_res calls the recursive function fix\_tdl; which will exhaustively apply the first-order strategy rs to cp in a top-down left-to-right fashion. The result is a new resolved constant pool that has no indexes in it. In this manner, the transformation mechanism of index\_resolution is modelled in ACL2.
5.5.3 Implementation of the Semantic Function $M_{dref}$

The ACL2 model of the semantic function $M_{dref}$ is called $dref$ and is shown in Figure 29. There are a few things that are worth mentioning about $dref$. From an operational perspective, $dref$ can be thought of as a function that, when given a data object model $p$ and a constant pool model $cp$, will directly try to resolve all indexes in $p$ by repeatedly replacing indexes with the data found in the corresponding constant pool entry. However, ACL2 requires that $dref$ also be a total function. What this means is that $dref$ must be defined for any structurally legal data object model and constant pool model. The totality constraint for $dref$ requires that two additional properties of well-formed constant pools be made explicit: (1) constant pool entries may not contain indexes whose resolution forms a cycle, e.g., $1 \Rightarrow 2 \Rightarrow 1$ or which contain a cycle as a sub-component, e.g., $1 \Rightarrow 1 \Rightarrow 2$, (2) every index in a data object must refer to an actual constant pool entry, e.g., if a constant pool has only 10 entries, then an index may not refer to the 22nd element of the constant pool.

![Figure 29: A fragment of the ACL2 model of $M_{dref}$](image)

In Figure 29, the functions $acyclicp$ and $acyclicp1$ model the constraint that index resolution sequences be acyclic. Similarly, the functions $well-formed$ and $well-formed1$ model the constraint that indexes fall within the domain of the constant pool.
5.6 Termination Phase

The definitional principle of ACL2 requires that a function be proven to terminate before it is admitted into the ACL2 verification framework. This is done to preserve the soundness of the underlying logic of ACL2. For example, before proving any theorem related to the function index_res_ACL2, we must first admit index_res_ACL2 and the other related functions in our model. The proof of termination for non-recursive functions is trivial. For recursive functions, it is, in some cases, straightforward and done automatically by ACL2; in other cases, it requires the user to identify a well-founded measure that decreases after each recursive call.

From the perspective of termination, index resolution represented by the function index_res_ACL2 can be abstractly seen as a reduction, \( \rightarrow \), on the ground term language \( \sum \) (defined in Section 5.5.1) where indexes, strings, and sequences consisting of indexes and strings form elements of \( \sum \). Termination can be demonstrated by constructing a function \( \varphi : \sum \rightarrow \mathbb{N} \) which is a monotonic embedding of \((\sum, \to)\) into \((\mathbb{N}, >)\). Specifically, it must be shown that \( \forall t.t' \in \sum : t \rightarrow t' \text{ implies } \varphi(t) > \varphi(t') \) from which the termination of the reduction system \((\sum, \to)\) is immediate.

5.6.1 Termination of index_res_ACL2

As presented in Figure 28, the function index_res_ACL2 calls the function index_res, and since both of these functions are non-recursive, the proof of termination for them is trivial; however, index_res calls the recursive function fix_tdl for which we have to provide a proof that the function has a decreasing measure after each recursive call. The function fix_tdl, which is the ACL2 model of FIX_TDL, takes as inputs two arguments: a strategy model \( rs \) and a constant pool model \( cp \). Operationally, fix_tdl traverses the \( cp \) in a top-down left-to-right fashion, and applies \( rs \) to all indexes encountered in every entry in \( cp \). This process is repeated until all indexes in \( cp \) are resolved. Then, the resolved version of \( cp \) is returned.

\[
\text{Theorem 3 The termination of fix_tdl.}
\]

\[
\forall cp, tail, rs : \ ok-cp(cp) \land \text{acyclic-indexes}(tail, cp) \land \text{acyclic-rhs-r}(rs, cp) \land \text{match}(rs, cp)
\]
\[
\Rightarrow \text{acount-cp(once_tdl(rs, tail, cp)) < acount-cp(cp)}
\]

Figure 30: The theorem capturing the termination of fix_tdl

The termination condition for fix_tdl is expressed via the function acount-cp whose implementation is shown in the bottom portion of Figure 28. Informally speaking, acount-cp captures the essence of the decreasing measure function \( \varphi \). More specifically, the function acount-cp accumulates the number of steps needed to resolve an index \( p \) in \( cp \), \( \forall p \in cp \). This function, also, serves as a wrapper for a more general function called acount-cp1.

