A Metacomputation Toolkit for a Subset of F♯ and Its Application to Software Testing

Towards Metacomputation for the Masses

Dimitur Krustev

IGE+XAO Balkan, Bulgaria
dkrustev@ige-xao.com

Abstract. We present an on-going experiment to develop a practical metacomputation toolkit for F#. There are – apart from the better known supercompilation – other mature and potentially useful methods stemming from metacomputation theory: program inversion and tabulation, neighborhood analysis. Although implementations of these methods have existed since many years, they are typically experimental tools, treating specifically designed small languages. We investigate if such methods can be made more readily available for practical use, by re-developing them for a reasonably large subset of a mainstream programming language. Practical technical challenges – together with possible solutions – are discussed. We also hint at a potential practical application – automatic generation of software test sets based on user specifications of "interesting" input partitioning.

1 Introduction

The metacomputation theory, developed originally by Turchin, gives rise to several interesting techniques. The best known is supercompilation [26], but there are others as well – such as neighborhood analysis, neighborhood testing, program tabulation and inversion [4]. While many supercompiler implementations already exist, some of them for large subsets of popular programming languages, the situation is different for other metacomputation techniques like neighborhood analysis or program tabulation/inversion. Apart from the pioneering work done in the context of Refal, there are few (publicly available) experimental implementations [4,1,2,15], treating specially designed small languages (S-Graph, TSG, XSG, SLL). While these implementations work well, and can perform impressive tasks, they are hardly usable by someone without extensive knowledge in metacomputation theory. For one thing, they require learning a new language, specific to the tool, typically first-order functional, often even flat (without nested function calls). While parsers and pretty-printers often exist, other tools like IDEs or even debuggers are not available. The main motivation for the current work is to try to transfer existing metacomputation techniques in the context of a larger and more popular programming language. We chose to work with a subset of F#, as F# has a clean and relatively simple functional core

(stemming from its ((O)Ca)ML heritage), together with some facilities greatly simplifying the creation of meta-programs, while at the same time it runs on a very popular platform, and is well supported by several powerful IDEs (Visual Studio, SharpDevelop, MonoDevelop [1]).

In line with the overall pragmatic orientation of this experiment, we also wanted an interesting and useful practical application to serve as a use case for the metacomputation machinery. While a very sophisticated testing method based on metacomputation already exists – neighborhood testing [4] – we guessed that it would pose more technical challenges to implement fully, and settled at first on a simpler (and less powerful) method for test generation. In particular, we are interested in ways to apply the equivalence partitioning method for black-box testing [5]. Given a partitioning specification, encoded in the subset of F# that our tool supports, we can apply program tabulation (a key first step in the URA-technique for program inversion) to generate test skeletons for each equivalence class of the chosen data partition. Afterward it would be easy to fill the skeleton holes with arbitrary values of the appropriate type.

To take a really simple example, consider a program (or a set of programs) dealing with binary trees. Important quantitative characteristics of trees are their size and height, so we may consider a partitioning based on the pair (size, height). We can easily create (Fig. 1), in F#, a description of the data domain, and the partitioning specification [2]. If we then make a program tabulation request for the expression shown in Fig. 2, we obtain the table shown in Fig. 3. We group input trees [3] by the result they produce, and we have filtered out those with result `None`.

```
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))

[<ReflectedDefinition>]
let rec treeHeight t =
    match t with
    | EmptyTree -> NZero
    | Node(_, l, r) -> NSucc (natMax (treeHeight l) (treeHeight r))
```

Fig. 1. Binary tree with size, height

[1] All product names mentioned in the article are trademarks of their respective owners.

[2] We assume some familiarity with the syntax of F#, or other languages in the ML family.

[3] Actually expressions representing sets of input trees.

```
<@ 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 @>
```

Fig. 2. Binary tree tabulation request by (size, height)

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

Fig. 3. Binary tree tabulation result

A couple of things to note: we have restricted both the size and the height of the trees we consider, in order to reduce the search space and to get a small number of tests. Each entry in the resulting table represents an equivalence class – a set of input values giving a particular output. The set is represented as a list of maps, each map specifying possible values for all parameters of the tabulated function. Further, the input expressions in the table contain free variables (`__3`, `__6`, ...), for places in the input trees, that bear no influence on the expression value. We could, however, instantiate those variables with suitable constants (based on the type) and get concrete tests as a final result. For example, the table entry `(Some (Tuple2 (NZero, NZero)), [map [(t_0, EmptyTree)]])` shows, that only the empty tree has both size and height 0.

We shall see a couple of more realistic examples of partitioning-based testing in Sect. 5. Before that, we discuss how F# features like code quotations facilitate the creation of "embedded" meta-programs operating on other parts of the F# program (Sect. 2). We also motivate the choice of the particular F# subset covered. Next we outline the implementation of the basic metacomputation algorithms – driving and process-tree creation (Sect. 3). We mostly follow an approach similar to existing implementations, but also discuss practical challenges and implementation tricks. The treatment of program tabulation and inversion (Sect. 4) is based on the classical approach used in URA [4,1,2], but the fact that we deal with a non-flat, higher-order functional language poses some unexpected obstacles. In fact, the current implementation is – as far as we know – the first that lifts URA-like methods to a higher-order language.

