# How to Verify Java Program with jStar: a Tutorial

DINO DISTEFANO
Queen Mary University of London, UK
ddino@dcs.qmul.ac.uk

MIKE DODDS
University of Cambridge, UK
Michael.Dodds@cl.cam.ac.uk

MATTHEW J. PARKINSON
University of Cambridge, UK
Matthew.Parkinson@cl.cam.ac.uk

June 14, 2010

**Abstract**

In this tutorial we describe how to use the tool jStar to verify Java programs. The document is meant to be an informal and basic introduction to jStar and many of the key notions behind this tool.

This is an evolving document, we hope to improve it along with the improvement of the tool.

## 1 Introduction

jStar [5] is an automatic verification tool based on separation logic aiming at object-oriented programs written in Java. jStar integrates theorem proving and abstract interpretation techniques. The user is required to provide specifications of pre/post conditions whereas loop invariants are synthesized automatically, minimizing the burden of verification. jStar uses succinct separation logic specification therefore pre/post specs in this language are often simpler than what would be in other formalisms.

jStar aims at those applications which present challenging problems for the other verification tools for object-oriented programs [1–3, 6, 7, 12]. Example of key challenges in verifying object-oriented programs are:

**Properties across multiple objects** where one needs to be able to express properties about several interacting objects from many different classes.

**Call-backs** Methods often make calls that in turn will call the original object. This schema can cause great difficulty when one tries to verify it using class/object invariant based approaches.

**Modular verification** Verification must deal with a single class in isolation. Adding new classes cannot invalidate the verification of pre-existing classes, and changing implementations while preserving specification should only require the re-verification of the changed code.

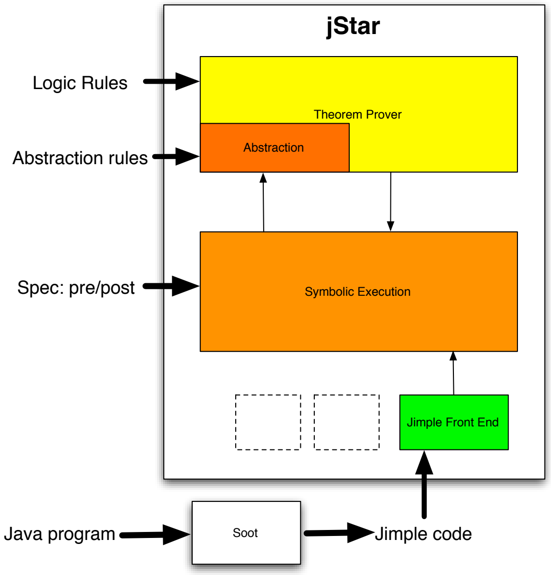

Figure 1: jStar architecture

There are two key technologies underlying jStar: the modular verification technique of *abstract predicate families* defined by Parkinson and Bierman [9,10] and the abstraction techniques for separation logic developed by Distefano *et al* [4].

## 2 Basics

jStar is implemented in OCaml. Its internal structure is depicted in Figure 1. It is composed by two main components: a theorem prover and the symbolic execution module. The prover is used by the symbolic execution during verification to decide implications. The symbolic execution module is responsible for the fixed point computation of loop invariants.

jStar has a small core language which is executed by the symbolic execution module. Different languages can be verified by jStar once a front-end translator module is provided. The current distribution contains a front end for Jimple, one the intermediate representations of the Soot toolkit [11]. We use Soot for parsing Java into Jimple. [1]

Other input files used by jStar for the verification of a Java program are:

**Pre / post-condition specifications.** This file specifies the pre and postconditions of the program's methods as well as the specifications of the methods called by it.

Details on pre / postcondition specifications are given in Section 3.

**Logic rules.** This file specifies the logical theory which is used by the theorem prover to decide entailment and other kinds of implications. The theory is specified by logical rules.

Details on logic rules are given in Section 4.

[1] Other languages can be verified by adding other front ends specific to a particular language. We encourage writing new front-ends, and if you write one and like to have it included in the official jStar distribution, please contact us.

**Abstraction rules.** This file defines the abstraction function used to ensure convergence in the fixed-point computation of loop invariants. The abstraction function is defined by means of abstraction rules that are an extension of the logical rules.

Details on abstraction rules are given in Section 5.

## 2.1 Getting started

To run jStar on a Java program we need first to compile it to the Jimple intermediated language. To do this the Soot [11] package must be installed and correctly working in your system. Consider the `LinkedList.java` program in Figure 2. Soot works on bytecode, so we first compile it to produce the binary `LinkedList.class` with the command

```
 javac LinkedList.java
```

We then run Soot to produce the Jimple version of our program:

```
java soot.Main LinkedList -f J -d .
```

At this point the file `LinkedList.jimple` should be in the current directory. To run jStar on the file `LinkedList.jimple` using the logical rules contained in `LinkedList.logic`, the abstraction rules in `LinkedList.abs` and the specification rules in `LinkedList.specs` simply type the command

```
jstar -l LinkedList.logic -a LinkedList.abs -s LinkedList.specs   \
      -f LinkedList.jimple
```

Here we are using the following jStar command-line options:[2]

**-l** specifies the file containing the *logic rules* used in this run of the verification.

**-a** specifies the file containing the *abstraction rules.*

**-s** specifies the file containing the pre/post specifications of the methods in the program.

**-f** specifies the Jimple translation of the program we want to verify.

jStar will either report that all methods have been verified (meaning that the `LinkedList.java` meets the specifications defined in `LinkedList.specs`), or report an error (indicating those methods that could not be verified), or it will not terminate (indicating that the abstraction function is not strong enough to ensure the termination of the invariant computation).