The function acount-cp1 recursively computes the number of steps needed to resolve each index in tail. Initially, the function acount-cp calls the function acount-cp1 with tail set to \( cp \). From a technical standpoint, the reason for defining the function acount-cp1 is that ACL2 is, by default, eager to expand non-recursive functions. Therefore, in any attempt to prove a theorem that includes the function acount-cp(cp), ACL2 will expand the function call to acount-cp1(cp, cp) for which induction will not work properly. However, this problem can be worked around by proving a more general theorem about the same function when it has two different inputs, for example acount-cp1(tail, cp), and then instantiating the theorem with tail set to \( cp \). This is a technique that is often used in ACL2 to achieve proofs of this type.
The proof that \textit{fix\_tdl} terminates is nontrivial and requires that a number of assumptions about the relationships among the indexes in the constant pool be made explicit. For instance, we can only resolve those indexes which are acyclic in \textit{cp}. Furthermore, note that \textit{fix\_tdl} calls \textit{once\_tdl} exhaustively. In contrast, \textit{once\_tdl} walks through \textit{cp} and applies \textit{rs} to every entry one time. Therefore, the conjecture of termination is really about the application of \textit{once\_tdl} under a variety of assumptions concerning acyclicity. As a result, the termination conjecture that needs to be proven is stated in Theorem 3 shown in Figure 30.

In this conjecture, the predicate \textit{ok-cp} returns \textit{t}, which represents true, if the \textit{cp} is a well-formed constant pool, i.e., \textit{cp} is a list of structures, the first component of each one of them is a natural number index and each index must not be reassigned anywhere else in the rest of structures of the \textit{cp}. The predicate \textit{acyclic-indexes} returns \textit{t} if every index in \textit{tail} is acyclic in \textit{cp}. The predicate \textit{acyclic-rhs-r} returns \textit{t} if the \textit{RHS} of every rule in \textit{rs} is acyclic in \textit{cp} and the predicate \textit{match} returns \textit{t} if there is a match between the \textit{LHS} of any rule in \textit{rs} and a specific entry in \textit{cp}. Informally, the conjecture states that: given an acyclic constant pool \textit{tail} whose indexes are acyclic in \textit{cp}, that has at least one entry that matches a rule in the set of the rewrite rules \textit{rs} and the \textit{RHS} of every rule in \textit{rs} is acyclic in \textit{cp}, then the sum of the number of steps needed to resolve the indexes of the new \textit{cp}, after applying the function \textit{once\_tdl} to it, is less that the sum of the those of the original \textit{cp}. In this manner, the termination of \textit{fix\_tdl} is proved.

\begin{verbatim}
Theorem 4 The main correctness conjecture (the ACL2-like version of Theorem 2).
\forall (p, c) : \text{wf-classfilep}(c) \land \text{acyclicp}(p, \text{get-cp}(c)) \Rightarrow \text{dref}(p, \text{index\_res\_ACL2}(c)) = \text{dref}(p, \text{get-cp}(c))

Lemma 1 The first main auxiliary lemma to the main correctness conjecture.
\forall (p, cp) : \text{acyclic-cp}(cp) \land \text{acyclicp}(p, cp) \Rightarrow \text{dref}(p, \text{index\_res}(cp)) = \text{dref}(p, cp)