2 "Decompiling" F# quotations

While typical meta-program implementations require re-implementing some phases of a standard compiler front-end (like lexing, parsing, de-sugaring, type-checking), with F# we can take a shortcut by using its built-it facilities for meta-programming – called "code quotations". In brief, F# quotations permit to instruct the compiler to store a high-level, abstract-syntax-tree (AST) representations of parts of the program, alongside (or instead of) the compiled low-level byte-code [24]. There are basically 2 ways to achieve this:

- by placing `[<ReflectedDefinition>]` in front of a top-level function (or method) definition (producing both a normal compiled definition and an AST representation);
- by enclosing any (syntactically complete) expression in `<@ ... @>` (which returns only the AST representation of the expression).

The use of both methods was already demonstrated in the introductory example. There are special facilities (the `MethodWithReflectedDefinition` active pattern, for example) for retrieving the AST corresponding to a reflected top-level function. The design of F# quotations resembles similar designs in MetaML and Template Haskell. There is no need to explicitly require reflecting type definitions – it is done by default in languages sitting on top the .NET run-time.

While in principle it would be possible to base driving and further metacomputation algorithms directly on the internal AST representation of quotations, we take a two-step approach: we first translate quotation ASTs (if possible) to another representation (Fig. 4) on which driving is then performed. The advantages of this approach are:

- we can precisely specify the subset of F# we treat;
- we can use a language representation more suitable for driving.

```
type BindPattern = PVar of VName | PWildcard
type Pattern = PCon of CName * BindPattern list
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
```

Fig. 4. F# subset definition

One can see immediately that we retain most typical features of modern functional languages (higher-order functions, let- and letrec-expressions, algebraic data types with pattern matching). Readers well familiar with F# can already deduce what is left out:

- all features related to object-oriented programming, and ensuring interoperability with other .NET languages (classes, structs, methods, etc.);
- all features related to mutable data structures, and side-effects in general

The inclusion of wild-card patterns adds some complexity to the metacomputation algorithms, but it greatly simplifies the conversion from the reflected AST representation.

Typing deserves a special note. Our intermediate language is untyped. On the other hand we rely on the fact, that F# quotations are type-checked by the compiler [4]. This fact gives 2 advantages:

- during driving (and other metacomputation algorithms) we can take fewer precautions if we can rely on an initially type-checked input;
- driving and other transformations preserve typing implicitly, so at the end we can recover a typed expression relatively easily, if needed.

As already hinted by the first example, the F# subset we accept features a rich type system, including tuples, records (with immutable fields) [5], unions, parametric polymorphism.

One more important restriction of our subset is the lack of access to the F# standard libraries. There are two main reasons for this decision:

- standard-library code is already compiled without reflection, and there is no easy way to make it reflected without modifying the library sources (where available) and recompiling a customized version of the standard libraries;
- metacomputation algorithms like driving are traditionally designed for algebraic data types. Adding support for some primitive data types (like `int` or `float`) would probably require the use of external constraint solver, and would in general complicate the implementation by at least an order of magnitude. Besides, many standard-library data types, such as arrays, are essentially mutable. Support for mutability is also deemed too complicated to contemplate at this point.

Still, we support some of the basic built-it types – like `bool`, `option`, `list` – because they can be treated as algebraic. The lack of standard-library support is compensated by a "reflected prelude" containing definitions of some useful data types and functions. (`natAdd` and `natMax` in the introductory example come from this prelude.)

The conversion from code quotation representation to our `Exp` type is mostly straightforward, as the F# compiler has already done its parsing, type inference, pattern-matching compilation and other de-sugaring, before emitting the quotation AST. Actually the amount of de-sugaring is more than we need here: F# match-expressions are converted to a combination of more primitive operations (if-expressions, predicates for testing for a particular head constructor, tuple projections). Our conversion maps those primitive operations back to special kinds of match-expressions, but without any non-local optimizations. The

[4] Unless built by direct calls to AST constructors, but we ignore that possibility here.
[5] Records are de-sugared into tuples, and do not appear explicitly in Fig. 4

net result can be seen in Fig. 5, which shows the conversion of the `treeSize` function (pretty-printed as F# code). The program in this form clearly contains redundancies, but driving is able to remove exactly this kind of redundancies well, so we do not need to perform any special optimizations at the level of our `Exp` representation.

```
let rec treeSize = (fun t ->
 match (match t with
        | EmptyTree -> False
        | Node (_,_,_) -> True) with
 | True ->
  (let r = match t with
           | Node (_,_,x) -> x in
   (let l = match t with
            | Node (_,x,_) -> x in
    NSucc ((ReflectedPrelude.natAdd (treeSize l)) (treeSize r))))
 | False -> NZero)
```

Fig. 5. F# quotation of the `treeSize` function

3 Process-tree Construction

We assume readers are familiar with the basic notions of driving, and do not reiterate them here [6], but instead focus on some of the implementation technical details. We follow the standard approach [4,23] and base our metacomputation toolkit on the notion of a process tree. While some implementations of supercompilation omit the explicit construction of process trees (e.g. [18,12,6,20]), we feel it is a useful concept, unifying different metacomputation techniques and making their implementation more modular.