# 3 Pre/post condition specifications

In this section we gives few practical notions and conventions needed for writing specifications in jStar. For a methodological introduction on how to write specification in jStar, and a collection of tricky problems we refer the reader to [5].

The program in Figure 2 contains two classes `NodeLL` representing a generic node for a linked list and the `LinkedList.java` combining nodes in singly-linked lists. To verify `LinkedList.java` in jStar we need to provide specification for:

[2] For the complete list of options, run `jstar -help`.

```
class LinkedList
{
 private NodeLL head ;
 private NodeLL tail ;
 /* head == tail == null -> empty list */

 void create()
 {
  head=null;
  while (true) {
   NodeLL n = new NodeLL() ;
   n.next=head;
   head=n;
  }
 }

 void reverseList()
 {
  NodeLL f = null ; // finished part of list
  NodeLL r = head ; // rest of list to do
  NodeLL t = head ; // swap head and tail
  head = tail ;
  tail = t ;
  // Assert: Loop Invariant: list is split
  // between f and r, f being finished, r
  // needs to be processed.
  while ( r!=null )
  {
   t = r ; // to process t
   r = r.next ; // loop will terminate
   t.next = f ; // put on head
   f = t ; // note: t has been used
  }
 }

 void insertAtHead( String newString )
 {
  NodeLL n = new NodeLL() ;
  n.content = newString ;
  n.next = head ;
  head = n ;
  if ( tail==null ) tail = head ;
 }

 void insertAtTail( String newString )
 {
  if ( head==null )
  {
   insertAtHead( newString ) ;
  }
  else
  {
   NodeLL n = new NodeLL() ;
   n.content = newString ;
   tail.next = n ;
   tail = n ;
  }
 }

 void printList()
 {
  NodeLL n = head ;
  while ( n!=null )
  {
   n = n.next ;
  }
 }
} //end LinkedList class


class NodeLL
{
 String content ;
 NodeLL next ;
}
```

Figure 2: `LinkedList.java` code.

- all `LinkedList.java`'s methods;
- the special method `<init>()` introduced by Jimple (representing a constructor); and
- all the methods used by `LinkedList.java`'s methods;

A specification is written as:

```
method_name(list parameter types):
   { Precondition formula }
   { Postcondition formula };
```

An example specification for the constructor of class `NodeLL` is:

```
  static void <init>() :
    {  }
    {  field(@this:,<NodeLL: NodeLL next>,nil()) *
       field(@this:,<NodeLL: java.lang.String content>,nil()) };
```

The spec above tells us several syntactic conventions used by jStar.

- An empty formula is is a shorthand for $\mathbf{true} \wedge \mathsf{emp}$.
- Following the Jimple notation, the Java special variable **this** is written `@this:`
- The *points-to* predicate of separation logic is expressed by the predicate `field(object, field name, content value)`. Hence $o.f \mapsto v$ is written as `field(o,f,v)`.
- Field names must be written using their complete signature (as it is done by Jimple). Signatures in Jimple are written `<ClassName: TypeOfField NameOfField>`. Hence `<NodeLL: java.lang.String content>` stands for the `content` field of class `NodeLL` which has type `java.lang.String`.

Intuitively the meaning of `<init>`'s specification is:

> *"Given an empty heap the* `init` *method returns an instance object of the* `node` *class expressed in terms of the* `field` *predicate."*

In separation logic we would write this as the following Hoare triple:

$$\{true \wedge \mathsf{emp}\} \quad \texttt{<init>()} \quad \{\mathbf{this}.next \mapsto \mathsf{nil} * \mathbf{this}.content \mapsto \mathsf{nil}\}$$

**Predicates definition.** When writing specifications it is often useful to define predicates which we will use inside specifications. This is similar to defining objects which will be then used in the program. Predicate definitions are of the form:

```
  define name_predicate(parameter_list) = Formula ;
```

An alternative syntax accepted by jStar replaces the equal sign by the keyword `as`

```
  define name_predicate(parameter_list) as Formula ;
```

Both syntaxes can be used interchangeably. For example, for writing specifications for the `LinkedList.java` class it useful to define a predicate denoting a linked list. (Specifications of `LinkedList.java`'s class do nothing more than express properties on linked lists.)

```
  define LL(x) =
       field(x,<LinkedList: NodeLL tail>,_t)
       * field(x,<LinkedList: NodeLL head>,_h)
       * ( _t = nil() * _h = nil()
        ||  _t!=nil() * lspe(_h,_t) * NodeLL(_t,nil()));
```

