Testing Completeness of Code Selector Specifications *

Helmut Emmelmann

GMD Research Center Karlsruhe
Vincenz-Prießnitz-Str. 1, 7500 Karlsruhe, Germany
email: emmel@karlsruhe.gmd.de

Abstract. The tree pattern matching approach for code selection has proven very successful. This paper presents an algorithm to test the completeness of a code selector specification. A specification is said to be complete, if it can produce code for every possible intermediate code tree produced by the front end.
To enable a code generator generator to test the completeness property current code selector descriptions have to be extended to describe the set of possible intermediate code trees. We use a tree grammar for this purpose.
The paper first presents the algorithm for testing completeness. Afterwards implementation techniques are discussed. Finally results of practical experiments using the algorithm are given.

1 Introduction

The tree pattern matching approach for code selection has proven very successful. It allows to specify code selectors using declarative specifications and to derive efficient code selectors automatically. The program which translates the specification into the final code selector program is called *code selector generator*. Sometimes it is integrated with generators for other parts of a code generator. However this paper deals only with the aspect of code selection.

The specifications are essentially rule based. The user only writes rules specifying how code can can be selected but not the exact way the rules have to be applied in order to get good code. This is determined automatically by the code generator generator system. The user annotates the rules with cost values and the system selects good code according to this cost measure.

The rule based approach has a lot of advantages which make it much superior to hand programming. However there is a small disadvantage: During hand programming one is usually forced to consider all possible constructs of the intermediate code as inputs for the code selector. When writing rules this is no longer the case. It is therefore much more likely to forget a rule for a certain construct of the intermediate code. This might show up as an error, only if the corresponding construct is used in a very special context.

This paper presents a method to detect statically if there are rules missing and if so to produce an appropriate error message. With such a test algorithm at hand

* This work was carried out within the Esprit Project #5399 (COMPARE)

the advantage of hand coding vanishes. One can be sure that the generated code selector can cope with every possible input.

More precisely a code selector description is said to be *complete*, if it specifies for every possible input at least one possibility to produce code for it. In other words, for every input there has to be a corresponding derivation using the rules of the code selector description. Current systems cannot check the completeness property statically. If there is no derivation the code selector blocks at code selection time when the corresponding input is supplied. So one has to rely on testing to determine whether the rule set specified is complete.

This paper presents a method to check the completeness statically in the code selector generator. To be able to check the completeness the generator needs to know about the set of possible inputs for the code selector, which is the intermediate code. Therefore we extend the code selector descriptions by another part that specifies the intermediate code, using the powerful mechanism of tree grammars.

Many code generator generator systems have the following nice property: If a derivation exists the generated code selector finds it. After a successful completeness test we are sure that for every possible input there is a corresponding derivation. From these two properties we can conclude that the generated code selectors can produce code for every possible input.

The rest of the paper is structured as follows: First we will give a short introduction to the tree pattern based specification technique for code selection and we define the completeness property. Then we present the basic completeness test algorithm. In the next section we consider it's efficient implementation. Finally we give some practical results using the algorithm.

2 General Remarks

2.1 Structure of Code Selector Descriptions

Input for our code selectors are *expression trees* made up by *operators of fixed arity*. Operators with arity zero are called *leaf operators*.

A code selector specification is made up by a set of *nonterminal* symbols N (disjoint from the set of operators) and a set of rewriting rules of the form $l \rightarrow r$ where $r \in N$ is a nonterminal and l is a tree pattern. A *tree pattern* is a tree that might contain nonterminals as leaves. In the extreme case l is a nonterminal, then the rule is called a *chain rule*. There is one special nonterminal Z named the *start symbol*. Rules are annotated with *cost values* and an *action part*.

Figure 1 contains an example. Ea (Effective Address), Areg (Address Register), Dreg (Data Register) and Z (start symbol) are the nonterminals of the description. The numbers given in the rules are the cost values. The action parts of the rules are given as english text or by stating the machine instruction which has to be emitted.

