Efficient Implementation of Subsequential Transducers

Garrick G. Trowsdale

Research Report CS-88-20

May 1988

Faculty of Mathematics

University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Efficient Implementation of Subsequential Transducers

Garrick G. Trowsdale

Research Report CS-88-20

May 1988

Efficient Implementation of Subsequential Transducers

by

Garrick Gavin Trowsdale

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, 1988

Abstract

Efficient Implementation of Subsequential Transducers

Ideally, software tools such as programming languages, database management systems, and parser generators should provide for high level specification and efficient solution of a particular class of problems; however, many systems meet only one of these criteria. INR is a program for specifying and optimizing finite automata, but not directly for their implementation. This thesis describes INRC, a program for compiling subsequential transducers computed by INR. Together, INR and INRC represent an efficient high level problem solving tool.

Acknowledgements

I have been very fortunate to write this thesis under the supervision of Howard Johnson, who suggested a particularly interesting topic, and who has provided encouragement and many ideas. Investigating strategies for the implementation of subsequential transducers has allowed me to pursue my interest in compilation issues and to achieve results in a reasonable amount of time.

Gaston Gonnet and Jose Icaza shared the research they had done on implementation of transducers, answering many questions, and providing valuable suggestions. The crucial idea of using dynamic programming, described in Chapter 3, belongs to them. Tim Bray, Gord Cormack, and Amar Sanmugasunderam all spent time discussing compile time and execution time performance issues; many of their ideas have been incorporated directly.

I thank Gaston Gonnet and Gord Cormack for taking time to read drafts and provide valuable comments.

Finally, I thank NSERC for the generous provision of scholarship funds.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Motivation

Efficiency is the strongest reason for using a low level programming language over a high level one; but even when performance suffers, high level languages are often chosen since they provide benefits such as understandability, maintainability, and portability. We want to specify *what* is to be done rather than *how* it is to be done. For instance, using the PROLOG programming language [SS86], complex algorithms may be written trivially, but efficiency remains an issue; nonetheless, work on the definition and construction of PROLOG machines demonstrates the value of high level specification of problems and their solutions.

A triumph of Computing Theory is LR parser generator theory and practice. LALR parser generators, such as YACC [Joh75], accept a simple specification of a language L and generate an *efficient* parser for L. Such a parser is a pushdown automaton which is "run" interpretively with an input string, producing a parse tree for the input. More specifically, the parser is a simple interpreter which parses input strings according to a parsing table prepared by the parser generator. This parsing table is a representation of a pushdown automaton for the language to be parsed.

INR is similar to a parser generator in the sense that it accepts a high level specification of a finite automaton and produces a representation suitable for interpretation or compilation; regular expressions and generalized rational expressions are used to denote automata in INR [Joh87a]. Two tape automata, called *transducers*, are one class of automata computed by INR, and are very useful for transforming text. The transformation specified by a transducer is a *transduction*. The *application* of a transduction to input text produces transformed output text; transduction application is analogous to LR parsing.

Instances where transduction is useful include ad-hoc transformation of large quantities of text and repeated application of a fixed transformation to small quantities of text. An example of a large body of text is the New Oxford English Dictionary; "one time" transformations can be extremely time consuming. An example of a fixed transformation is the Soundex coding scheme, which is used in reservation systems, and which is applied frequently to customer surnames. In both cases, efficiency is clearly important.

There are two fundamental approaches to the application of a transduction to an input string. The first is to run a transducer for the transduction interpretively, and the second is to compile it into a non-interpretive program which then applies the transduction to its input. INRC is a compiler for a strict sub-class of finite transductions, namely subsequential functions. The motivation behind INRC is to complement INR's high level specification capabilities and automaton optimization with efficient implementation.

1.2. Previous Work

The programs developed at the University of Waterloo which form the context in which INRC was developed are listed below.

INR	●	J. H. Johnson
	-	computes finite automata
lsim	●	J. H. Johnson and F. W. Tompa
	-	applies general non-deterministic transductions
	-	interpretive
	-	provides error recovery mechanisms
*gsm*1	●	J. H. Johnson
	-	applies only subsequential transductions
	-	interpretive
mscan	●	G. H. Gonnet and J. I. Icaza
	-	compiles general non-deterministic transducers into C language programs
	-	provides error recovery mechanisms

Johnson wrote INR in conjunction with the development of string similarity models, which includes analysis of the expressive power of rational relations, and subsequential functions in particular [Joh83]. The *lsim* and *gsm*1 programs read a transducer and input text, transforming the input text using the transducer to produce output text. Kazman utilizes INR and *lsim* to augment the structural information of the New Oxford English Dictionary as one step in its computerization [Kaz86]. The *gsm*1 program applies only subsequential transductions, and realizes a ten-fold speed increase over *lsim*.

The work of Gonnet and Icaza on efficient implementation of general non-deterministic two tape automata led to the *mscan* program. Their approach is to generate C language programs which implement the states of an automaton directly. By permitting non-determinism, mechanisms for trying different paths through the machine are necessary. This problem is complicated further by cycles, which necessitate limits on how far alternative paths are followed. Moreover, retreating from an incorrect path involves undoing the operations performed on the two tapes during the exploration. In addition to these types of problems, many compiler restrictions were encountered, such as program and subroutine size, switch statement size, and long jump code generation errors. Despite these difficulties, compiled automata run significantly faster than they can be interpreted.

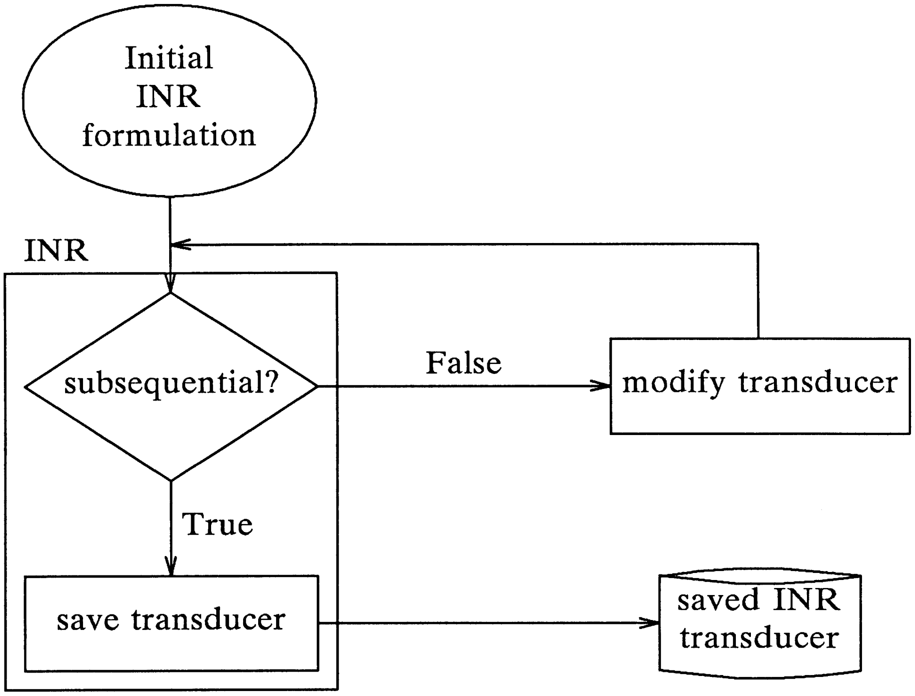
1.3. INRC — A Transducer Compiler

Informally, a subsequential transducer is a deterministic finite automaton (DFA) with output; using a computationally convenient characterization, a subsequential transducer has input and output states. Input states, which cause one input symbol to be read, are analogous to non-final states in a DFA; output states simply cause one output symbol to be written. INRC is a compiler for subsequential transducers computed by INR.

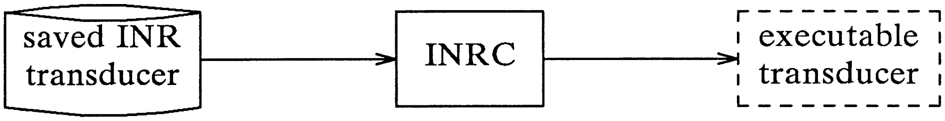
Problem solving using INR and INRC is illustrated in Figure 1.1. The design goals of INRC include the following:

- transducer modularity
  - INRC generates user callable transducers, but a standard driver is included by default, yielding executable programs
  - transducers support assembler and C language calling conventions
- highly efficient transducer object code
- practical transducer compile time complexity

1. Formulate the problem as a transduction.
2. Convert to INR syntax / semantics.
3. Iterate to obtain a subsequential transduction:



4. Compile the saved transducer using INRC:



5. Execute the transducer:

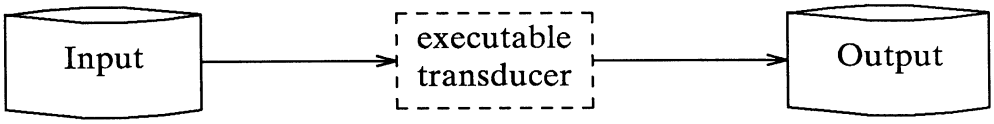


Figure 1.1: Problem solving steps using INR and INRC

In order to achieve the goal of highly efficient transducer object code, two "layers" of optimization are performed. The first layer of optimization is performed by INR's automaton state minimization. The second layer, performed by INRC, includes transducer input state optimization and span dependent branch optimization. Transducer input state optimization is essentially the same as case statement optimization for high level languages; output states are so simple that they provide little opportunity for optimization. Span dependent branch optimization is very important since a large number of the instructions executed by a transducer are branch instructions.

1.4. Implementations

INRC is fully operational on two computer systems: a VAX 8650 running 4.3 BSD Unix, and an IBM 4341 running VM/CMS. The New Oxford English Dictionary project at the University of Waterloo has successfully used INR and INRC to perform tag transduction and invoke index construction routines for the entire text (470 megabytes) of the New Oxford English Dictionary.

1.5. Thesis Outline

The expressive power and computability of the sub-class of rational relations which may be computed by subsequential transducers is the subject of Chapter 2. Optimization techniques employed in INRC are detailed in Chapters 3 through 5. Results obtained with the VAX and IBM implementations are presented in Chapters 6 and 7.

Chapter 2

Subsequential Functions

The most powerful transducers are rational transducers, which characterize rational relations. Removing the non-determinism of rational transducers leads to subsequential transducers, which characterize subsequential functions. Machine models for rational and subsequential transducers are formally defined, followed by important results which make the computation of subsequential transducers practical. Chapter 3 relies heavily on the final characterization of subsequential transducers presented.

2.1. Alphabets, Languages, and Machines

Definition: An *alphabet* is a finite, non-empty set of atomic symbols.

Definition: A *string* over an alphabet Σ is the catenation of a finite sequence of symbols from Σ; the *empty string*, denoted ϵ, contains no symbols.

Definition: Σ^* contains all strings over Σ; Σ^* is called the universal language.

Definition: A *language* L over an alphabet Σ is a set of strings which satisfies $L \subseteq \Sigma^*$.

Definition: A *language relation* R over alphabets Σ and Δ is a set of string pairs which satisfies $R \subseteq \Sigma^* \times \Delta^*$.

From the theory of languages over a single alphabet, we may say that deterministic finite automata and non-deterministic pushdown automata *accept* or *recognize* regular and context free languages. The distinction between a language L and an instance of a machine model that recognizes L is important: the *behaviour* of a machine M is the language accepted by M. Thus, the behaviour of a deterministic finite automaton is a regular language and the behaviour of a non-deterministic pushdown automaton is a context free language. Similarly, the behaviour of a transducer is a language relation. Formally, a transducer may be viewed as an acceptor of string pairs, but in practice, a transducer is viewed as a traditional program which reads input and writes it as output after transformation.

Definition: The *behaviour* of a machine M, denoted $|M|$, is the language (relation) it accepts.

Definition: The family of all languages recognizable by instances of the machine model M is denoted L_M.

2.2. Disjoint Unions

In the definitions of transducers that follow, the disjoint union of input and output alphabets is used to represent symbols from both alphabets as symbols of single alphabet. Disjoint union is defined operationally rather than by the properties it must satisfy, since this is what we need.

Definition: The *disjoint union* of two finite sets A and B is

$$A \dot{\cup} B \equiv (\{0\} \times A) \cup (\{1\} \times B)$$

with the two partial projection functions

$$\pi_A((0,a) = \begin{cases} a & \text{if } a \in A \\ \Omega & \text{otherwise} \end{cases}$$

$$\pi_B((1,b)) = \begin{cases} b & \text{if } b \in B \\ \Omega & \text{otherwise} \end{cases}$$

Any equivalent formalism which has the same behaviour may be considered $A \dot{\cup} B$. For instance, If Σ is an input alphabet and Δ is an output alphabet, INR forms the disjoint union $\Sigma \dot{\cup} \Delta$ as $(\{0.\} \times \Sigma) \cup (\{1.\} \times \Delta)$ with

$$\pi_\Sigma(0.a) = a$$
$$\pi_\Delta(1.a) = a$$

where a tuple $(0.,a)$ is abbreviated as $0.a$. Another example is Algol68 united modes, which require the use of a conformity clause to access alternate mode values: the conformity clause selects and applies the appropriate projection function.

In order to define the behaviour of transducers whose alphabet is $\Sigma \dot{\cup} \Delta$, we define two total "string" projection functions

$$\Pi_\Sigma : (\Sigma \dot{\cup} \Delta)^* \rightarrow \Sigma^*$$
$$\Pi_\Delta : (\Sigma \dot{\cup} \Delta)^* \rightarrow \Delta^*$$

Let $\dot{a}$ denote any element of $\Sigma \dot{\cup} \Delta$. By adding ϵ to the range of π_Σ and π_Δ, they may be made total.

$$\pi_\Sigma^T(\dot{a}) = \begin{cases} a & \text{if } \pi_\Sigma(\dot{a}) = a \\ \epsilon & \text{otherwise} \end{cases}$$

$$\pi_\Delta^T(\dot{a}) = \begin{cases} a & \text{if } \pi_\Delta(\dot{a}) = a \\ \epsilon & \text{otherwise} \end{cases}$$

Now Π_Σ and Π_Δ may be defined in terms of π_Σ^T and π_Δ^T as follows:

$$\Pi_\Sigma(\dot{a}_1\dot{a}_2 \cdots \dot{a}_n) = \pi_\Sigma^T(\dot{a}_1)\ \pi_\Sigma^T(\dot{a}_2)\ \cdots\ \pi_\Sigma^T(\dot{a}_n)$$
$$\Pi_\Delta(\dot{a}_1\dot{a}_2 \cdots \dot{a}_n) = \pi_\Delta^T(\dot{a}_1)\ \pi_\Delta^T(\dot{a}_2)\ \cdots\ \pi_\Delta^T(\dot{a}_n)$$

For instance, using INR's disjoint union,

$$\Pi_{\Sigma}(0.a\ 1.b\ 1.c\ 0.b) = ab$$
$$\Pi_{\Delta}(0.a\ 1.b\ 1.c\ 0.b) = bc$$

2.3. Regular Languages and Finite Automata

The well known deterministic finite automaton model is defined here to give the flavour of the transducer definitions to follow.

Definition: A language $L \subseteq \Sigma^*$ is a *regular* language if there exists a deterministic finite automaton whose behaviour is L.

Definition: A deterministic finite automaton (DFA) is a 5-tuple:

$$<\Sigma, Q, s, F, \delta>$$

where

Σ	is an alphabet
Q	is a finite set of states
$s \in Q$	is the start state
$F \subseteq Q$	is a set of final states
δ	is a transition function

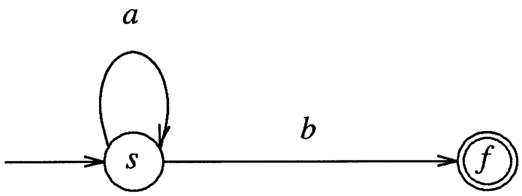
and

$$\delta : Q \times \Sigma \rightarrow Q$$

The behaviour of a DFA M is

$$|M| = \{ x \mid x \in \Sigma^* \text{ and } \exists \text{ a path from } s \text{ to } f \in F \text{ such that the catenation of transition labels is } x \}$$

A DFA whose behaviour is $\{ a^i b \mid i \geq 0 \}$ is pictured below.



2.4. Rational Relations and Rational Transducers

Definition: a language relation $R \subseteq \Sigma^* \times \Delta^*$ is a *rational relation* if there exists a *rational transducer* whose behaviour is R.

Three equivalent machine models of rational transducers are presented. The first corresponds most closely with how one might think of a transducer; it is also convenient for proving properties. The second model is an intermediate between the first and third models. And, the third model is the most computationally convenient—it is the model used by INR.

Definition: A rational transducer (T_{R1}) is a 6-tuple:

$$<\Sigma,\Delta,Q,s,F,E>$$

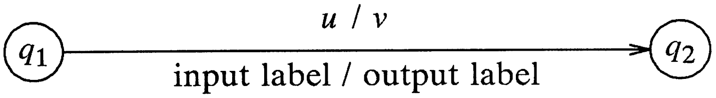
where

Σ is an input alphabet
Δ is an output alphabet
Q is a finite set of states
$s \in Q$ is the start state
$F \subseteq Q$ is a set of final states
E is a finite transition relation

and

$$E \subseteq Q \times \Sigma^* \times \Delta^* \times Q$$

A transition (q_1,u,v,q_2) is pictured thus:



The behaviour of a T_{R1} rational transducer T is

$|T| = \{ (u,v) \mid u \in \Sigma^*,\ v \in \Delta^*,$ and $\exists$ a path from s to $f \in F$ such that the catenation of the input labels is u, and the catenation of the output labels is v $\}$

We may restrict the transition relation so that transition labels contain at most one alphabet symbol without reducing the size of the family of languages recognized.

