Tag Space Machines

Lee Spector
lspector@hampshire.edu

July 28, 2012

DRAFT — WORK IN PROGRESS — COMMENTS WELCOME

A tag space machine (TSM) is an abstract computational device that operates on a set of typed data stacks.[1]

Tag space machines were developed to facilitate the automatic synthesis of computational systems through algorithms based on natural selection, as in genetic programming [2].

While the utility of tag space machines is unknown, the hypothesis motivating the development of the model is that program representations that are structured by *tag references* will be more evolvable than other representations that have been used previously. The *tag* concept used here was initially developed by Holland [1]: tags are initially arbitrary labels that come to have meaning with use. Notably, tags can be matched inexactly; this allows tag usage to grow incrementally over a sequence of system transformations (e.g. mutations). The tag concept has been developed further in the context of evolutionary game theory (e.g. [3, 6]), and more recently it has been explored in the context of genetic programming with stack-based and tree-based programs [8, 4, 5]. The TSM model presented here is based on prior stack-based models, but in contrast to the previous models all program structure in a TSM is mediated by tag references.

A TSM may include stacks for a variety of common data types including integers, floating-point numbers, Boolean values, strings, etc. Two stacks that are always present—the `x` (or "execution") stack and the `ts` (or "tag space") stack—play a special role in TSM execution and are treated differently than other stacks; this is explained further below.[2]

[1]It would be interesting to develop a single-stack variant of TSM using Bill Tozier's duck-typing ideas or Maarten Keijzer's push-forth ideas.

[2]For those familiar with the Push programming language, the stack system in a TSM is similar to that in version 3 of Push [7] except that:

- there is no code data type
- there is no code stack
- the execution stack is handled differently
- program structure is encoded via tag spaces.

The TSM execution rule specifies how a single execution step transforms a set of data stacks. TSM execution involves the repeated processing of execution steps, transforming an initial set of stacks into subsequent sets of stacks and perhaps into a final set of stacks.

1 States

We notate sets of stacks using Clojure's map notation, mapping type names (as keywords) to vectors representing stacks with the top elements first. Here is an example:

```
{:x [], :ts [], :integer [1 2 3], :boolean [false true], :float [3.24],
 :string ["Hello" "world"]}
```

Here we show the `x` and `ts` stacks as empty because we have not yet explained what they may contain. The `integer` stack contains three items (with 1 on top), the `boolean` stack contains two, and the `float` stack contains one.

A particular set of stacks, including the items on the stacks, fully characterizes a particular TSM at a particular time. We will refer to such sets of stacks as "TSM states" or more simply as "TSMs."

We will use the term "literal" to refer to *instructions* (described below) and instances of any of the data types, such as integers and Boolean values, that can be handled by the TSM system under discussion. Note that *pairs* and *tag spaces*, also explained below, are *not* literals.

2 Minimal TSMs

We will sometimes consider the special case of a TSM running on a set of stacks that are initially empty except for one item on the `ts` stack and one item on the `x` stack. In such cases we may describe the item that is initially on the `ts` stack as "the program," the item that is initially on the `x` stack as the "entry point," and the final state of the TSM as the result of running the program, starting at the entry point. We will use the term "minimal TSM" to refer to a TSM that is intended to be used in this way. It should be straightforward to prove that there there exists, for any TSM, a functionally equivalent minimal TSM.

3 Instructions

Instructions are entities that, when executed, may access and potentially alter the stacks.

Some instructions will be notated as symbols; for example `integer_add` is the instruction that pushes onto the `integer` stack the sum of two integers popped from the `integer` stack.

Other instructions may be notated as Clojure maps, facilitating the inclusion of pseudo-arguments that are "baked in" to the instructions. These *instruction maps* (or *imaps*) will mainly be useful for tag-related instructions (see below), but a simpler example would be an instruction such as `{:imap integer_add_const :const 3}`. Such an instruction would presumably take one item from the integer stack and push back the item plus 3.

The execution of any instruction that requires data that is not present on the stacks will have no effect; we call this the "no-op rule."[3]

Although many instructions will treat the data stacks as true stacks, accessing or changing only the top-most elements at any given time, others may treat stacks as random access collections and/or rearrange items on the stack for subsequent top-down access. "Deep stack" access instructions like `yank` and `shove` can be included to move items from/to arbitrary locations in a stack based on an index taken from the `integer` stack (and taken modulo the size of the stack in question.