3.1 Driving and Process Trees

Our representation of process trees is quite standard, very similar to [23,13,15]. We separate the implementation of individual driving steps in a stand-alone function, and then the main driving function simply unfolds the results of driving steps into a tree (Fig. 6). Each driving step takes a "configuration", encoding the current state of the driving process, and produces one of 4 kinds of results, corresponding to the 4 kinds of process tree nodes described below. As the process tree is often infinite and F# is a strict language, we need to explicitly make the process-tree data type lazy enough. (We use the `Lazy<'X>` data type from the F# standard libraries for this purpose.)

We distinguish 4 kinds of process-tree nodes:

- leaves – used when driving cannot proceed further at all (because we reach an atomic value or a free variable);

[6] Good introductions can be found, for example in [23,4].

```
type DriveStep<'Conf, 'ContrHead, 'Contr> =
    | DSDone
    | DSTransient of 'Conf
    | DSDecompose of 'Conf list * ('Conf list -> 'Conf)
    | DSBranch of 'ContrHead * ('Contr * 'Conf) list
let driveStep (conf : DriveConf)
    : DriveStep<DriveConf, VName, Pattern> = ...
type PrTree<'Conf, 'ContrHead, 'Contr> =
    | PTDone of 'Conf
    | PTTransient of 'Conf * Lazy<PrTree<'Conf, 'ContrHead, 'Contr>>
    | PTDecompose of ('Conf * Lazy<PrTree<'Conf, 'ContrHead, 'Contr>>)
        list * ('Conf list -> 'Conf)
    | PTBranch of 'ContrHead * ('Contr * 'Conf
        * Lazy<PrTree<'Conf, 'ContrHead, 'Contr>>) list
let rec drive (conf : DriveConf)
    : PrTree<DriveConf, VName, Pattern> = ...
```

Fig. 6. Driving and process trees

- branches – used when driving gets blocked by a match-expression scrutinizing a free variable. The edges to the sub-trees of such a node are labeled with "contractions", which encode the conditions for selecting one of the sub-trees. It turns useful to factor out the common part of all contractions in the branch node itself – represented as `'ContrHead` in Fig. 6;
- transient nodes – when the driving step amounts to (deterministic) weak reduction;
- decomposition nodes – used in different cases:
  - when driving reaches a constructor of non-0 arity, we can continue driving the constructor arguments in parallel;
  - when we need to perform generalization during supercompilation, in order to obtain a finite (but imperfect) process tree;
  - when driving get stuck at an expression with sub-expressions, and none of the above cases applies.

In the current setting, the last subcase amounts to reaching an expression of the form `match` $fx_1x_2...x_n$ `with ...`, where f is a free variable of a function type, and $n > 0$. In the ideal case, we should create a branch node, but that would mean using a more complicated representation of contractions, and some form of higher-order unification. As in other supercompilers for higher-order languages, we take a simpler route by decomposing such expressions and continuing to drive their sub-expressions in parallel.

3.2 Driving Configurations

A technical subtlety arises with the choice of configuration representation: our F# subset preserves the possibility of the full language to have arbitrary nested let(rec) expressions. Direct treatment of let-, and especially letrec-, expressions can complicate substantially the definition of driving, so it is common to assume, that at least recursive definitions are lambda-lifted to the top level [18,20,17].

But first, we make an important decision – to stick to call-by-name (CBN) driving, even if our object language is call-by-value (CBV). This approach is classic – taken already by the original work on driving for Refal. In the absence of side effects, the semantic difference between CBN and CBV is mostly of theoretical interest. More important pragmatically is the potential loss of sharing, compared with call-by-need driving methods [18,17,6]. We argue, however, that this loss of sharing is critical mostly in the context of supercompilers used as a part of optimizing compilers. As we are interested in other applications of metacomputation techniques, the simplicity and the potential for more reductions offered by call-by-name driving is deemed more important.

Sticking to call-by-name enables us to introduce yet another intermediate representation of expressions, based on closures (Fig. 7), inspired by [7,8]. It is, obviously, easy to convert between the `Exp` and `ClosedExp` representations. The new encoding brings the following advantages:

- no need to deal explicitly with let-expressions or substitutions during driving;
- an especially unobtrusive treatment of recursion;
- no worries about variable capture, at least in the context of weak reductions.

```
type CaseAlts = (Pattern * Exp) list
type EnvEntry<'V> =
    | EEVarBind of VName * 'V
    | EERecDefs of (VName * Exp) list
type Env<'V> = EnvEntry<'V> list
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>
```