Definition: A rational transducer (T_{R2}) is a 6-tuple:

$<\Sigma,\Delta,Q,s,F,E>$

where

Σ	is an input alphabet
Δ	is an output alphabet
Q	is a finite set of states
$s \in Q$	is the start state
$F \subseteq Q$	is a set of final states
E	is a finite transition relation

and

$$\begin{aligned} E \subseteq\ & Q \times \Sigma \times \{\epsilon\} \times Q\ \cup \\ & Q \times \{\epsilon\} \times \Delta \times Q\ \cup \\ & Q \times \{\epsilon\} \times \{\epsilon\} \times Q \end{aligned}$$

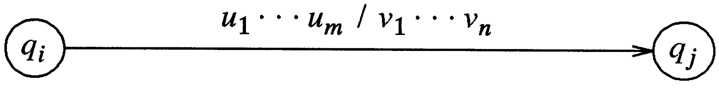
This means that each label specifies either one input symbol, one output symbol, or a ϵ-transition. The behaviour of a T_{R2} is defined the same way as for a T_{R1}.

Lemma 2.4.1: $L_{T_{R1}} = L_{T_{R2}}$

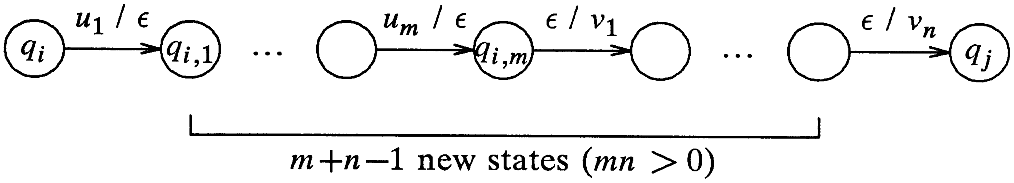
($L_{T_{R2}} \subseteq L_{T_{R1}}$) Any T_{R2} is also a T_{R1}.

($L_{T_{R1}} \subseteq L_{T_{R2}}$) Let $T_1 = <\Sigma,\Delta,Q_1,s,F,E_1>$ be a T_{R1}. We define an equivalent T_{R2} $T_2 = <\Sigma,\Delta,Q_2,s,F,E_2>$ by constructing Q_2 and E_2 from Q_1

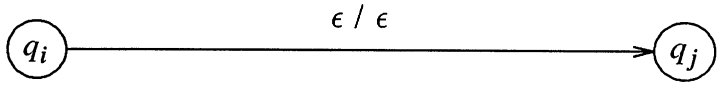
and E_1. Initially, define $Q_2 = Q_1$ and set $E_2 = \varnothing$; then, for each transition in E_1



add to E_2 the set of transitions



and add $\{ q_{i,1}, q_{i,2}, \ldots, q_{i,m+n-1} \}$ to Q_2. The transition



is added to E_2 without any new states in Q_2. By construction, $|T_1| \subseteq |T_2|$. Note that all "new" states in Q_2 are non-final states; thus, any path from s to $f \in F$ in T_2 including a new state q must contain the *complete* path between the two states in Q_1 whose transition label in E_1 caused q's addition to Q_2. Now suppose that $(u,v) \in |T_2|$; then there is a corresponding path $p_2 = s \cdots f \in F$ in T_2. But there is a path $p_1 = s \cdots f \in F$ in T_1 obtained by deleting all new states from p_2. Finally, $(u,v) \in |T_1|$ since we choose the transitions in E_1 that gave rise to the new states deleted from p_2. ■

The final model is derived from T_{R2}s by renaming transition labels. A bijection is defined between semantically equivalent T_{R2} and T_{R3} transition labels; thus, the models are trivially equivalent. Transition labels of a T_{R3}

are elements of the disjoint union of the input and output alphabets and $\{\epsilon\}$, enabling us to view a T_{R3} as a non-deterministic finite automaton over a special alphabet.

Definition: A rational transducer (T_{R3}) is a 6-tuple:

$$<\Sigma,\Delta,Q,s,F,E>$$

where

Σ	is an input alphabet
Δ	is an output alphabet
Q	is a finite set of states
$s \in Q$	is the start state
$F \subseteq Q$	is a set of final states
E	is a finite transition relation

and

$$E \subseteq Q \times ((\Sigma \dot{\cup} \Delta) \cup \{\epsilon\}) \times Q$$

The behaviour of a T_{R3} rational transducer T is defined as

$|T| = \{ (u,v) \mid u \in \Sigma^*, v \in \Delta^*$, and $\exists$ a path from s to $f \in F$ such that if L is the catenation of the transition labels, then $\Pi_\Sigma(L)$ is u, and $\Pi_\Delta(L)$ is v $\}$

Lemma **2.4.2**: $L_{T_{R2}} = L_{T_{R3}}$

A bijection between T_{R2} and T_{R3} transition labels is given below. ■

T_{R2} transition	T_{R3} transition
(q_1, a, ϵ, q_2)	$(q_1, 0.a, q_2)$
(q_1, ϵ, b, q_2)	$(q_1, 1.b, q_2)$
$(q_1, \epsilon, \epsilon, q_2)$	(q_1, ϵ, q_2)

2.5. Subsequential Functions and Subsequential Transducers

Definition: A function $f : \Sigma^* \rightarrow \Delta^*$ is a *subsequential function* if there exists a *subsequential transducer* whose behaviour is f.

As with rational transducers, three equivalent machine models of subsequential transducers are presented. The first is the mathematical model used by Berstel [Ber79]. The second model corresponds to deterministic generalized sequential machines (DGSMs) with endmarkers [Gin66]. The third model is computationally convenient, and is the one used by INR. In the same way that it is sometimes more convenient to use non-deterministic finite automata instead of deterministic finite automata in proving properties about regular languages, it is sometimes more convenient to use Berstel's model rather than the computational model when proving properties about subsequential functions.

Definition: A subsequential transducer (T_{S1}) is a 7-tuple:

$$<\Sigma,\Delta,Q,s,\delta,\lambda,\rho>$$

where

Σ	is an input alphabet
Δ	is an output alphabet
Q	is a finite set of states
$s \in Q$	is the start state
δ	is a finite transition function
λ	is a finite output function
ρ	is a finite final output function

and

$$\left.\begin{array}{l}\delta : Q \times \Sigma \rightarrow Q \\ \lambda : Q \times \Sigma \rightarrow \Delta^*\end{array}\right\} \text{ specify transition labels}$$

$\rho : Q \rightarrow \Delta^*$ specifies final states

The input and output functions δ and λ are defined over the same domain:

$\delta(q,a)$ is defined $\Leftrightarrow$ $\lambda(q,a)$ is defined

Note that δ, λ, and ρ are not necessarily total functions. A T_{S1} starts in state s and makes transitions using δ like a DFA; while input is read using δ, output is written using λ. When the input is exhausted, the machine is in some state q. If $\rho(q)$ is defined, the final output $\rho(q)$ is written; otherwise, the machine hangs in state q. Together, δ and λ serve to define transition labels $\in Q \times \Sigma \times \Delta^* \times Q$, and ρ serves to designate final states by where it is defined since a T_{S1} has no final state set. Thus, the behaviour of a T_{S1} subsequential transducer T is

$|T| = \{ (u,v) \mid u \in \Sigma^*, v \in \Delta^*$, and $\exists$ a path from s to f such that
$\rho(f)$ is defined and 1) the catenation of input labels
is u, and 2) the catenation of output labels and $\rho(f)$
is v $\}$

A T_{S1} subsequential transducer is shown in Figure 2.1.

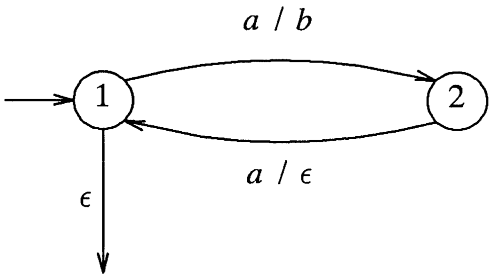
An alternative to the final output function ρ is to terminate each input string with a readable endmarker symbol $\dashv \notin \Sigma$, and add a single final state with no outgoing transitions.

Definition: A subsequential transducer (T_{S2}) is a 7-tuple:

$<\Sigma \cup \{\dashv\}, \Delta, Q, s, f, \delta, \lambda>$

where

Σ	is an input alphabet
Δ	is an output alphabet
Q	is a finite set of states
$s \in Q$	is the start state
$f \in Q$	is the unique final state
δ	is a finite transition function



$$\begin{array}{lcll}
\Sigma & = & \{a\} & \\
\Delta & = & \{b\} & \\
Q & = & \{1,2\} & \\
s & = & 1 & \\
\delta & = & (1,a) & \rightarrow 2 \\
 & & (2,a) & \rightarrow 1 \\
\lambda & = & (1,a) & \rightarrow b \\
 & & (2,a) & \rightarrow \epsilon \\
\rho & = & 1 & \rightarrow \epsilon
\end{array}$$

Figure 2.1: T_{S1} subsequential transducer for $f(a^{2i}) \rightarrow b^i \quad i \geq 0$

λ is a finite output function

and

$$\delta : Q \times (\Sigma \cup \{\dashv\}) \rightarrow Q$$
$$\lambda : Q \times (\Sigma \cup \{\dashv\}) \rightarrow \Delta^*$$

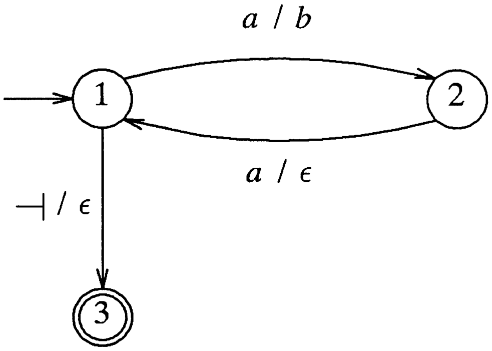
The input and output functions δ and λ are defined over the same domain; all endmarker ($\dashv$) transitions lead to the final state; and the final state may have no outgoing transitions:

$$\delta(q,a) \text{ is defined } \Leftrightarrow \lambda(q,a) \text{ is defined}$$
$$\delta(q,\dashv) \notin Q - \{f\} \; \forall \; q \in Q$$
$$\delta(f,a) \text{ is undefined } \forall \; a \in (\Sigma \cup \{\dashv\})$$

Note that δ, λ, and ρ are not necessarily total functions. The behaviour of a T_{S2} subsequential transducer T is

$|T| = \{ (u,v) \mid u \in \Sigma^*, v \in \Delta^*$, and $\exists$ a path from s to f such that the catenation of input labels is $u\dashv$, and the catenation of output labels is v $\}$

A T_{S2} subsequential transducer is shown in Figure 2.2.



$$\begin{array}{lcll}
\Sigma & = & \{a\} & \\
\Delta & = & \{b\} & \\
Q & = & \{1,2,3\} & \\
s & = & 1 & \\
f & = & 3 & \\
\delta & = & (1,a) & \rightarrow 2 \\
 & & (1,\dashv) & \rightarrow 3 \\
 & & (2,a) & \rightarrow 1 \\
\lambda & = & (1,a) & \rightarrow b \\
 & & (1,\dashv) & \rightarrow \epsilon \\
 & & (2,a) & \rightarrow \epsilon
\end{array}$$

Figure 2.2: T_{S2} subsequential transducer for $f(a^{2i}) \rightarrow b^i \quad i \geq 0$

Lemma 2.5.1: $L_{T_{S1}} = L_{T_{S2}}$

($L_{T_{S1}} \subseteq L_{T_{S2}}$) Let $T_1 = <\Sigma,\Delta,Q_1,s,\delta,\lambda_1,\rho>$ be a T_{S1}. We define an equivalent T_{S2} $T_2 = <\Sigma \cup \{\dashv\},\Delta,Q_2,s,f,\delta,\lambda_2>$ as follows:

$$Q_2 \quad = \quad Q_1 \cup \{f\}$$

$$
\begin{array}{lcl}
\delta_2 & = & \delta_1 \cup \{\, (q,\dashv) \rightarrow f \mid \rho(q) \text{ is defined } \} \\
\lambda_2 & = & \lambda_1 \cup \{\, (q,\dashv) \rightarrow x \mid \rho(q) \rightarrow x \,\}
\end{array}
$$

($L_{T_{S2}} \subseteq L_{T_{S1}}$) Let $T_2 = <\Sigma \cup \{\dashv\}, \Delta, Q_2, s, f, \delta, \lambda_2>$ be a T_{S1}. We define an equivalent T_{S1} $T_1 = <\Sigma, \Delta, Q_1, s, \delta, \lambda_1, \rho>$ as follows:

$$
\begin{array}{lcl}
Q_1 & = & Q_2 - \{ f \} \\
\delta_1 & = & \delta_2 - \{\, (q,\dashv) \rightarrow f \mid \delta_2(q) \rightarrow f \,\} \\
\lambda_1 & = & \lambda_2 - \{\, (q,\dashv) \rightarrow x \mid \lambda_2(q) \rightarrow x \,\} \\
\rho & = & \{\, q \rightarrow x \mid \delta_2(q,\dashv) \rightarrow f \text{ and } \lambda_2(q,\dashv) \rightarrow x \,\}
\end{array}
$$

■

The third and final model restricts transition labels to elements of the disjoint union of the input and output alphabets.

Definition: A subsequential transducer (T_{S3}) is a 6-tuple:

$<\Sigma \cup \{\dashv\}, \Delta, Q, s, f, \delta>$

where

Σ	is an input alphabet
Δ	is an output alphabet
Q	is a finite set of states
$s \in Q$	is the start state
$f \in Q$	is the unique final state
δ	is a finite transition function

and

$$\delta : Q \times ((\Sigma \cup \{\dashv\}) \dot{\cup} \Delta) \rightarrow Q$$

The final state may have no outgoing transitions:

$\delta(f,a)$ is undefined $\forall\ a \in (\Sigma \cup \{\dashv\})$

The state set may be partitioned into input states and output states:

$$Q = Q_I \cup Q_O \cup \{ f \}$$

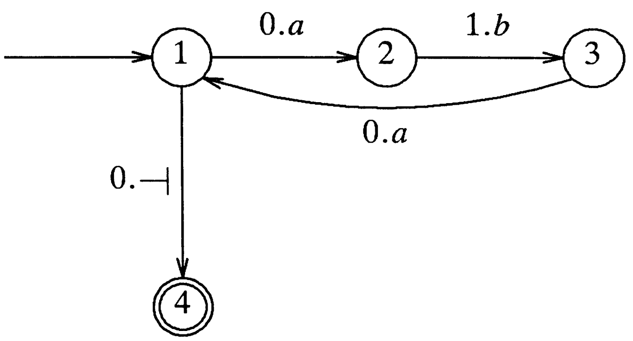
All transitions from an input state must specify input symbols, and there must be only one transition from an output state, which specifies an output symbol:

$$\begin{array}{ll} \forall\ q \in Q_I & : \delta(q,a) \text{ is defined} \Leftrightarrow \Pi_\Sigma(a) \text{ is defined} \\ \forall\ q \in Q_O & : \delta(q,a) \text{ is defined} \Leftrightarrow \Pi_\Delta(a) \text{ is defined} \\ \forall\ q \in Q_O & : \delta(q,a) \text{ defined and } \delta(q,b) \text{ defined} \Rightarrow a = b \end{array}$$

The behaviour of a T_{S3} subsequential transducer T is

$$|T| = \{ (u,v) \mid u \in \Sigma^*,\ v \in \Delta^*, \text{ and } \exists \text{ a path from } s \text{ to } f \text{ such that if } L \text{ is the catenation of transition labels, then } \Pi_\Sigma(L) \text{ is } u\dashv \text{ and } \Pi_\Delta(L) \text{ is } v \ \}$$

A T_{S3} subsequential transducer is shown in Figure 2.3.

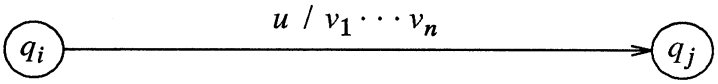


$$\begin{array}{lcll} \Sigma & = & \{ a \} & \\ \Delta & = & \{ b \} & \\ Q & = & \{1,2,3,4\} & \\ s & = & 1 & \\ f & = & 4 & \\ \delta & = & (1,0.a) & \rightarrow 2 \\ & & (1,0.\dashv) & \rightarrow 4 \\ & & (2,1.b) & \rightarrow 3 \\ & & (3,0.a) & \rightarrow 1 \end{array}$$

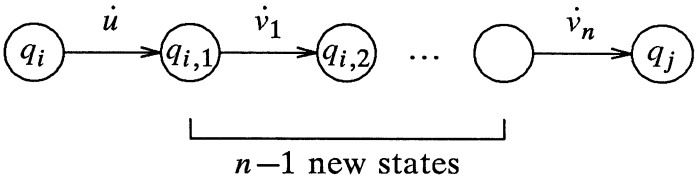
Figure 2.3: T_{S3} subsequential transducer for $f(a^{2i}) \rightarrow b^i \quad i \geq 0$

Lemma 2.5.2: $L_{T_{S2}} = L_{T_{S3}}$

$(L_{T_{S2}} \subseteq L_{T_{S3}})$ let $T_2 = <\Sigma \cup \{\dashv\},\Delta,Q_2,s,f,\delta_2,\lambda_2>$ be a T_{S2}. We define an equivalent T_{S3} $T_3 = <\Sigma \cup \{\dashv\},\Delta,Q_3,s,f,\delta_3>$ by constructing Q_3 and δ_3 from Q_2, δ_2, and λ_2. Initially, define $Q_3 = Q_2$ and set δ_3 undefined everywhere; then, for each transition in T_2 ($\delta_2(q_i,u) = q_j$ and $\lambda(q_i,u) = v_1 \cdots v_n$)



add to δ_3 the set of transitions



where $\pi_\Sigma(\dot{u}) = u$, $\pi_\Delta(\dot{v}_i) = v_i$ for $1 \leq i \leq n$, and add $\{ q_{i,1},q_{i,2},\ldots,q_{i,n-1} \}$ to Q_3. By construction, $|T_2| \subseteq |T_3|$. Since all "new" states in Q_3 are non-final and have one ingoing and one outgoing transition, any path from s to f in T_3 including a new state q must contain the *complete* path between the two states in Q_2 whose transition label specified by δ_2 and λ_2 caused q's addition to Q_3. Now suppose that $(u,v) \in |T_3|$; then there is a corresponding path $p_3 = s \cdots f$ in T_3. But there is a path $p_2 = s \cdots f$ in T_2 obtained by deleting all new states from p_3. By the construction of T_3, $(u,v) \in |T_2|$.