The task of code selection is performed by reducing the input tree to Z using the rules of the description. Usually there are several different derivations, each corresponds to correct code but the code quality varies. So one has to select a derivation of minimal cost. The cost of a derivation is the sum of the costs of all rules applied. After a minimal derivation has been determined the action parts of

(1)	c	→ Ea;	0	make immediate addressing mode
(2)	Dreg	→ Ea;	0	make data register direct addressing mode
(3)	cont (Areg)	→ Ea;	2	make address register indirect addr. mode
(4)	bb	→ Areg;	0	block base is a dedicated register
(5)	Ea	→ Dreg;	2	mov Ea,Dreg
(6)	Ea	→ Areg;	2	mov Ea,Areg
(7)	minus (Dreg, Ea)	→ Dreg;	2	sub Ea,Dreg
(8)	plus (Areg, Ea)	→ Areg;	2	adda Ea,Areg
(9)	plus (Dreg, Ea)	→ Dreg;	2	add Ea,Dreg
(10)	assign(Areg, Ea)	→ Z;	2	mov Ea,[Areg]
(11)	assign(Areg, Areg)	→ Z;	2	mov Areg,[Areg]

Fig. 1. Example Code Selector Description

the rules are executed as side effects of the rule applications and produce the desired output code.

For testing the completeness, the set of trees for that code can be produced is interesting. This is the set of trees that can be reduced to Z. Here we are not interested in the cost or the actions of a derivation. So when testing completeness we can ignore the cost values and action parts of the rules.

We call the set of rules without costs and actions together with the set of non-terminals and the start symbol Z a *reversed tree grammar* G. It is a *tree grammar*, in which the rules are written in the reverse direction (with swapped left and right hand sides). There are no big differences in theory, it is only a matter of writing the derivation relation in the other direction.

We write $\rightarrow$ to denote the derivation relation and $\Rightarrow$ for its transitive and reflexive closure. The set of trees that can be processed by the code selector, is the set of trees that can be reduced to $Z : L(G) = \{t : t \Rightarrow Z\}$. This is the language of the reversed tree grammar.

2.2 Definition of Completeness

As stated above a code selector specification is *complete*, if it specifies how to select code for every possible input tree. The set of all possible input trees is called I. So we have to test, if $I \subset L(G)$ holds.

Usually code selector specifications do not describe the set I. Therefore the code selector generator does not even know about I and cannot perform a completeness test. So it is necessary to extend the code selector descriptions by a part describing the set of possible input trees for the code selector.

There are several possible description mechanisms which could be used to describe I. For example we could use a simple type system or a type system with subsorting. We decided to use a tree grammar for that purpose, which is an even more general mechanism. So we extend the code selector descriptions by a tree grammar T specifying the set of possible input trees I. This means $L(T) = I$. We call T the *intermediate code description*. It is written using the same syntax as for the

code selector description: as reversed tree grammar. So the completeness test has to decide if $L(T) \subset L(G)$ holds.

(T1)	bb	→ A;	block base address
(T2)	c	→ D;	a constant value
(T3)	plus (A, D)	→ A;	address add
(T4)	plus (D, D)	→ D;	integer add
(T5)	minus(D, D)	→ D;	integer subtract
(T6)	cont(A)	→ D;	content resulting in address
(T7)	cont(A)	→ A;	content resulting in data
(T8)	assign(A, D)	→ T;	assign a data value
(T9)	assign(A, A)	→ T;	assign an address

Fig. 2. Description of the Intermediate Code

Figure 2 gives an example. The nonterminal D denotes a data value, A denotes an address value and T is the start symbol. T describes how the intermediate code looks like. For example it defines that there are no absolute addresses. An address is either bb the block base register, the content operation that is used to load a pointer, or the sum of an address and a data value. So T poses a proper restriction on the possible input trees.