In separation logic this translates to:

$$\begin{array}{rcl} LL(x) & = & \exists t', h'. \quad x.tail \mapsto t' * x.head \mapsto h' * \\ & & \qquad (t' = \mathsf{nil} * h' = \mathsf{nil} \wedge \mathsf{emp}) \vee (t' \neq \mathsf{nil} \wedge \mathsf{lseg}(h', t') * NodeLL(t', \mathsf{nil})) \end{array}$$

This specification uses the predicate $NodeLL(a, b)$ denoting an instance of the class `NodeLL` and which can be written in separation logic as:

$$NodeLL(a, b) \quad = \quad a.next \mapsto b * x.content \mapsto c'$$

The formula $LL(x)$ denotes either a list with only one element or a list with the tail pointing to a node object. With the predicate $LL(x)$ we can write easy specifications for the methods in `LinkedList`. All the specifications related to a class need do be enclosed between a `class` construct:

```
class LinkedList {
 ...
  void reverseList() :
   {  LL$(@this:) }
   {  LL$(@this:) };
 ...
}
```

Here the $ sign is used to give a dual specification standing for both a dynamic and a static specification at the same time (see Section 3.1 for details on dynamic and static specifications and the use of $).

**On logical variables.** Logical variables with names beginning with an underscore are quantified variables. Within a formula, a variable `_x` is *existentially* quantified (there is no need to write the quantifier explicitly). For example observe how `_t`, `_h` in the example above correspond to $t'$ and $h'$ in the associated separation logic formula.

However, if a variable `_x` occurs in both the pre and post-condition then it is *universally* quantified (as in Hoare's logic). For example the specification:

```
{ Val$(@this:, {content=_X})}
  get()
{ _X=$ret_var * Val$(@this:, {content=_X}) };
```

corresponds to the Hoare triple:

$$\forall \_X.\{\ Val(\mathbf{this}, \_X)\ \}\ \ \texttt{get()}\ \ \{\_X = ret\_var * Val(\mathbf{this}, \_X)\}.$$

**On initialisation.** Every specification file must contain a specification for the `<init>` method of the class `java.lang.Object`. Normally this specification has an empty pre and postcondition:

```
class java.lang.Object
{
static void <init>(): {  } {  };
}
```

## 3.1 Advanced specifications using inheritance

We illustrate how inheritance can be dealt with in jStar by mean of the following classes Cell and Recell:

```
class Cell {                     class Recell extends Cell {
    int val;                         int bak;

    void set(int x) {                void set(int x) {
        val=x;                           bak=super.get(); super.set(x);
    }                                }

    int get() {                      int get() {
        return val;                      return super.get();
    }                                }
}                                }
```

The Cell class has a val field. This is updated by the set method and its value is returned by the get method. The subclass Recell has an additional field bak, which stores the previous value the object held.

To specify the Cell class, we define a property *Val* describing a Cell's contents. This is done with the following specification, defined inside the class Cell:

```
define Val(x,content=y) as x.<Cell:int val> |-> y ;            (1)
```

This defines the property $\mathit{Val}(x, \{\text{content}{=}\,y\})$ to mean that the field *val* of object $x$ has contents $y$, *provided* $x$ is precisely of dynamic type Cell.

If $x$ is *not* of this type, then this definition does not constrain the meaning of the property. We call *Val* an *abstract predicate family* [9]. (Recall that *Val* is defined within Cell's scope.) One can view this like a method definition: the definition of a method specifies its behaviour for a single class, not for all classes.

Definition (1) also (implicitly) defines another, related, property

$$\mathit{Val}\$\text{Cell}(x, \{\text{content}{=}\,y\})$$

$\mathit{Val}\$\text{Cell}(x, \{\text{content}{=}\,y\})$ refers to the definition of *Val* for type Cell. This property is visible (and therefore can be used) both inside Cell and in any of Cell's subclasses. Since its scope covers Cell and its subclasses, the property $\mathit{Val}\$\text{Cell}(x, \{\text{content}{=}\,y\})$ is independent of the actual dynamic type of $x$. That is, it always asserts $x$'s *val* field contains $y$ no matter where it is used (either Cell or a subclass specifications) and consequently no matter what $x$'s type is (either Cell or a subtype).

We refer to this second property as the *internal property* as it corresponds precisely to the body of the definition for a particular class. Formally, we have the following two axioms:

$$\begin{array}{l}\mathrm{type}(x, \mathrm{Cell}) \Longrightarrow \\ \qquad (\mathit{Val}(x, \{\mathrm{content}{=}\, y\}) \iff \mathit{Val}\$\mathrm{Cell}(x, \{\mathrm{content}{=}\, y\}))\end{array}$$

$$\mathit{Val}\$\mathrm{Cell}(x, \{\mathrm{content}{=}\, y\}) \iff x.\mathit{val} \mapsto y$$

(Here, $\mathrm{type}(x, \mathrm{Cell})$ means $x$ is precisely of dynamic type Cell.)

The first axiom relates the internal property to the general property. Note that, this does not specify the meaning of $\mathit{Val}(x, \{\mathrm{content}{=}\, y\})$ if $x$ is not of dynamic type Cell. The second axiom specifies the internal property. This can be used to verify the class Cell, but is not available when verifying other classes. That is, other classes (including subclasses) must be independent of the specific internal definition of the predicate (in the case of $\mathit{Val}$, $x.\mathit{val} \mapsto y$), but may mention the predicate abstractly (i.e., $\mathit{Val}\$\mathrm{Cell}(x, \{\mathrm{content}{=}\, y\})$).