$(L_{T_{S3}} \subseteq L_{T_{S2}})$ Suppose that T is a T_{S3}. The procedure for constructing an equivalent T_{S2} transducer from T consists of the following steps:

1. Partition the states of T into input and output states.

   Transition labels from input states specify symbols in $\Sigma \cup \{\dashv\}$, and transition labels from output states specify symbols in Δ; the final state is special since it has no outgoing transitions.

2. Accumulate output "strings" and attach them to the immediately preceding input state.

   Each sequence of consecutive output states is finite and does not form a cycle since a T_{S3} subsequential transducer is deterministic and must read the $\dashv$ symbol. Thus, output symbols specified by each consecutive sequence of output states may be catenated to form a finite length output string. The T_{S2} transducer states correspond to the T_{S3} input states, and the T_{S2} transition labels are constructed from each T_{S3} input transition, and the immediately following output string. A small problem arises when the T_{S3} transducer performs output before reading any input, which leads to the next step.

3. Deal with "leading output" if present.

   Output states preceding the first input state may only occur if the start state is an output state; furthermore, there may be at most one sequence of leading output states. It is not sufficient to move a leading output string to the *subsequent* input state if that state has any cycles involving output; thus, extra states must be introduced to solve this problem.

■

Consider the T_{S3} transducer T, Figure 2.4, whose behaviour is

$$|T| = \{\, a^i \rightarrow lead b^i ,\ b^i \rightarrow lead c^{2i} \,\}$$

and where

$((\Sigma \cup \{\dashv\}) \dot{\cup} \Delta)$ is formed as $\{ \{0.\} \times (\Sigma \cup \{\dashv\}) \} \cup \{ \{1.\} \times \Delta \}$

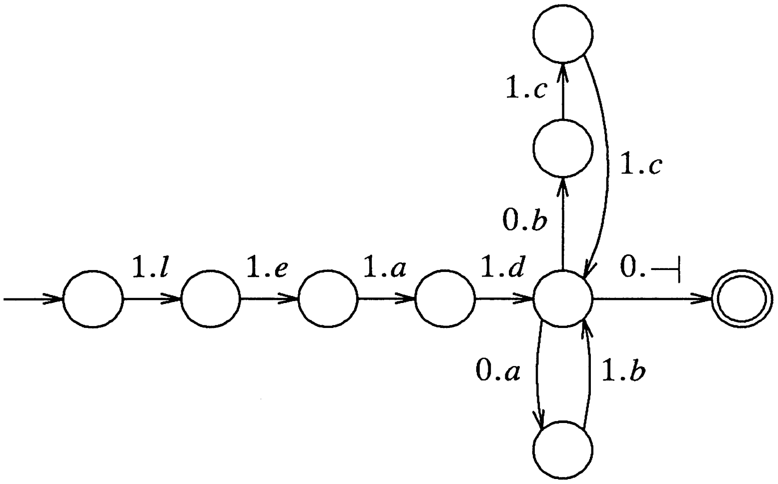


Figure 2.4: T_{S3} transducer T

Applying steps 1 and 2 leads to the transducer shown in Figure 2.5.

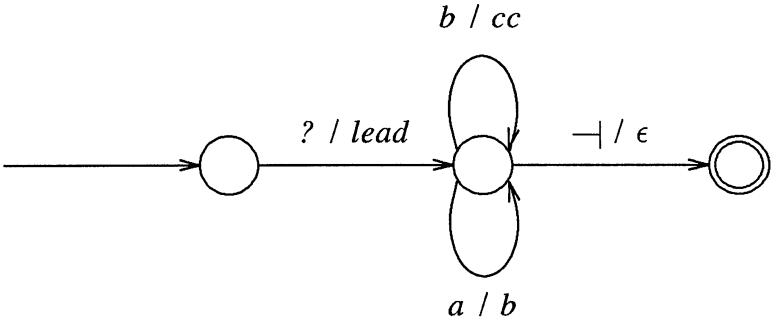


Figure 2.5: Result of applying steps 1 and 2 to T

And finally, step 3 yields the equivalent T_{S2} transducer shown in Figure 2.6.

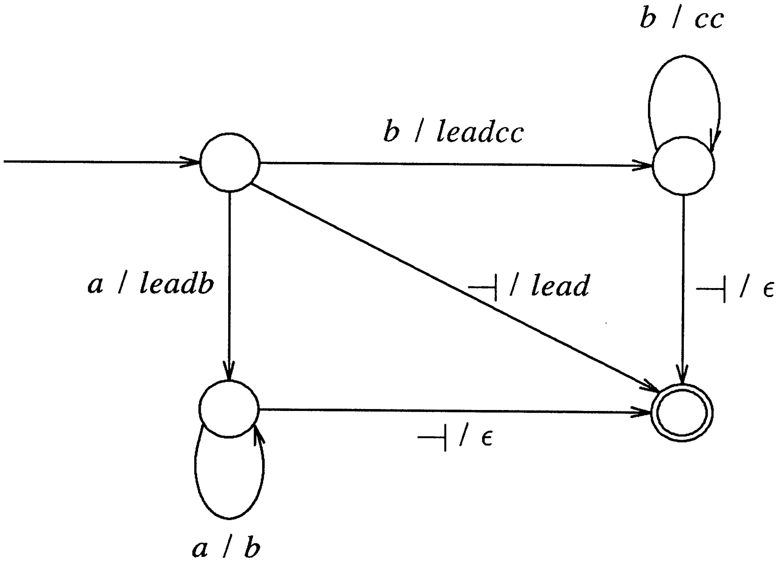


Figure 2.6: T_{S2} transducer equivalent to T

2.6. Important Results

In addition to rational relations, we may consider the language class of rational *functions*; similarly, in addition to subsequential functions, we may consider *finite* functions:

$L_{Rational}$ = rational relations: $R \subseteq \Sigma^* \times \Delta^*$
L_{RatFcn} = rational relations that are also functions: $F : \Sigma^* \longrightarrow \Delta^*$
$L_{Subsequential}$ = subsequential functions: $f : \Sigma^* \longrightarrow \Delta^*$
L_{Finite} = finite functions: $f : \Sigma^* \longrightarrow \Delta^*$

These language classes are related by the following hierarchy:

$$L_{Finite} \subset L_{Subsequential} \subset L_{RatFcn} \subset L_{Rational}$$

For each langauge class, membership in a more restricted class is decidable;

furthermore, a machine belonging to a more restricted class can be computed from a machine belonging to a more general class providing that membership in both classes holds.

Theorem 2.6.1: $L_{Finite} \subseteq L_{Subsequential}$.

Proof: Suppose $f : \Sigma^* \rightarrow \Delta^*$ is finite. Then f is a language relation $L = \{ (u_i,v_i) \mid 1 \leq i \leq n \}$ such that $u_i = u_j \Rightarrow i = j$. Therefore, we can construct a trie for the domain of L. A T_{S1} subsequential transducer whose behaviour is L is constructed from the trie:

$T = <\Sigma,\Delta,Q,s,\delta,\lambda,\rho>$

where

Σ	is the alphabet of the domain $\{ u_i \mid 1 \leq i \leq n \}$
Δ	is the alphabet of the range $\{ v_i \mid 1 \leq i \leq n \}$
Q	contains one state for each node in the trie
s	is the root of the trie
δ	is defined from the trie
λ	has value ϵ for all $Q \times \Sigma$ combinations in the trie
ρ	is $\{ (q_i,v_i) \mid q_i$ is the terminal node for $u_i \}$

■

Theorem 2.6.2: $L_{Finite} \neq L_{Subsequential}$.

Proof: The function $f(a^i) \rightarrow b^i$ for $i \geq 0$ is subsequential, but not finite. ■

Theorem 2.6.3: $L_{Subsequential} \subseteq L_{RatFcn}$.

Proof: Without loss of generality, we may assume that any subsequential transducer is a T_{S3} subsequential transducer. But a T_{S3} subsequential transducer is also a T_{R3} rational transducer. Furthermore, every subsequential transducer's behaviour is a function. ■

Lemma 2.6.4: The function

$$f(a^i) = \begin{cases} a^i & \text{if } i \text{ is even} \\ \epsilon & \text{otherwise} \end{cases}$$

is a rational function.

Proof: A T_{R1} rational transducer T such that $|T| = f$ is given in Figure 2.7.

■

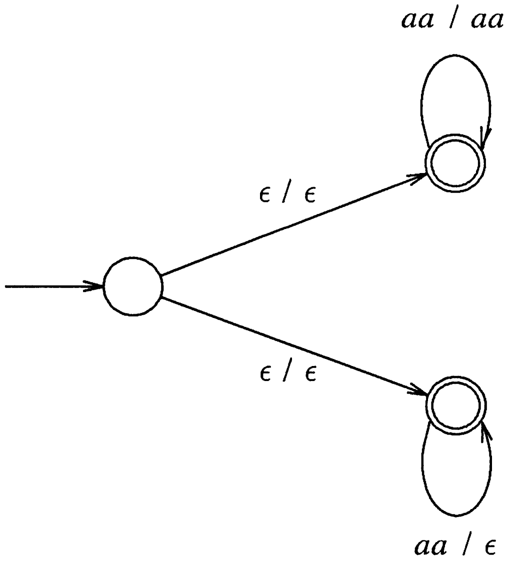


Figure 2.7: Proof of Lemma 2.6.4

Lemma 2.6.5: The function

$$f(a^i) = \begin{cases} a^i & \text{if } i \text{ is even} \\ \epsilon & \text{otherwise} \end{cases}$$

is not a subsequential function.

Proof: Assume that there exists a T_{S1} subsequential transducer T whose behaviour is f. Let k be the maximum number of symbols in the defined range values of λ and ρ. After reading a^{2t+1}, T must have written ϵ; if not, and the input string is a^{2t+1}, then $|T| \neq f$. If the input string is a^{2t+2}, then T must write a^{2t+2}. But T cannot write a^{2t+2} if $k < t + 1$ since it may make only one more λ-transition and write a ρ-output string. ■

The significance of the following theorems is that the expressive power of subsequential functions is not as great as the expressive power of rational relations. Some desirable transductions are rational but not subsequential. For instance, suppose that we would like to extract all dictionary entries that are "interesting", where an arbitrary proportion of an entry must be examined to determine if it is interesting—this represents a non-subsequential transduction. Functions like this, and the function of Lemmas 2.6.4 and 2.6.5, are "inherently non-deterministic" since a deterministic transducer would need to examine, and either remember or count, an arbitrarily large number of input symbols.

***Theorem* 2.6.6**: $L_{Subsequential} \subset L_{RatFcn}$.

Proof: Theorem 2.6.3 and Lemmas 2.6.4 and 2.6.5. ■

***Theorem* 2.6.7**: $L_{RatFcn} \subset L_{Rational}$.

Proof: Every rational function is also a rational relation; but all relations are not functions. ■

The next two theorems ensure that a subsequential transducer may be computed if one exists.

***Theorem* 2.6.5** (*Choffrut*): It is decidable if the behaviour of a rational transducer is a subsequential function [Ber79].

***Theorem* 2.6.6** (*Choffrut*): There is an algorithm for computing an equivalent subsequential transducer from a rational transducer whose behaviour is a subsequential function [Cho77].

Chapter 3

Input State Transition Function Optimization

The first major opportunity for optimization is presented by transducer input states. Input states correspond to non-final states of a deterministic automaton; equivalently, input states are traditional case statements provided by high level languages. Furthermore, the case selector is of type character, limiting the number of alternatives to roughly 2^8. The restricted nature of input state transition functions permits highly specialized optimization.

3.1. Notation

We will augment the definitions of Chapter 2 by making the following assumptions:

- the input alphabet Σ contains the special endmarker symbol $\dashv$
- the input alphabet Σ is totally ordered

With these assumptions, a subsequential transducer T is a 6-tuple:

$$T = <\Sigma, \Delta, Q, s, f, \delta>$$

where

Σ	is an ordered input alphabet $\{ a_1, a_2, \ldots, a_{\vert\Sigma\vert} = \dashv \}$
Δ	is an output alphabet
Q	is a finite set of states
$s \in Q$	is the start state

$f \in Q$ is the unique final state
δ is a finite transition function

and

$$\delta : Q \times (\Sigma \dot{\cup} \Delta) \rightarrow Q$$

The final state may have no outgoing transitions:

$$\delta(f,a) \text{ is undefined } \forall\ a \in \Sigma$$

The state set may be partitioned into input states and output states:

$$Q = Q_I \cup Q_O \cup \{f\}$$

All transitions from an input state must specify input symbols, and there must be only one transition from an output state, which specifies an output symbol:

$$\begin{aligned}
&\forall\ q \in Q_I : \delta(q,a) \text{ is defined} \Leftrightarrow \Pi_\Sigma(a) \text{ is defined}\\
&\forall\ q \in Q_O : \delta(q,a) \text{ is defined} \Leftrightarrow \Pi_\Delta(a) \text{ is defined}\\
&\forall\ q \in Q_O : \delta(q,a) \text{ defined and } \delta(q,b) \text{ defined} \Rightarrow a = b
\end{aligned}$$

It will be convenient to define the *completion*, T^c, of a subsequential transducer $T = <\Sigma,\Delta,Q,s,f,\delta>$ as follows:

$$T^c = <\Sigma,\Delta,Q^c,s,f,\delta^c>$$

where

$$Q^c = Q \cup \{\Omega\}$$

$$\delta^c(q,a) = \begin{cases} \delta(q,a) & \text{if } \delta(q,a) \in Q \\ \Omega & \text{otherwise} \end{cases}$$

Note that δ is only augmented for input states. We will say that T^c hangs when it enters the "dead state" Ω during its computation; that is, T^c cannot hang because the transition function is undefined. Without loss of generality, any subsequential transducer T is complete.

In order to discuss the transition function for a particular input state q, we define the function

$$\delta_q : \Sigma \rightarrow Q$$

which satisfies

$$\delta_q(a) = \delta(q,a) \quad \forall\ a \in \Sigma$$

The idea of the following *δ-diagram* is to emphasize the sequences of consecutive input symbols which share the same target state.

q_1	q_2	$\cdots$	q_n
$a_1a_2\cdots a_{l_1}$	$a_{l_1+1}a_{l_1+2}\cdots a_{l_1+l_2}$		$\cdots a_{\mid\Sigma\mid}$
l_1	l_2		l_n

It describes pictorially the function $\delta_q : \Sigma \rightarrow Q$ for some state q, which satisfies

$$\begin{array}{ll} \delta_q(a_i) = q_1 & i \in [\ 1,l_1\ ] \\ \delta_q(a_i) = q_2 & i \in [\ l_1{+}1,l_1{+}l_2\ ] \\ \vdots & \\ \delta_q(a_i) = q_n & i \in [\ \sum\limits_{i=1}^{n-1} l_i + 1,\mid\Sigma\mid] \end{array}$$

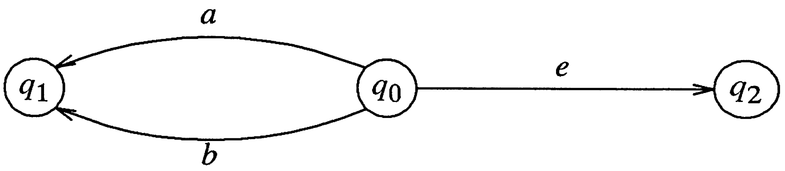
The $\{ q_i \}$ represent states in Q, and the $\{ l_i \}$ are positive integers. Figure 3.1 provides a concrete example. In order to describe δ_q over a restricted domain $D \subseteq \Sigma$, the notation $\delta_{q_{i,j}}$ will be used to mean

q_i	$\cdots$	q_j
$a_{i,1}a_{i,2}\cdots a_{i,l_i}$		$a_{j,1}a_{j,2}\cdots a_{j,l_j}$
l_i		l_j

And, the first and last symbols within a segment will be denoted

$$\begin{array}{l} \text{First}\ (\ i\ )\ =\ a_{i,1} \\ \text{Last}\ (\ i\ )\ =\ a_{i,l_i} \end{array}$$

let $\Sigma = \{ a,b,c,d,e \}$ and let part of the transducer T be



then, δ_{q_0} would be pictured thus:

q_1	Ω	q_2
ab	cd	e
2	2	1

Figure 3.1: Example of δ-diagram

Thus, the δ-diagram above covers the range of symbols

First(i), . . . ,Last(j) $= a_{i,1}, \ldots ,a_{j,l_j}$

3.2. Problem Definition

The fundamental local optimization problem is the generation of intermediate level code which implements the transition function δ_q for an input state q of a subsequential transducer T. Suppose that the current input symbol is a; then the generated code must find the state q_0 such that $\delta_q(a) = q_0$. Action appropriate when the machine hangs must be taken in the case that $\delta_q(a) = \Omega$. A deliberate distinction between intermediate level code and assembly level code has been made in order to yield a computationally practical problem. That is, it is not practical to attempt to compute an optimal assembly level code sequence for reasons which will be described in the section on optimality.

A common implementation of the case statement construct, such as the *switchon* command in BCPL, consists of two logical components [Sal81]. The first is the generation of a sequence of labelled *case actions* which correspond to the statements to be executed for the case labels, and which is preceded by an unconditional branch to *case selection*. The second, *case selection*, is a segment of code that determines which case label the selector matches and transfers control to the appropriate *case action*. This ordering is natural for one-pass compilers, such as for Pascal, and is the method used by the Unix C compiler.

