

# An Abstract Machine Approach to Finite State Transduction Over Large Character Sets

Thilo Goetz                      Holger Wunsch  
IBM Watson Research Center      University of Tübingen

April 27, 2001

## Abstract

We describe an implementation of finite state transducers in terms of an abstract machine. That is, a transducer is compiled to binary code, which is then interpreted by the abstract machine. This is different from traditional approaches to transduction, where transducers are normally compiled into fixed-form transition tables. We will argue that our approach offers increased flexibility, extensibility and efficiency when compiling transducers over large character sets.

## 1 Introduction

Abstract machine implementations are standard techniques in compiler construction, where instead of a hardware machine, the target for compilation is an abstract machine implemented in software. Examples include implementations of functional and logic languages (e.g., [War83]), but also scripting languages like Perl, and, more recently, Java [JSGB00]. The advantages of an abstract machine implementation include increased efficiency with respect to interpreted approaches, and increased portability with respect to native compilation.

This work was carried out as a study on the applicability of abstract machine technology to finite state transduction. We were mostly interested in the increased flexibility such an approach offers. We intend to apply our work to information extraction for multiple languages (including Asian) over text retrieved from the internet. Without going into the details of character encoding headaches, it will generally be useful to be able to transduce over Unicode text.

When transducing over the full set of Unicode characters, it is immediately clear that the traditional table generation methods will run into trouble: the size of the character set is  $2^{16} = 65536$ , as opposed to  $2^8 = 256$  for a regular 8-bit character set<sup>1</sup>. It is therefore necessary to allow for a very compact transition encoding, but at the same time desirable to provide an efficient encoding for the many cases where a compact encoding is not necessary.

In our approach, it is possible to decide the encoding on a *per state* basis. That is because we do not encode all transitions in a monolithic table, but individually in abstract machine instructions.

Similarly, we use the flexibility of our approach for the efficient treatment of output. Output instructions are only generated when output is actually specified in the transducer. That means that we can, for example, compile a finite state recognizer with the same compiler to the same instruction set, with no additional overhead.

---

<sup>1</sup>The size of the Unicode character set is not exactly  $2^{16}$ , but for the sake of brevity, we will gloss over those issues here (e.g., see [Lun99]).

In future research, we will extend this work to transduction over non-atomic, structured objects. For an account of transducing over structured objects, as well as low-level compilation, see [Bra98].

In the rest of the paper, we will concentrate on the compiler and the instruction set. For space reasons, we can not go into the details of the run-time environment, but we will mention run-time issues as we go along.

## 2 The Compiler and the Instruction Set

For reasons of efficiency and simplicity of the run-time environment, we decided to compile transducers into bima-chines<sup>2</sup>, using the algorithms from [RS97]. In the light of the remarks in [vNG01] concerning the correctness of the functionality test, we may need to revisit this decision.

The compiler works roughly in three phases. In a first phase, the input language (a variant of the usual regular expressions) is translated into an internal FST representation. In a second phase, the internal representation is manipulated to generate an equivalent bimachine. In a final step, the low-level code for the bimachine is created from the internal FST representation. We will return to this step later.

In addition to the instructions, the generated code contains two data areas: a symbol table for the output of the transducer, and the emission function for the bimachine (see [RS97]). The emission function is used in the right-to-left traversal of the input to generate the actual output.

Below we list the instructions, together with their arity and a brief explanation.

**0: NONFINAL (1)**

A non-final state has been reached. The argument is a state id.

**1: FINAL (1)**

A final state has been reached. The argument is a state id.

**2: PUSHSTATEID (0)**

Push the last seen state on the stack. The stack remembers states seen on the left-to-right traversal. This information is needed when generating output later.

**3: POP (0)**

Pop a state id from the stack.

**4: BACKWARDS (0)**

Instruct the run-time environment to start right-to-left processing.

**5: FORWARDS (0)**

Instruct the run-time environment to start left-to-right processing.

**6: OUTPUT\_BM (0)**

Emit the output symbol that the emission function assigns to the triple  $\langle s_1, s_2, i \rangle$ , where  $s_1$  is the top state on the state stack,  $s_2$  is the current state in the right-to-left traversal, and  $i$  is the current input symbol.

**7: TRANS\_LIST (n+1)**

Give the transitions from a state as a list of input symbol/jump address pairs. The first cell after the instruction gives the length  $n$  of the list.