In jStar we must provide two kinds of specification for each method: *static* and *dynamic* [3, 10]. The static specification gives a precise specification of the code's behaviour, while the dynamic gives a more abstract view of how the method behaves. More specifically, the static specification is used for **super** and private calls and for verifying that it is sound to inherit a method. The dynamic specification is used for dynamic dispatch and hence must be implemented by all subclasses (behavioural subtyping [8]).

We can specify the behaviour of the get method by:

```
int get() static:
  {@this:.val |-> _X }
  { _X=return * this.val|-> _X  };
int get() dynamic:
  { Val$(@this:, {content= _X }) }
  { _X= return * Val$(@this:, {content= _X }) };
```

The first specification, the static specification, describes precisely how the method updates the *val* field.[3] The precondition

$$\mathbf{this}.\mathit{val} \mapsto X$$

specifies that the *val* field of **this** has the value $X$ (a logical variable). The postcondition

$$X = \mathbf{return} * \mathbf{this}.\mathit{val} \mapsto X$$

specifies that the field still has the same value, and that this value is returned, $X = \mathbf{return}$.

The second specification, the dynamic specification, describes how the method alters the more abstract Val property, rather than the concrete fields. This enables subclasses to satisfy this specification while changing the concrete behaviour, that is, modifying different fields.

**Notation 1** *In* jStar *only the* `static` *qualifier needs to be explicitly stated. Without it a method specification is assumed to be dynamic.*

[3] In this case, the update of the method get() is the identity function.

The static specification we have given for get is very concrete, since it is expressed in terms of the *val* fields. In order for subclasses of Cell to use it, it must be restated more abstractly using the *Val*\$Cell property.

```
int get() static :
  { Val$(this, {content=_X})}
  { _X=return * Val$(@this:, {content=_X}) };
```

This specification describes for all subclasses precisely what the method body does, but it does not reveal the fields that are actually modified.

As this pattern of specification is common, jStar provides a shorthand, which defines both the dynamic specification given above, and the second static specification, with the single specification.

```
int get() :
  { Val$(this, {content=_X})}
  { _X=return * Val$(@this:, {content=_X}) };
```

A property postfixed by \$ is interpret by jStar as the standard property in the dynamic specification, that is without the \$, and as the internal property for the current class in the static specification, in this case *Val*\$Cell. Hence, both static and dynamic specifications for set can be provided with only the following:

```
void set(int x) :
  { Val$(this, {content=_})}
  { Val$(this, {content=x}) };
```

The underscore stand for some value. Remember that the behaviour of the constructor must be specified:

```
void <init>() : { } { Val$(this,{content=_ }) };
```

This specification stipulates that constructing an object gives the property

$$\exists\hat{X}.\,\mathit{Val}(\mathbf{this},\{\text{content}=\hat{X}\})$$

and that a call to a subclass's **super** constructor results in the internal property

$$\exists\hat{X}.\,\mathit{Val}\$\text{Cell}(\mathbf{this},\{\text{content}=\hat{X}\})$$

This enables subclasses to use this property without knowing its meaning.

We can now give the complete specification of the class Cell :

```
class Cell {
  define Val(x,content=y) as x.<Cell: int val> |-> y ;

  void <init>() : { } { Val$(this,{content=_ }) };

  int get() :
    { Val$(this, {content=_X})}
    { _X=return * Val$(@this:, {content=_X}) };

  void set(int x) :
    { Val$(this, {content=_})}
    { Val$(this, {content=x}) };
}
```

**Recell.** Now, let's consider Recell, a subclass of Cell. First of all we must define what the *Val* property means for the Recell class.

```
 define Val(x,content=y;oldcon=z) as  Val$Cell(x, {content=y})
                                      * x.<Recell: int bak>|-> z
```

If $x$ is of type Recell, this defines $\mathit{Val}(x, \{\text{content}{=}\, y;\ \text{oldcon}{=}\, z\})$ as the composition of the property associated to *Val* from the superclass Cell, and the new field oldcon with value $z$. As we have defined the property with an additional labelled parameter, oldcon, the system will also provide a meaning to the property with only the content parameter, for use when casting from Cell to Recell. In effect, we have width subtyping on the labelled parameters, where missing parameters are existentially quantified. Hence, this defines $\mathit{Val}(x, \{\text{content}{=}\, y\})$ for the Recell class as

```
Val(x, content=y) as Val$Cell(x,content= y )
                            * x.<Recell: int bak>|-> _z
```

Importantly, by containing the *Val*$Cell property it allows the Recell to make super calls to the Cell's methods, and also inherit methods, as the precondition of *Val*$Cell can be provided for the calls. The specifications for Recell is as follows:

```
class Recell {
  define Val(x,content=y;oldcon=z) as Val$Cell(x, {content=y})
                                      * x.<Recell: int bak>|-> z;

  void <init>() : { } { Val$(this, {content=_x; oldcon=_z}) };

  int get() :
  { Val$(this, {content=_X; oldcon=_Y})}
  { _X=return * Val$(this, {content=_X; oldcon=_Y}) };

  void set(int x) :
  { Val$(@this:, {content=_X; oldcon=_})}
  { Val$(@this:, {content=x; oldcon=_X}) };
}
```

We must verify that these specifications are valid behavioural subtypes, which follows from width subtyping of labelled parameters.