Implementing an input state transition function may then be viewed as implementing a case statement where each *case action* is an unconditional branch to a state in the transducer. Therefore, *case actions* may be integrated with *case selection*, and thus *case selection* as defined above is an alternative problem statement.

3.3. Related Work

Recent compiler texts devote little attention to the topic of code generation for case statements [TS85, BBGC86, ASU86]. The most extensive treatment is presented in *Compilers: Principles, Techniques, and Tools* [ASU86], which suggests the use of one of the following techniques:

- generation of a static linear search
- construction of a (case label, case action address) table at compile time with a dynamic linear search
- construction of a hash table of case action addresses at compile time
- implementation of traditional jump tables

However, a method for combining techniques is not discussed, and the criteria for the choice of technique are vague. Efficient implementation of case statements has been addressed in a series of papers in *Software — Practice and Experience*.

The first in the series provides a comprehensive list of potentially useful techniques [Sal81], namely:

- jump tables
- static linear searches
- static tree searches (usually binary)
- tabular searches (static table with dynamic linear, binary, hash, etc. search)

Machine independent space and time complexity formulae are proposed for the first three of these, and formulae dependent on the Burroughs B6700/7700 architecture are given for all four, utilizing a special "linear search" machine instruction for the tabular technique. Plots of these functions lead to the exclusion of the linear and tabular search strategies in the Pascal compiler considered. In conclusion, either a jump table or a tree search is used to implement any particular case statement with the following simple selection criteria: use a jump table unless a tree search requires less space.

An important extension is to *combine* jump table and tree search techniques in implementing a single case statement [HM82]. Although not rigorously defined, their notion of a "cluster" is a sorted sequence of case labels $L_\alpha = l_1, l_2, \ldots, l_n = L_\beta$ such that

$$\frac{n}{L_\beta - L_\alpha + 1} \approx 1$$

Implementing a case statement involves constructing a binary tree search to distinguish amongst clusters (leaves), and generating jump tables for clusters. The algorithm for choosing when and how to split a range of case labels and introduce a new node in the comparison tree is heuristic, but the authors have found it effective in practice.

Bernstein expands on the idea of using more than one technique in implementing a single case statement, and further parameterizes the selection algorithm [Ber85]. The notion of clustering is used more precisely and is defined in terms of *case density* which, for a sorted sequence of case labels $L_\alpha = l_1, l_2, \ldots, l_n = L_\beta$, is

$$\text{casedensity}(l_1 \cdots l_n) = \frac{n}{L_\beta - L_\alpha + 1}$$

In addition to jump tables and binary tree searches, linear searches are also incorporated. The initial nodes of the binary search tree serve to distinguish amongst clusters whose case density meets a minimum parameter value (MinCaseDensity). Bernstein contends that finding the smallest set(s) of clusters which meet the requirement is NP-complete, and thus a heuristic is used to find the set of clusters. Two additional parameters (MinForJumpTable and MinForBinSearch) control the application of jump tables and binary searches respectively; each specifies the minimum case label range size for which the method is acceptable. Clusters which do not become jump tables will give rise to additional nodes in the binary search tree or will become a simple linear search segment. Finally, it is noted that case label probability data could be used in ordering the searches.

Aho and Ullman [AU77] describe a data structure for representing a lexical analyzer DFA which permits constant time implementation of the transition function for each state; its objective is to reduce the space required by a transition matrix, indexed by the current state and input symbol, while maintaining fast access. The traditional transition matrix method requires a single two-dimensional array access while the data structure method involves at least three one-dimensional array accesses plus one or two explicit additions. Thus, both methods implement the transition function in constant time. The space savings are realized by exploiting the similarities of different states, and the authors have found the space required by their representation "little more than the minimum possible".

The problem of input state transition function optimization has been addressed directly by Gonnet and Icaza in their work on compiling nondeterministic two tape automata into C language programs. Their work led to the *mscan* program. Their approach involves discovering special terminal cases and using dynamic programming to construct an optimal binary search tree whose leaves are instances of terminal cases. An instance of a terminal case is shown in Figure 3.2. The cost function used in the dynamic programming algorithm is a rough approximation and apart from several terminal cases (less than ten), only the binary tree search method is used. However, the application of dynamic programming makes it feasible to compute an optimal solution with an accurate cost function.

q_1	q_0	q_1
$a_i \cdots a_{i+A-1}$	a_{i+A}	$a_{i+A+1} \cdots a_{i+A+B}$
A	1	B

```
if( input_symbol = $a_{i+A}$ )
    goto $q_0$
else
    goto $q_1$
```

Figure 3.2: Sample terminal case used by dynamic programming algorithm

3.4. Incorporating Probability

The time cost functions used in the algorithms described above assume that all case label values are equally likely, and are intended to represent average time for case selection. If a discrete probability function P which reflects the relative frequencies of $\{ a_i \} = \Sigma$ in the input stream is available, then it may be used to more accurately compute expected execution cost of transducer input states. If English text is being processed, then the standard alphabet frequency table could be used; or, if the quantity of input is very large (e.g. a dictionary), P may be determined directly from the input. As a last resort, $P(a_i) = 1 \ / \ | \Sigma \ |$ will suffice.

This probability function P will be tailored to each input state q by defining $\Sigma_q \subseteq \Sigma$ and P_q as follows.

$$\Sigma_q = \{ a_i \mid \delta_q(a_i) \neq \Omega \}$$

$$\overline{\Sigma}_q = \Sigma - \Sigma_q$$

$$P_q(a_i) = \begin{cases} P(a_i) \, / \sum\limits_{a \in \Sigma_q} P(a) & \text{if } a_i \in \Sigma_q \\ 0 & \text{if } a_i \in \overline{\Sigma}_q \end{cases}$$

This implies a simple requirement of P, namely

$$\sum_{a \in \Sigma_q} P(a) > 0 \qquad \forall \; q \in Q$$

Intuitively, the idea of P_q is to permit the code which determines that $\delta_q(a) = \Omega$ to be arbitrarily expensive. In terms of an optimal binary tree, the nodes corresponding to the $\{ a_i \} = \overline{\Sigma}_q$ will be "pushed down" to the leaves, and will be equally as likely, allowing them to be placed anywhere at the bottom; see Figure 3.3.

3.5. Optimality

In a traditional optimizing compiler, optimization techniques are applied to an intermediate code in a separate phase; the effectiveness of the transformations performed are rarely measured, and the resulting code is not necessarily optimal. Our approach is to generate optimal intermediate level code (ILC) directly. This is accomplished by

- defining a simple but accurate model of computation M and a corresponding ILC,
- defining a machine parameterized execution cost (objective) function z, and
- computing an optimal ILC segment with respect to z under M using dynamic programming

let $\Sigma = \{ a, b, c, d, e \}$, and suppose that P is given by

	a	b	c	d	e
P	$\frac{1}{10}$	$\frac{99}{120}$	$\frac{1}{20}$	$\frac{99}{120}$	$\frac{1}{30}$

if δ_{q_0} for some input state q_0 is

Ω	q_1	Ω	q_2	Ω
a	b	c	d	e
1	1	1	1	1

then, $\Sigma_q = \{ b, d \}$, and P_q is given by

	a	b	c	d	e
P_q	0	$\frac{1}{2}$	0	$\frac{1}{2}$	0

Figure 3.3: Example of P_q

These points are discussed in more detail below.

It is not feasible to compute optimal machine level code segments due to the presence of span dependent machine instructions; in particular, the following requirements are contradictory.

1) In order to determine the shortest form of each span dependent instruction, the relative location and target of *every* span dependent instruction must be known.

2) In order to compute the relative location and target of the span dependent instructions for any particular input state q, the length of each ILC instruction (including span dependent instructions) must be known. This is a requirement of z.

3.6. Model of Computation

Because of the restricted kinds of operations needed to implement a transducer, an appropriate model of computation closely resembles most real architectures. It is simplified by restricting operand addressing modes to registers and literals, and one in particular for each instruction; some operands are implicit – such as the current input symbol, which is always in a specific register. Table 3.1 contains the instruction set of M, with symbolic names for time and space cost for instructions used in input state optimization.

The choice of instructions is intended to be one-to-one for comparison and branching; however, span dependence of branch instructions has been filtered out, so that the problem mentioned above may be avoided. The remaining instructions are "high-level" in the sense that a number of machine instructions will be needed to implement them. For instance, input and output buffering may be implemented in any way desired and hidden in the `get` and `put` instructions.

Code generators have been written for the VAX and IBM/370 architectures, and despite their substantial differences, good results were obtained with both. Implementation of a SUN code generator was planned but not done, since the SUN architecture is very similar to the VAX architecture. More detail may be found in Chapters 6 and 7, but the important implication is that the choice of ILC is a good one.

Instruction	Meaning	Time	Space
cmp <i-symbol>	compare input symbol with <i-symbol> and set condition code	CT	CS
ubr <label>	unconditional branch to <label>	UBT	UBS
bxx <label>	conditional branch to <label>; xx = lo,le,eq,ne,ge,hi	CB*	CBS
jth <base><size>	jump table header for a jump table for the range of input symbols <base> ... <base>+<size>-1	JTT	JTHS
jte <label>	jump table entry target for input symbol <base>+d where d is this jte's distance from the closest jth	-	JTES
get	obtain an input symbol	-	-
put <o-symbol>	place the output symbol <o-symbol> on the output	-	-
hng <state>	hung in state <state>	-	-
acp	transduction completed and successful	-	-

* Actually, there are two conditional branch time constants, namely:

CBTT = the time for a conditional branch *taken*, and

CBNT = the time for a conditional branch *not* taken.

Table 3.1: Machine instructions in model *M*

3.7. Execution Cost Function

Generating fast code on a real machine requires that the execution cost function z truly reflect the target machine. Exact timing information is usually unavailable from vendors, and benchmarking is a tricky, error prone process. Due to the unpredictable effects of cache memory, instruction prefetch, and memory faults, a good approximation is sought. Our approach to measuring the cost of a machine instruction is to sum the space occupied by the instruction (in bytes) and the number of memory accesses it performs. This is a common approach, and is valid since no "expensive" instructions are used – examples of costly instructions include integer division, floating point, and character string instructions. These instruction cost quantities correspond to the symbolic constants in Table 3.1, and are calculated for each target machine. For span dependent instructions, the most common format is used to calculate space and time constants.

Given an input state q, its transition function δ_q, its input alphabet Σ_q, and its probability function P_q, z is defined as a linear function of the *expected* execution cost and machine code size:

$$z = k_1 \sum_{a \in \Sigma_q} P_q(a)\, T(a) + k_2 \text{ Space}$$

where $T(a)$ denotes the time required if the input symbol is a, and Space is the number of bytes of object code generated; k_1 and k_2 are compilation parameters each with a default value of 1. This function allows large problems to be solved using solutions to smaller problems. Functions such as Time × Space and Time × Space^2 are examples of functions that cannot be optimized directly using dynamic programming; however, a linear function may be used iteratively to optimize these non-linear objective functions.

Through experimentation, large changes in k_1 and k_2 appear to have little effect on the ultimate solution found. Practically, the limits of 32-bit integer arithmetic could be reached easily in the non-linear case; using floating point arithmetic will solve this problem but was found to slow down the dynamic programming algorithm significantly.

3.8. Dynamic Programming

A problem is amenable to a dynamic programming solution providing that it satisfies the *Principle of Optimality*. Essentially, this permits the combination of optimum solutions of sub-problems in determining the optimal solutions of larger sub-problems leading finally to an optimal solution of the original problem. The problem we wish to solve is similar to finding an optimal binary search tree; Knuth shows how to solve this problem using dynamic programming [Knu71, Knu73].

Recall that $\delta_q : \Sigma \longrightarrow Q$ for an input state q may be written as a δ-diagram:

q_1	q_2	$\cdots$	q_n
$a_{1,1}a_{1,2}\cdots a_{1,l_1}$	$a_{2,1}a_{2,2}\cdots a_{2,l_2}$		$a_{n,1}a_{n,2}\cdots a_{n,l_n}$
l_1	l_2		l_n

Let each maximal contiguous sequence of input symbols $a_\alpha \cdots a_\beta$ satisfying $\delta_q(a_i) = \delta_q(a_j)$ for $\alpha \leq i < j \leq \beta$ be called a *segment* of the δ-diagram. (Segments are delimited by vertical lines.) The *problem size* of $\delta_q = \delta_{q_{1,n}}$ is the number of segments, n, in the δ-diagram; and in general, the problem size of $\delta_{q_{i,j}}$ is $j - i + 1$. For an input state q, there is no obvious relationship between $|\Sigma_q|$ and the problem size of δ_q. Consider the examples in

Table 3.2.

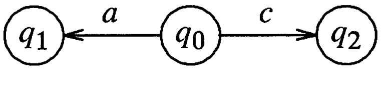
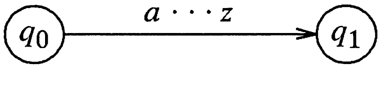
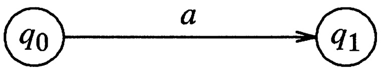
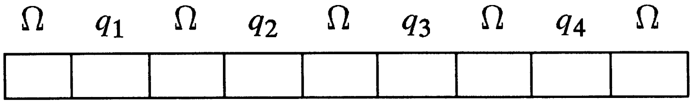
δ_q	$\lvert\Sigma_q\rvert$	problem size
	2	5
	26	3
	1	3

Table 3.2: Relationship of $\lvert\Sigma_q\rvert$ and problem size

A crude bracket for the problem size n for an input state q is the following:

$$1 \leq n \leq \min(\, 2\lvert\Sigma_q\rvert + 1, \lvert\Sigma\rvert)$$

In the best case, $\delta_q(a_i) = \delta_q(a_j)$ for $1 \leq i < j \leq \lvert\Sigma\rvert$ with a problem size of one. In the worst case, $\delta_q(a_i) \neq \delta_q(a_{i+1})$ for $1 \leq i < \lvert\Sigma\rvert$ with a problem size of $\lvert\Sigma\rvert$. When $\lvert\Sigma_q\rvert < \lvert\Sigma\rvert / 2$†, we can find a better upper bound. The idea is to count the maximum number of segments possible. Consider the following partial δ-diagram for $\lvert\Sigma_q\rvert = 4$:



There are at most $\lvert\Sigma_q\rvert - 1$ "separating" segments, plus one leading and one

† Assume $\lvert\Sigma\rvert$ is a power of 2.

trailing segment. Hence, the problem size is at most $2|\Sigma_q| + 1$.

The execution cost function z may now be redefined in terms suitable for dynamic programming. For $\delta_{q_{i,j}}$ we define:

$$\begin{aligned}
\sigma_{i,j} &= \{\,\text{First}(i), \ldots, \text{Last}(j)\,\} \\
w_{i,j} &= \sum_{a \in \sigma_{i,j}} P_q(a) \\
s_{i,j} &= \text{bytes of code to implement } \delta_{q_{i,j}} \\
\eta_{i,j} &= \text{time to execute code for } \delta_{q_{i,j}} \text{ if input is } a \in \sigma_{i,j}
\end{aligned}$$

Then, the cost of $\delta_{q_{i,j}}$ is

$$z_{i,j} = k_1 \sum_{a \in \sigma_{i,j}} P_q(a)\, \eta_{i,j}(a) + k_2\, s_{i,j}$$

and the cost of δ_q is $z_{1,n}$. The dynamic programming approach involves terminal and non-terminal cases. If a terminal case is detected, a predefined implementation is used to solve the problem; if no terminal pattern matches, the problem is divided into sub-problems with code to distinguish between them. In the strategy used here, non-terminal cases give rise to two sub-problems distinguished by a single input symbol comparison. (Actually, there may be a third trivial sub-problem of size one.) Now, a recurrence relation relating a problem P to its sub-problems L and R is needed.

Suppose that $P = z_{i,j}$ and one (binary) decomposition of P is $L = z_{i,k}$ and $R = z_{k+1,j}$. Further, suppose that

$$\begin{aligned}
\eta_L &= \text{time units to reach code for } L \\
\eta_R &= \text{time units to reach code for } R \\
s_P &= \text{bytes of code to distinguish } L \text{ and } R
\end{aligned}$$

To demonstrate that the Principle of Optimality holds, it is necessary to show that for any pair of solutions L_1^* and L_2^* to L, and any particular solution R^* to R

$$L_1^* \leq L_2^* \quad \Rightarrow \quad P_1^* \leq P_2^*$$

where P_1^* and P_2^* are the solutions to P corresponding to L_1^* and L_2^*. By definition,

$$z_{i,j} = k_1 \sum_{a \in \sigma_{i,j}} P_q(a)\,\eta_{i,j}(a) + k_2 s_{i,j}$$

We may express the cost of the solution to P as a function of k, the split point, as

$$z_{i,j}^k = k_1 \sum_{a \in \sigma_{i,k}} P_q(a)(\eta_{i,k}(a) + \eta_L)$$

$$+ k_1 \sum_{a \in \sigma_{k+1,j}} P_q(a)(\eta_{k+1,j}(a) + \eta_R)$$

$$+ k_2(s_{i,j} + s_P + s_{k+1,j})$$

Expanding, and rearranging the terms, the required recurrence is:

$$\boxed{z_{i,j}^k = z_{i,k} + z_{k+1,j} + k_1(w_{i,k}\,\eta_L + w_{k+1,j}\,\eta_R) + k_2 s_P}$$

Letting

$$Y = z_{k+1,j} + k_1(w_{i,k}\,\eta_L + w_{k+1,j}\,\eta_R) + k_2 s_P$$

we have

$$z_{i,j}^k = z_{i,j} + Y$$

Therefore,

$$P_1^* = L_1^* + Y$$

$$P_2^* = L_2^* + Y$$

And clearly, $L_1^* \leq L_2^* \Rightarrow P_1^* \leq P_2^*$.