The code selector specification of Fig. 1 is complete with respect to the intermediate code description of Fig. 2. However it is not complete if arbitrary input trees are allowed. For example the tree $assign(minus(c, bb), minus(c, bb))$ cannot be translated. However, the tree cannot occur as input to the code selector, because it conflicts with the intermediate code description. In fact, taking the negative value of the block base does not lead to anything reasonable.

On the other hand the code selector description can generate code for more trees than can actually occur in the intermediate code. For example for $assign(c, c)$ code can be produced, although in the intermediate code no absolute addresses are allowed. So in fact the machine description describes too much. One could change the description, to accept only legal intermediate code trees, however this is quite unnatural. It would be necessary to remove the possibility to move a value from a data register into an address register.

To illustrate the algorithm we will use a different example. Suppose we leave out rule number 6 of the example code selector description. Then the code selector cannot load the content of a memory cell into an address register. Calling the completeness test algorithm would tell us that the description is not complete and produce a counterexample: $assign(cont(bb), bb)$.

2.3 Related Work

A lot of work has been done on code selection using tree pattern matching for example see [FeSe92, HaCh86, HoDo82]. The problem of completeness however is handled only in very few papers.

In [GiSchm88, Gie90] a completeness test working on the generated code selection tables is described. The main part of the paper uses an order sorted signature Q to specify the intermediate code. However Giegerich also briefly gives the necessary extensions when using a tree grammar as intermediate code description. Our paper handles these algorithms in detail by using the theoretical term of a product automaton. A more refined version of the algorithm is given and the implementation is described. An important point is that our algorithm finds out counterexamples in case the description is not complete. These counterexamples are very useful to find errors in the specification.

3 Algorithm

In Sect. 2.2 it has been shown that the completeness test basically is the problem of testing the subset property $L(T) \subset L(G)$ of two regular tree languages. The theory of tree automata tells us that this test is decidable. In the next section we will explain, how the solution looks like from the automata theoretic point of view. Although this forms the basis of our algorithm, we have tried not to use too much theory afterwards.

3.1 Automata Theoretic Solution

Languages generated by regular tree grammars are called regular tree languages. They have properties quite similar to those of regular (string) languages. Especially it is possible to test the subset relation in the following way: First we construct non-deterministic automata accepting $L(G)$ and $L(T)$ respectively. Then a deterministic automaton accepting $L(G)$ can be produced by using the subset construction. It can easily be modified to accept the complement $\overline{L(G)}$, that is the set of all trees that are not in $L(G)$. Now we build the product automaton of the deterministic automaton accepting $L(G)$ and the non-deterministic automaton accepting $L(T)$. The product automaton accepts the intersection of the languages accepted by the two factors. So it accepts $L(T) \cap \overline{L(G)}$ which is $L(T) \setminus L(G)$. Finally we can check if $L(T) \setminus L(G)$ is empty, by looking at the product automaton. If so, $L(G) \subset L(T)$ holds, otherwise the description was incomplete and all trees in $L(T) \setminus L(G)$ are counterexamples.

3.2 Normalizing Rules

Before we describe the main algorithm we have to transform the code selector description and the intermediate code description into what we call *non-deterministic tree automata*. This transformation is quite simple, in fact only big patterns are split and simulated by several rules with simpler patterns.

Consider a rule $l \rightarrow r$ where l is a tree pattern. Suppose l has the form $\theta(l_1 \ldots l_\alpha)$ where α is the arity of the operator θ and $l_1 \ldots l_\alpha$ are the children. Suppose one of them for instance l_i is not a nonterminal but a more complex tree. Then we can replace the rule $l \rightarrow r$ by the following two rules:

$$\theta(l_1 \ldots l_{i-1} n l_{i+1} \ldots l_\alpha) \rightarrow r$$

$$l_i \rightarrow n$$

n is supposed to be a new nonterminal. This transformation step does not change the accepted language. It is applied repeatedly as long as a rule exists it can be applied on. Finally we end up with a rule system, in which each rule has the special form $\theta(l_1 \ldots l_n) \rightarrow r$ and all l_i are nonterminals. We call such a system a *non-deterministic tree automaton.* It still accepts the same tree language as the original tree grammar. The "nonterminals" of a tree automaton are usually called *states.*