## 4 Logic Rules

jStar's prover needs logic rules for deciding entailments. There are two kinds of rule:

**Pre-defined rules** are very general structural rules which need to be used with any kind of logic systems. These rules are hard-coded into the prover.

**User-defined rules** are problem-specific rules. They define a particular logic theory needed to solve the problem at hand. They must be provided by the user. Note that jStar does not check that user-defined rules are consistent. Therefore, the user should use special care in ensuring the soundness of rules.

In this tutorial we focus on the second type of rules. Rules are introduced with the syntactic construct `rule`:

```
rule my_rule :
 | Conclusion-L |- Conclusion-R
if
 SubtFPre | Premise-L |- Premise-R
```

The keywords `if` separates the conclusion from the premise. Mathematically this syntax corresponds to

$$\frac{H_s * \texttt{SubtFPre} \;\mid\; H_a * \texttt{Premise-L} \;\vdash\; H_g * \texttt{Premise-R}}{H_s \;\mid\; H_a * \texttt{Conclusion-L} \vdash\; H_g * \texttt{Conclusion-R}}\;\texttt{my\_rule}$$

for some $H_s, H_a, H_g$. The rules work with sequents of the form

$$\Sigma \;\mid\; F_1 \;\vdash\; F_2$$

We call $F_1$ the *assumed formula*, $F_2$ the *goal formula*, and $\Sigma$ the *subtracted (spatial) formula*. The semantics of a judgement is:

$$F_1 * \Sigma \Longrightarrow F_2 * \Sigma$$

The subtracted formula $\Sigma$ is used to allow predicates to be removed from both sides without losing information. Notice that rules are *implicitly framed* — the semantics of the rule includes the contexts $H_s$, $H_a$ and $H_g$. This means rules can fire inside large context heaps. Hence the following example rule

```
rule numeric_eq_right :
   | |- numeric_const(?x) = numeric_const(?y)
if
   | |- ?x=?y
```

encodes the implication:

$$\frac{H_s \mid H_a \vdash x = y * H_g}{H_s \mid H_a \vdash \texttt{numeric\_const}(x) = \texttt{numeric\_const}(y) * H_g}\;\texttt{rule numeric\_eq\_right}$$

Informally, this rule says: in any context $H_s, H_a, H_g$, if we want to prove the equality $\texttt{numeric\_const}(x) = \texttt{numeric\_const}(y)$ then we can prove the simpler fact $x = y$ in the same context.

**Notation 2** *As specified in previous sections, existentially quantified variables start with an underscore. Variables starting with a question mark as in the rule above are used for denoting any expressions (i.e., quantified variables, non-quantified variables, constant values, etc.).*

**Without clauses.** Consider the following rule:

```
rule field_remove1:
| field(?x,?f,?y) |-  field(?x,?f,?t)
without
  ?y!=?t
if
  field(?x,?f,?y) | |- ?y=?t
```

This rule states that if we have a field `?f` for the object `?x` in both the assumed and goal formula of an implication, to then move the field to the subtracted formula, and to add the proof obligation that the values `?y` and `?t` are the same to the goal formula.

The clause `without` prevents the firing of the rule if the inequality `?y!=?t` is present in the assumed formulae of the conclusion. In other words, it prevents the rule from firing when the assumed formula already says that the fields have different values. This clause is used to avoid the rule being applied an infinite number of times.

We also allow the clause

```
without
   WO-L |- WO-R
```

to allow the user to specify on which side the formula does not occur.

**Contradictions.** A rule with an empty premise can be used to force the termination of the proof search when we reached a contradictory statement. For example consider the rule `field_nil_contradiction`:

```
rule field_nil_contradiction :
 | field(nil(),?f,?z) |-
if
```

This rule fires when a field for `nil()` is assumed to exist. Clearly, in that case we have found a contradiction, and `field_nil_contradiction` will stop the proof search.

**With.** We allow the user to specify a match to the obligation that may be either in the already framed formula, or in the unmatched formula.

```
rule my_rule :
 With-L | Conclusion-L |- Conclusion-R
if
 SubtFPre | Premise-L |- Premise-R
```

Mathematically this syntax corresponds to

$$\dfrac{\begin{array}{c} H_s * \texttt{SubtFPre} \mid H_a * \texttt{Premise-L} \ \vdash H_g * \texttt{Premise-R} \\ H_s * H_a \vdash \texttt{With-L} \end{array}}{H_s \mid H_a * \texttt{Conclusion-L} \vdash H_g * \texttt{Conclusion-R}} \ \texttt{my\_rule}$$

for some $H_s, H_a, H_g$.

We can improve the previous rule by using the with formula.

```
rule field_nil_contradiction :
 field(nil(),?f,?z) | |-
if
```

This allows contradictions to be found in the already framed heap.

