A Metacomputation Toolkit for a Subset of F# and Its Application To Software Testing

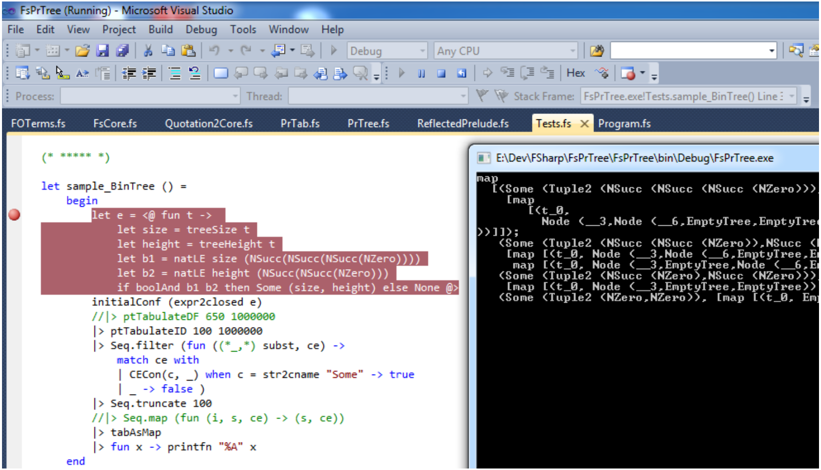
Towards Metacomputation for the Masses

Dimitur Krustev

IGE+XAO Balkan

6 July 2012 / META 2012

Driving and Tabulation inside Visual Studio®



Outline

1. Introduction
   - Supercompilation $\subsetneq$ Metacomputation
   - Making Metacomputation (More) Practical
   - Sample Application – Equivalence-partitioning Tests
2. Program Tabulation for a HO FL
   - F# – Subset, Code Quotations
   - Driving, Optimizations
   - Program Tabulation
   - Tabulation Limitations
3. Application to Testing, Possible Extensions
   - Equivalence Partitioning by Program Tabulation
   - Partition Testing – Another Example
   - Possible Extensions

Supercompilation ⊊ Metacomputation

- Supercompilation – currently most popular metacomputation technique

1.	ftp://ftp.botik.ru/pub/local/Sergei.Abramov/Scp-project	TSG
2.	http://botik.ru/pub/local/scp/refal5/refal5.html	Refal
3.	http://community.haskell.org/~ndm/supero/	Haskell subset
4.	http://hackage.haskell.org/package/optimusprime	Haskell subset
5.	http://hackage.haskell.org/package/supero	Haskell subset
6.	http://users.dsic.upv.es/grupos/elp/peval/	Curry
7.	http://users.ecs.soton.ac.uk/mal/systems/ecce_Download/	Prolog
8.	http://web.archive.org/web/20050819015639/http://www.dina.kvl.dk/~jesper/CASE/	Haskell subset
9.	http://www.evil-wire.org/~jacobian/supercompiler.tgz	Prolog
10.	http://www.supercompilers.ru/	Java subset
11.	https://github.com/batterseapower/chsc	Haskell subset
12.	https://github.com/ilya-klyuchnikov/hosc	Haskell subset
13.	https://github.com/ilya-klyuchnikov/sc-mini	Haskell subset
14.	https://github.com/jasonreich/FliterSC	Haskell subset
15.	https://github.com/spsc	Haskell subset
16.	https://sites.google.com/site/dkrustev/Home/publications/fpsc20030102.zip?attredirects=0	FP subset

- Other powerful techniques exist (neighborhood analysis, neighborhood testing, program tabulation, program inversion)

1.	ftp://ftp.botik.ru/pub/local/Sergei.Abramov/book.appndx	TSG
2.	http://www.botik.ru/~xsg/	XSG
3.	https://github.com/ilya-klyuchnikov/sll-meta-haskell	Haskell subset