Our two example descriptions of Fig. 1 and 2 already have the desired form, i.e. they are non-deterministic tree automata. To illustrate the transformation look at the following example: The rule *plus(Dreg, cont(Areg))* $\rightarrow$ *Dreg* can be split into the two rules *cont(Areg)* $\rightarrow$ *n* and *plus(Dreg,n)* $\rightarrow$ *Dreg. n* is a new nonterminal not used before.

3.3 The Subset and Product Automaton Construction

Input for the algorithm are the two non-deterministic tree automata G (the code selector description) and T (the intermediate code description). The set of states of G is named $\mathcal{S}_G$ and the set of states of T is named $\mathcal{S}_T$. The start symbols of both automata are named $\mathcal{Z}_G$ and $\mathcal{Z}_T$ respectively.

Our examples of Fig. 1 and Fig. 2 will be used to illustrate the algorithm. Here $\mathcal{S}_G = \{\text{Ea, Dreg, Areg, Z}\}$, $\mathcal{S}_T = \{\text{A,D,T}\}$, $\mathcal{Z}_G = \text{Z}$, and $\mathcal{Z}_T = \text{T}$ hold.

We assume that both tree automata T and G do not contain unreachable or not finalizable states. A state is unreachable, if no input tree can be reduced into this state. A state is not finalizable if there is no way for any tree t containing the state to be reduced to the final state. If T and G contain such states they can be simply deleted together with all rules referencing them. This does not change the accepted languages.

The algorithm works in two steps. First a set of states called U is calculated (Sect. 3.4). Afterwards this set is checked for a certain property (Sect. 3.5). A state is a tuple $[P;q]$ where $P \subset \mathcal{S}_G$ is *a set of* states of G and $q \in \mathcal{S}_T$ is *one* state of T.

U is a set of such states with the following property: A state $[P;q]$ is in U iff a tree t exists with

- t can be reduced into q using T
- t can be reduced into every state $s \in P$ using G and it cannot be reduced into any other state $s \in \mathcal{S}_G$.

The set of all trees t which fulfill this property is named the language $L([P;q])$ of the state $[P;q]$. So a state $[P;q]$ is in U iff $L([P;q])$ is not empty.

When we look at a fixed tree t we can determine the set P of all states of G which can be derived from t: $P = \{s \in \mathcal{S}_G | t \overset{G}{\Rightarrow} s\}$. We also define the set $Q = \{s \in \mathcal{S}_T | t \overset{T}{\Rightarrow} s\}$. Then we know that U contains all the states of the form $[P;q]$ where q is an element of Q.

The set U is in fact the state set of the product automaton mentioned in Sect. 3.1. The first component P of a state $[P;q]$ is a state of a deterministic automaton accepting $\overline{L(G)}$. This deterministic automaton is constructed by subset construction from the non-deterministic automaton G. The second component q is simply a state

of a possibly non-deterministic automaton accepting $L(T)$. The transitions of the product automaton are not recorded by the algorithm, because they are not required to decide completeness. It is sufficient to construct the set of reachable states of the product automaton.

3.4 Calculating U

For the calculation of U another definition is more suitable: U is the smallest set of states which fulfills the following property:

> Suppose there are an operator θ with arity α and states $[P_1, q_1], \ldots, [P_\alpha, q_\alpha] \in U$. Let
>
> $$P = \{s | \exists s_1 \in P_1 \ldots s_\alpha \in P_\alpha : \theta(s_1 \ldots s_\alpha) \overset{G}{\Rightarrow} s\}$$
>
> $$Q = \{s | \theta(q_1 \ldots q_\alpha) \overset{T}{\Rightarrow} s\}$$
>
> Then all the states of the form $[P, q]$ where q is an element of Q have to be elements of U.