## 5 Abstraction Rules

Abstraction rules are similar in syntax to logic rules, but they have a different purpose. They help the convergence of jStar's computation when dealing with infinite domains. The idea is that an abstraction rule simplifies a formula so that it remains within a restricted class of heaps for which simplification is known to converge. The simplification is done by *rewriting*. In general, simplification is achieved by removing existentially quantified variables that do not appear anywhere else in the heap.

Abstraction rules are of the form:

$$\frac{\text{condition}}{H * H' \leadsto H * H''}\ \text{(Abs Rule)}$$

Heap $H'$ is replaced by $H''$ if the condition holds. $H''$ should be more abstract (simpler) than $H'$ since some unnecessary information is removed (abstracted away). Heap $H$ is an arbitrary context preserved by the abstraction rule.

Given a set of abstraction rules, jStar tries to use any rules that can be applied to a heap. When no rules are applicable the resulting heap is maximally abstracted. Note that to ensure termination of this strategy, abstraction rules should be chosen so that each application strictly simplifies the heap. Moreover, for soundness, the abstraction rules *must* be true implications in separation logic. In other words, it must hold that:

$$H * H' \Longrightarrow H * H''$$

When designing abstraction rules checking this implication gives a easy sanity condition for the soundness of the resulting fixed-point computation.

The syntax for abstraction rules uses the `abstraction` construct and it is as follows:

```
abstraction my_abstraction_rule:
InitialFormula ~~>   AbstractFormula
where  Condition
```

The semantics of such a formula is:

$$\frac{\texttt{Condition}}{H * \texttt{InitialFormula}\ \leadsto\ H * \texttt{AbstractFormula}}\ \texttt{my\_abstraction\_rule}$$

The following is an abstraction rule for combining list predicates:

```
abstraction ls_ls:
 ls(?x,_x) * ls(_x,nil()) ~~>  ls(?x,nil())
where
  _x notincontext;
  _x notin ?x
```

Condition `_x notincontext` says that `_x` does not occur syntactically in the rest of the heap, i.e., in $H$. The condition `_x notin ?x` says that `?x` cannot be instantiated to `_x`. The rule would be written mathematically as:

$$\frac{\_x \notin Var(H) \cup \{y\}}{H * ls(y, \_x) * ls(\_x, \mathsf{nil}) \quad \rightsquigarrow \quad H * ls(y, \mathsf{nil})}$$

This rule abstracts away from the useless quantified variable `_x`. It reduces the original heap involving two list-segment predicates to a simpler heap with only one such predicate. ✯

**Acknowledgment.** Distefano and Parkinson are both supported by Royal Academy of Engineering research fellowships. Dodds is supported by EPSRC.

## References

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec$^\sharp$ programming system: An overview. In *CASSIS*, pages 49–69, 2005.

[2] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and Erik Poll. An overview of JML tools and applications. In *FMICS*, pages 73–89, 2003.

[3] W.-N. Chin, C. David, H.H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In *Proceedings of POPL*, 2008.

[4] D.Distefano, P.O'Hearn, and H.Yang. A local shape analysis based on separation logic. In H. Hermanns and J. Palsberg, editors, *In 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006)*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.

[5] Dino Distefano and Matthew J. Parkinson. jstar: towards practical verification for java. In *OOPSLA*, pages 213–226, 2008.

[6] M.B. Dwyer, J.Hatcliff, M.Hoosier, and Robby. Building your own software model checker using the bogor extensible model checking framework. In *17th Conference on Computer-Aided Verification (CAV 2005)*, volume 3576. Springer, 2005.

[7] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[8] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, 1994.

[9] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.

[10] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of POPL*, pages 75–86. ACM, 2008.

[11] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[12] X.Deng, J.Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE 2006*, pages 157–166. IEEE, 2006.

# A Syntax of logic

```
iop      ::=  +
          |   -
          |   <<
          |   >>
          |   >>>
          |   %
          |   /
          |   &


bop      ::=  <
          |   >
          |   =
          |   !=
          |   <=
          |   >=

expression ::= (expression iop expression)
            |  field_signature
            |  integer_const
            |  - integer_const

variable ::=  name
          |   ? name
          |   _ name

formula  ::=  (formula)
          |   formula * formula
          |   formula || formula
          |   expression bop expression
          |   expression . expression |-> expression
          |   name ( expression , ... , expression )
          |   ! name ( expression , ... , expression )
          |   expression : name
          |   variable
          |   formula | formula                  [deprecated, lhs must be pure]
          |   False
```

# B A full Example

In this section we report the specifications, logic, and abstraction rules to analyze the `LinkedList` class reported in Figure 2.