---

<sup>2</sup>This means that we can't handle non-functional transducers.

#### 8: TRANS\_ARRAY ( $n+2$ )

Give the transition from a state as a dense list of jump addresses. The first argument gives an offset, the smallest input symbol for which the transition is defined. The second argument is  $n$ , the length of the list. That is, the largest input symbol for which the transition is defined is  $\text{offset} + n$ .

As one can see, only a small set of instructions is needed. This is not surprising, since FSTs are very simple machines. So what is the advantage of our approach? Consider the last two instructions, which encode the actual transitions.

We ended up with two ways of encoding transitions. TRANS\_ARRAY is a traditional way to encode a transition. It is a dense array that defines transitions from a smallest up to a largest input symbol (character). Of course, the values of the array are in our case code addresses, and not state IDs.

Encoding transitions as dense arrays is wasteful if the transition is sparse. That is, if for most input symbols the array is defined for, the transition is actually undefined. For that case, we defined the TRANS\_LIST instruction. Here the transitions from a certain state are simply encoded as a key-sorted list of character/address pairs. The run-time environment uses binary search on this list to find the correct transition (if it exists), resulting in  $\log n$  access time, where  $n$  is the number of outgoing arcs. Notice that this is effectively equivalent to sorted row-ordered storage as discussed in [Kir97].

The compiler decides for each state individually how the outgoing transitions should be encoded, based on the size of the resulting code. It speculatively constructs both representations, and then compares the size. It then chooses the smaller one, modulo a user-defined factor that one can use to favor one representation over another.

For the most part, the code will consist of sequences of instructions as follows:

1. a FINAL or NONFINAL instruction,
2. a OUTPUT\_BM instruction if in BACKWARDS mode, and output is defined,
3. a PUSHSTATEID if in FORWARDS mode, or a POP when in BACKWARDS mode,
4. a TRANS\_LIST or TRANS\_ARRAY instruction.

For experimentation, we mainly used tokenization and stemming over English and French literary data, using dictionaries and data from INTEX [Sil99]. We also built a names recognizer for some English and Japanese MUC data that we translated into Unicode.

The simple, list based representation works surprisingly well. For the core run-time (not counting I/O), we generally experienced less than 10% slowdown compared to the dense representation. This may be due to the kind of automata we used for experimentation since even with the Japanese data, the character set we were using was still relatively small ( $< 1000$ ). We did experience the expected compression advantages as noted in [Kir97].

### 3 Conclusion and Outlook

The architecture presented here is clearly only a start. What we hope to have shown is that this kind of setup lends itself to relatively easy experimentation and extension, without sacrificing efficiency.

Among the things we plan to do in the future are:

- Add weighting. This should be a simple addition of an instruction.

- Experiment with other transition encodings as we build transducers with truly large character sets.
- Add support for non-functional transducers with factorization à la [Kem].

We are also working on extending the architecture for transduction over typed feature structures (cf. [Bra98]). However, this adds significant complexity to both the compiler and the run-time environment; to the compiler, because standard FST results no longer apply, as we change the transduction domain from a finite set of incomparable symbols to some sort of (usually infinite) partial order of feature structures; and to the run-time environment, since it needs to support some appropriate matching operation like subsumption or unification.

## References

- [Bra98] Sascha Brawer. Patti: Compiling unification-based finite-state automata into machine instructions for a superscalar pipelined risc processor. Master's thesis, Universität des Saarlandes, Saarbrücken, 1998.
- [JSGB00] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2000.
- [Kem] André Kempe. Factorization of ambiguous finite state transducers. In Pre-Proceedings of the 5th International Conference on Implementation and Application of Automata (CIAA). To be published.
- [Kir97] George Anton Kiraz. Compressed storage of sparse finite-state transducers. In *Workshop on Implementing Automata WIA99 – Pre-Proceedings*. 1997.
- [Lun99] Ken Lunde. *CJKV Information Processing*. O'Reilly, 1999.
- [RS97] Emmanuel Roche and Yves Schabes. Introduction. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 1–66. MIT Press, 1997.
- [Sil99] Max Silberstein. Text indexation with INTEX. *Computers and the Humanities*, 33(3):265–280, 1999.
- [vNG01] Gertjan van Noord and Dale Gerdeman. Finite state transducers with predicates and identity. Draft, 2001.
- [War83] David H. D. Warren. An abstract Prolog instruction set. Technical note 309, SRI International, 1983.