Fig. 7. Closure-based expression representation

To keep track of the context of the current redex we use a stack of evaluation contexts, similar to [6]. We also need a counter for generating fresh variables when reduction descends under a binder. The final representation of driving configurations is shown on Fig. 8.

```
type DriveContextEntry =
    | DCEApp of ClosedExp
    | DCECase of CaseAlts * Env<ExpOrClosed>
type DriveContext = DriveContextEntry list
type DriveConf = int * ClosedExp * DriveContext
```

Fig. 8. Driving configurations

3.3 Optimizations of the Driving Implementation

While the closure-based representation of expressions has many advantages, it comes at a cost: we have to constantly switch between the 2 representations (`Exp` and `ClosedExp`) during driving and further metacomputation algorithms (when we look up a variable binding inside an environment; when driving reaches a (weak) redex, whose head expression has a binder; when we need to perform driving under a binder; when we propagate a contraction inside the sub-trees of a branch node). The repeated re-traversals and re-creations of parts of the driven expression incur a high cost both in terms of processor time and memory consumption. We found 2 measures, that helped reduce significantly this cost.

Firstly, the closure-based representation can be modified slightly, as shown on Fig. 9. At certain key places (simple environment bindings, and the scrutinee of a match-expression) we permit both representations. The advantage of this flexibility is that we can delay in many situations the conversion from `Exp` to `ClosedExp` or vice-versa. Further, when the same sub-expression undergoes a sequence of conversions from one form to the other, adjacent pairs of such conversions can now cancel each other. We have to force the conversion typically only when driving reaches a variable reference, whose binding in the corresponding environment is not in the form needed.

The second measure involves performing a limited form of environment pruning (currently – only when converting an `EVar` into a `CEVar`). The limited form we have chosen is inexpensive both in terms of time and memory allocations, while permitting in many cases to seriously reduce the size of environments stored in closures. As these environments may need to be converted back to let(rec)-expressions at a certain point, the saving can be substantial.

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

Fig. 9. Optimized closure-based representation

4 Process-tree Based Program Tabulation and Inversion

Let's briefly recall the idea of program tabulation, which was historically developed as a key initial step in the URA method for program inversion [4,1]. Assuming a programming language L, a data domain D, and evaluation function $[\![\_]\!] : L \to D \to D$, the tabulation of a program $p \in L$ can be defined in

general as producing – from a program and a subset of the data domain – a sequence of pairs:

$$Tab(p, D_{in}) = (D_{in}^{(1)}, f_1), (D_{in}^{(2)}, f_2), ...$$

where:

$$D_{in}, D_{in}^{(i)} \subseteq D;\ D_{in}^{(i)} \cap D_{in}^{(j)} = \emptyset\ for\ i \neq j;\ \bigcup_{i=1}^{\infty} D_{in}^{(i)} = D_{in};$$
$$f_i : D_{in}^{(i)} \to D, i = 1, 2, ...$$
$$\forall i \forall d \in D_{in}^{(i)} (f_i(d) = [\![p]\!]\ d)$$

and with the further requirement that, given any i, for each $d \in D_{in}^{(i)}$ the computation of $[\![p]\!]\ d$ takes the same path in the branches of the process tree.

We follow the established approach for performing tabulation ([4,1,2]): first build the process tree, then traverse it in breath-first order (to ensure completeness), collecting contractions from branch nodes along the way. Each time we reach a leaf in the tree, we can produce a new input-output pair in the table: each $D_{in}^{(i)}$ is the composition of contractions along the path from the root applied to D_{in} (all $D_{in}^{(i)}$ take thus the form of first-order substitutions); and $f_i = e$, where e is the expression extracted from the leaf configuration. Note that both $D_{in}^{(i)}$ and e can contain free variables; binding them to different values gives different elements in the set $D_{in}^{(i)}$, and substituting these values in e gives the ground result associated with the corresponding input.

The algorithm outlined above considers only process trees containing leaves and branch nodes. Indeed, the original implementations of URA [4,1] dealt with first-order flat functional languages (S-Graph, TSG), Extensions were later developed for non-flat, but still first-order languages [2,15]. Still, it appears that those extensions either do not produce transient/decomposition nodes, or ignore the possibility of such nodes during tabulation. Transient nodes are easy – we can simply skip them during tabulation. As for decomposition nodes, recall that in the current setting they can be of 3 kinds:

1. corresponding to a constructor of non-0 arity;
2. corresponding to a λ-expression, which will appear in the program result;
3. corresponding to an expression of the form `match` $fx_1x_2...x_n$ `with ...`, with $n > 0$ and f a free variable of a function type

We decided to ignore the latter 2 kinds during tabulation, with one exception: nodes of the second kind immediately below the root of the process tree. Such nodes correspond to the program arguments, and we simply traverse them collecting the set of free variables, to build D_{in}. As for other non-constructor decomposition nodes appearing inside the tree, they seem not so important for the practical applications we currently have in mind, as their presence would imply at least one of the following:

– first-class functions embedded inside the result of the tabulated function;

– first-class functions appearing as (sub-values of) arguments of the tabulated function.