Lemma 2 The second main auxiliary lemma to the main correctness conjecture.
\forall (p, rs, cp) : \text{acyclic-cp}(cp) \land \text{acyclic-rhs-r}(rs, cp) \land \text{acyclicp}(p, cp)
\Rightarrow \text{dref}(p, \text{fix\_tdl}(rs, cp)) = \text{dref}(p, cp)

Lemma 3 The function \text{index\_res\_ACL2} preserves acyclicity.
\forall (p, c) : \text{wf-classfilep}(c) \land \text{acyclicp}(p, \text{get-cp}(c)) \Rightarrow \text{acyclicp}(p, \text{index\_res\_ACL2}(c))

Lemma 4 The function \text{index\_res\_ACL2} preserves acyclicity.
\forall (p, cp) : \text{acyclic-cp}(cp) \land \text{acyclicp}(p, cp) \Rightarrow \text{acyclicp}(p, \text{index\_res}(cp))

Lemma 5 The function \text{fix\_tdl} preserves acyclicity.
\forall (p, rs, cp) : \text{acyclic-cp}(cp) \land \text{acyclic-rhs-r}(rs, cp) \land \text{acyclicp}(p, cp) \Rightarrow \text{acyclicp}(p, \text{fix\_tdl}(rs, cp))
\end{verbatim}

Figure 31: Selected main theorems and lemmas
5.7 Verification Phase

At this point we have reached the verification phase. We are now in a position to prove that the transformation mechanism defined by index_resolution is correctness-preserving with respect to \( \mathcal{M}_{dref} \). Theorem 4 shown in Figure 31 is a statement of the correctness theorem in terms of the ACL2 constructions that model index_resolution.

Lemma 1 and Lemma 2 are essentially two versions of Theorem 4. Lemma 1 is derived from Theorem 4 by unfolding the definition of index_res_ACL2. Lemma 2 is derived from Lemma 1 by unfolding index_res. In order to prove Lemma 2, one major supporting lemmas is needed. Lemma 5 asserts that the application of the rule list \( rs \) to the constant pool model \( cp \) by the function \( fix_tdl \) also results in a constant pool model that is acyclic. After proving Lemma 2, our target is to prove Lemma 1. The proof is straightforward given that Lemma 4 has been proven. Lemma 4 asserts that the application of the function index_res to a constant pool model \( cp \) will result in a “transformed” constant pool model that is still acyclic. After proving Lemma 1, the proof of the main correctness conjecture, i.e., Theorem 4, is also straightforward given that one is able to prove Lemma 3. Lemma 3 asserts that the application of the function index_res_ACL2 to a constant pool model \( cp \) will result in a “transformed” constant pool model that is still acyclic.

This concludes our summary of the discussion on the verification of the correctness of the abstracted version of index resolution presented in this paper. In its entirety, the ACL2 model of the index resolution of the SSP class loader consists of 68 functions and predicates and more that 200 main theorems and lemmas. Our current goal is complete the verification of all the activities of the SSP class loader to provide strong evidence that it behaves as intended. On the long run, we intend to build an interpreter for TL in ACL2 so that we will not only be able to verify the TL implementation of class loading activities of the SSP, but also other TL applications.

6 Related Work

In this section we discuss related work in the areas of rewriting/strategic programming as well as the verification style we are pursuing.

6.1 Rewriting and Strategic Programming

There are a number of rewriting and strategic systems where a rewrite-based implementation of the class loader core could be considered. Among these systems are ELAN [2], Stratego [21], and ASF+SDF [1]. One of the unique features of TL is its use of higher-order strategies as the mechanism to aggregate data (e.g., all indexes in a constant pool and the Utf8 data to which they resolve) and first-order strategy application as the mechanism to distribute data (e.g., index resolution) throughout a term structure (e.g., a class file or application). In contrast, both ELAN and ASF+SDF [17], use parameterization to collect and distribute data throughout a term structure. That is, parameters are added to rewrite rules which can then be passed down and applied at various points within a term structure. In a parameter-based approach, aggregations of data (e.g., all indexes in the constant pool) are typically converted into an internal representation such as a list and must be accompanied with an associated lookup function.

