Rigorous Software Development
CSCI-GA 3033-009

Instructor: Thomas Wies

Spring 2013

Lecture 8

Run-time Checking vs. Verification

- Runtime Assertion Checking
  - finds errors at run time,
  - tests for violation during execution,
  - requires appropriate test cases.
- Verification
  - finds errors at compile time,
  - proves that there is no violation
  - high degree of confidence,
  - often requires additional annotations/proof guidance.

Dafny

- Dafny is an object-oriented programming language and verifying compiler developed at Microsoft Research
- Compiles to Microsoft .NET
- Compiler statically checks:
  – absence of runtime errors
  – termination of loops/method calls
  – correctness of user-defined contracts
- Project website:
  http://research.microsoft.com/en-us/projects/dafny/

Dafny Language

- Object-based language
  - generic classes, no subclassing
  - object references, dynamic allocation
  - sequential control
- Built-in specifications
  - pre- and postconditions
  - loop invariants, inline assertions
  - termination
  - framing
- Specification support
  - sets, sequences, algebraic datatypes
  - user-defined functions
  - ghost variables

Top-Level Grammar

- Program ::= Type*
- Type ::= Class | Datatype
- Class ::= **class** Name { Member* }
- Member ::= Field | Method | Function
- Datatype ::= **datatype** Name { Constructor* }

Generic (that is, accepts type parameters)

Methods

- A method is declared in the following way:

```
method Abs(x: int) returns (y: int)
{
  if (x < 0) { return -x; }
  else { y := x; }
}
```

- Note that the return parameter is declared explicitly.

Pre- and Postconditions

- Pre and postcondition are specified using `requires` and `ensures` clauses like in JML
- Example:

```
method MultipleReturns(x: int, y: int)
returns (more: int, less: int)
  requires 0 < y;
  ensures less < x < more;
{
  more := x + y;
  less := x - y;
}
```

Functions

- classes can also define functions and predicates
- predicates are functions with return type bool
- functions are like pure ghost methods in JML and can be used inside contracts
- like pure methods, functions
  - must not have side effects
  - must always terminate
- both properties are checked by the verifier

Functions: Example

```
function fib(n: nat) : nat
{
  if (n < 2) then n
  else fib(n-2) + fib(n-1)
}

method computeFib(n: nat) returns (m: nat)
  ensures m == fib(n);
{ ... }
```

Arrays and Quantification

- Dafny has built-in generic arrays.
- Arrays can be `null` and have a built-in length function.
- Example:

```
predicate sorted(a: array<int>)
  requires a != null;
{
  forall j, k :: 0 <= j < k < a.Length ==>
                        a[j] <= a[k]
}
```

Ghost Fields and Ghost Methods

- Dafny supports ghost fields and ghost methods but not model fields.
- Model fields can be emulated using ghost fields and functions/predicates.
- Functions and predicates are ghost by default.
- Dafny has no inbuilt support for class invariants.
- Class invariants can be encapsulated in predicates that are explicitly conjoined to pre/postconditions of methods.

Example: Array Sets

```
class ArraySet<T(==)> {
  var values : array<T>;
  var size : int;
  ghost var content : set<T>;

  predicate Valid() { /* relates values and content */ }

  method add(x: T) returns (b: bool)
    requires Valid();
    ensures Valid();
    ensures b ==> content == old(content) + {x};
    ensures !b ==> content == old(content);
  { ... }
}
```

Useful Specification Constructs

- Sets
  - `var s0 := {1, 2, 3}; // finite sets`
  - `var s1 := s0 + {4, 5}; // set union`
  - `var s2 := s0 * {1, 4}; // set intersection`
  - `var s3 := (set x | 0 <= x < 5); // comprehension`
- Sequences (functional lists)
  - `var s0 := [1, 2, 3, 4, 5]; // finite sequence`
  - `var e := s0[0]; // indexed access`
  - `var s1 := s0[..|s|-1]; // slice`
  - `var s2 := s0 + s1; // concatenation`

How Dafny works: Modular Checking

- The Dafny verifier checks each method in each class in isolation.
- Each method body is transformed into straight-line code with inlined specs, but with all method calls and loops eliminated.
- Straight-line code is then transformed into logical formulas that are given to an automated theorem prover.

`assume` and `assert`

The basic specifications in Dafny are `assume` and `assert`.

```
assume this.next != null;
this.next.prev := this;
assert this.next.prev == this;
```

- Dafny proves that if the `assume` statement holds in the pre-state, the `assert` statement holds in the post-state.
- Such a triple of specification and code is called Hoare triple.

Checking for Runtime Errors

To check for runtime errors Dafny automatically inserts appropriate `assert` statements:

```
a[x] := 0;
```

becomes

```
assert a != null && 0 <= x < a.Length;
a[x] := 0;
```

Caution with assume

assume statements can be useful for debugging specifications but should be avoided otherwise.