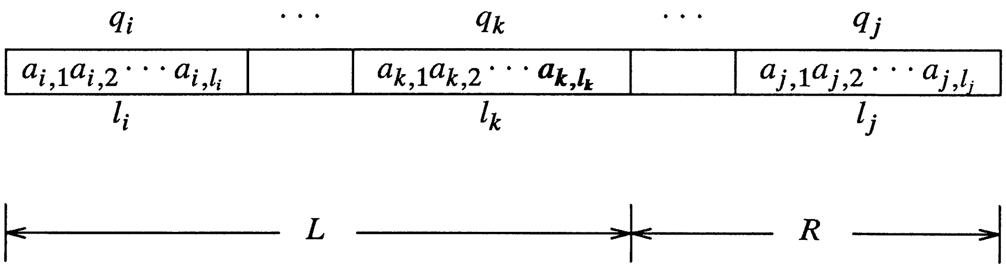
3.9. Non-Terminal Cases

Problem decomposition is straight forward: a single input symbol comparison is made to determine which of two, and sometimes three, execution paths to follow. These decompositions will be called binary and ternary *splits* respectively. One sub-problem of a ternary split will always have problem size one which corresponds to the case where the input symbol is equal to the symbol against which it is compared. This sub-problem is trivial to solve, and instances where it is applicable will be described below.

Dobosiewicz argues that optimal binary trees are not optimal under the comparison based model when search algorithms are implemented in a high level language [Dob86]. The reason is that the result of a relational operator is boolean while one of *three* relationships must be determined. If p and q are elements of a totally ordered set, such as the ASCII character set, then $p < q$, $p = q$, or $p > q$. Consequently *two* comparisons are needed at some nodes of a binary search tree. By generating assembly language programs, it is possible to distinguish all three cases with a single "key" comparison which sets the condition code. Each subsequent conditional branch tests the condition code, but this involves only a few bits of the program status word. In our implementation, a ternary split includes one more conditional branch than a binary split; moreover, the cost metric is accurate since comparisons and (conditional) branches are counted separately.

3.9.1. Binary Splits

We may decompose $P = \delta_{q_{i,j}}$ using a binary split by choosing k such that $i \le k < j$ yielding the left sub-problem $L = \delta_{q_{i,k}}$ and the right sub-problem $R = \delta_{q_{k+1,j}}$. This may be visualized as follows:


q_i
. . .
q_k
. . .
q_j
$a_{i,1}a_{i,2} \cdots a_{i,l_i}$
$a_{k,1}a_{k,2} \cdots a_{k,l_k}$
$a_{j,1}a_{j,2} \cdots a_{j,l_j}$
l_i
l_k
l_j
L
R

noting that the ellipses represent omitted segments which are not important to the split. Sub-problems L and R are always distinguished by comparing the input symbol with a_{k,l_k}.

Following the comparison of the input symbol with a_{k,l_k}, there are two possible orderings of ILC instructions:

```
     Method M1                    Method M2
----------------------      ----------------------
       cmp   a_{k,l_k}             cmp   a_{k,l_k}
       ble   lbl                   bhi   lbl
       <R>                         <L>
lbl:   <L>                  lbl:   <R>
```

In these ILC code fragments, `<L>` and `<R>` denote the ILC code for the two sub-problems. Recall that the cost function is

$$z_{i,j}^{k} = z_{i,k} + z_{k+1,j} + k_1(w_{i,k}\,\eta_L + w_{k+1,j}\,\eta_R) + k_2 s_P$$

Let us define $\alpha < \beta$ as follows

$$\begin{aligned} \alpha &= \mathrm{CT} + \mathrm{CBNT} \\ \beta &= \mathrm{CT} + \mathrm{CBTT} \end{aligned}$$

Then, the two methods may be compared by eliminating common terms:

Method	Cost Component
M1	$\beta w_{i,k} + \alpha w_{k+1,j}$
M2	$\alpha w_{i,k} + \beta w_{k+1,j}$

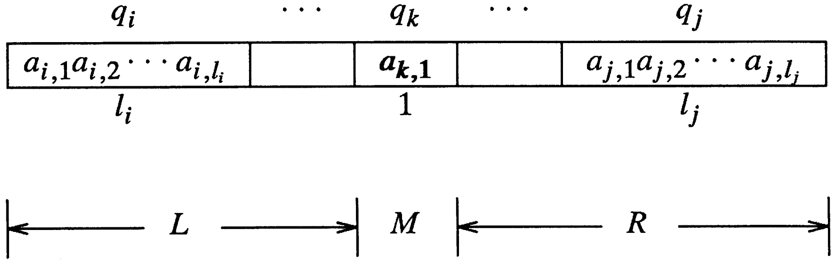
Since $\alpha < \beta$, the selection algorithm is

```
if( $w_{i,k} \leq w_{k+1,j}$ )
    use M1
else
    use M2
```

This means that a single comparison between $w_{i,k}$ and $w_{k+1,j}$ is sufficient to chose the best ILC instruction ordering for a binary split.

3.9.2. Ternary Splits

A ternary decomposition of $P = \delta_{q_{i,j}}$ is possible for k satisfying $i < k < j$ when the segment $\delta_{q_{k,k}}$ contains a single input symbol:



q_i
. . .
q_k
. . .
q_j
$a_{i,1}a_{i,2} \cdots a_{i,l_i}$
$a_{k,1}$
$a_{j,1}a_{j,2} \cdots a_{j,l_j}$
l_i
1
l_j
L
M
R

The solution to sub-problem M is a branch to state q_k, thus it is incorporated into the code which distinguishes the three sub-problems by comparing the input symbol with $a_{k,1}$.

Following the comparison of the input symbol with $a_{k,1}$, there are four possible orderings of instructions:

Method M1

```
          cmp    $a_{k,1}$
          blo    lbl
          beq    $q_k$
          <R>
lbl:      <L>
```

Method M2

```
          cmp    $a_{k,1}$
          bhi    lbl
          beq    $q_k$
          <L>
lbl:      <R>
```

Method M3

```
          cmp    $a_{k,1}$
          beq    $q_k$
          blo    lbl
          <R>
lbl:      <L>
```

Method M4

```
          cmp    $a_{k,1}$
          beq    $q_k$
          bhi    lbl
          <L>
lbl:      <R>
```

Note that there is no fragment <M> since it is precisely the instruction beq q_k. The cost function must be rewritten to take account of the sub-problem M:

$$z_{i,j}^{k} = z_{i,k} + z_{k+1,j} + k_1 (w_{i,k-1} \eta_L + w_{k,k} \eta_M + w_{k+1,j} \eta_R) + k_2 s_P$$

We define $\alpha < \beta < \gamma$ as follows

$$\begin{aligned} \alpha &= \text{CT} + \text{CBTT} \\ \beta &= \text{CT} + \text{CBNT} + \text{CBNT} \\ \gamma &= \text{CT} + \text{CBNT} + \text{CBTT} \end{aligned}$$

Eliminating the common terms we have:

Method	Cost Component
M1	$\alpha w_{i,k-1} + \gamma w_{k,k} + \beta w_{k+1,j}$
M2	$\beta w_{i,k-1} + \gamma w_{k,k} + \alpha w_{k+1,j}$
M3	$\gamma w_{i,k-1} + \alpha w_{k,k} + \beta w_{k+1,j}$
M4	$\beta w_{i,k-1} + \alpha w_{k,k} + \gamma w_{k+1,j}$

Unfortunately, selecting the best method is more complex than for binary splits. Analysis of all orderings of the three weights (e.g. $w_{i,k-1} \leq w_{k,k} \leq w_{k+1,j}$) leads to the following selection algorithm.

```
if( $w_{i,k-1} \leq w_{k+1,j}$ )
   if  ( $w_{k,k} \leq w_{i,k-1}$ ) use M2
   elif( $w_{k+1,j} \leq w_{k,k}$ ) use M3
   elif( $(\alpha - \beta)(w_{k,k} - w_{i,k-1}) < (\beta - \alpha)(w_{k+1,j} - w_{k,k})$ ) use M2
   else use M3
else
   if  ( $w_{k,k} \leq w_{k+1,j}$ ) use M1
   elif( $w_{i,k-1} \leq w_{k,k}$ ) use M4
   elif( $(\alpha - \beta)(w_{k,k} - w_{k+1,j}) < (\beta - \alpha)(w_{i,k-1} - w_{k,k})$ ) use M1
   else use M4
```

Experimentation suggests that evaluating the cost of ternary splits is an inefficient use of time since they are rarely chosen but require an expensive selection algorithm.

3.10. Terminal Cases

While we could use binary and ternary splits exclusively, there are certain cases where it is possible to generate a more efficient solution; these cases are called *terminal cases*. Jump tables are also terminal cases, and are considered for every problem of size greater than one. Since there is only one instruction sequence for implementing a jump table, there is no selection algorithm and cost evaluation is simple; however, all other terminal cases (except the most trivial one) involve a selection algorithm with correspondingly complicated cost evaluation.

Terminal cases identified by Gonnet and Icaza are illustrated below; each pattern δ-diagram is shown with its associated C code implementation. The current input symbol is denoted by θ. For reference, we shall assign names to these cases, beginning with the trivial case, T_1:

q_i
$a_{i,1}a_{i,2}\cdots a_{i,l_i}$
l_i

`goto` q_i

There are two cases with problem size three, namely T_{3a} and T_{3b}; T_{3a} requires that $|\sigma_{k,k}| = 1$ and that there are two distinct target states:

q_i	q_k	q_i
	$a_{k,1}$	
l_i	1	l_j

`if(` $\theta = a_{k,1}$ `) goto` q_k `else goto` q_i

T_{3b} is similar to T_{3a} except that $|\sigma_{k,k}| > 1$:

q_i	q_k	q_i
	$a_{k,1}a_{k,2}\cdots a_{k,l_k}$	
l_i	$l_k > 1$	l_j

`if(` $a_{k,1} \leq \theta$ && $\theta \leq a_{k,l_k}$ `) goto` q_k `else goto` q_i

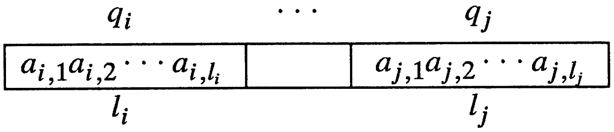
Finally, T_5 solves a problem of size five when $|\sigma_{i+1,i+1}| = |\sigma_{i+3,i+3}| = 1$ and there are only two distinct target states:

q_i	q_{i+1}	q_i	q_{i+1}	q_i
	$a_{i+1,1}$		$a_{i+3,1}$	
l_i	1	l_{i+2}	1	l_{i+4}

`if(` $a_{i+1,1} = \theta$ || $\theta = a_{i+3,l}$ `) goto` q_{i+1} `else goto` q_i

Note that for T_{3a}, T_{3b}, and T_5, exactly two target states are permitted, and thus each is implemented as an if-then-else statement. Similarly, each node in the binary search tree whose leaves are instances of these terminal cases is implemented as an if-then-else statement. Therefore, every δ_q is implemented as a sequence of nested if statements.

Often the fastest technique, jump tables are an important terminal case which may be applied to any problem of size greater than one. There is only one ILC instruction ordering so it is not necessary to compare various methods like those which arise with the non-terminal cases. For an arbitrary problem $\delta_{q_{i,j}}$


q_i
. . .
q_j
$a_{i,1}a_{i,2} \cdots a_{i,l_i}$
$a_{j,1}a_{j,2} \cdots a_{j,l_j}$
l_i
l_j

the ILC instruction sequence generated is

$$\begin{array}{lll} \texttt{jth} & a_{i,1}, |\sigma_{i,j}| & \\ \texttt{jte} & q_i & (l_i \text{ times}) \\ \vdots & & \\ \texttt{jte} & q_j & (l_j \text{ times}) \end{array}$$

with a cost of

$$z_{i,j} = k_1(w_{i,j}\,\mathrm{JTT}) + k_2(\mathrm{JTHS} + |\sigma_{i,j}|\,\mathrm{JTES})$$

A consequence of this formula is that the representation of δ_q becomes important from an efficiency point of view – for a problem of size n, $|\sigma_{i,j}|$ is computed for $1 \leq i < j \leq n$.

Pattern δ-diagrams for the remaining terminal cases we have implemented appear below. Since there are different ILC instruction orderings possible, implementations are not shown here. As before, the trivial case, $T_{trivial}$ is:

q_i
$a_{i,1}a_{i,2}\cdots a_{i,l_i}$
l_i

The first of two cases for problems of size three, T_{3-1-eq}, is equivalent to T_{3a}:

q_i	q_k	q_i
	$a_{k,1}$	
l_i	1	l_j

And the second, T_{3-1-ne}, is similar to T_{3-1-eq} except that there are three distinct target states:

q_i	q_k	q_j
	$a_{k,1}$	
l_i	1	l_j

The idea of T_5 may be generalized to problems of odd size n greater than three satisfying $|\sigma_{k,k}| = 1$ for k in $\{ i+1, i+3, \ldots, i+n-2 \}$:

q_i	q_{i+1}	q_i	q_{i+1}	$\cdots$	q_{i+1}	q_i
	$a_{i+1,1}$		$a_{i+3,1}$		$a_{i+n-2,1}$	
l_i	1	l_{i+2}	1		1	q_{i+n-1}

This set of cases will be called T_{linear} (due to the implementation strategy).

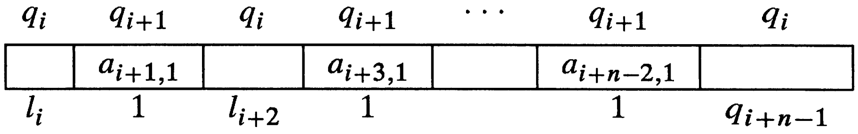
Which terminal cases are used and why? Factors to consider include the following:

- execution speed of code generated for terminal cases
- compilation cost incurred by terminal cases
- identification of terminal cases

Although a particular terminal case may deliver high execution speed, if it is expensive to compute, or it is selected infrequently, it is likely not worth including. One possible strategy for identifying terminal cases is to generate and test possible code sequences mechanically. Simplicity is also an important consideration.

The terminal cases T_{3-1-eq}, T_{3-1-ne}, and T_{linear} result in code that is superior to code resulting from "pure" binary splits. However, if subproblems of T_{linear} cases are optimally solved with jump tables, binary splits may be used to decompose the entire problem. Consequently, jump tables are tried for *all* problem sizes; binary splits are considered for T_{linear} cases, but not for T_{3-1-eq} and T_{3-1-ne} cases.

The number of possible ILC instruction orderings for a problem of size n which is an instance of T_{linear} is $((n - 1) / 2)$!



This may be seen by considering the form of any such order

```
cmp    $a_{k_1}$
beq    $q_{i+1}$
 ⋮
cmp    $a_{k_{n-1/2}}$
beq    $q_{i+1}$
ubr    $q_i$
```

where $a_{k_1} a_{k_2} \cdots a_{n-1/2}$ is some permutation of $a_{i+1,1}\, a_{i+3,1} \cdots a_{i+n-2,1}$. Consequently, only the two permutations $a_{i+1,1}\, a_{i+3,1} \cdots a_{i+n-2,1}$ and $a_{i+n-2,1}\, a_{i+n-4,1} \cdots a_{i+1,1}$ are considered. Evaluating the corresponding costs requires $O(n)$ multiplications and $O(n)$ additions. If we define $\alpha < \beta$ as

$$\begin{array}{lcl} \alpha & = & \mathrm{CT} + \mathrm{CBNT} \\ \beta & = & \mathrm{CT} + \mathrm{CBTT} \end{array}$$

then the cost of the first permutation is given by

$$z_{i,j} = k_1 \left(\sum_{k=1}^{\frac{n-1}{2}} ((k-1)\alpha+\beta) w_{i+2k-1,i+2k-1} \right.$$

$$\left. + \quad ((\frac{n-1}{2})\alpha+\mathrm{UBT}) \sum_{k=0}^{\frac{n-1}{2}} w_{i+2k,i+2k} \right)$$

$$+ k_2 \left((\frac{n-1}{2})(\mathrm{CS} + \mathrm{CBS}) + \mathrm{UBS} \right)$$

and the second is computed analogously.

3.11. Computational Complexity

Our discussion of dynamic programming as been informal since this is a straight forward extension to the well known dynamic programming solution of the optimal binary search tree problem [AHU74]. Brown presents an overview of computer science applications of dynamic programming with an intermediate level of formalism [Bro79]; and there are many formal treatments [CC81, DL77, Glu72]. Yao gives the *quadrangle inequality* condition for reducing the complexity of dynamic programming solutions [Yao80].

Significant effort was required in order to obtain acceptable performance from the dynamic programming algorithm. Its complexity is $O(n^3)$, but the leading constant may be reduced; the current version runs approximately 100 times faster than the initial implementation. Large performance improvements were realized through experimentation with the following issues.

- cost calculations
  - integer arithmetic is much faster than floating point arithmetic
  - integer arithmetic requires overflow checking
- matrix accesses
  - the two-dimensional dynamic programming matrix is implemented as a one-dimensional array with an efficient mapping (one shift and one add)
  - when $z_{i,j}$ is being computed, the address of the corresponding matrix element is computed and saved to speed up subsequent accesses
- representation of ILC labels
  - ILC labels are encoded as integers instead of strings yielding efficient comparison and copy operations
  - string operations are slow, and lead to storage management problems

Although the primary objective is to generate efficient code, the compilation process must be reasonably efficient to be useful in practice.

Chapter 4

Span Dependent Branch Optimization

Although there is still controversy about whether compilers should produce assembly or relocatable object code, there is general agreement that assembly time is a logical phase during which to perform span dependent branch optimization. Table 4.1 summarizes the capabilities of three assemblers.