## B.1 Specifications for the LinkedList class

```
class java.lang.Object
{
void <init>() static: { } { };
}

class NodeLL
{
  void <init>() static:
    { }
    { field(@this:,<NodeLL: NodeLL next>,nil()) *
        field(@this:,<NodeLL: java.lang.String content>,nil())  };
}

class LinkedList
{
  define LL(x) =
       field(x,<LinkedList: NodeLL tail>,_t)
       * field(x,<LinkedList: NodeLL head>,_h)
       * ( _t = nil() * _h = nil()
        ||  _t!=nil() * lspe(_h,_t) * NodeLL(_t,nil()));

  void <init>() :
   { }
   { field(@this:,<LinkedList: NodeLL tail>,nil())
       * field(@this:,<LinkedList: NodeLL head>,nil()) };

  void create() :
   { field(@this:,<LinkedList: NodeLL tail>,nil())
       * field(@this:,<LinkedList: NodeLL head>,nil()) }
   { LL$(@this:) };

  void reverseList() :
   { LL$(@this:) }
   { LL$(@this:) };

  //This is called by insertAtTail with dynamic dispatch.
  void insertAtHead(java.lang.String) :
   { LL$LinkedList(@this:) }
   { LL$LinkedList(@this:) };
  void insertAtTail(java.lang.String) :
   { LL$(@this:) }
   { LL$(@this:) };

  void printList() :
   { LL$(@this:) }
   { LL$(@this:) };
}
```

### B.2 Logic Rules for LinkedList

```
import "../../src/prover/tests/field_logic";
import "../../src/prover/tests/boolean_logic";

/***************************************
 *  This file defines
 *
 *  NodeLL
 *  ls
 *  lspe
 *
 ***************************************/
rule NodeLL_not_nil:
  NodeLL(nil(),?y) |  |-
if

rule ls_not_nil:
  ls(nil(),?y) |  |-
if

rule NodeLL_not_nil:
  lspe(nil(),?y) |  |-
if
  | ?y=nil()  |-

rule NodeLL_not_nil:
  NodeLL(?x,?y) |  |- ?x!=nil()
if
  |  |-

rule NodeLL_not_eq:
  NodeLL(?x,?y) * NodeLL(?x,?w) |  |-
if

/*************************************
 * Rule for unpacking Nodell
 *
 *  These rules could potentially cycle forever
 *  but due to their order cannot.
 *************************************/

//Unroll NodeLL if we are looking for its next field
rule field_remove1a:
  |   NodeLL(?x,?e1) |-  field(?x,"<NodeLL: NodeLL next>",?e2)
if
    field(?x,"<NodeLL: NodeLL next>",?e1)
|  field(?x,"<NodeLL: java.lang.String content>",_w)  |-  ?e1=?e2

//Unroll NodeLL if we are looking for its content field
rule field_remove1b:
  |   NodeLL(?x,?e1) |-  field(?x,"<NodeLL: java.lang.String content>",?e2)
if
     field(?x,"<NodeLL: java.lang.String content>",w)
|   field(?x,"<NodeLL: NodeLL next>",?e1)   |-  w=?e2

//Roll up a complete NodeLL if we have both fields.
rule field_remove2:
  |  field(?x,"<NodeLL: NodeLL next>",?e1) *
      field(?x,"<NodeLL: java.lang.String content>",?z) |-
if
  |  NodeLL(?x,?e1) |-


/*************************************
 * Simple subtraction rules
 *************************************/
rule ls_unroll_exists :
|  ls(?x,?y) |- | field(?x,?w,?Z)
if
|  NodeLL(?x,_fooz) * lspe(_fooz,?y)  |-  field(?x,?w,?Z)

rule ls_ls_match :
  ls(?z,?w) |  ls(?x,?y) |-  ls(?x,?z)
if
  ls(?x,?y) |  |-  lspe(?y,?z)

rule ls_NodeLL_match :
  NodeLL(?z,?w) |  ls(?x,?y) |-  ls(?x,?z)
if
  ls(?x,?y) |  |-  lspe(?y,?z)

rule ls_field_match :
  field(?z,?f,?w) | ls(?x,?y) |-  ls(?x,?z)
if
```

```
  ls(?x,?y) |  |- lspe(?y,?z)

rule nl_ls_match :
  ls(?z,?w) |  NodeLL(?x,?y) |-  ls(?x,?z)
if
  NodeLL(?x,?y) |  |-  lspe(?y,?z)

rule nl_NodeLL_match :
  NodeLL(?z,?w) |  NodeLL(?x,?y) |-  ls(?x,?z)
if
  NodeLL(?x,?y) |  |- lspe(?y,?z)

rule nl_field_match :
  field(?z,?f,?w) | NodeLL(?x,?y) |-  ls(?x,?z)
if
  ls(?x,?y) |  |-  lspe(?y,?z)

rule lspe_left :
  |  lspe(?x,?y) |-
if
  |  ls(?x,?y) |-  ;
  | ?x=?y  |-

rule lspe_right :
  |  |-  lspe(?x,?y)
if
  |  |-  ls(?x,?y)
or
  |  |- ?x=?y

/************************************
 * rules for contradictions
 ************************************/
rule ls_field_contradiction1 :
ls(?x,?t) * field(?x,"<NodeLL: NodeLL next>",?z) |  |-
if

rule ls_field_contradiction2 :
ls(?x,?t) * field(?x,"<NodeLL: java.lang.String content>",?z) |  |-
if

rule ls_node_contradiction :
ls(?x,?t) * NodeLL(?x,?z) |  |-
if

rule ls_ls_contr :
ls(?x,?t) * ls(?x,?z) |  |-
```

```
if

rule ls_ls_contr :
 |  |- | ls(?x,?t) * ls(?x,?z)
if
 |  |- x!=x
```