This definition immediately leads to an algorithm. Initially we start with an empty set U. Then we look for an operator θ and states $u_1 \ldots u_\alpha \in U$ which do not fulfill the above property. In a first step this will be an operator of arity zero $\alpha = 0$ because at the beginning U is empty. Then we calculate all the states which have to be in U according to the definition above. If they are not already present we add them to U. This process is continued until U does not change any more. As the set of all possible states is finite the algorithm will stop.

In an actual implementation one usually handles the 0-ary operators separately, because the states they produce are independent of any other states in U. So one can first add all the states produced by unary operators to U and afterwards forget about these operators.

Please note the use of $\Rightarrow$ rather than $\rightarrow$ in the definitions of the sets P and Q. It is indeed possible that several rules have to be applied, first a rule of the form $\theta(s_1 \ldots s_\alpha) \rightarrow s_0$ and then several chain rules to derive $s_0 \Rightarrow s$. On the other hand it is not necessary to check for the applicability of chain rules in the operands $s_1 \ldots s_\alpha$ or $q_1 \ldots q_\alpha$ because when building the sets P_i and U the chain rules have already been considered.

3.5 Detecting Completeness by Examination of U

Using U it is easy to decide the completeness:

If there is a state $[P; \mathcal{Z}_T]$ in U with $\mathcal{Z}_G \notin P$ then the code selector specification is not complete. According to Sect. 3.3 the language of $[P; \mathcal{Z}_T]$ is not empty because $[P; \mathcal{Z}_T] \in U$. Let t be a tree in $L([P; \mathcal{Z}_T])$. The tree t can be reduced to $\mathcal{Z}_T$ in T. But it cannot be reduced to $\mathcal{Z}_G$ in G, because P is the set of all states it can be reduced to. So t is in $L(T)$ but not in $L(G)$ and hence $L(T) \not\subset L(G)$.

On the other hand, if no such state exists, the code selector specification is complete. Suppose we take an arbitrary tree t out of $L(T)$. Then because of t there are some states of the form $[P; \mathcal{Z}_T]$ contained in U. As for all such states $\mathcal{Z}_G \in P$ holds it follows $t \overset{G}{\Rightarrow} \mathcal{Z}_T$ and $t \in L(G)$.

3.6 Calculating Counterexamples

If the completeness test fails the user needs a meaningful error message. Therefore we compute a counterexample. This is a tree which is accepted by the intermediate code description T but not by the code selector description G. Suppose the completeness test failed. Then there is a state $c = [P'; \mathcal{Z}_T] \in U$ with $\mathcal{Z}_G \notin P'$. According to the definition of U in Sect. 3.3 a tree t exists in the language of c. In Sect. 3.5 it has been shown that t is a counterexample.

To calculate a counterexample we give a recursive algorithm which calculates for a given state $[P; \mathcal{Z}_T] \in U$ one example tree $t \in L([P; \mathcal{Z}_T])$. During the calculation of U we record for each state one possibility how the state was built. Therefore all states are numbered. For each state we store the operator θ and the numbers of the states $[P_1, q_1], \ldots, [P_\alpha, q_\alpha] \in U$ it has been produced from.

Then the construction of an element t of $L([P; q])$ for a state $[P, q]$ works as follows: If $[P, q]$ has been produced by a leaf operator θ then $\theta \in L([P; q])$ is the desired tree. If θ has some children, then we know the numbers of the states $[P_1, q_1], \ldots, [P_\alpha, q_\alpha] \in U$ from which $[P, q]$ can be build. For each of these states we construct a tree $t_i \in L([P_i, q_i])$ recursively. Then $t = \theta(t_1, \ldots, t_\alpha)$ is the desired tree $t \in L([P; q])$. The algorithm cannot run into an infinite loop, because the state numbers become smaller in every step, since each state is always produced from states with smaller numbers.