Assembler	Type of Branch Optimization
Unix SUN	unknown degree of branch optimization
Unix VAX-11	partial optimization with branch chaining
IBM/370	no branch optimization

Table 4.1: Branch optimization provided by common assemblers

The SUN documentation does not state the extent of branch optimization performed by its assembler, but the VAX-11 and IBM/370 assemblers definitely do not perform comprehensive branch optimization. Furthermore, jump tables are not considered span dependent instructions, and thus they are not optimized in any of these assemblers. Consequently, span dependent branch optimization was implemented in INRC.

4.1. Notation

A span dependent instruction (*sdi*) is one whose machine instruction length is a function of the proximity of its operand.

> *Definition*: An instruction is said to be *span-dependent* if 1) the instruction exists in two forms of differing length, 2) the shorter form of such an instruction can be used at machine location m only if that instruction's operand has an address between $m + a$ and $m + b$ where a and b are fixed (and possibly negative) integer constants, 3) the longer form of such an instruction can always be used in place of a shorter form [Szy78].

Subsequently, the restriction to two instruction forms will be relaxed to permit sdis with a finite number of forms.

The distance between an instance of an sdi and its operand is the instruction *span*. Typically, there are two or three sdi *formats*, each of which accommodates a different range of spans. In some cases, a particular sdi format may be synthesized from primitive branch instructions if no single instruction exists. The *forward reach* of an sdi format is the maximum possible span in the forward direction and the *reverse reach* is the maximum possible span in the reverse direction. To simplify the description of the algorithms that follow, the reach of an sdi format will serve to denote its forward *and* reverse reaches; any comparison between the reach of an sdi format and the span of an sdi will take into account the forward and reverse reaches. Figures 4.1 and 4.2 illustrate sdi formats for the VAX and IBM/370 machines.

Short	Medium	Long
bgtr lbl	bleq skip brw lbl skip:	bleq skip jmp lbl skip:

Figure 4.1: VAX formats for the ILC instruction `bhi lbl`

Short	Long
BH LBL	BNH SKIP L Rn,=A(LBL) BR Rn SKIP DS 0H

Figure 4.2: IBM/370 formats for the ILC instruction `bhi lbl`

These terms are summarized by the following definitions:

sdi(t,f) = sdi instruction of type t and format f
reach(t,f) = reach of sdi(t,f)
growth(t,f) = size(sdi(t,successor(f))) − size(sdi(t,f))

A good example of the "asymmetry" of forward and reverse reaches is provided by the base and displacement branch instructions of the IBM/370: using one base register declared to contain the relocatable address b, a branch instruction operand must have a relocatable address l satisfying $b \leq l \leq b + 4095$. That is, the forward and reverse reaches depend on the location of the branch instruction relative to the location declared to be in the base register.

In order to avoid confusion in the discussion of jump tables, the term "jump table *element*" will be used in the same sense as the term "unconditional branch". That is, a jump table element specifies a target location for a particular value of the case selector; an ILC `jte` instruction is an instance of a jump table element. A jump table *entry* is a displacement, or virtual address value that is stored in a machine code implementation of jump table.

4.2. Assumptions

A fundamental assumption about sdi format reaches is that once an sdi grows to the long format, it can *always* reach its target; this assumption can be violated by a program whose size is nearly equal to the size of the virtual address space, but this is highly unlikely. An important assumption made in the section on jump tables is that at most two jump table element formats are used, and that the longer of the two is a long format (as opposed to short and medium, for instance).

4.3. Problem Definition

Once the ILC instruction sequence has been generated for a subsequential transducer, the span dependent branch optimization phase is performed; for an ILC sequence containing sdis 1 .. n ordered by increasing location, the problem is to find the shortest possible format for each one. Branch optimization may only be performed after all ILC has been generated for the reasons described in Section 3.6.

An alternative problem definition is appropriate if branch "chaining" is permitted. For instance, instead of using a synthesized long format, a branch to another branch sharing the same target could be used. The space savings are clear, but whether or not there are execution time benefits over synthesized branch formats is not. An algorithm for finding chains introduces the need for a data structure that contains for each label l a list of all branch instructions whose target is l [WJWHG75]. For large transducers (e.g. 10,000 states) this represents a tremendous space requirement; moreover, finding *minimal* branch chains appears to be difficult. Consequently, branch chaining was not implemented.

4.4. Basic Algorithms

Two algorithms which compute optimal solutions were considered as the basis for the INRC implementation. Szymanski gives an $O(n^2)$ algorithm which constructs and traverses a "branch dependency graph" (BDG) for span dependent branches limited to two formats [Szy78]. Fischer and Patterson introduce the monotonic priority set (MPS) data structure and apply it to the branch assembly problem for span dependent branches of finitely many formats [FP85]. Their algorithm has complexity $O(n \log n)$ but is significantly more complicated: a set of balanced binary trees must be constructed and maintained.

Initially, the BDG algorithm was selected for three reasons. First, it is simpler than the MPS algorithm: the BDG need not be constructed explicitly—the data structures used in this case are simple lists. Second, the BDG algorithm is more flexible than the MPS algorithm: if there is insufficient virtual memory to hold large arrays, sequential files may be used instead. Third, the BDG algorithm lends itself to extensions (such as permitting more formats) naturally. In practice, performance of the algorithm implemented in INRC is satisfactory, reinforcing the algorithm choice.

The directed branch dependency graph $G = (V,E)$ for a code segment containing sdis numbered 1 .. n is defined as follows:

$$V = \{ \ v_i \ | \ v_i \text{ denotes the } i^{th} \text{ sdi, } 1 \leq i \leq n \ \}$$
$$E = \{ \ (v_i,v_j) \ | \text{ the } j^{th} \text{ sdi lies between the } i^{th} \text{ sdi and its target } \}$$

Nodes are labeled i:*type*(i):*format*(i) where *type*(i) is the type of sdi i and *span*(i) is its current span. Short formats are assigned to all sdis when computing the initial node labels. The significance of an edge $v_i \longrightarrow v_j$ is that if *format*(j) grows from short to long then *span*(i) increases. Once G is constructed, it is processed using Algorithm 4.1 yielding $G' = (V',E')$.

```
while $\exists\ v_i$ such that $span(i) >$ reach $(type(i),$short$)$ do
    grow $\leftarrow$ growth$(type(i),$short$)$
    $\forall\ v_j$ such that $(v_j,v_i) \in E$ do
        $span(j) \leftarrow span(j) +$ grow
    od
    remove $v_i$ from $G$
od
```

Algorithm 4.1: Szymanski's BDG algorithm

Each sdi i whose corresponding node v_i *does* appear in V' is assigned its short format; each sdi j whose corresponding node v_j does *not* appear in V' is assigned its long format.

4.5. Algorithm Extensions

The BDG algorithm described above is designed to optimize sdis with two formats, namely short and long. In order to take advantage of VAX support for branch instructions, it is necessary to extend the branch optimization algorithm to incorporate an additional intermediate sdi format. Since jump tables are not directly supported in either the IBM/370 or Motorola/68020 instruction sets, they may be implemented using a table of displacements; these displacements may be one, two, or four bytes in size. For this reason, jump tables represent an excellent opportunity for sdi optimization. And because of the kind of *case* instructions supported, jump tables are also excellent candidates for sdi optimization on the VAX. Hence, inclusion of jump tables is another important extension.

4.5.1. Incorporating More sdi Formats

The branch dependency graph $G = (V,E)$ is defined as before, except that nodes are now labeled i:*type*(i):*format*(i):*span*(i). The difference in is that the format of each sdi is represented explicitly in its corresponding node rather than by that node's presence in or absence from the graph. Algorithm 4.2 shows how G is processed.

```
while $\exists\ v_i$ such that $span(i) >$ reach $(type(i), format(i))$ do
    grow $\leftarrow 0$
    repeat
        grow $\leftarrow$ grow + growth$(type(i), format(i))$
    until $span(i) \leq$ reach$(type(i), format(i))$
    $\forall\ v_j$ such that $(v_j, v_i) \in E$ do
        $span(j) \leftarrow span(j)$ + grow
    od
od
```

Algorithm 4.2: Multi-format BDG algorithm

To see that Algorithm 4.2 is correct, suppose that it does not produce the optimal solution. If it fails, then there exists and sdi I with an assigned format which is either too short or unnecessarily long. Suppose that I is too short to reach its target; this is a contradiction since the termination condition is that no sdi has insufficient reach. Now suppose that I has an unnecessarily long format. Initially, I has the shortest possible format, and since *format*(I) is only changed to the next larger format when *span*(I) is greater than reach(*type*(I),*format*(I)), the only way *format*(I) could be too large is if the subsequent growth of another sdi caused the span of I to shrink. Szymanski calls an sdi exhibiting the behaviour of I *pathological.* But this also leads to a contradiction since the target of all ILC sdis are labels, and hence the span of every sdi is monotonically increasing as sdis grow.

Suppose next that the maximum number of sdi formats is m, and consider the time complexity of Algorithm 4.2 As mentioned above, the BDG is not explicitly constructed; both finding a node requiring growth and finding all parents of a node which has grown are achieved by a brute force sequential search of a list of all sdis. Thus, each growth requires $O(n)$ time. Since each node may grow a maximum of $m-1$ times, the total complexity is $O((m-1)n^2)$ or $O(n^2)$.

4.5.2. Incorporating Jump Tables

In practice, jump tables will be selected frequently during input state transition function optimization because of their execution speed. A basic property of a jump table implementation is that the same number of bytes must be used to specify each jump table entry—these entries are displacements or virtual addresses. Thus, the entry size of a jump table determines the reach of *each* jump table element. The span of a jump table element e is the distance between the jump table containing it and e's target label. For some jump tables it is not possible to select an entry size which is optimal for all elements—either some entries will be too small or some will be unnecessarily large. Consequently, the problem is to select the entry size and ensure the reach of each element is sufficient.

The most obvious way to solve this problem is to select the minimal entry size such that the reach of each element is sufficient. However, suppose that the machine architecture provides special instructions for jump tables with a shorter entry size. How can the reach of individual jump table elements be increased? Without increasing the entry size, the reach of a `jte` instruction may be *extended* by adding an unconditional long branch whose target is the `jte` instruction's original target and which is the new target of the `jte`. The Unix C compiler uses this technique in case selection for *every* case action in a switch statement, regardless of whether or not it is necessary.

Consequently, our approach for the VAX is to utilize the special *case* instruction and extend the reach of only those jump table elements requiring it. Jump tables must be implemented without the aid of special instructions on the IBM/370 and thus entries may be one, two, or four byte displacements or four byte virtual addresses. A potential optimization strategy for this situation is to begin with two byte displacement entries and grow each entry to a four byte virtual address upon detection of an element with insufficient reach. This corresponds to selecting the minimal entry size such that no extensions are required.

The BDG algorithm may be used to optimize jump tables in addition to span dependent conditional and unconditional branches provided that the branch dependency graph G correctly reflects the dependencies introduced by jump table elements. Recall that $G = (V,E)$ was defined as follows:

$$\begin{array}{lll} V = \{ & v_i & \mid v_i \text{ denotes the } i^{th} \text{ sdi}, 1 \le i \le n \ \} \\ E = \{ & (v_i,v_j) & \mid \text{the } j^{th} \text{ sdi lies between the } i^{th} \text{ sdi and its target} \ \} \end{array}$$

Sdis, including `jte` instructions, are numbered sequentially from one by increasing location, thus the definition of V remains correct; however, the definition of E must be augmented:

$$\begin{array}{lll} E = \{ & (v_i,v_j) & \mid \text{the } j^{th} \text{ sdi lies between the } i^{th} \text{ sdi and its target} \ \} \cup \\ \quad\ \ \{ & (v_i,v_j) & \mid \text{the } i^{th} \text{ and } j^{th} \text{ sdis are elements of the same jump} \\ & & \ \ \text{table, and extension of sdi } j \text{ increases the span of} \\ & & \ \ \text{sdi } i \ \} \end{array}$$

Branch dependency between any pair of sdis—except two `jte` instructions from the same jump table—is defined easily because they have different locations. (Locations refer to the relocatable addresses initially assigned to sdi instructions.) Whether or not there is a dependency between two jump table elements from the same jump table depends on the directions of their respective targets and on how extensions for forward and reverse elements are implemented.

Suppose that all extensions are implemented by placing the unconditional branches directly preceding the jump table (leading method). Then there is an edge (v_p,v_q) in E due to the jump table element sdis p and q if and only if the target of sdi p precedes the jump table. Therefore, each reverse jump table element depends on every other element in the table. By symmetry, if all extensions are placed following the table (trailing method) then there is an edge (v_p,v_q) in E due to the jump table element sdis p and q if and only if the target of sdi p follows the jump table. A third possibility is to place reverse element extensions preceding the jump table, and forward element extensions following it (mixed method). Which method gives rise to the fewest number of edges? Assuming that jump table of size s† has the same number of forward and reverse elements on average, then the number of edges introduced by both the leading and trailing methods is

$$\frac{s}{2}\,(s-1) = \frac{s^2}{2} - \frac{s}{2}$$

The number of edges introduced by the mixed method is

† Assume s is even

$$2\left[\frac{s}{2}(\frac{s}{2}-1)\right]=\frac{s^2}{2}-s$$

It is clear that jump tables give rise to a large number of edges, and that the three extension methods differ very little.

To summarize, the BDG algorithm remains the same, but the branch dependency graph is redefined to include each jump table element as an sdi. The problem size, n, now includes regular branch sdis and all jump table element sdis; since the algorithm does not change, it is correct, and its complexity remains $O(n^2)$.

Chapter 5

Global Optimization

Traditional optimization techniques such as common subexpression elimination and dead code elimination are absent from INRC since INR, which produces the input for INRC, computes optimized automata. For instance, state minimization ensures that no dead code will be generated and that equivalent states will be combined. Many other optimizations are unnecessary because of the simplicity of the generated code: there are no arithmetic expressions, variables, or subroutines. The determinism of subsequential transduction eliminates the need for complex backtracking—a major implementation issue in non-deterministic transduction. The small set of operations *required* is described in Table 3.1 of Section 3.6. There are, however, important global optimization strategies included in INRC. The most complex global optimization, span dependent branch optimization, is discussed in Chapter 4; the others are described in the following sections.

5.1. Register Allocation

In addition to the usual register assignments (e.g. stack pointer, base register) which establish addressability or operands and support calling conventions, five global register are allocated (Table 5.1); remaining registers are only used for subroutine calls.

Name	Description
CurIB	Pointer to current position in input buffer
EndIB	Pointer to end of input buffer
CurOB	Pointer to current position in output buffer
EndOB	Pointer to end of output buffer
CurSym	Current input symbol
TmpJT	Work register for jump tables (IBM only)

Table 5.1: Global register allocation

The get and put ILC instructions affect these registers in the expected way: when a buffer is exhausted, an appropriate I/O routine is called and the buffer pointer pair is updated.

5.2. Code Factoring

A significant code size reduction was achieved with both the VAX and IBM implementations by factoring long machine code translations of the ILC instructions get, put, and hng. The machine instruction sequence common to all instances of an ILC instruction may be factored out, and invoked efficiently using only two machine instructions. This technique was included in the IBM implementation from the outset, but was added to the VAX implementation later.

Because of the high number of instructions required to accomplish a subroutine call on the IBM, it would be unreasonable to generate the calling sequence for each ILC instruction requiring a call. Therefore, the calling sequence is generated only once for each subroutine called; such a call site is activated by a BAL instruction, and returns control using a BR instruction. In this case, code factoring is a necessary optimization.

On the other hand, the VAX calling sequence is very short, and generating multiple in-line subroutine calls is reasonable. Code factoring was implemented since it yields a reduction in code size, and, although the user program execution time increases, the elapsed program execution time does not increase. The VAX instructions used to activate the factored code are the JSB and RSB instructions.

Chapter 6

VAX - Unix Implementation

6.1. System Interface

There are three distinct categories to consider: the interface between INR and INRC, the interface between the compiled transducer and the operating system, and the interface between INRC and its user. Implementation language choice has a great impact on the first two categories, but less on the third. The quality of the user interface is a function of the modularity of the code that INRC produces. The Unix operating system, our development environment, provides excellent support for the C programming language: it is efficiently compiled, provides access to system services, and uses simple subroutine linkage. INR is written in C.

The unit of communication between INR and INRC is an automaton. Thus INRC's knowledge of INR data structures may be limited to the automaton data type; sufficiently detailed knowledge is required for automaton validation and an important extension, action routines, described in Section 6.2. However, INR contains a sophisticated memory management module useful in implementing INRC. Therefore, simple communication and sharing of library utilities may be achieved by writing INRC in the same language as INR. For these reasons, and the level of support for C under Unix, INRC is also written in C.

In the simplest configuration, the transducer produced by INRC is invoked by a default driver program which performs input and output file handling, communicates with the operating system to perform block I/O, and terminates execution if the transduction fails (Figure 6.1).

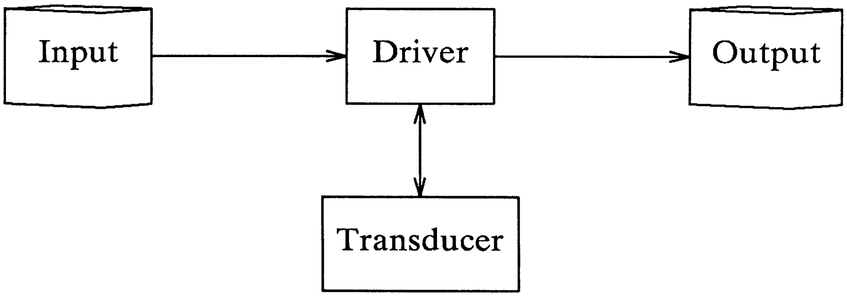


Figure 6.1: Simple run-time organization of a compiled transducer

In a more complex configuration, a user may write his own driver—which may be another application program—in order to control the input and output of the transduction. For example, the contents of memory may be transduced instead of the contents of a disk file. In any configuration, I/O routines are supplied by the transducer's *caller*. Therefore, it is only necessary for the transducer to communicate with its caller, and this implies that the transducer must follow the linkage conventions of its caller. For the reasons listed above, the language chosen for the VAX driver was C.

