Mouldable Programming

Séminaire Performance et Généricité

2008-04-30

Magne Haveraaen
University of Bergen

Brittle Programs

Currently programs are brittle

- **Programming is slow and errorprone**
  - **program specifications are impresise**
  - **program architecture not well documented**
  - **programming languages are (semantically) too complex**
  - **lack of connection between specification and code**
  - **little reuse of software components**
- **Maintenance is a (witch) craft**
  - **documentation / specification out of date**
  - **modifications break program structure**
  - **support tools are rudimentary**

Programs break if you touch them

Mouldable Programming

Try to make programs tough and mouldable

- **precise notions and relationships between them**
  - **interface (signature)**
  - **specification**
  - **program (model):**
    semantical and algorithmic properties
  - **a model *satisfies* a specification**
- **pragmatics**
  - **one formal framework:**
    reasoning
    ***mathematical* program constructions (functors)**
  - **unified syntactic framework: support tools**

Taking the brittle out of programming

Contents

Too much of everything?

- **parameter passing modes**
- **method declaration and use:**
  expression-terms versus method calls
- **undefinedness is a many-splendored thing**
- **typing regimes**

Mould the signature notion according to needs

Parameter passing modes in C++

```
void m (const & A, & B);
void m (const A, & B);
void m (A, & B);

B m (const & A);
B m (const A);
B m (A);

const & B m (const & A);
...
```

Parameter passing modes I

```
procedure swap1 ( T & a, T & b )
{  T
   t := a;          (* a=a0, b=b0, t=a0 *)
   a := b;          (* a=b0, b=b0, t=a0 *)
   b := t;          (* a=b0, b=a0, t=a0 *)
}
```

Parameter passing modes I

```
procedure swap1 ( T & a, T & b )
{  T
   t := a;          (* a=a0, b=b0, t=a0 *)
   a := b;          (* a=b0, b=b0, t=a0 *)
   b := t;          (* a=b0, b=a0, t=a0 *)
}

procedure swap2 ( G & a, G & b )
{  a := a – b;      (* a=a0–b0, b=b0 *)
   b := a + b;      (* a=a0–b0, b=a0 *)
   a := b – a;      (* a=b0, b=a0 *)
}
```

Parameter passing modes I

```
procedure swap1 ( T & a, T & b )
{  T                (* assume &a   &b *)   (* call swap(a,a) *)
   t := a;          (* a=a0, b=b0, t=a0 *)  (* a=b=a0, t=a0 *)
   a := b;          (* a=b0, b=b0, t=a0 *)  (* a=b=a0, t=a0 *)
   b := t;          (* a=b0, b=a0, t=a0 *)  (* a=b=a0, t=a0 *)
}

procedure swap2 ( G & a, G & b )
{  a := a – b;      (* a=a0–b0, b=b0 *)     (* a=b=0 *)
   b := a + b;      (* a=a0–b0, b=a0 *)     (* a=b=0 *)
   a := b – a;      (* a=b0, b=a0 *)        (* a=b=0 *)
}
```

What about a and b having common substructures?

Parameter passing modes II

```
procedure swap1 ( T in-out a, T in-out b )
{  T                                        (* call swap(a,a) *)
   t := a;                                  (* a=a0, b=a0, t=a0 *)
   a := b;                                  (* a=a0, b=a0, t=a0 *)
   b := t;                                  (* a=a0, b=a0, t=a0 *)
}

procedure swap2 ( G in-out a, G in-out b )
{  a := a – b;                              (* a=0, b=a0 *)
   b := a + b;                              (* a=0, b=a0 *)
   a := b – a;                              (* a=a0, b=a0 *)
}
```

What about a and b having common substructures?

Too many parameter passing modes

Parameter passing modes

- gives the programmer
  - control of storage use / addressing
  - efficiency considerations
- takes away abstract reasoning about programs
  - abuse of call forces algorithm to be part of reasoning
  - same algorithm, different parameter passing modes: different results

Information passing modes

Modes

- *obs*erve – use data in the parameter (const&)
- *upd*ate – use and (possibly) modify data in parameter (&)

Syntactic constraint

- variable used in upd position: not allowed elsewhere in call

Conceptual constraints

- disallow selectors and pointers: eliminates hidden aliases

Abstract reasoning ~ as for functional programs
Allows tools to choose parameter passing mechanism

Method call confusion

Object oriented

```
p0.m(p1,p2);
```

Procedural

```
m(p0,p1,p2);
```

Expression

```
p0 = m(p0,p1,p2);
```

How do these relate to specifications?