- …but not so well-known ⇒ no practical applications developed

Supercompilation $\subsetneq$ Metacomputation

- Supercompilation – currently most popular metacomputation technique

1.	ftp://ftp.botik.ru/pub/local/Sergei.Abramov/Scp-project	TSG
2.	http://botik.ru/pub/local/scp/refal5/refal5.html	Refal
3.	http://community.haskell.org/˜ndm/supero/	Haskell subset
4.	http://hackage.haskell.org/package/optimusprime	Haskell subset
5.	http://hackage.haskell.org/package/supero	Haskell subset
6.	http://users.dsic.upv.es/grupos/elp/peval/	Curry
7.	http://users.ecs.soton.ac.uk/mal/systems/ecce_Download/	Prolog
8.	http://web.archive.org/web/20050819015639/http://www.dina.kvl.dk/˜jesper/CASE/	Haskell subset
9.	http://www.evil-wire.org/˜jacobian/supercompiler.tgz	Prolog
10.	http://www.supercompilers.ru/	Java subset
11.	https://github.com/batterseapower/chsc	Haskell subset
12.	https://github.com/ilya-klyuchnikov/hosc	Haskell subset
13.	https://github.com/ilya-klyuchnikov/sc-mini	Haskell subset
14.	https://github.com/jasonreich/FliterSC	Haskell subset
15.	https://github.com/spsc	Haskell subset
16.	https://sites.google.com/site/dkrustev/Home/publications/fpsc20030102.zip?attredirects=0	FP subset

- Other powerful techniques exist (neighborhood analysis, neighborhood testing, program tabulation, program inversion)

1.	ftp://ftp.botik.ru/pub/local/Sergei.Abramov/book.appndx	TSG
2.	http://www.botik.ru/˜xsg/	XSG
3.	https://github.com/ilya-klyuchnikov/sll-meta-haskell	Haskell subset

- ...but not so well-known $\Rightarrow$ no practical applications developed

Making Metacomputation (More) Practical

- Existing metacomputation implementations
  - small special languages
  - no tool support (IDE, debugger)
- Why F#?
  - Simple functional core (language in the ML family)
  - Relatively Popular
    - created/supported by Microsoft (.NET language)
    - open-source (runs on Mono as well)
  - Good Tools (Visual Studio, SharpDevelop, . . .)
  - Built-in support for writing meta-programs
    - code quotations
    - parsing, type inference, de-sugaring – handled by the F# compiler

Equivalence-partitioning Tests

- Equivalence partitioning:
  - define an equivalence relation on the input domain
  - . . . which partitions the domain into a (finite) number of equivalence classes
  - select just one test from each equivalence class
- Motivation:
  - if partitioning is chosen well
  - then the program under test will behave “in the same way” for all data points in a given equivalence class
  - hence it suffices to test on a single data point from each class

Example – Tests for Binary Trees

```
type BinTree<'T> =
  | EmptyTree | Node of 'T * BinTree<'T> * BinTree<'T>
[<ReflectedDefinition>]
let rec treeSize t = match t with
  | EmptyTree -> NZero
  | Node(_, l, r) ->
      NSucc (natAdd (treeSize l) (treeSize r))
```

```
<@ fun t ->
  let size = treeSize t
  let height = treeHeight t
  let b1 = natLE size (NSucc(NSucc(NSucc(NZero))))
  let b2 = natLE height (NSucc(NSucc(NZero)))
  if boolAnd b1 b2 then Some (size, height) else None @>
```

Example – Results

```
[(Some (Tuple2 (NSucc (NSucc (NSucc (NZero))),
  NSucc (NSucc (NZero)))),
  [map [(t_0, Node (__3,Node (__6,EmptyTree,EmptyTree),
    Node (__9,EmptyTree,EmptyTree)))]]);

 (Some (Tuple2 (NSucc (NSucc (NZero)),
  NSucc (NSucc (NZero)))),
  [map [(t_0, Node (__3,
    Node (__6,EmptyTree,EmptyTree),EmptyTree))];
   map [(t_0, Node (__3,EmptyTree,
     Node (__6,EmptyTree,EmptyTree)))]]);
 ...]
```