Care has to be taken to get small counterexamples. If one somewhere builds bigger counterexamples than necessary these can multiply and finally result in very big trees. In fact counterexamples can grow exponentially, however, this seldom happens. To get smaller counterexamples the algorithm building up U should use the states introduced by smaller trees t first to combine new states.

3.7 Example

Number	State $[P, q]$	Stored Information	Element of $L([P, q])$
1	[{Areg};A]	bb	bb
2	[{Ea, Dreg, Areg};D]	c	c
3	[{Ea, Dreg, Areg};A]	cont(1)	cont(bb)
4	[{Z};T]	assign(2,1)	assign(bb,c)

Fig. 3. U of example description

For our example description the set U is given in Fig. 3. The table contains the state number, the state $[P, q]$ itself, and the information how the state was created, and an element of $L([P, q])$. The first 3 columns need to be stored in the implementation, while the last column can be calculated from the information of the first 3 columns as described in Sect. 3.6. Obviously U fulfills the criterion of Sect. 3.5 and therefore the description is complete.

If we leave out rule number 6 from the description in Fig. 1 we get an incomplete description. U is given in Fig. 4. State number 7 does not fulfill the criterion.

Number	State $[P,q]$	Stored Information	Element of $L([P,q])$
1	[{Areg};A]	bb	bb
2	[{Ea, Dreg};D]	c	c
3	[{Ea, Dreg};A]	cont(1)	cont(bb)
4	[{Z};T]	assign(1,2)	assign(bb,c)
5	[{};D]	cont(3)	cont(cont(bb))
6	[{};A]	cont(3)	cont(cont(bb))
7	[{};T]	assign(3,1)	assign(cont(bb),bb)

Fig. 4. U of example description without rule 6

Therefore the description is not complete. The completeness test would produce the counterexample *assign(cont(bb),bb)*.

4 Refined Version of the Algorithm

4.1 Representation of States

For a more efficient implementation we use a different representation for U: Instead of states we store tuples $\langle P;Q\rangle$ where $P \subset \mathcal{S}_G$ and $Q \subset \mathcal{S}_T$. U is represented by a set R of these tuples. Each tuple $\langle P;Q\rangle$ represents all the states $[P;q]$ with $q \in Q$. There is at most one tuple $\langle P;Q\rangle$ with first component P in R.

In other words: Rather than storing each state $[P;q]$ of U we store one tuple $\langle P;Q\rangle$ for every distinct P with $Q = \{q|[P;q] \in U\}$. The tuples of R can be efficiently implemented by structs containing two bitsets.

The condition given in Sect. 3.4 for U can be redefined for the direct calculation of R: R is the smallest set of states which fulfills the following property:

> Suppose an operator θ with arity α and tuples $\langle P_1,Q_1\rangle,\ldots,\langle P_\alpha,Q_\alpha\rangle \in R$ exist. Let
>
> $$P = \{s|\exists p_1 \in P_1,\ldots,p_\alpha \in P_\alpha\ \theta(p_1\ldots p_\alpha) \stackrel{G}{\Rightarrow} s\} \quad (1)$$
>
> $$Q = \{s|\exists q_1 \in Q_1,\ldots,q_\alpha \in Q_\alpha\ \theta(q_1\ldots q_\alpha) \stackrel{T}{\Rightarrow} s\} \quad (2)$$
>
> If $Q \neq \emptyset$ then there has to be a tuple $\langle P,Q'\rangle$ in R and Q has to be a subset of Q'.

This definition immediately leads to an algorithm. The algorithm starts with empty R. Then it performs the following steps for each operator θ with arity α and every combination of the tuples $\langle P_1,Q_1\rangle,\ldots,\langle P_\alpha,Q_\alpha\rangle \in R$ including the tuples which are added to R while performing the loop. It calculates P and Q according to the specification above. If $Q = \emptyset$ there is nothing to do. Otherwise if there is no tuple with first component P in R then the tuple $\langle P;Q\rangle$ is added to R. If there is already a tuple $\langle P;Q'\rangle$ in R we check $Q \subset Q'$. If not we replace the tuple $\langle P;Q'\rangle$ by $\langle P;Q' \cup Q\rangle$ and we mark the tuple as new, so that it will be considered again when testing combinations.