Specification versus algorithm

Specifications often use functional style notation

- **many inputs, one output**

  f:s1, …, sn –> s

→makes nice expression syntax

Algorithms naturally

- **take many inputs**
- **produce several outputs**
- **in-situ modification of variables**

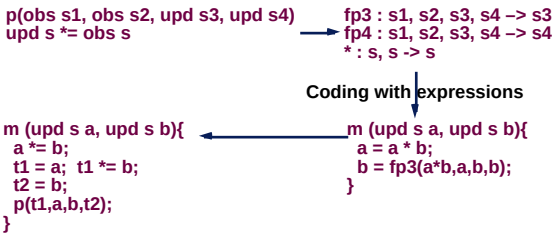
  p(obs s1, …, upd sn)

→Imperative signature: *obs*erves data, *upd*ates variables

Moulding interfaces

Functionalisation:

- **from algorithm-style to specification-style interface**



Mutification:

- **from expressions to imperative calls**

Undefinedness

- Specifying operations
- Implementing algorithms
- Using operations

that may be undefined for some arguments

Undefinedness: specifications

There are many meanings to undefinedness

- partial, i.e., no return value:

  f : s1,…,sn -p-> s, dom_f ⊂ s1,…,sn
- underspecified: will decide the meaning later
- don't want to talk about it: guarding (precondition)

  / : R, R -> R, guard x/y by y≠0

  a * (b/a) = b
  - x/0 may be outside domain (partial)
  - value of x/0 may be context sensitive

    f(a)=0, g(a)=0, then f(a)/g(a) = f'(a)/g'(a)
  - possibility of alert to be captured later

Guarding gives us the freedom of not worrying about expressions that do not have to follow the specification

Undefinedness: algorithms

There are many meanings to undefinedness

- (semi)infinite loop: longer than I want to wait
- termination outside of normal control flow
  - exceptions
    - operation throws an exception: division-by-zero
    - system throws an exception: out-of-memory
    - environment throws an exception:

      user interrupt, system going down
- algorithm flags incomplete result: key-not-in-dictionary
- special return values: nan
- precondition on use of operation (guard)

Why many alert reporting mechanisms

An algorithm may fail for many reasons

- getting inconsistent arguments s.t. it cannot proceed

  returning the middle value of an empty interval

  precondition
- dysfunctional data set detected alongside normal behaviour

  switch on input data, missing case sets return flag

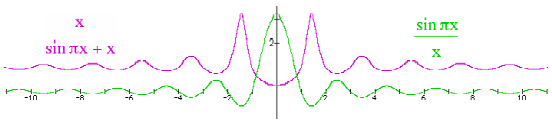
  postcondition
- breach of requirements detected deep inside algorithm

  unexpected null pointer

  exception

Replacement values

```
float purple ( float x )
{ return x/(sin(pi*x)+x); }
```




```
float green ( float x )
{ return sin(pi*x)/x); }
```

Handling alerts

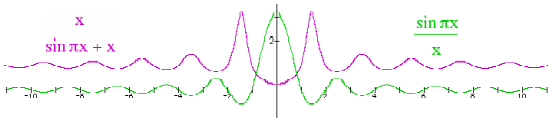
Precondition detection
- default: ignore error, undefined behaviour and return state
- handling: if statement around every function call

Postcondition detection
- default: ignore error, undefined return state
- handling: if statement after every function call

Exception detection
- default: propagate up the call stack
- handling: try-catch around every function call

Handling alerts: replacement value

```
float purple ( float x )
{ return x/(sin(pi*x)+x) <: division_by_0 : 1/(pi+1); }
```




```
float green ( float x )
{ return sin(pi*x)/x) <:: pi; }
```

Moulding undefinedness and alerts

Specifications
- standardise on guarded formulation

Implementations
- algorithm chooses alerting strategy
  functionalisation to guarded formulation
- user chooses handling strategy
  moulded to actual alert

→ transformations between user's handler and needed handler

Dependent typing

Dependent typing simplifies coding

```
template<typename R, int n, int m, int p>
Matrix<R,n,m> mmult ( Matrix<R,n,p> A, Matrix<R,p,m> B )
    { Matrix<R,n,m> C;
      for(int i=0; i<n; ++i)
        for(int j=0; j<m; ++j)
          {  C[i,j] = 0;
            for(int k=0; k<p; ++k)
                C[i,j] += A[i,k] * B[k,j];
          }
      return C;
    }
```

Run-time typing

Run-time typing simplifies use

```
Matrix mmult ( Matrix A, Matrix B )
    { if (A.elementClass() != B.elementClass() ) alert;
      if (A.dimensions != 2 ) alert;
      if (B.dimensions != 2 ) alert;
      if (A.length(2) != B.length(1) ) alert;
      Matrix C(A.elementClass(),A.length(1),B.length(2));
      for (int i=0; i<n; ++i) for (int j=0; j<m; ++j)
          {  C[i,j] = 0;
            for (int k=0; k<p; ++k) C[i,j] += A[i,k] * B[k,j]; }
      return C;
    }
```

Typing schemes

Very strict typing (dependent typing)

- helps ensuring correctness
- simplifies implementing algorithm
- not so good for usage (problem proving type correctness)

Weaker typing

- (somewhat) increases useability
- less transparent implementation
- code becomes more error prone, but
  correctness can be assured by systematic guarding

Moulding strict typing to weaker typing

Summary: simplify programming

– Specifications:
  Functional style declarations with guards
  Dependent types
– Implementations:
  Imperative declarations with alerts
  Dependent types
  Moulding tools:
  - Functionalisation – mutification and alert handling
  - Dependent types into run-time type checking

Automated testing
  Specification-Implementation relationship

Taking the brittle out of programming

WWW

http://www.ii.uib.no/mouldable
http://www.ii.uib.no/saga