F# Code Quotations

- Similar in spirit to MetaML and Template Haskell
- Give access to ASTs of selected code fragments
  - `[<ReflectedDefinition>]` makes the AST of a top-level definition accessible (the definition is still compiled as well)
  - `<@ ... @>` returns the AST of the enclosed (syntactically complete) code fragment, instead of evaluating it
  - AST can be processed like a normal algebraic data type

```
match e with
| Var(var) -> ...
| Application(e1, e2) -> ...
| Lambda(v, e1) -> ...
...
```

F# Subset

```
type Exp =
    | EVar of VName
    | EApp of Exp * Exp
    | ELam of BindPattern * Exp
    | ELet of VName * Exp * Exp
    | ELetRec of (VName * Exp) list * Exp
    | ECon of CName * Exp list
    | ECase of Exp * (Pattern * Exp) list
```

- higher-order!
- tuples, union types, records (de-sugared to tuples)
- full support for let- and letrec-expressions
- NO: destructive updates, OOP (classes, inheritance, . . .)

Driving Step Results

- `DSDone` – no more driving possible – make a leaf in the process tree
- `DSTransient of 'Conf` – deterministic static reduction performed
- `DSBranch of 'ContrHead * ('Contr * 'Conf) list` – match-expression scrutinizing a variable – leads to a branching node in the tree
- `DSDecompose of 'Conf list * ('Conf list->'Conf)` – "decomposition" node – several sub-cases possible:
  - non-nullary constructor
  - lambda-expression (`fun x -> ...`)
  - `f x y ...`, where `f` is a free variable
  - `match f x y ... with ...`, where `f` is a free variable

Configuration Representation – Closures

- Configurations: context + closure-based expression representation (explicit environments)
  - easier, transparent treatment of let-expressions
  - easier, transparent treatment of letrec-expressions!!
  - less worries about variable capture/freshness

```
type ClosedExp =
    | CEVar of VName * Env<ClosedExp>
    | CEClosure of BindPattern * Exp * Env<ClosedExp>
    | CEApp of ClosedExp * ClosedExp
    | CECon of CName * ClosedExp list
    | CECase of Exp * CaseAlts * Env<ClosedExp>
```

Configuration Representation – Optimizations

- Need to optimize to achieve acceptable (memory-related) performance
  - delay conversion between closure-based and standard expression representations whenever possible (hoping that some conversions may cancel each other)
    - accept both kinds of expression representations in closure environments
  - limited form of environment pruning (when making a closure from a variable, skip environment bindings until one for this variable found)

Program Tabulation – Definition

- Key initial step in the URA technique for program inversion
- Reconstruct the input-output relation of the program
  - on a subset of the data domain $D_{in} \subseteq D$
  - as a possibly infinite table $(D_{in}^{(1)}, f_1), (D_{in}^{(2)}, f_2), \ldots$
  - where $D_{in}^{(i)}$ form a partition of D_{in}
  - and f_i are expressions representing functions $D_{in}^{(i)} \to D$
  - Also: computation on each $d \in D_{in}^{(i)}$ must take the same path in the perfect process tree of the program

Program Tabulation – Classic Approach

- Algorithm outline:
  - build and traverse a (perfect) process tree of the program
  - when passing through a branch node, collect contractions in each branch
  - when reaching a leaf, its configuration is f_i, and the composition of contractions along the way is an encoding of $D_{in}^{(i)}$
- No transient or decomposition nodes considered
- Transient nodes: easy – just skip them
- Decomposition nodes?

Decomposition Node Treatment

- Classic approach: breadth-first process tree traversal – complete, BUT:
  - memory-hungry
  - not clear how to treat decomposition nodes
- Iterative deepening – less memory-hungry alternative, easier to treat decomposition nodes:
  - tabulate each subtree of decomposition node, resulting in a table tab_i (finite, because traversal is depth-limited!)
  - construct the Cartesian product of all tab_i
  - from each product element $((D_{in}^{(i_1)}, f_{i_1}), \ldots, (D_{in}^{(i_n)}, f_{i_n}))$ build table entry for decomposition node: $(D_{in}^{(i_1)} \cap \cdots \cap D_{in}^{(i_n)}, C(f_{i_1}, \ldots, f_{i_n}))$