Our current strategy is to skip processing the corresponding sub-tree, which renders the method incomplete, but at least allows us to find other existing solutions, even if the tabulation request violates the above assumptions. Note also that this restriction in no way prevents many typical uses of higher-order functions – for example in conjunction with list functions such as `map` or `filter`. (See the examples in the next section.) We should only avoid higher-order functions used as data – nested inside input or output values. Thus our restriction is very similar to existing restrictions in most higher-order functional languages on performing input/output with values embedding higher-order functions.

This still leaves us with the problem of how to handle constructor decomposition nodes during tabulation. In fact, we have not yet found a suitable solution for the tabulation algorithm based on breadth-first search. BFS in itself proved to be a practical problem, as it can be very memory-consuming (because of the need to keep a queue with all nodes of the level of the tree currently being scanned). There is a well-known alternative to BFS, which trades a small constant-factor increase in time complexity for big asymptotic improvement in memory consumption – iterative deepening [21]. While we have not implemented yet a version of tabulation that performs full iterative-deepening search, we have one based on depth-limited search, which is the key component of iterative deepening. To further optimize memory consumption, we do not build the process tree explicitly before tabulation, but use directly `driveStep` (Fig. 6), producing a "virtual" version of the process tree for traversal.

This version of depth-limited tabulation also enables an easy (even if somewhat brute-force) solution to the problem of inner constructor nodes. The basic idea is as follows:

– build all input-output tables – tab_i $(i = 1..n)$ – corresponding to the n subtrees of the constructor node; as we are using depth-limited search, each tab_i is guaranteed to be finite;
– construct the Cartesian product of all tab_i:

$$cp = \{((s_1, e_1), ..., (s_n, e_n))| \ \forall i(s_i, e_i) \in tab_i\}$$

– for each $((s_1, e_1), ..., (s_n, e_n)) \in cp$, if `mergeManySubsts [` s_1; ...; s_n `] = Some` s, we can yield a new table entry for the constructor node: $(s, C(e_1, ..., e_n))$, where C is the corresponding constructor.

The function `mergeManySubsts` simply combines several first-order substations, ensuring that if the same variable is bound in several substitutions, the corresponding most general unifier exists, and the variable is bound to the unification result.

5 Black-box Tests Based on Partitioning Specifications

We return to the discussion of one potential application of our F# metacomputation toolkit – generation of partition-based black-box tests. Black-box testing

requires creation of "interesting" test sets for programs $p \in L$, whose internal structure is unknown. One well-proven heuristics for building such tests is the method of "equivalence partitioning" [5]. The idea is to devise a partition (typically – finite) of the data domain: $D = \bigcup_i D_i$ $(i \neq j \rightarrow D_i \cap D_j = \emptyset)$; each such partition defines an equivalence relation over D. The intuition behind the method is that if we define a suitable partition, the data points in each of the corresponding equivalence classes will be treated in a similar manner by the program under test, and so it will be sufficient to take only one or a few tests from each equivalence class. One way to specify such a partition is by a function $f : D \rightarrow X$, where $X = \{x_1, x_2, ..., x_n\} \subseteq D$ is some finite data type. Then take $D_i := \{d \in D \mid f(d) = x_i\}$. If we also assume that the specification is complete, in the sense that $\forall i \exists d \in D \ : \ f(d) = x_i$, we can use the program tabulation method described in the previous section to generate representatives of the equivalence classes. It suffices to tabulate f, and to group entries in the resulting table by output value (x_i). Then, for each x_i, we can select one or several input tests from the corresponding $D_{in}^{(i)}$.

Let's look at a few more examples illustrating this idea. First, consider a program dealing with a very simple imperative language, containing assignments, sequences and while-loops (Fig. 10). We leave unspecified the data types for variables and expressions. Two simple quantitative measures on programs are introduced – statement count, and loop-nesting depth, and we use them to define partitioning of the space of programs – the expression used in the tabulation request is shown on Fig. 11. (Note that we limit the search to programs of nesting ≤ 2 and statement count ≤ 3.) The result of tabulation appears in Fig. 12; we can see 6 equivalence classes, and by instantiating the free variables with some suitable values we can obtain test inputs for each class.

```
type Stmt<'V, 'E> =
    | Assign of 'V * 'E
    | Seq of Stmt<'V, 'E> * Stmt<'V, 'E>
    | While of 'E * Stmt<'V, 'E>

[<ReflectedDefinition>]
let rec stmtCount (s: Stmt<'V, 'E>) : Nat = ...

[<ReflectedDefinition>]
let rec loopNesting (s: Stmt<'V, 'E>) : Nat = ...
```

Fig. 10. Simple imperative language

The next example is from a different domain (a scaled-down version of a real-world use case): programs dealing with electrical diagrams often have wire-lists as inputs. Wire connections usually form an acyclic graph, with each connected component representing an equipotential. We can thus represent wire-lists as forests, with each tree being a set of electrically connected wires and pins (Fig. 13). A possible tabulation request – defining a partition by (equipotential-count,

```
<@ fun s ->
    let sCnt = WhileL.stmtCount s
    let nestCnt = WhileL.loopNesting s
    if boolAnd (natLE nestCnt (NSucc(NSucc(NZero))))
       (natLE sCnt (NSucc(NSucc(NSucc(NZero)))))
    then Some (nestCnt, sCnt) else None
    @>
```