The computational power of a particular TSM model will depend on the available instruction set. It should be straightforward to prove the Turing completeness of many such sets that we would consider using.

4 Execution

The `x` stack—also called the *execution* stack—can contain only literals. The TSM execution rule is:

- If the `x` stack is empty then do nothing.
- Else
  - Pop the `x` stack and call the popped item `i`.
  - If `i` is an instruction then execute it.
  - Else push `i` onto the appropriate stack.

Some computations can be expressed without the use of the `ts` stack. For example, consider this TSM state:

```
{:x [integer_add integer_mult], :ts [], :integer [1 2 3]}
```

Executing this state two or more times produces:

```
{:x [], :ts [], :integer [9]}
```

[3]This is the same behavior as specified in Push. It would be interesting to develop a TSM variant in which instructions that are missing arguments are curried rather than skipped, following an idea of Bill Tozier's.

5 Pairs and Tag Spaces

More interesting computations can be expressed using tag spaces.

Tag spaces are constructed out of tags, literals, and pairs.

Tags, here, are represented as floating-point numbers.

Pairs are 2-tuples of literals, notated with Clojure vector notation (square brackets: [1 2]).[4]

Tag spaces are mappings from tags to literals and pairs, sorted by tag. Here too we will use Clojure map notation:

```
{-6.0 3.14, -0.5 [integer_add 4.2], 0.0 [integer_mult integer_sub],
 22.0 [1 false], 50.5 integer_div}
```

Tag spaces are stored on the `ts` stack (which is so-named because "ts" is an abbreviation for "tag space"). Instructions that alter or access tag spaces always operate on the top tag space or the one immediately beneath it, and all of these except for the instruction `ts_pop` do so without popping the `ts` stack. Note that this is an exception to the normal behavior of TSM instructions, which otherwise always pop the stacks from which they take data.

The simplest tag space instructions are `ts_tag` and `ts_tagged`, both of which are expressed as imaps. The `ts_tag` instruction adds an association to the top tag space on the `ts` stack, associating a tag that is embedded within the imap with the next item on the `x` stack (which is popped). For example, if we start with this TSM state:

```
{:x [{:imap ts_tag :tag 0.5} 27], :ts [{}]}
```

then after one execution step we will have:

```
{:x [], :ts [{0.5 27}]}
```

If the top tag space already contains an association with the given tag then it will be overwritten with the new association.

A pair can be tagged by means of `ts_tag_pair`, which tags a new pair that contains the top and then second items on the `x` stack.

[4]Why do we provide only pairs, rather than tuples of any length? Pairs provide structure that is not mediated by tags, which is something that we want to avoid in the TSM model. But they provide the minimum necessary non-tag-mediated structure. General tuples would provide more non-tag-mediated structure than we need, and we want as little as we can get away with.

The `ts_tagged` instruction looks up an embedded tag in the top tag space and pushes the associated data onto the `x` stack.

For example, if we start with this TSM state:

```
{:x [{:imap ts_tagged :tag 10.0}], :ts [{10.0 "foo"}]
 :string ["bar"]}
```

then after one execution step we will have:

```
{:x ["foo"], :ts [{10.0 "foo"}]
 :string ["bar"]}
```

and after one more, which will "execute" the string literal, pushing it onto the appropriate stack:

```
{:x [], :ts [{10.0 "foo"}]
 :string ["foo" "bar"]}
```

If the tagged data is a pair then the two elements of the pair are both pushed onto the `x` stack, with the first item on top. So, for example:

```
{:x [{:imap ts_tagged :tag 10.0}], :ts [{10.0 ["foo" "bar"]}]
 :string ["baz"]}
```

will produce, after one execution:

```
{:x ["foo" "bar"], :ts [{10.0 ["foo" "bar"]}]
 :string ["baz"]}
```

and then, after two more:

```
{:x [], :ts [{10.0 ["foo" "bar"]}]
 :string ["bar" "foo" "baz"]}
```

6 Inexact Matching

The match between the tag in the `ts_tagged` instruction and the tag of the retrieved data may be inexact, and it is guaranteed that data will be pushed if the tag space is non-empty. The matching association for tag t is the one with the numerically lowest tag m such that $m \geq t$, or the one with the lowest tag overall if none are greater than or equal to t. That is, we wrap around to the bottom of the tag space.