Testing the completeness property using R is very simple: The description is not complete iff there is a tuple $\langle P;Q\rangle \in R$ with $\mathcal{Z}_G \notin P$ and $\mathcal{Z}_T \in Q$.

Recording the information needed to calculate counterexamples is a bit more complex, because it is not sufficient to store one possibility to produce a tuple. In contrast the information has to be recorded whenever the set Q of a tuple grows.

4.2 Restricting Combinations

When processing operators θ with arity α greater than one, the algorithm has to consider every combination of α tuples from R. Obviously there are $|R|^\alpha$ combinations which can be very many. To drastically reduce the number of combinations to consider we use the following two optimizations.

For each operator θ we look at each operand position n separately. Let $W \subset \mathcal{S}_T$ be the set of states which occur as the n-th operand of a rule with operator θ in the description of the intermediate code T.

$$W = \{s_n \in \mathcal{S}_T | \theta s_1 \ldots s_\alpha \rightarrow s_0 \in T\}$$

When building combinations we can ignore every tuple $\langle P_n;Q_n\rangle$ with $W \cap Q = \emptyset$, because when using such a tuple the Q in equation 2 will be empty.

The idea of the second optimization was first introduced in [Cha87]: Let $S \subset \mathcal{S}_G$ be the set of states which occur as the n-th operand of a rule with operator θ in G.

$$S = \{s_n \in \mathcal{S}_G | \theta s_1 \ldots s_\alpha \rightarrow s_0 \in G\}$$

Consider two different tuples $\langle P_n;Q_n\rangle$ and $\langle P'_n;Q'_n\rangle$ with $P_n \cap S = P'_n \cap S$. They can be merged into a tuple $\langle P_n \cap S, Q_n \cup Q'_n\rangle$. Then the merged tuple can be used instead of the two original tuples on this operand position without changing the final result. This is the case because P_n and P'_n differ only in states of $\mathcal{S}_G$ which are not relevant, because no rule references them. So P in equation 1 is not effected and Q in equation 2 of the merged tuple becomes the union of the Qs of the two tuples $\langle P_n;Q_n\rangle$ and $\langle P'_n;Q'_n\rangle$.

According to this rule we merge for every operand position as many states as possible. This usually leads to a substantial reduction of combinations to process.

4.3 A Variant using Tables

A possible inefficiency of the algorithm occurs when a tuple $\langle P;Q\rangle$ should be inserted while there is already a tuple $\langle P;Q'\rangle$ in R. If $Q \subset Q'$ there is nothing to do and hence no problem. But if Q brings in something new we have to replace $\langle P;Q'\rangle$ by $\langle P;Q' \cup Q\rangle$ and to mark the state as new. This means that all the calculations done on $\langle P;Q'\rangle$ are void and have to be redone, because for some states reachable from $\langle P;Q'\rangle$ the Q-component might grow.

This can be improved by building up automata tables when states are inserted the first time and recalculating the Q-components based on the tables. This saves the time of calculating the P-components in the main step of the algorithm in Sect. 4.1. It is unclear how much this speeds up the algorithm, but it is clearly advantageous if the tables are needed anyway.

Another variant of the algorithm could run on the tables produced by an existing tree parser generator: Then the tree parser generator would in a first step perform the calculation of the P-components of all tuples contained in R. Afterwards one can calculate the Q-components using the tables. However there is a disadvantage when proceeding this way: When building up tuples also tuples of the form $\langle P, \emptyset \rangle$ will be produced. These tuples can be reached only by trees which are not correct intermediate code trees or parts thereof. So these tuples are in fact redundant and are produced for nothing. They do not occur in the algorithm of Sect. 4.1.