Fig. 11. Tabulation request for imperative language programs

```
[(Some (Tuple2 (NSucc (NSucc (NZero)),NSucc (NSucc (NSucc (NZero))))),
  [map [(s_0, While (__2,While (__4,Assign (__6,__5))))]]);
 (Some (Tuple2 (NSucc (NZero),NSucc (NSucc (NSucc (NZero))))),
  [map [(s_0, While (__2,Seq (Assign (__6,__5),Assign (__8,__7))))];
   map [(s_0, Seq (While (__4,Assign (__8,__7)),Assign (__6,__5)))];
   map [(s_0, Seq (Assign (__4,__3),While (__6,Assign (__8,__7))))]]);
 (Some (Tuple2 (NSucc (NZero),NSucc (NSucc (NZero)))),
  [map [(s_0, While (__2,Assign (__4,__3)))]]);
 (Some (Tuple2 (NZero,NSucc (NSucc (NSucc (NZero))))),
  [map
     [(s_0, Seq (Seq (Assign (__6,__5),Assign (__8,__7)),Assign (__10,__9)))
   map
     [(s_0, Seq (Assign (__4,__3),Seq (Assign (__8,__7),Assign (__10,__9)))
 (Some (Tuple2 (NZero,NSucc (NSucc (NZero)))),
  [map [(s_0, Seq (Assign (__4,__3),Assign (__6,__5)))]]);
 (Some (Tuple2 (NZero,NSucc (NZero))), [map [(s_0, Assign (__2,__1))]])]
```

Fig. 12. Imperative language tabulation result

wire-count) – is shown in Fig. 14. The list of results is too long to include, but the number of different entries in each equivalence class (after taking only the first 30 non-`None` results, with a depth limit of 25) is summarized in Table 1.

Table 1. Wire-list tabulation results

Equipotential count	Wire count	Number of entries
2	3	12
2	2	5
2	1	2
2	0	1
1	3	5
1	2	2
1	1	1
1	0	1
0	0	1

What is remarkable in this example is, that the partition specification requires some slightly more involved processing than the previous ones. We can see, though, that the familiar use of higher-order library functions (like `listFoldl`, `listNubBy`) keeps the code succinct. We could no doubt code the same functions

```
type RoseTree<'N, 'E> = RTNode of 'N * ('E * RoseTree<'N, 'E>) list
type PinData<'EQ> = {pinTag: Nat; eqTag: 'EQ}
type WireData<'WC> = {wireColor: 'WC}
type Equipotential<'EQ, 'WC> = RoseTree<PinData<'EQ>, WireData<'WC>>

[<ReflectedDefinition>]
let rec equipotStats (ep: Equipotential<'EQ, 'WC>) : Nat * 'EQ list =
    match ep with
    | RTNode(pd, wds_ts) ->
        let ts = listMap (fun (_, x) ->x) wds_ts
        let wcount1 = listLength wds_ts
        let (wcount2, eqs) = listFoldl (fun (wc, eqs) t ->
            let (wc1, eqs1) = equipotStats t
            (natAdd wc1 wc, listAppend eqs1 eqs)) (NZero, []) ts
        (natAdd wcount1 wcount2, pd.eqTag::eqs)

[<ReflectedDefinition>]
let equipotsStats (eqTagEq: 'EQ -> 'EQ -> bool)
    (eps: Equipotential<'EQ, 'WC> list) : Nat * Nat * 'EQ list =
    let (wc, eqs) = listFoldl (fun (wc1, eqs1) ep ->
        let (wc2, eqs2) = equipotStats ep
        (natAdd wc2 wc1, listAppend eqs2 eqs1)) (NZero, []) eps
    (listLength eps, wc, listNubBy eqTagEq eqs)
```

Fig. 13. Wire-list as a forest, with simple statistics functions

```
<@ fun eps ->
    let (epCnt, wireCnt, eqs) = WList.equipotsStats boolEq eps
    if boolAnd (natLE epCnt (NSucc(NSucc(NZero))))
        (natLE wireCnt (NSucc(NSucc(NSucc(NZero)))))
    then Some (epCnt, wireCnt) else None
    @>
```

Fig. 14. A tabulation request for wire-list test generation

in a first-order language, but some code-size increase and loss of modularity seems inevitable, even in such small cases. We can also note the use of another trick for controlling the size of the search space: the type for equipment tags is abstracted as a parameter in the wire-list definition; we instantiate it with `bool` in the tabulation request, to limit the number of different equipment tags appearing in test cases.

We have performed further experiments, which we do not describe in detail here. Still it is worth noting that we stumbled upon certain restrictions of this test-generation technique. One of the examples defines a Church-style version of the simply-typed lambda calculus, together with a type-checker. If we try to generate directly only well-typed lambda terms for testing, with a reasonably small depth limit, the method tends to generate mostly trivially well-typed terms of the form $\lambda x_1 x_2 ... x_n . x_i$. On the other hand, the tabulation technique is also very flexible – in the same lambda-calculus sample, it is possible to "nudge" the tabulator towards more interesting well-typed terms, by partially specifying the shape of types in the typing environment of the type-checker, ensuring that there are more suitable types for forming applications.