In contrast to parameterization, Stratego supports an approach that arguably can be considered the first-order cousin of higher-order strategy construction mechanisms of TL. Stratego is a first-order strategic programming system that has two constructs related to the higher-order strategies presented in the paper: contextual rules and scoped dynamic rewrite rules. In [19], contextual rules are used to distribute data within a term structure and can be seen as a first-order cousin of the higher-order rules presented in this paper.
In [20], an approach to the distributed data problem is taken that is similar to what we have described. Here the distributed data problem is viewed from a context-free/context-sensitive perspective. In particular, semantic relationships between portions of a term are seen as representing context-sensitive relationships. Dynamic rewrite rules are developed as a mechanism for capturing context-sensitive relationships between portions of a term. Dynamic rewrite rules are named rewrite rules that can be instantiated at runtime (i.e., dynamically) yielding a rule instance which is then added to the existing rule base. Similar to the higher-order approach taken by TL, in Stratego the program itself is the driver behind the instantiation of dynamic rule variables. The lifetime of dynamic rules can be explicitly constrained in strategy definitions by the scoping operator \{ | ... | \}.

Primary differences between our approach and the scoped dynamic rules described in [20] are the following:

1. In our approach, we view the rule base as a strategy that is created dynamically. Various combinators provide the user explicit control over the structure of this strategy.

2. Though the transient combinator has no direct analogy within scoped dynamic rewrite rules, its effects can be simulated in Stratego. However, it is somewhat unclear whether a single approach/method can be used in Stratego to simulate all the behaviors resulting from the interaction between higher-order strategies and transients.

3. The hide combinator has no analogy in Stratego.

6.2 Verification

State transition approach was investigated in [15, 16, 14, 13]. In [4], Boyer and Yu used Nqthm [3], the predecessor of ACL2, to formalize a substantial subset of a commercial microprocessor, the Motorola MC68020 [M85]. Based on this model, they were able to verify many binary machine code programs produced by commercial compilers from source code in such high-level languages as Ada, Lisp, and C. In [13], Moore also used the same approach to model Piton, an assembly programming language that is implemented on a microprocessor, the FM8502, via a compiler, an assembler, and a linker. A Piton interpreter was coded in the ACL2 logic in which given an initial state \( p_0 \) you obtain state \( p_n \) by running Piton forward \( n \) steps. However, the alternative approach is to map \( p_0 \) to down to a FM8502 state (or core image), through a function defined in ACL2, run the FM8502, and map the resulting state back up. The compiler, assembler and linker were also defined as functions in the ACL2 logic. The implementation of Piton was mechanically proved correct. In [15, 16, 14] ACL2 has been used in verifying the JVM by analyzing the bytecode produced for it. The general approach was to model a significant subset of the JVM operationally using ACL2. This model was used to execute certain Java programs by compiling them into bytecode. The model consists of a state of the JVM and state transition function for each JVM bytecode instruction in the subset. Basically, the state is a triple containing a thread table, a heap, and a class table. The transition function takes an instruction, a thread, and a state, and returns a new state that is the result of executing the given instruction on the given thread in the given state. The new state is a modification of the previous state.

In [10] an executable ACL2 model of the Java Virtual Machine (JVM) is presented. The model is referred to as M6 and represents the most complete JVM model in a series of JVM models that have been developed for many years by the ACL2 group at the University of Texas at Austin. The M6 model is derived from the C implementation of the Java Virtual Machine KVM. It supports most of the features of J2ME JVM, including class loading, class initialization, synchronization via monitors, and exception handling. The model does not support access permissions checking and floating point operations.
The M6 model is built in the specification language of the automatic theorem prover ACL2. Therefore, it represents both: a simulator that can be used for extensive validation tests and a specification that is amenable to formal reasoning.