### B.3 Abstraction Rules for LinkedList class

```
//Roll up a complete NodeLL if we have both fields.
abstraction field_remove2:
   field(?x,"<NodeLL: NodeLL next>",?e1) * field(?x,"<NodeLL: java.lang.String content>",_y)
~~>
   NodeLL(?x,?e1)

abstraction type_remove :
  !type(?x,"NodeLL") * NodeLL(?x,?y)  ~~> NodeLL(?x,?y)

/****************nil() rules*****************/

abstraction nil_neq_remove_nodell :
  ?x != nil() * NodeLL(?x,?y) ~~> NodeLL(?x,?y)

abstraction nil_neq_remove_field :
  ?x != nil() * field(?x,?f,?y) ~~> field(?x,?f,?y)

abstraction nil_neq_remove_ls :
  ?x != nil() * ls(?x,?y) ~~>  ls(?x,?y)

/*************** Junk Rules ******************/

abstraction garbage_garbage :
  Garbage * Garbage  ~~>  Garbage


abstraction gb1_ls :
 ls(_x,?e) ~~>  Garbage
where
   _x notincontext






abstraction gb1_ast :
  Ast(_x,?e) ~~>
where
```

```
   _x notincontext


abstraction gb1_pto :
  NodeLL(_x,?e) ~~>  Garbage
where
   _x notincontext


abstraction gb2_ls_ls:
  ls(_x,_y) * ls(_y,_x) ~~>  Garbage
where
   _x,_y notincontext


abstraction gb2_ls_pto:
  ls(_x,_y) * NodeLL(_y,_x) ~~>  Garbage
where
   _x,_y notincontext


abstraction gb2_pto_pto:
  NodeLL(_x,_y) * NodeLL(_y,_x) ~~>  Garbage
where
   _x,_y notincontext

/*************** End Junk Rules *******************/
/*************** Abs1 Rule *******************/
abstraction ls_ls:
 ls(?x,_x) * ls(_x,nil()) ~~> ls(?x,nil())
where
  _x notincontext;
  _x notin ?x


abstraction ls_pto:
  ls(?x,_x) * NodeLL(_x,nil()) ~~>   ls(?x,nil())
where
  _x notincontext;
  _x notin ?x




abstraction pto_ls:
  NodeLL(?x,_x) * ls(_x,nil()) ~~>  ls(?x,nil())
where
```

```
  _x notincontext;
  _x notin ?x


abstraction pto_pto:
  NodeLL(?x,_x) * NodeLL(_x,nil()) ~~>  ls(?x,nil())
where
  _x notincontext;
  _x notin ?x

/*************** End Abs1 Rule *******************/

/*************** Abs2 Rule *******************/
abstraction ls_ls_ls:
   ls(?x,_x) * ls(_x,?y) * ls(?y,?z) ~~>   ls(?x,?y) * ls(?y,?z)
where
  _x notincontext;
  _x notin ?x;
  _x notin ?y;
  _x notin ?z


abstraction ls_ls_pto:
   ls(?x,_x) * ls(_x,?y) * NodeLL(?y,?z) ~~>   ls(?x,?y) * NodeLL(?y,?z)
where
  _x notincontext;
  _x notin ?x;
  _x notin ?y;
  _x notin ?z


abstraction ls_pto_ls:
   ls(?x,_x) * NodeLL(_x,?y) * ls(?y,?z) ~~>   ls(?x,?y) * ls(?y,?z)
where
    _x notincontext;
  _x notin ?x;
  _x notin ?y;
  _x notin ?z






abstraction ls_pto_pto:
   ls(?x,_x) * NodeLL(_x,?y) * NodeLL(?y,?z) ~~>   ls(?x,?y) * NodeLL(?y,?z)
where
```

```
  _x notincontext;
  _x notin ?x;
  _x notin ?y;
  _x notin ?z


abstraction pto_ls_ls:
   NodeLL(?x,_x) * ls(_x,?y) * ls(?y,?z) ~~>   ls(?x,?y)  * ls(?y,?z)
where
  _x notincontext;
  _x notin ?x;
  _x notin ?y;
  _x notin ?z


abstraction pto_ls_pto:
   NodeLL(?x,_x) * ls(_x,?y) * NodeLL(?y,?z) ~~>   ls(?x,?y)  * NodeLL(?y,?z)
where
  _x notincontext;
  _x notin ?x;
  _x notin ?y;
  _x notin ?z


abstraction pto_pto_ls:
   NodeLL(?x,_x) * NodeLL(_x,?y) * ls(?y,?z) ~~>   ls(?x,?y) * ls(?y,?z)
where
  _x notincontext;
  _x notin ?x;
  _x notin ?y;
  _x notin ?z


abstraction pto_pto_pto:
   NodeLL(?x,_x) * NodeLL(_x,?y) * NodeLL(?y,?z) ~~>   ls(?x,?y) * NodeLL(?y,?z)
where
  _x notincontext;
  _x notin ?x;
  _x notin ?y;
  _x notin ?z
```