For example, if we start with this TSM state:

```
{:x [{:imap ts_tagged :tag 10.0}], :ts [{0.5 27, 23.0 integer_add}]
 :integer [1 2]}
```

then after one execution step we will have:

```
{:x [integer_add], :ts [{0.5 27, 23.0 integer_add}]
 :integer [1 2]}
```

and after one more we will have:

```
{:x [], :ts [{0.5 27, 23.0 integer_add}]
 :integer [3]}
```

Note that because of the "wrapping around" provision one step of execution from:

```
{:x [{:imap ts_tagged :tag 100.0}], :ts [{0.5 27, 23.0 integer_add}]
 :integer [1 2]}
```

will produce:

```
{:x [27], :ts [{0.5 27, 23.0 integer_add}]
 :integer [1 2]}
```

7 Program Structure

Tagged pairs permit the expression of long programs that are "threaded" through the tag space. For example, consider this TSM state:

```
{:x [{:imap ts_tagged :tag 0.0}],
 :ts [{0.0 [0.3 {:imap ts_tagged :tag 1.0}],
       1.0 [0.2 {:imap ts_tagged :tag 2.0}],
       2.0 [0.1 {:imap ts_tagged :tag 3.0}],
       3.0 [float_add float_mult]}]}
```

When executed to completion this will leave 0.09 on the `x` stack. How? When the single imap on the `x` stack is executed it will cause the elements of the pair `[0.3 {:imap ts_tagged :tag 1.0}]` to be pushed on the `x` stack. Then 0.3 will be processed and pushed onto the `float` stack. Then the imap `{:imap ts_tagged :tag 1.0}` will be executed, retrieving `[0.2 {:imap ts_tagged :tag 2.0}]` and pushing its elements onto the `x` stack. And so on.

Note that `ts_tagged` imaps can occur as both elements of a pair. This means that tree-structured programs can be expressed as well as linearly-structured programs.[5]

8 Control Structure

Instructions that manipulate the `x` stack, such as `x_dup`, `x_swap` and `x_rot`, in conjunction with tag-related instructions and conditionals such as `x_if` (which executes either or the top or the second item on the `x` stack, discarding the other, depending on the value on top of the `boolean` stack) can be used to implement complex control structures.

The `ts_tagged_under` instruction is like `ts_tagged` except that it inserts the retrieved literals *under*, rather than on top of, the top item on the `x` stack. This allows one to re-tag any literal or pair by following the `ts_tagged_under` instruction with a tagging instruction.

9 Multiple Tag Spaces

So far we have only seen programs that use (at most) one tag space. Although there are no tag space literals, additional tag spaces can be created during execution by means of the `ts_new` instruction, which pushes a new, empty tag space onto the `ts` stack. The tag-related instructions described above each operate on whatever tag space is on top of the `ts` stack at the time of execution. However, for each of these instructions there is a similar instruction with a suffix of 2—for example `ts_tagged_2`—that operates instead on the *second* tag space on the `ts` stack. Combinations of these instructions can be used to copy associations from one tag space to another, which can permit the expression of modules with local data spaces to which arguments are passed and from which values are returned. Stack manipulation instructions such as `ts_dup`, `ts_swap`, and `ts_rot` can be used to

[5]As noted by Bill Tozier these programs may be DAGs or general graphs in terms of their execution semantics. The point here is that a program that would be represented as a tree or a nested sequence in a conventional programming language could be expressed in a tag space by means of tagged pairs that may contain `ts_tagged` imaps as both elements.

achieve other effects. Because of the special "non-popping" treatment of the `ts` stack, a tag space can only be removed from the `ts` stack with an explicit call to `ts_pop`.

Why do we need multiple tag spaces?[6] While they may not be absolutely necessary the motivation here is to provide mechanisms like local variable scope in traditional programming languages. The availability of multiple tag spaces may also simplify proofs of computational power/equivalence by allowing arbitrary computations to be composed without having tag usage interfere between sub-computations.