Never assume something that is not true, otherwise the verifier will be able to prove anything:

```
var a := new int[3];
assume a.Length > 3;
a[-3] := 2;
```

```
> dafny BadAssume.dfy
Dafny program verifier finished with 1 verified,
0 errors.
```

Inlining `requires` and `ensures`

The method contract is just translated into assume and assert statements:

```
method m(n: int) returns (m: int)
  requires n > 0;
  ensures m == n * n;
{
  body
}
```

becomes

```
assume n > 0;
body
assert m == n * n;
```

Eliminating Method Calls

And if method `m` is called, the roles of `assume` and `assert` are interchanged:

```
...
y := m(x);
...
```

becomes

```
...
assert x > 0;
y := m_x; // m_x fresh variable
assume y == x*x;
...
```

Handling Loops

- Dafny cannot know at compile-time how often a `while` loop is executed.
- However, the verifier needs to consider **all** possible paths through the program.
- Loop invariants enable the verifier to eliminate all loops in the program by using induction.
- A loop invariant is a Boolean expression that
  - holds before the loop is entered for the first time
  - is maintained by each iteration of the loop

Adding Loop Invariants

```
method computeFib(n: nat) returns (m: nat)
  ensures m == fib(n);
{
  var i := 0;
  var k := 1;
      m := 0;
  while (i < n)
  {
    m, k := k, m + k;
    i := i + 1;
  }
}
```

```
> dafny Fibonacci.dfy
... A postcondition might not hold on this return path ...
```

Adding Loop Invariants

Loop invariants can be annotated using `invariant` expressions.

```
method computeFib(n: nat) returns (m: nat)
  ensures m == fib(n);
{
  var i := 0;
  var k := 1;
      m := 0;
  while (i < n)
    invariant 0 <= i <= n;
    invariant k == fib(i+1) && m == fib(i);
  {
    m, k := k, m + k;
    i := i + 1;
  }
}
```

Termination and Ranking Functions

- Dafny proves that all loops and (recursive) method and function calls terminate.
- The termination argument can be provided in the form of a ranking function.
- A ranking function (aka variant) is a function that
  - maps program states into some well-founded domain (e.g. the natural numbers)
  - decreases with every loop iteration / recursive call
- Programmers can provide ranking functions using `decreases` expressions.
- Dafny checks that these expressions are indeed ranking functions.

Ranking Functions: Example

```
var i := 0;
while (i < n)
  invariant i <= n;
  decreases n - i;
{
  i := i + 1;
}
```

- In many cases, Dafny is able to infer an appropriate `decreases` expression automatically.

Lexicographic Ranking Functions

- Dafny also supports lexicographic ranking functions
- Example: Ackermann function

```
function ack(m: nat, n: nat): nat
  decreases m, n;
{
  if m == 0 then n + 1
  else if n == 0 then ack(m - 1, 1)
  else ack(m - 1, ack(m, n - 1))
}
```

Either `m` decreases or `m` remains the same and `n` decreases.

Framing

- Functions and methods need to specify their memory footprint, i.e., the locations they might access or modify.
- A set of memory locations is called a frame.
- Frame conditions:
  - `reads S;`
    specifies that a function reads only locations in frame S
  - `modifies S;`
    specifies that a method modifies only locations in frame S
- Functions may read only those locations specified by their **`reads`** clauses.
- Methods may access any location but may only modify those locations specified by their **`modifies`** clauses.

Example of reads clause

```
predicate sorted(a: array<int>)
  requires a != null;
  reads a;
{
  forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}

method BinarySearch(a: array<int>, key: int)
returns (index: int)
  requires a != null && sorted(a);
  ensures ...
{
  ...
}
```

Predicate `sorted` may read any cell of array `a`.

There are limits to what Dafny can prove

```
predicate isPrime (x: nat)
{ x > 1 && forall y :: 1 < y < x ==> x % y != 0 }
predicate isOdd (x: nat) { x % 2 != 0 }
ghost method VinogradovsTheorem()
  ensures exists k : nat ::
    forall x :: x >= k && isOdd(x) ==>
      exists y1 : nat, y2 : nat, y3 : nat ::
        isPrime(y1) && isPrime(y2) && isPrime(y3) &&
        x == y1 + y2 + y3;
{ }
```

```
> dafny Vinogradov.dfy
Dafny program verifier finished with 2 verified, 1 error.
```

Dealing with Incompleteness

Common sources of incompleteness

- quantifiers (in particular, if nested and alternating)
  `exists ... :: forall ... :: exists :: ...`
- non-linear integer arithmetic
- properties that require induction proofs

Often, problems with incompleteness can be resolved by guiding the proof search, e.g. by

- inserting intermediate assertions,
- providing witnesses for existential quantifiers,
- making induction explicit.

Demos

- Fibonacci numbers
- Binary search
- Array sets
- Schorr-Waite algorithm

Many more examples included in the Boogie source code distribution.