Providing a good user interface for INRC is simple because of the modularity of its compiled transducers. By default, a driver is linked with the transducer to produce an executable program which will transduce the standard input to the standard output. If, however, this default is unsatisfactory, the user may provide his own driver. This arrangement is simple and flexible.

6.2. Extensions

Since INRC is meant to be a practical tool, extensions were made to permit the invocation of *action routines* during a transduction. The first requirement of this extension is that the specification of action routines be made easily in the framework of INR; and the second is that the changes to INRC be integrated logically. The New Oxford English Dictionary project has used transduction with action routines to process the text of the Oxford Dictionary and build index data structures.

Action routines may be invoked at any point during a transduction, and are supplied with the number of symbols read and the number of symbols written so far. No other information may be passed to an action routine during transduction. For example, if input symbols already read are needed by an action routine, the driver program must maintain an input buffering scheme through which the desired input symbols may be extracted using the number of symbols read.

A notational convention is adopted by INRC which allows the specification of action routine calls within INR. Action routine calls are denoted by *action tokens* in the specification of a transducer; these are special output symbols that do not correspond to ordinary characters. During transduction, special output symbols will result in an action routine call rather than a write symbol operation.

Since the names of action routines are not included in the transducer definition, it is necessary to associate action tokens with action routines. This is accomplished as follows. A mapping between action tokens and small integers is established by INRC during compilation of the transducer; when transduction begins, an initialization routine is called to store this mapping. During transduction, a *master* action routine is invoked with an integer code corresponding to an action token, and, using the mapping stored, invokes the desired action routine.

A direct consequence of action routines is that there is increased communication between the transducer and its driver. Incorporating action routines in INRC was extremely helpful in refining the modularity of its compiled transducers. In the same way that a default driver is provided, default initialization and master action routines are provided in the user interface. Figure 6.2 illustrates the organization of a transducer with user supplied driver and action routines.

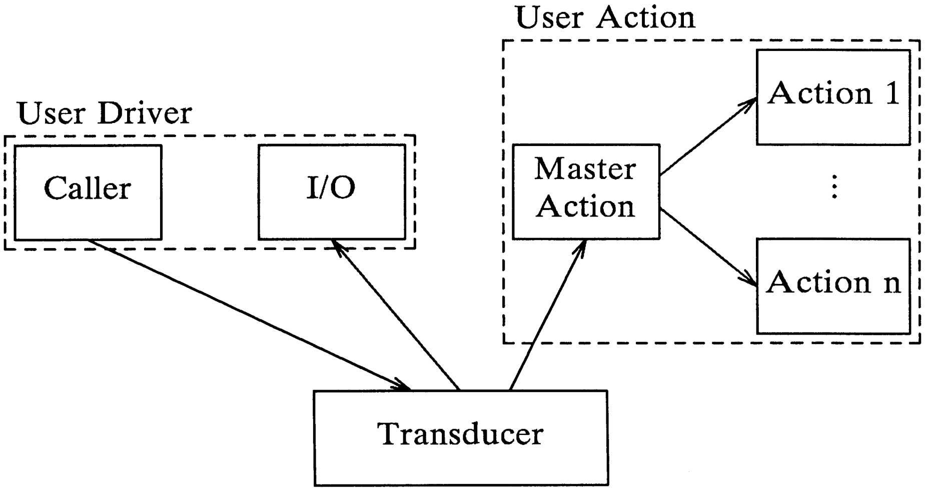


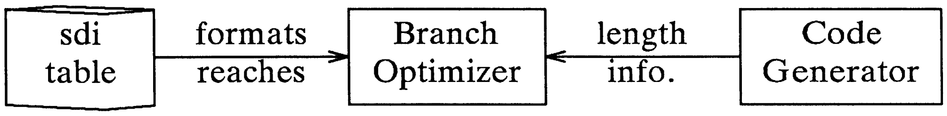
Figure 6.2: Sophisticated run-time organization of a compiled transducer

6.3. Code Generation

The VAX code generator produces assembly language output—this leads to simple code generation and rapid debugging. Generating relocatable object code is the alternative, but the compile time improvement gained does not justify the increased complexity. Code generation for the VAX was particularly easy for two reasons. First, operand addressability is unlimited; in particular, long branches may access the entire address space using program counter relative addressing. And second, C calling conventions use the VAX hardware stack: arbitrary calling sequences between the driver, the transducer, and system services are simple.

6.4. Branch Optimization

To a large extent, span dependent branch optimization may be abstracted from machine dependent details; it is, however, necessary for the branch optimization module to have knowledge of the available sdi format sizes and reaches, and the ability to compute the initial sdi spans. The number of formats and their reaches is localized to a table that is created for each machine, and computation of initial spans is accomplished through a code generator function which returns the size of the machine translation of each ILC instruction.



ILC instruction length information could also be represented in a table, but it is kept within the code generator so that ILC length information will be updated in conjunction with changes to ILC instruction translations.

Unconditional branch instructions are available in three single instruction formats; conditional branch instructions are also available in three formats, but two of these are synthesized.

Format	Unconditional Branch	Conditional Branch Reaches
short	$-2^7+2\ ..\ 2^7+1$	$-2^7+2\ ..\ 2^7+1$
medium	$-2^{15}+3\ ..\ 2^{15}+2$	$-2^{15}+5\ ..\ 2^{15}+4$
long	$-\infty\ ..\ \infty$	$-\infty\ ..\ \infty$

Table 6.1: VAX reaches relative to sdi location

The VAX provides direct support for jump tables whose elements have a reach similar to that of the medium branch format. Thus, by extending only elements requiring it, jump table elements are span dependent instructions with two formats, namely medium and long.

The BDG algorithm described in Chapter 4 is capable of handling sdis whose forward and reverse reaches are different, but it is simpler if they are the same. By calculating instruction format reaches relative to the "updated program counter" location, rather than the location of the sdi instruction, the forward and reverse reaches differ by only one byte. Moreover, using the same technique for jump table elements, the forward and reverse reaches of the medium and long formats are the same as for the corresponding branch formats. Therefore, it is possible to use the same three format reaches for every sdi instruction type. The one extra byte of reverse reach was sacrificed to obtain the symmetric forward and reverse reaches common to all sdis shown in Table 6.2.

Format	Sdi Reach
short	$-2^7 + 1 \,..\, 2^7 - 1$
medium	$-2^{15} + 1 \,..\, 2^{15} - 1$
long	$-\infty \,..\, \infty$

Table 6.2: VAX reaches relative to updated-PC location

6.5. Performance

Analysis of INRC may be carried out separately for compile time and execution time phases. Compile time considerations are computational effort and transducer size capacity; execution time considerations are speed and object code size. Good performance at compile time is important in practice but highly efficient executable code is of paramount importance.

Performance measurements for transducers compiled and executed on a VAX 8650 are presented. The first transducer realizes the Soundex code. Soundex encoding is often used in search applications involving surnames; for instance, airline reservation systems may use Soundex encoding to locate bookings for customers whose names were transcribed incorrectly. Johnson summarizes the literature related to the Soundex code, in the context of string similarity models, and proves that it is a subsequential function [Joh83]. His functional description is reproduced in Figure 6.3, and one corresponding INR formulation appears in Appendix A.

The second and third transducers were developed to process the entire tagged text of the Oxford English Dictionary (OED). They will be called the NoedTag and NoedAct transducers. Rick Kazman gives an explanation of OED tagging in *Structuring the Text of the Oxford English Dictionary through Finite State Transduction* [Kaz86]. The nature of the data and the New OED transduction is outlined below.

1. Remember the first character for later in, say, a variable X.
2. Map the letters according to the following scheme:
   b, f, p, v → 1
   c, g, j, k, q, s, x, z → 2
   d, t → 3
   l → 4
   m,n → 5
   r → 6
   a, e, h, i, o, u, w, y → 7
3. Wherever a sequence of like digits occur, delete all but the first of the sequence.
4. Drop the first digit of the number.
5. Remove all sevens from the number.
6. Add three zeros at the end of the number.
7. Take the first three digits of the number with the remembered first character appended in front.

Figure 6.3: Functional description of Soundex code

The text of the OED is received in electronic form, with structural elements tagged in SGML style [ISO85]. That is, each structure is delimited by a pair of *tags*:

<structure_tag> ... text of structure ... </structure_tag>

Closing tags are formed from opening tags by prepending a slash character; tags are distinguished syntactically by their enclosing angle brackets, which cannot appear elsewhere in the text. The data includes many instances of SGML style "tag attributes". For instance, the tags

<sen status=obs no=13> ... text of sense number 13 ... </sen>

delimit the text of sense number 13, which is obsolete. Since the input data was human generated, it contains a substantial number of tag nesting errors. The OED contains roughly 35 million tags, which account for approximately 45% of the input data.

The transduction process accomplishes several goals in parallel; the main ones are:

- *Tag Shortening*

  Tags are unnecessarily verbose, interfering with the readability of the text; furthermore, since tags account for such a high percentage of the data, they occupy a large amount of unnecessary space. Thus, tags are shortened; for example,

  <entry> becomes <E>, and <quot> becomes <Q>

- *Attribute Promotion*

  The use of tag attributes is unnecessary, since the structure they describe may be equivalently described by attribute-free tags alone. For instance,

  <sen status=obs no=13>
  becomes
  <sen><st>obs</st><#>13</#>

  This greatly simplifies on-line searching and automatic processing of the text.

- *Index Construction*

  As the transducer encounters opening and closing tags, an index structure is constructed. It consists of the following three components.

1. A "tag" file which contains pointers to the beginning and end of each structure in the text.

2. A "descriptive grammar" file which contains productions describing the nested structures of each tag pair.

3. A "rule" file which maps each pair of pointers is the tag file to the corresponding production in the descriptive grammar.

- *Error Processing*

  Tag nesting errors are corrected in the index, but not in the text. The transducer maintains a stack of the tags which have been opened but not yet closed. When a closing tag is encountered, the matching opening tag should be on the top of the stack. If the opening tag is not anywhere in the stack, the closing tag is discarded; if the opening tag is on the stack, but not on the top, each opening tag above it is popped, with a corresponding closing tag recorded in the index.

Tag shortening and attribute promotion represent the actual transduction, while index construction and error processing are accomplished through the use of action routines.

The New OED transducers are good examples since the machines are quite large, the action routine facility is thoroughly exercised, and the application is a real one. A substantial amount of time is spent building the indices, which occupy approximately half as much space as the dictionary itself. Both of the New OED transducers perform the same transformations on tags, but only NoedAct invokes action routines to build index data structures. The reason for this distinction is to enable measurement of "pure" transduction speed by comparison of INRC with other programs, and to provide true measurements of dictionary transduction time.

Unix provides three timers for measuring the execution speed of a program, namely:

real elapsed clock on the wall time
user time spent executing the user program
system time spent executing system services on behalf of the user program

When execution performance is being analyzed, these three quantities may be used to identify programs that are I/O or CPU bound; in addition, system loading will be reflected by the ratio of real to user times. For comparison purposes, timing measurements for an INR copy transducer, InrCopy (Figure 6.4),

```
SIGMA     =   '\x01\x02 ... \x7f' :alph;
InrCopy   =   SIGMA * :sseq;
```

Figure 6.4: InrCopy transducer

and two C copy programs, ChrCopy (Figure 6.5), and BlkCopy (Figure 6.6) are presented.

6.5.1. Compile Time

If all available compilers for a programming language offer poor performance, then chances are good that the language will not be used. Ada is a sophisticated high level language that is extremely difficult to compile—even implementations of subsets of Ada are slow. Consequently, a basic requirement of INRC is that it compile efficiently; furthermore, since complex subsequential transducers have many states and transitions, INRC must be able

```
#include <stdio.h>
int main( )
{
        int     c;
        while( ( c = getchar( ) ) != EOF )
                putchar( c );
        return( 0 );
}
```

Figure 6.5: ChrCopy C program

```
#include <stdio.h>
int main( )
{
    int     n;
    char    buffer[8192];

    while( ( n = read( 0, buffer, 8192 ) ) > 0 )
        write( 1, buffer, n );

    return( 0 );
}
```

Figure 6.6: BlkCopy C program

to compile very large machines. The tables below show compilation times, broken down into input state optimization and branch optimization phases, for the Soundex, New OED, and InrCopy transducers.

Input state optimization is performed for each state individually using the dynamic programming algorithm described in Chapter 3. This algorithm has complexity $O(n^3)$, where n is the problem size of the input state (Section 3.8), and since $n < |\Sigma|$, where Σ is the alphabet of input symbols (e.g. ASCII), the complexity of optimizing any input state is $O(|\Sigma|^3)$.

Soundex Transducer		Compile Time (sec)		
		real	user	system
States	75	–	–	–
Transitions	1,270	–	–	–
Input States	23	10.0	9.7	0.1
Span Dep. Instrs.	1,243	1.2	1.2	0.0
Total Time		12.0	11.5	0.1
Object Bytes	4,684	–	–	–

NoedTag Transducer		Compile Time (sec)		
		real	user	system
States	8,029	–	–	–
Transitions	11,464	–	–	–
Input States	245	58.0	51.1	1.0
Span Dep. Instrs.	6,821	260.9	247.3	2.2
Total Time		372.2	305.6	3.7
Object Bytes	146,647	–	–	–

NoedAct Transducer		Compile Time (sec)		
		real	user	system
States	8,222	–	–	–
Transitions	11,657	–	–	–
Input States	245	54.5	51.3	0.5
Span Dep. Instrs.	6,835	254.1	248.0	2.0
Total Time		317.5	306.7	3.1
Object Bytes	149,594	–	–	–

InrCopy Transducer		Compile Time (sec)		
		real	user	system
States	131	–	–	–
Transitions	257	–	–	–
Input States	1	6.8	6.5	0.1
Span Dep. Instrs.	257	0.4	0.3	0.0
Total Time		7.5	7.0	0.2
Object Bytes	2,529	–	–	–

Therefore, the complexity of the input state optimization phase is linear in the number of transducer input states. The dynamic programming matrix requires $O(|\Sigma|^2)$ space, which is acquired at the start and released at the end of this phase.

Branch optimization, however, is performed for all span dependent instructions (sdis) at once using the algorithm described in Chapter 4. This algorithm has complexity $O(n^2)$, where n is the number of sdis present in the intermediate level code (ILC). The space required for a transducer with $|Q|$ states and $|\delta|$ transitions is $O(|Q|+|\delta|)$; thus, space requirements may be a problem for large transducers. Although a significant constant factor slow down will result, the branch optimization algorithm may use files instead of memory without changing either time or space complexity.

6.5.2. Execution Time

In order to put the execution speeds measured for INRC compiled transducers into perspective, they are compared with execution speeds of the *lsim* and *gsm*1 programs. These results are considered in light of the nature of timing statistics provided by the operating system. And finally, the impact of action routines on the interpretation of timing results is considered.

Both *lsim* and *gsm*1 are load and simulate programs. *lsim* is capable of applying non-deterministic transductions and thus has elaborate mechanisms to permit back-tracking. *gsm*1 applies only subsequential transductions; it builds a hash table in order to achieve fast transition function lookup. Neither of these programs permit the incorporation of action routines. Timing results for the Soundex and New OED transducers appear in the tables below.

Soundex	Input	Output	Execution Time (sec)		
	bytes	bytes	real	user	system
lsim	720,896	491,520	200.1	185.5	1.3
*gsm*1	–	–	18.4	15.9	0.4
INRC	–	–	4.0	1.4	0.6

NoedTag	Input	Output	Execution Time (sec)		
	bytes	bytes	real	user	system
lsim	16,201,271	13,580,177	4,265.0	4,006.0	39.4
*gsm*1	–	–	444.7	396.2	9.8
INRC	–	–	57.4	42.2	4.7

NoedAct	Input	Output	Execution Time (sec)		
	bytes	bytes	real	user	system
INRC	16,201,271	13,580,177	174.6	131.5	20.6

Timing results for the InrCopy transducer and C copy programs appear below.

Copy	Bytes Copied	Execution Time (sec)		
Program		real	user	system
InrCopy	720896	3.5	1.6	0.3
ChrCopy	–	3.4	2.2	0.2
BlkCopy	–	2.8	0.0	0.2
InrCopy	16,201,271	47.6	36.9	4.0
ChrCopy	–	58.6	49.7	4.3
BlkCopy	–	30.4	0.0	3.0

Comparison of these timing results indicates that INRC transducers without action routines run as fast as the ChrCopy C program; the BlkCopy program beats the competition by a significant 50%. On the basis of *user* times, INRC achieves an order of magnitude improvement over *gsm*1, and two orders of magnitude improvement over *lsim*; On the basis of real times, INRC still achieves a five-fold improvement over *gsm*1. The BlkCopy measurements suggest that INRC transducers are *not* I/O bound, but experimentation with the VAX code generator suggests otherwise. Increasing the number of instructions executed by compiled transducers increased user but not real execution time.

Therefore, if pure transduction is performed on files, a five-fold elapsed time improvement is realized. However, if transduction of memory is performed, a greater improvement will be realized; if transduction of files involves CPU bound action routine processing, little or no increase in elapsed transduction time will occur.

Chapter 7

IBM/370 - CMS Implementation

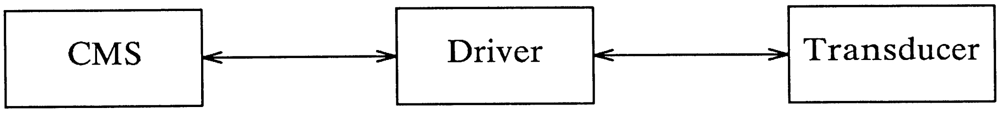
7.1. System Interface

The purpose of the IBM/370 implementation is to demonstrate that the ILC may be translated effectively on an architecture that is substantially different from the VAX architecture. Unlike the VAX, the 370 is not a stack machine. Moreover, it does not provide the rich set of addressing modes, such as autoincrement and program counter relative, that the VAX does. Consequently, the primary goal of the IBM implementation is to show that code generation can be done. The next most important goal is to generate transducers with the same degree of modularity as those generated for the VAX.