Decomposition Node Treatment

- Classic approach: breadth-first process tree traversal – complete, BUT:
  - memory-hungry
  - not clear how to treat decomposition nodes
- Iterative deepening – less memory-hungry alternative, easier to treat decomposition nodes:
  - tabulate each subtree of decomposition node, resulting in a table tab_i (finite, because traversal is depth-limited!)
  - construct the Cartesian product of all tab_i
  - from each product element $((D_{in}^{(i_1)}, f_{i_1}), \ldots, (D_{in}^{(i_n)}, f_{i_n}))$ build table entry for decomposition node: $(D_{in}^{(i_1)} \cap \cdots \cap D_{in}^{(i_n)}, C(f_{i_1}, \ldots, f_{i_n}))$

Tabulation Restrictions – HO Results

- Decomposition nodes:
  - non-nullary constructors – OK!
  - lambda-expressions – ?
  - calls to unknown function (free variable) – ?
- HO functions in result

```
<@ fun b ->
  if b then (b, fun x -> boolNot x)
  else (boolNot b, fun x -> x) @>
```

  - We must recover a finite, closed function body from a (potentially infinite) process tree (we need a supercompiler)
  - interesting use cases?

Tabulation Restrictions – HO Results

- Decomposition nodes:
  - non-nullary constructors – OK!
  - lambda-expressions – ?
  - calls to unknown function (free variable) – ?
- HO functions in result

```
<@ fun b ->
  if b then (b, fun x -> boolNot x)
  else (boolNot b, fun x -> x) @>
```

  - We must recover a finite, closed function body from a (potentially infinite) process tree (we need a supercompiler)
  - interesting use cases?

Tabulation Restrictions – HO Inputs

- HO functions in inputs

```
<@ fun p xs -> listFilter p (listFilter p xs) @>
```

  - Tabulation must deal with **`match`** `p x` **`with`** `...`, where p is free
    - some sort of higher-order unification needed?
- instead of adding higher-order unification to tabulation ...
- ...we can make a meta-system transition:
  - higher-order input ⇒ first-order function encoding
  - calls to HO parameter ⇒ calls to an encoding interpreter

Tabulation Restrictions – HO Inputs

- HO functions in inputs

```
<@ fun p xs -> listFilter p (listFilter p xs) @>
```

  - Tabulation must deal with **`match`** `p x` **`with`** `...`, where p is free
    - some sort of higher-order unification needed?
- instead of adding higher-order unification to tabulation ...
- ...we can make a meta-system transition:
  - higher-order input ⇒ first-order function encoding
  - calls to HO parameter ⇒ calls to an encoding interpreter

Avoiding Restrictions – Example

```
module NatToXRepr =
    type Stream<'a> = | SCons of 'a * Lazy<Stream<'a>>
    [<ReflectedDefinition>]
    let rec streamNth n (SCons(x, xs1)) =
      match n with
      | NZero -> x
      | NSucc(n1) -> streamNth n1 (xs1.Force())
    [<ReflectedDefinition>]
    let eval tbl n = streamNth n tbl
```

```
<@ fun p_tbl xs ->
    let p = NatToXRepr.eval p_tbl
    listFilter p (listFilter p xs) @>
```

Avoiding Restrictions – Result

```
map
  [(Cons (NSucc (NZero),Empty),
    [map
       [(p_tbl_0, SCons (__6,SCons (True,__7)));
        (xs_1, Cons (NSucc (NZero),Empty))]]);
   (Cons (NZero,Empty),
    [map [(p_tbl_0, SCons (True,__4));
          (xs_1, Cons (NZero,Empty))]]);
   ...]
```

Using Tabulation for Equivalence Partitioning

- Recall main idea of equivalence partitioning – build a finite partition of the input domain: $D_1 \cup D_2 \cup \cdots \cup D_n = D$, $D_i \cap D_j = \emptyset$
- We can specify such a partition by a function $f : D \to X$ where $X = \{x_1, x_2, \ldots, x_n\}$ is finite (with small number of elements):
  - $D_i := \{d \in D \mid f(d) = x_i\}$
- If f is coded in our F# subset, we can use program tabulation on f to build the partition:
  - $Tab(f, D) = (D'_1, f_1), (D'_2, f_2), \ldots$

Using Tabulation for Equivalence Partitioning (cont.)