The model of M6 is an interpreter, which represents the semantics of executing bytecode instructions in JVM, and is modelled in ACL2 as a function, called run, that takes as inputs a schedule sched and a state s. The schedule is a list of thread ids and the state is a seven-tuple consisting of a global program counter, a current thread register, a heap, a thread table, an internal class table that records the runtime representation of the loaded classes, an environment that represents the source from which classes are to be loaded, and a fatal error flag used by the interpreter to indicate an unrecoverable error. The interpreter repeatedly executes the next instruction from the thread as indicated in the schedule until no further instructions are left. One of the main contribution of this work was to formally prove an invariant of dynamic class loading in the presented JVM model. The invariant says that if class A is assignable to a slot of type class B, then, class B must be already loaded. However, the authors assumes that all symbolic references to fields and methods have already been dereferenced to their corresponding symbolic descriptors.

The work in [11] investigates the same ACL2 model of JVM that was presented in [10], namely M6, but from a verification standpoint. M6 is referred to as deep embedding as opposed to shallow embedding. In deep embedding, models of programs and their environments are formally represented in the logic of ACL2. In contrast, in a shallow embedding, a model represents the semantics of a specific program. Although shallow embedding requires less effort in building models of specific programs and typically produce simpler conjectures, a deep embedding has the advantage that it allows formal reasoning about not only a given program, but also about relations between programs. Furthermore, deep embedding allows the user to formally reason about the semantics of the programming environment itself.

One of the main difficulties that the authors identified in deep embedding is that, the reasoning process involves not only a specific program, but also the semantics of the programming language itself. To justify their use of deep embedding, even though it involves challenges both in modelling phase and verification phase, the authors investigate in the rest of the paper how to overcome some of these challenges. They introduced ACL2 concepts such as equivalence and congruence lemmas and books which are helpful in constructing some library lemmas that can be reused in proving some other theorems that is related to the properties of other programs that has similarities among them. They supported their argument by two small program examples: a simple addition program and a factorial program. They developed a library of lemmas while proving some property about the first program and stored these lemmas in a book. Then, they tried to prove some property about the second program. During this second proof, ACL2 reused some lemmas that were developed during the course of the first proof. This reuse reduced the verification effort associated with the second program.

7 Conclusion

From a conceptual standpoint, we believe that transformation provides a natural framework in which the functionality of the class loader core can be considered. However, the intricacy of data interactions as well as the structural complexity of Java class files presents a number of challenges to traditional rewriting and strategic frameworks. Foremost among these challenges is the treatment of term-specific data and its distribution throughout a term structure.

Though table construction and parameterization are techniques capable of realizing data distribution, their use departs from rewriting in its purest sense. Our research is based on the premise that higher-order rewriting provides a mechanism for dealing with the treatment and distribution of term-specific data conforming to the tenets of rewriting. In a higher-order framework, the use of such data is expressed as a
rule. Instantiation of such rules can be done using standard (albeit higher-order) mechanisms controlling rule application (e.g., traversal).

Typically, a traversal-driven application of a higher-order rule will result in a number of instantiations. If left unstructured, these instantiations can be collectively seen as constituting a rule base whose creation takes place dynamically. However, such rule bases again encounter difficulties with respect to confluence and termination. In order to address this concern the notion of strategy construction is lifted to the higher-order as well. That is, instantiations result in rule bases that are structured to form strategies.

Nevertheless, in many cases, simply lifting first-order control mechanisms to the higher-order does not permit the construction of strategies that are sufficiently refined. This difficulty is alleviated though the introduction of the transient and hide combinators. The interplay between these combinators, higher-order rules, and more traditional control mechanisms enables a the functionality of the class loader core to be concisely expressed. In spite of this, reasoning about the correctness of higher-order strategies is conceptually somewhat of a departure from the reasoning used when considering first-order rewrite rules. Our current efforts in using ACL2 reflects our initial efforts in formalizing our reasoning process in an automatable fashion. This effort involves mapping our approach to reasoning about TL strategies onto proven approaches to reasoning about software.

References