In any implementation of INRC, the important interface categories are: the interface between INR and INRC, the interface between the compiled transducer and the operating system, and the interface between INRC and its user. However, since CMS is not the development environment, there are constraints imposed by the initial (VAX) implementation and the existing implementation of INR on CMS. Since CMS provides a C compiler, CW†, INR was ported from Unix with minor system dependent changes. Similarly, INRC was ported from Unix with an IBM/370 code generator.

† Waterloo C compiler (version 1.31 IBM 370)

The goal of modularity requires that transducers be compiled as subroutines. This enables the caller to select the input and output of the transduction, and to handle errors.



Since the transducer communicates with its caller, the language in which the caller is written determines the linkage conventions that the transducer must follow. In order to provide direct access to system services, the transducer uses the traditional IBM/370 assembler conventions internally. Therefore, writing the caller in a language using these conventions would be a logical choice. Unfortunately, the CW compiler uses its own linkage mechanism. Writing the driver in assembler simplifies the transducer linkage but complicates I/O whereas writing it in C complicates the transducer linkage but leaves system dependent I/O to the compiler.

This dilemma was solved by using the Unix driver and writing macros to permit linkage between C and assembler routines. Specifically, special entry and exit macros follow the CW linkage conventions for the caller, but provide the standard S-LINKAGE environment within the assembler subroutine they enclose; a related call macro enables such an assembler subroutine to invoke another C routine. These macros work in conjunction by using the C stack for register save areas and parameter passing to and from C routines. This arrangement permits arbitrary calling sequences between C and assembler programs. Since these macros require complete knowledge of the CW calling conventions, they were difficult to write.

The user interface provided under CMS is the same as the one provided under Unix. By default, a transducer is compiled and linked with a default driver and default master action routine to produce an executable program which will transduce the standard input to the standard output. If, however, these defaults are unsatisfactory, the user may provide his own driver and action routines.

7.2. Addressability

Base and displacement addressing is the IBM/370's primary method of addressing relocatable operands. But because of its limitations, it is not possible to establish addressability for large program which are not divided into subroutines. Structured programming methodology demands the decomposition of large programs into modules; however, machine generated programs that are neither read nor modified may be unstructured for performance reasons. Since the programs generated by INRC are so simple, the use of subroutines does not even make sense; the only reason for artificially introducing them would be to alleviate base and displacement addressability problems.

Consider an IBM/370 machine language program of length L with byte locations numbered $0 \,..\, L$. Suppose that a single base register B_1 is declared to contain the location l_{B_1}; then the locations $l_{B_1} \,..\, l_{B_1} + 4095$ are addressable using B_1. If m base registers $B_1 \,..\, B_m$ are declared such that B_i will contain the location $l_{B_1} + 4096(i-1)$ then the locations $l_{B_1} \,..\, l_{B_1} + 4096m$ are addressable. since there are 16 general registers, and some must be reserved for linkage purposes, $m < 16$. Therefore, it is not possible to establish addressability for a program whose length L is greater than 2^{16} by using a set of base registers whose contents do not change during execution of the program.

In addition to base and displacement addressing, the IBM/370 provides A type (address) and V type (external symbol) constants which are resolved at load time. These constants permit the specification of arbitrary locations; however, they are also relocatable quantities. Branching to an arbitrarily distant location may be achieved using a load/branch register instruction pair (Figure 7.1).

```
         L          BASEREG,=A(TARGET)  ─┐
         BR         BASEREG              │  base and
         :                               │  displacement
         LTORG                           │
         =A(TARGET)                     ─┤
         :                               │  resolved
                                         │  by loader
TARGET   DS         0H                  ─┘
```

Figure 7.1: IBM/370 long branch format

Note that the load instruction still uses base and displacement addressing to retrieve the address constant from the literal pool. Thus, there are two branch formats, namely short and long; short formats are supported directly by base and displacement branch instructions, and long formats are synthesized as described above.

The solution to the addressing problem is to establish addressability separately for each transducer state. That is, a base register is declared and a literal pool is generated for each state so that

- one base register will easily address all of the code for a single state
- all literals referenced within a state will be assembled at the end of the corresponding code segment

By using the *same* base register in each state, efficient branches between states are possible. The crucial assembly instructions and assembler directives for establishing addressability for each state are shown in Figure 7.2.

```
        BALR    BASEREG,*     target of short branch
STATEi  DS      0H            target of long branch
        USING   *,BASEREG     declare base register
        <body of state i>
        LTORG                 generate literal pool
```

Figure 7.2: Addressability within a transducer state code segment

Figure 7.3 shows that both short and long branches require execution of only two instructions.

```
        BALR    BASEREG,*
STATEi  DS      0H
        USING   *,BASEREG
          ⋮
        B       STATEj-4             short branch to state j
          ⋮
        L       BASEREG,=A(STATEj)   long branch to state j
        BR      BASEREG
          ⋮
        LTORG
        BALR    BASEREG,*
STATEj  DS      0H
        USING   *,BASEREG
          ⋮
        L       BASEREG,=A(STATEi)   long branch to state i
        BR      BASEREG              (short branch impossible)
          ⋮
        LTORG
```

Figure 7.3: Branching between transducer states

7.3. Code Generation

Like the VAX code generator, the IBM/370 code generator produces assembly language output. The major difference, however, is its extensive use of macros: there is one macro for each ILC instruction. These ILC macros, in conjunction with macros that handle communication between C and assembler, represent a virtual machine. Thus, the complexity of code generation is shifted from the code generator to the set of macros supporting the virtual machine.

After successful code generation using macros, folding macro processing into the code generator was considered; it was rejected for the following reasons. The IBM/370 macro processor is very powerful, supporting flexible macros capable of producing the same instructions that the INRC code generator would produce. Furthermore, the INRC macros localize complexity in a format that is more easily understood than the source code of an equivalent code generator. The drawback of using macros is the additional assembly time they incur; however, assembly time represents a small fraction of the time spent in compiling a transducer. Finally, the macros may be used by driver and action routines written in assembler.

7.4. Extensions

The ability to invoke user subroutines during transduction has been incorporated in the CMS implementation in the same way as in the Unix implementation. Section 6.2 describes the *action routine* facility in detail.

7.5. Branch Optimization

Branch optimization is not performed in the CMS implementation. The reasons for this are two-fold: first, the objective of the CMS implementation is to demonstrate feasibility; and second, there are significant difficulties posed by the IBM/370 architecture.

The primary difficulty in implementing span dependent branch optimization for the IBM/370 is the peculiar nature of the "growth" from short to long formats. Specifically, a long branch requires that a nearby literal pool contain the target (virtual) address, and this address may or may not be in the appropriate literal pool. Thus, growth involves two components, namely growth at the branch site, and possible growth of a literal pool. Address entries in the literal pool must be fullword aligned; therefore, if an entry must be added, two extra pad bytes may be necessary. Although literal pools do not perform transfer of control, they are span dependent instructions (sdis) in the sense that their growth affects genuine sdis. This problem does not occur with the VAX architecture since PC-relative branch instructions are self-contained.

Another difficulty is that the forward and reverse reaches of a short format branch instruction depends on its distance from the current base register. Since there may be intervening sdis, the forward and reverse reaches of a short branch may change as other sdis grow. This behaviour is unlike PC-relative branches whose forward and reverse reaches are fixed and symmetrical.

7.6. Performance

Sample measurements of the Soundex transducer, and C copy programs described in Chapter 6 are presented in Table 7.1.

Soundex	Input bytes	Output bytes	Execution Time (elapsed sec)
*gsm*1	41884	28531	5.70
INRC	–	–	2.94
ChrCopy	41884	41884	4.09
BlkCopy	–	–	2.78

Table 7.1: CMS execution time results

Clearly, I/O is the crucial factor. Since the transducer driver is written in C, permitting simple input and output redirection, I/O time is much higher than it would be if it were written in assembler. A simple copy program coded in assembler yielded a 50% improvement over the BlkCopy C program.

Presently, CMS INRC transducers achieve a two-fold improvement over *gsm*1; however, optimization of I/O handling would definitely enhance this improvement.

Chapter 8

Conclusions

We have detailed optimization techniques used to compile subsequential transducers; two particularly important sources of optimization are:

- transducer input states—essentially "case statements"
- machine specific span dependent branch instructions

Transducers compiled by INRC achieve a five-fold improvement in execution time performance over the closest competitor (*gsm*1), and appear to be I/O bound. Moreover, the compilation speed of INRC makes it practical. The transducer input state optimization techniques described here are directly applicable to case statement compilation of high level languages.

There are many opportunities to increase the utility of subsequential transduction as a problem solving tool. These opportunities fall into one of two categories:

- improvements in the specification of subsequential transducers (INR)
- extensions to the pure model of subsequential transducers (INRC)

Work has already begun in both of these areas.

A lexical analyzer uses the rule that the token which matches the *longest* prefix of the remaining input is the desired one. This is a "disambiguating" rule which is natural and obvious, but which is difficult to express in INR. Johnson describes possible criteria for obtaining meaningful single-valued finite transductions from ambiguous (multi-valued) finite transductions [Joh87b], and has implemented corresponding functions in INR which compute subsequential transducers.

Action routines were added to INRC, enabling subroutine calls during transduction. At present, only the number of symbols read and written so far may be passed as parameters; the capability of passing parameters containing symbols from the input string would be very useful. Another extension is to permit transducers to be invoked and suspended in order to provide the type of master slave relationship between a parser and a lexical analyzer.

We have shown that subsequential transducers *can* be implemented efficiently. The New Oxford English Dictionary transducer, which performs tag transduction and invokes index construction routines, demonstrates the effectiveness of INRC: total processing time for 470 megabytes of text is less than five hours.

Appendix A

Transducer Grammars

A.1. Soundex Transducer

```
lc          = {a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z};
digit       = {0,1,2,3,4,5,6,7};

copy        = (0,0);       # copier from recognizer
double      = (0,0 0);     # copier which doubles each input symbol

lower       = { ({A,a},a), ({B,b},b), ({C,c},c), ({D,d},d), ({E,e},e),
                ({F,f},f), ({G,g},g), ({H,h},h), ({I,i},i), ({J,j},j),
                ({K,k},k), ({L,l},l), ({M,m},m), ({N,n},n), ({O,o},o),
                ({P,p},p), ({Q,q},q), ({R,r},r), ({S,s},s), ({T,t},t),
                ({U,u},u), ({V,v},v), ({W,w},w), ({X,x},x), ({Y,y},y),
                ({Z,z},z) };

mapcode     = { ({b,f,p,v},1), ({c,g,j,k,q,s,x,z},2), ({d,t},3), (l,4),
                ({m,n},5), (r,6), ({a,e,h,i,o,u,w,y},7) };

compress    = (1+ ,1) % (2+ ,2) % (3+ ,3) % (4+ ,4) % (5+ ,5) %
              (6+ ,6) % (7+ ,7);

dropfirst   = ((digit,^) (digit* $copy));

elimseven   = ({1,2,3,4,5,6} $copy) | (7,^);

append      = ((digit* $copy) ['000']);

truncate    = (((digit digit digit) $copy) (digit*,^));

Soundex     = lower* @
              ((lc $double) (lc* $copy)) @
              ((lc $copy  ) mapcode*   ) @
              ((lc $copy  ) compress   ) @
              ((lc $copy  ) dropfirst  ) @
              ((lc $copy  ) elimseven* ) @
              ((lc $copy  ) append     ) @
              ((lc $copy  ) truncate   );
```

References

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974

[AU77] Alfred V. Aho, and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass., 1986

[BBGC86] William A. Barrett, Rodney M. Bates, David A. Gustafson, and John D. Couch, *Compiler Construction*, second edition, Science Research Associates, Chicago, 1986

[Ber85] R. L. Bernstein, "Short Communication: Producing Good Code for the Case Statement", *Software — Practice and Experience*, **15**:1021-1024, 1985

[Ber79] Jean Berstel, *Transductions and Context-Free Languages*, B. G. Teubner, Stuttgart, Germany, 1979

[Bro79] K. Q. Brown, "Dynamic Programming in Computer Science", Department of Computer Science Report CMU-CS-79-106, Carnegie-Mellon University, 1979

[Cho77] Christian Choffrut, "Une caractérisation des fonctions séquentielles et des fonctions sous-séquentielles en tant que relations rationnelles", *Theoretical Computer Science*, **5**:325-338, 1977

[CC81] Leon Cooper, and Mary W. Cooper, *Introduction to Dynamic Programming*, Pergamon Press, 1981

[Dob86] W. Dobosiewicz, "Optimal Binary Search Trees", *International Journal of Computer Mathematics*, **19**:135-151, 1986

[DA77] Stuary Dreyfus, and Averill Law, *The Art and Theory of Dynamic Programming*, Academic Press, 1977

[FP85] M. J. Fischer, and M. S. Paterson, "Dynamic Monotone Priorities on Planar Sets", *26th Annual Symposium on Foundations of Computer Science*, pp 289-292, IEEE Computer Society, 1985

[Gin66] Seymour Ginsburg, *The Mathematical Theory of Context-Free Languages*, McGraw-Hill, New York, 1966

[Glu72] Brian Gluss, *An Elementary Introduction to Dynamic Programming: A State Equation Approach*, Allyn and Bacon, 1972

[HM82] J. L. Hennessy, and N. Mendelsohn, "Compilation of the Pascal Case Statement", *Software — Practice and Experience*, **12**:879-882, 1982

[ISO85] International Organization for Standardization DIS8879, "Information processing — text and office systems — Standard Generalized Markup Language (SGML)", 1985

[Joh83] J. Howard Johnson, "Formal Models for String Similarity", Ph.D. thesis, available as Research Report CS-83-32, University of Waterloo, 1983

[Joh87a] J. Howard Johnson, "INR: A Program for Computing Finite Automata", Unpublished report, University of Waterloo, 1987

[Joh87b] J. Howard Johnson, "Single-Valued Finite Transduction", *Proceedings of the 14th ICALP — Lecture Notes in Computer Science*, **267**:202-211, Springer-Verlag, 1987

[Joh75] Steven C. Johnson, "Yacc: Yet Another Compiler-Compiler", Technical Report CSTR 32, Murray Hill New Jersey, 1975

[Kaz86] Rick Kazman, "Structuring the Text of the Oxford English Dictionary through Finite State Transduction", M. Math thesis, available as Research Report CS-86-20, University of Waterloo, 1986

[KR78] Brian W. Kernighan, and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978

[Knu71] D. E. Knuth, "Optimum Binary Search Trees", *Acta Informatica*, **1**:14-25, 1971

[Knu73] D. E. Knuth, *The Art of Computer Programming Vol 3: Searching and Sorting*, Addison-Wesley, Reading, Mass., 1973

[Sal81] A. Sale, "The Implementation of Case Statements in Pascal", *Software — Practice and Experience*, **11**:929-942, 1981

[SS86] L. Sterling, and E. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, Mass., 1986

[Szy78] T. G. Szymanski, "Assembling Code for Machines with Span-Dependent Instructions", *Communications of the ACM*, **21**:300-308, 1978

[TS85] Jean-Paul Tremblay, and Paul G. Sorenson, *The Theory and Practice of Compiler Writing*, McGraw-Hill, New York, 1985

[WJWHG75] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*, American-Elsevier, New York, 1975

[Yao80] F. F. Yao, "Efficient Dynamic Programming Using Quadrangle Inequalities", *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, pp 429-435, Los Angeles, 1980

Index

Printing Requisition/Graphic Services

21116

1. Please complete unshaded areas on form as applicable.
2. Distribute copies as follows: White and Yellow to Graphic Services. Retain Pink Copies for your records.
3. On completion of order the Yellow copy will be returned with the printed material.
4. Please direct enquiries, quoting requisition number and account number, to extension 3451.

TITLE OR DESCRIPTION CS-88-20

DATE REQUISITIONED June 15/88

DATE REQUIRED ASAP

ACCOUNT NO. 1 2 6 6 1 5 4 4 1

REQUISITIONER-PRINT Sue DeAngelis.

PHONE 6462

SIGNING AUTHORITY

MAILING INFO - NAME SUE DEANGELIS DEPT. C.S. BLDG. & ROOM NO. DC 2314 [✓ □ DELIVER] □ PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES 112 NUMBER OF COPIES 50

TYPE OF PAPER STOCK
☑ BOND □ NCR ____ PT. ☑ COVER □ BRISTOL ☑ SUPPLIED □ ________

PAPER SIZE
☑ 8½ x 11 □ 8½ x 14 □ 11 x 17 □ ________

PAPER COLOUR
☑ WHITE □ ________

INK
□ BLACK □ ________

PRINTING
□ 1 SIDE ____ PGS. ☑ 2 SIDES ____ PGS.

NUMBERING
FROM TO

BINDING/FINISHING 3 down left side
☑ COLLATING ☑ STAPLING □ ____ HOLE PUNCHED □ PLASTIC RING

FOLDING/PADDING

CUTTING SIZE

Special Instructions

With front + backs enclosed

COPY CENTRE OPER. NO. BLDG. MACH. NO.

DESIGN & PASTE-UP OPER. NO. TIME LABOUR CODE

D01

D01

D01

TYPESETTING QUANTITY

PAP00000 T01

PAP00000 T01

PAP00000 T01

PROOF

PRF

PRF

PRF

NEGATIVES QUANTITY OPER. NO. TIME LABOUR CODE

FLM C01

FLM C01

FLM C01

FLM C01

FLM C01

PMT

PMT C01

PMT C01

PMT C01

PLATES

PLT P01

PLT P01

PLT P01

STOCK

001

001

001

001

BINDERY

RNG B01

RNG B01

RNG B01

MIS000000 B01

OUTSIDE SERVICES

$ COST

TAXES - PROVINCIAL □ FEDERAL □