It might also be useful to include instructions that move items directly between multiple tag spaces on the `ts` stack, possibly using integer indices to refer to tag spaces deep in the stack.[7]

10 Use

The TSM model was developed to serve as a substrate for genetic programming, but different uses of TSMs may make sense in different genetic programming contexts. For example, one might want to do different things when searching for a program that solves a symbolic regression problem than when searching for a problem to control an agent in a dynamic simulation environment.

In some cases one may wish to think in terms of evolving minimal TSMs that compute results by starting with the execution of a single `ts_tagged` instruction, operating on a tag space that encodes the "program" along with any inputs (which might be tagged in the tag space prior to execution). In others one may want to evolve full, non-minimal TSM states. In some cases one might want to evolve minimal TSMs but allow stacks to persist across multiple interactions in an agent's lifetime.

In any case, outputs might be found in the resulting tag space or on data stacks.

Acknowledgments

Thanks to Bill Tozier, Maarten Keijzer, Thomas Helmuth, Omri Bernstein, Kyle Harrington, Emma Tosch, Kwaku Yeboah Antwi, and Zeke Nierenberg for helpful discussions. This material is based upon work supported by the National Science Foundation under Grant No. 1017817. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

[6]Thanks to Omri Bernstein for the question.

[7]This is a suggestion from Bill Tozier, who suggests instructions such as `tagged_item_yank`, which pulls up a tagged item from a tag space deep in the `ts` stack and tags it in the top tag space, along with analogous `tagged_item_shove` and `tagged_item_swap` instructions.

Revisions

20120726 Fixed typos, added motivations in intro, added acknowledgments and references.

20120727 More detail in intro, deep stack access and power in Instructions, other minor changes, all tags floats, major errors fixed in tagging/tagged examples.

20120728 Fixed math error in threaded program example, clarified comments about tree structure, need for multiple tag spaces, deep tag space access, rationale for pairs rather than general tuples.

References

[1] J. H. Holland. *Hidden order: how adaptation builds complexity.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[2] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[3] R. L. Riolo, M. D. Cohen, and R. Axelrod. Evolution of cooperation without reciprocity. *Nature*, 414(6862):441–443, Nov. 2001.

[4] L. Spector, K. Harrington, B. Martin, and T. Helmuth. What's in an evolved name? the evolution of modularity via tag-based reference. In R. Riolo, E. Vladislavleva, and J. H. Moore, editors, *Genetic Programming Theory and Practice IX*, Genetic and Evolutionary Computation, chapter 1, pages 1–16. Springer, Ann Arbor, USA, 12-14 May 2011.

[5] L. Spector, K. I. Harrington, and T. Helmuth. Tag-based modularity in tree-based genetic programming. In T. Soule and J. H. Moore, editors, *GECCO*, pages 815–822. ACM, 2012.

[6] L. Spector and J. Klein. Genetic stability and territorial structure facilitate the evolution of tag-mediated altruism. *Artificial Life*, 12(4):1–8, 2006.

[7] L. Spector, J. Klein, and M. Keijzer. The push3 execution stack and the evolution of control. In H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, W. Banzhaf, C. Blum, E. W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G. R. Raidl, T. Soule, A. M. Tyrrell, J.-P. Watson, and E. Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA, 25-29 June 2005. ACM Press.

[8] L. Spector, B. Martin, K. Harrington, and T. Helmuth. Tag-based modules in genetic programming. In N. Krasnogor, P. L. Lanzi, A. Engelbrecht, D. Pelta, C. Gershenson, G. Squillero, A. Freitas, M. Ritchie, M. Preuss, C. Gagne, Y. S. Ong, G. Raidl, M. Gallager, J. Lozano, C. Coello-Coello, D. L. Silva, N. Hansen, S. Meyer-Nieberg, J. Smith, G. Eiben, E. Bernado-Mansilla, W. Browne, L. Spector, T. Yu, J. Clune, G. Hornby, M.-L. Wong, P. Collet, S. Gustafson, J.-P. Watson, M. Sipper, S. Poulding, G. Ochoa, M. Schoenauer, C. Witt, and A. Auger, editors, *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1419–1426, Dublin, Ireland, 12-16 July 2011. ACM.