6 Related Work

As already mentioned on several occasions, the pioneering work on metacomputation – both supercompilation and related techniques – was done by Turchin and his students, using Refal [26]. Later many key methods were re-developed using simpler languages like S-Graph [9,4,1,3]. In recent years, there is renewed interest and activity in different supercompiler implementation, for example [18,14,12,11,6,20], to cite a few. In comparison, other metacomputation methods seem neglected, maybe with the exception of [15].

There is extensive literature on software testing, and in particular, on black-box and equivalence partitioning testing [5]. Another interesting testing method is based on metacomputation techniques as well – neighborhood testing [4,3]. While a detailed comparison with the technique outlined here is out of scope, we can note important high-level differences. Neighborhood testing is a white-box testing method, requiring access to the source of the unit under test (and, if available, also its executable specification). It can be used – in principle – for any language, as long as we have an interpreter for that language, written in the language, for which a neighborhood analyzer exists. We consider neighborhood testing being a "second-order" meta-technique – requiring 2 meta-system transitions – to be the main difficulty for practical application. These two levels of interpretative overhead will probably make it harder to achieve sufficient performance in practical implementations. On the other hand, neighborhood testing is a much more powerful method (in certain formal sense subsuming all existing test coverage criteria), so its successful practical application remains an interesting area of study.

Other test-generation methods apply techniques similar to driving – usually under the umbrella term of "symbolic execution". A recent variation – dynamic

symbolic execution – is employed in successful test-generation tools like Microsoft Pex [10]. Again, we omit a detailed comparison between dynamic symbolic execution and driving, noting instead some pragmatic differences with Pex. Pex is heavily geared towards supporting well idiomatic code written in OO .NET languages like C#. It handles surprisingly well complicated program logic using a combination of different built-in types. When faced with data types exhibiting deep recursive structure, however, it usually requires help from the user to generate good tests [7]. The approach we propose here deals well with recursive algebraic data types, so it can be seen as complementary to tools like Pex.

There are many methods for generating test data with specific structure (for example grammar-based, or XML-Schema-based [5]). Specialized tools exist for generating correct test programs in particular programming languages [16,19]. A generic technique for test generation, like the one presented here, can hardly compete in efficiency with such specialized techniques. At the same time, driving-based exploration is very flexible (as seen in most of our examples) and can probably be combined with similar techniques to reduce the search space for test generation (as we hinted for the lambda-calculus example).

We have already noted, that F# code quotations [24] – which helped a lot in making our toolkit feasible with relatively small effort – are very similar in design to languages like MetaML, and language extensions like Template Haskell [25,22]. Similar facilities are starting to appear for other popular languages. We can thus imagine a version of our toolkit for such languages, leveraging on the corresponding meta-programming support.

7 Conclusions and Future Work

We describe an on-going experiment to make more accessible some of the less well-known metacomputation methods (like program tabulation/inversion or neighborhood analysis) – by re-implementing them for a large subset of a popular standard language (F#), which is supported by an efficient run-time and a rich IDE and other developer tools. The F# support for meta-programming (code quotations) greatly facilitated this effort.

To the best of our knowledge, this is the first implementation of URA-like program tabulation for a higher-order functional language. Aside from the special challenges posed by higher-order functions, we rely on already established metacomputation techniques. We described parts of the implementation in more detail, in the hope that some of the insights, gained in attempting to make driving and tabulation reasonably efficient, can be useful in similar contexts. An interesting practical application of program tabulation is also described in some detail – generating black-box tests from partitioning specifications.

Our system, in its current state, can handle successfully moderately-sized programs, and generate small-sized partition-based tests (similar in size or slightly

[7] It is perfectly possible that many of the current limitations of Pex will be lifted in future versions.

larger than the examples in Sect. 5). Our experiments indicate, that more optimization effort is needed to make the test-generation approach practical in a wider variety of scenarios. It is not clear at this point if modest low-level fine-tuning will be enough, or we need to more drastically re-think some of the key algorithms, such as the treatment of decomposition nodes during tabulation. Another interesting optimization option to try would be to re-use the underlying F# run-time for performing weak reductions, and revert to interpretative driving for treating reduction under binders, and for information propagation inside match-expression branches.

There are a lot of possibilities to make the toolkit more "user-friendly". As suggested by one of the reviewers, we could automatically convert some primitive data types (such as `int`) into suitable replacements from the reflected prelude. A non-trivial issue is how to select the best replacement in each case. (We could supply an implementation of arbitrary signed integers in the prelude, but it would be less efficient during tabulation in cases, where the user actually needs only natural numbers.) Another improvement, which is in progress, is to fill holes in generated tests by arbitrary values of the appropriate type and convert back each test into a normal F# value.

