A brief user's guide to Jedd

Ondřej Lhoták

May 25, 2004

1 Preliminaries

This mini-tutorial assumes that the reader has read the paper [LH04] about Jedd that was presented at PLDI 2004, and is available at `http://www.sable.mcgill.ca/publications/papers/#pldi2004`.

2 Example

The Jedd distribution contains a directory called `examples` containing sample Jedd code. Currently, it contains a single example, `examples/pointsto`. This is a Jedd version of the BDD-based points-to analysis from [BLQ+03].

3 Jedd source files

Source files to be processed by Jedd must have one of the extensions `.jedd` or `.java`. It is customary to use the extension `.jedd` for files containing Jedd-specific constructs, and `.java` for files containing plain Java.

Jedd files should import the package `jedd.*` from the Jedd runtime library. This package contains interface classes with methods that can be called by Jedd programs. In particular, the `jedd.Jedd` class is a singleton containing methods affecting the behaviour of Jedd in general, and `jedd.Relation` is an interface listing the methods that can be called on any Jedd relation type. Jedd files should not import the package `jedd.internal.*`.

4 Defining numberers, domains, attributes and physical domains

The first step in writing a Jedd program is to define the numberers, domains, attributes, and physical domains that will be used. A numberer is a class that generates and maintains a mapping between objects and non-negative integers, and must be implemented by the programmer. A domain is a set of objects that will form the basis of Jedd relations. Each domain must have an associated numberer for its objects. An attribute is a domain with an assigned name, used to distinguish multiple instances of a domain within the same relation. A physical domain is a set of BDD bit positions to which Jedd may map an attribute of a relation.

A numberer is a plain Java class implementing the `jedd.Numberer` interface. See the file `examples/pointsto/src/domains/IntegerNumberer.java`, which assigns to `Integer` objects the integer that is their value.

A domain is similar to a Java class, but is defined with a slightly different syntax: the class name is immediately followed by an integer constant in parentheses. See the file `examples/pointsto/src/domains/Var.jedd` for an example. The integer constant specifies how many bits are to be used to represent the domain. The maximum number of objects in the domain is 2^b, where b is the number of bits specified. Each domain must extend the `jedd.Domain` class and implement the `numberer()` method, which returns the numberer for the domain.

An attribute is defined similarly to a domain, but the integer constant number of bits is replaced with the name of the domain of the attribute. See the file `examples/pointsto/src/attributes/var.jedd` for an example with the domain `Var`. An attribute must extend the `jedd.Attribute` class. However, it should not implement its abstract method `domain()`; Jedd will implement it for you.

A physical domain is defined similarly to a domain or attribute, but the parentheses following its name are empty. See the file `examples/pointsto/src/physical_domains/V1` for an example. Each physical domain must extend the class `jedd.PhysicalDomain`.

5 Selecting a backend

Jedd currently supports four different BDD libraries as backends: BuDDy, CUDD, SableJBDD, and JavaBDD. BuDDy is the backend which has the most complete support in Jedd, which is the most tested, and which tends to perform best. BuDDy and CUDD are C libraries, so they require that their shared library (`.so` or `.dll`) files be available on the `LD_LIBRARY_PATH`. Before using Jedd in you program, you must select one of the backends by calling `jedd.Jedd.v().setBackend()`. The argument to this method should be one of `"buddy"`, `"cudd"`, `"sablejbdd"` or `"javabdd"`.

6 Selecting a physical domain ordering (optional)

By default, Jedd places the various physical domains one after the next in the BDD. For performance reasons, you may want to select a different ordering. This is done by calling `jedd.Jedd.v().setOrder()`. The first argument is an array containing the physical domains in the order in which they should appear in the BDD, from the top of the BDD to the bottom (constant nodes). To get the physical domain instance (to be placed in the array) for a given physical domain, call the method `v()` on it (see the example in `examples/pointsto/src/Prop.jedd`). To interleave the bits of several physical domains, place them in a separate array, and then include this array as an element of the main array. The `jedd.Jedd.setOrder()` method takes a second, boolean argument that determines the order of bits within each domain. Set it to `true` to order the bits from most significant to least significant when moving down the BDD from top to bottom, or to `false` for the opposite order.

7 Writing Jedd code

The Jedd grammar and explanations of its operators appear in [LH04], and are outside the scope of this guide. The paper also includes various examples of Jedd code. Refer also to the points-to analysis example in `examples/pointsto/src/Prop.jedd`.

The javadoc documentation of the (rather small) API available to Jedd programs is available in `doc/api`. In particular, this includes the `jedd.Jedd` class with methods controlling the behaviour of Jedd in general, and the `jedd.Relation` interface of methods that can be called on any relation type.

8 Compiling Jedd code

The Jedd compiler is invoked with the command `java jedd.Main`. It uses the same command-line format as Polyglot, with two additional switches for specifying the path to a SAT solver (`-s`) and a SAT core extractor (`-sc`). The simplest way to compile a project is to list all the `.jedd` files on the command line. This will compile them to `.java` files, and run `javac` on them to compile them to classfiles. The `-c` switch disables the `javac` pass. If your project consists of both `.jedd` and `.java` files, you can put them all on the command line, but be warned that Polyglot will overwrite your `.java` files unless you specify an alternate output directory with the `-d` switch.

The points-to analysis example includes a simple Ant build file which can be modified for use in other projects.

9 Using the profiler (optional)

To use the profiler, it must be enabled before the computation to be profiled begins by calling `jedd.Jedd.v().enableProfiling()`. At the end of the computation, the recorded profiling data can be written to a file in SQL format by calling `jedd.Jedd.v().outputProfile()` with a `java.io.PrintStream`. See the file `examples/pointsto/src/Prop.jedd` for an example use of the profiler.

Viewing the profile data requires an SQL database and a CGI-capable web server. The CGI scripts (found in the `profile_view` directory in the Jedd distribution) are specific to SQLite, but should work with any web server. They expect the profiling data in a database called `profile.db`, in the same directory as the scripts. This file can be generated by piping the SQL file to SQLite with the command

```
cat profile.sql | sqlite profile.db
```

(assuming the SQL file is `profile.sql`). thttpd can be started with the command:

```
/usr/sbin/thttpd -d /directory/with/cgi/scripts -p 8080 -c ’*.cgi’
```

(where `/directory/with/cgi/scripts` is replaced with the directory containing the Jedd CGI scripts from `profile_view`). This starts the web server on port 8080. To view the profiling data, point your web browser to `http://127.0.0.1:8080/main.cgi`.

References

[BLQ+03] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 103–114. ACM Press, 2003.

[LH04] Ondřej Lhoták and Laurie Hendren. Jedd: A BDD-based relational extension of Java. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. ACM Press, 2004.