- Assume f is "reasonably" defined:
  - all f_i are constant functions ($f_i(d) = x_j$ for some j)
  - there is a finite prefix of the table (of length n), such that $\{f_1(d_1), f_2(d_2), \ldots, f_n(d_n)\} = X$ (where $d_i \in D'_i$ are arbitrary)
- We can then obtain our partition of the input domain:
  - $D_i := \bigcup\{D'_k \mid f_k(d_k) = x_i, d_k \in D'_k, k \in \{1, \ldots, n\}\}$
- When partition is defined, selecting actual tests from each equivalence class is (usually) a simple task (fill arbitrary well-typed values in place of free variables)

Another Example: Well-typed STLC Terms

```
type Ty = Tiota | Tarr of Ty * Ty
type Exp = V of Nat | A of Exp * Exp | L of Ty * Exp
[<ReflectedDefinition>]
let rec typeOf (tenv: Ty list) (e: Exp) : Ty option =
  match e with
   | V n -> listNth n tenv
   | A(e1, e2) ->
     match typeOf tenv e1, typeOf tenv e2 with
     | Some (Tarr(t11, t12)), Some t2
         when tyEq t11 t2 -> Some t12
     | _, _ -> None
   | L(ty, e1) ->
     match typeOf (ty::tenv) e1 with
     | Some ty1 -> Some (Tarr(ty, ty1))
     | None -> None
```

Well-typed STLC Terms – Tabulation Query

```
<@ fun tenv e ->
 let cond1 = natEq (LCSample.lamCount e) NZero
 let appc = LCSample.appCount e
 let cond2 = natLE (NSucc(NSucc(NZero))) appc
 let cond3 = natLE appc (NSucc(NSucc(NSucc(NZero))))
 if boolAnd cond1 (boolAnd cond2 cond3) then
   match LCSample.typeOf tenv e with
   | None -> false
   | _ -> true
 else false @>
```

Well-typed STLC Terms – Results

```
[(e_1, A (V (NZero),A (V (NZero),V (NSucc (NZero)))));
 (tenv_0, Cons (Tarr (Tiota,Tiota),Cons (Tiota,__14)))]
[(e_1, A (V (NSucc (NZero)),
          A (V (NSucc (NZero)),V (NZero))));
 (tenv_0, Cons (Tiota,Cons (Tarr (Tiota,Tiota),__12)))]
[(e_1, A (A (V (NSucc (NZero)),V (NZero)),V (NZero)));
 (tenv_0, Cons (Tiota,
   Cons (Tarr (Tiota,Tarr (Tiota,__16)),__12)))]
...
[(e_1, A (V (NZero),A (V (NZero),
    A (V (NZero),V (NSucc (NZero))))));
 (tenv_0, Cons (Tarr (Tiota,Tiota),Cons (Tiota,__17)))]
...
```

Toolkit Improvements

- Make the toolkit even more user-friendly
  - extend toolkit library of standard types and operations (binary-arithmetic integers, maps, sets, . . .)
  - extend built-in conversions from/to standard F# types (especially `int`)
- Make the toolkit faster (current space usage reasonably good already)
  - speed up driving?
    - byte-code-based driving?
    - parallelization?
  - prune process tree branches?
  - faster treatment of decomposition nodes?

Toolkit Extensions

- Add a supercompiler
  - many potential practical applications (property verification, ...)
  - full treatment of higher-order functions inside tabulation results
- Neighborhood analyzer
- Neighborhood testing
  - Potentially very useful in practice!
    - property-based test generation
    - ...
  - Possible problem: performance
    - neighborhood testing requires 2 levels of interpretation

Summary

- A practical implementation of metacomputation techniques for a large subset of F#
  - first implementation of program tabulation for a HO FL
- With a practical application: generating equivalence-partitioning tests
- Interesting optimization tricks (especially w.r.t. space usage)

- Outlook
  - Make toolkit even more easier to use (e.g. special support for numbers)
  - Further optimizations (especially time of driving, tabulation)
  - Implement other practically useful metacomputation techniques (neighborhood testing?)