Apart from the potentially interesting future tasks hinted in the previous section, we can list a few more interesting ideas for future experiments:

- extend driving to a full supercompiler for the F# subset; check if program tabulation/inversion can be made more efficient by using the process graphs generated during supercompilation, instead of the potentially infinite process trees;
- modify driving to use call-by-need instead of call-by-name, and see what performance benefits it can bring to metacomputation methods (other than supercompilation);
- implement neighborhood analysis, and experiment with potential practical applications (like neighborhood testing).

8 Acknowledgments

The author would like to thank Ilya Klyuchnikov and Neil Mitchell for the helpful comments and suggestions for improving the presentation in this article.

References

1. Abramov, S., Glück, R.: The universal resolving algorithm and its correctness: inverse computation in a functional language. Science of Computer Programming 43(2-3), 193–229 (2002)
2. Abramov, S., Glück, R., Klimov, Y.: An universal resolving algorithm for inverse computation of lazy languages. Perspectives of Systems Informatics pp. 27–40 (2007)

3. Abramov, S.M.: Metacomputation and program testing. In: Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging. pp. 121–135. Linköping University, Linköping, Sweden (1993)
4. Abramov, S.M.: Metavychisleniya i ih primenenie (Metacomputation and its applications). Nauka, Moscow (1995)
5. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2008)
6. Bolingbroke, M., Peyton Jones, S.: Supercompilation by evaluation. In: Proceedings of the third ACM Haskell symposium on Haskell. pp. 135–146. ACM (2010)
7. Clément, D., Despeyroux, T., Kahn, G., Despeyroux, J.: A simple applicative language: Mini-ML. In: Proceedings of the 1986 ACM conference on LISP and functional programming. pp. 13–27. ACM (1986)
8. Coquand, T., Kinoshita, Y., Nordström, B., Takeyama, M.: A simple type-theoretic language: Mini-TT. From Semantics to Computer Science: Essays in Honour of Gilles Kahn (2009)
9. Glück, R., Klimov, A.V.: Occam's razor in metacomputation: the notion of a perfect process tree. In: Cousot, P., Falaschi, M., Filé, G., Rauzy, A. (eds.) Static Analysis. Proceedings. Lecture Notes in Computer Science, vol. 724, pp. 112–123. Springer-Verlag (1993)
10. Godefroid, P., de Halleux, P., Nori, A., Rajamani, S., Schulte, W., Tillmann, N., Levin, M.: Automating software testing using program analysis. Software, IEEE 25(5), 30–37 (2008)
11. Hamilton, G.: Distillation: extracting the essence of programs. In: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. pp. 61–70. ACM (2007)
12. Jonsson, P.A., Nordlander, J.: Positive supercompilation for a higher order call-by-value language. SIGPLAN Not. 44(1), 277–288 (Jan 2009)
13. Klyuchnikov, I.: The ideas and methods of supercompilation. Practice of Functional Programming (7) (2011), in Russian
14. Klyuchnikov, I., Romanenko, S.: Proving the equivalence of higher-order terms by means of supercompilation. In: Perspectives of Systems Informatics'09. pp. 193–205. Springer (2010)
15. Klyuchnikov, I.: GitHub project: "Metacomputation and its Applications" - now for SLL (2011), `https://github.com/ilya-klyuchnikov/sll-meta-haskell`, [Online; accessed 13-March-2012]
16. Koopman, P., Plasmeijer, R.: Systematic synthesis of functions. In: H. Nilsson (ed.), Selected papers from the Seventh Symposium on Trends in Functional Programming (TFP06), Nottingham, United Kingdom, 19-21 April 2006. pp. 35–54. Bristol: Intellect Books (2006)
17. Mitchell, N.: Rethinking supercompilation. In: ACM SIGPLAN Notices. vol. 45, pp. 309–320. ACM (2010)
18. Mitchell, N., Runciman, C.: A supercompiler for core Haskell. In: et al., O.C. (ed.) IFL 2007. LNCS, vol. 5083, pp. 147–164. Springer-Verlag (May 2008)
19. Reich, J.S., Naylor, M., Runciman, C.: Lazy generation of canonical programs. In: 23rd Symposium on Implementation and Application of Functional Languages (2011)
20. Reich, J., Naylor, M., Runciman, C.: Supercompilation and the Reduceron. In: Proceedings of the Second International Workshop on Metacomputation in Russia (2010)
21. Russell, S., Norvig, P.: Artificial intelligence: a modern approach. Prentice Hall series in artificial intelligence, Prentice Hall (2003)

22. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. pp. 1–16. ACM (2002)
23. Sørensen, M.H., Glück, R.: Introduction to supercompilation. In: Hatcliff, J., Mogensen, T., Thiemann, P. (eds.) Partial Evaluation: Practice and Theory. Lecture Notes in Computer Science, vol. 1706, pp. 246–270. Springer-Verlag (1999)
24. Syme, D.: Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In: Proceedings of the 2006 workshop on ML. pp. 43–54. ACM (2006)
25. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theoretical computer science 248(1-2), 211–242 (2000)
26. Turchin, V.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 292–325 (July 1986)