So it would be in fact advantageous to proceed the other way round and build this algorithm into existing tree parser generators, because it helps to get rid of some redundant states. [MöWe86] mentions this but only for tree matching rather than tree parsing. As for code selection usually cost values are encoded into the tables it would be necessary to extend our method in this way. It is an open question how the algorithm behaves in this context. Practically this is useful when we have a machine description processing a big intermediate language (e.g. an intermediate code with constructs of several different source languages) and we want to generate a code selector which handles only part of it (e.g. a code selector for only one source language).

5 Practical Results

5.1 Experimental Implementation

The completeness test algorithm has been implemented as a prototype. The first version of the algorithm that does not build up tables has been used.

There are no problems to process even big code selector descriptions in acceptable execution times. A code selector description with 188 rules and 10 nonterminals was processed within 0.7 sec user time on DECStation 3100. This time measures only the completeness algorithm and does not include the time for reading in the code selector and the intermediate code specification and building up internal data structures for them. The intermediate code description had 100 rules and 5 nonterminals. The set U contained 46 states and R contained 37 tuples. These rather low numbers come from the fact that the states do not contain costs. So many of the rules are in fact redundant and do not produce new states. They only become interesting to produce *good* code. In other words, a rather small set of rules is enough to produce code for every possible intermediate code tree, the other rules are only required to increase code quality.

The times measured show that the completeness test is not as time critical as one might expect.

5.2 Handling of Conditions

Some code selector generators allow rules to contain conditions. For example in the BEG system ([ESL89]) conditions have to be specified as expressions in the target programming language. It is very difficult or even impossible to analyze the condition itself to get the information to perform a proper completeness test. The only thing we can do with descriptions containing conditions is the following:

In a first step all conditions are assumed to be true all the time. When the completeness test fails one is sure, that the description is not complete. One can analyze the counterexample and add some more rules and try again.

Afterwards we assume all conditions to be false. That means in fact deleting all rules with conditions. Then often the code selector description is not complete any more. This happens for instance when there are two rules which differ only in the cost, action and condition parts, in such a way that always one of the two conditions is true. Then taking the special form of the conditions into account the description is, in fact, complete but the test cannot find out. In these cases the user can tell the generator to assume for the completeness test that the condition of a certain rule is true.

5.3 Application of the Completeness Test on Existing Code Selector Specifications

We used several existing BEG descriptions. As BEG allows conditions we had to cope with the problems explained in Sect. 5.2. Several descriptions turned out to be complete, however one experimental description was incomplete. In fact there were two counterexamples, which belonged to two different classes of error:

In the first case it turned out that the intermediate code description was not correct. The code generator assumed that one operand of a certain operator was always a constant. In fact this was no compiler error, because the front end only produced code in the required way. However the documentation of the intermediate code did not describe this restriction. This lack of documentation might have led to complications when a new front end had to be interfaced.

The other counterexample was a proper compiler error. A certain floating point operation could be compiled only when it's operand was in memory and not in a register. This error has not been found by testing the compiler, since the special floating point operation was rarely used, and obviously never in a case where the operand was only available in a register and not in memory.

This shows that the completeness test is not only of theoretical interest, but is useful in practical compiler development to improve the quality of the generated compilers.

6 Conclusion

We have presented an algorithm to perform completeness tests on code generator descriptions. Therefor the code generator descriptions had to be extended by a description of the intermediate code. Then the completeness test can prove that every possible intermediate code tree can be processed successfully.

The algorithm has been implemented and it turned out that there are no efficiency problems even when processing realistic code generator descriptions.

The completeness test was then applied on some existing code selector specifications. Although most of the descriptions analyzed were complete, an incomplete description was found. In one case this helped to remove a real compiler error, in another case the documentation of the intermediate code had to be extended.

So finally we can conclude that the completeness test can be implemented efficiently and helps to find errors in real life compiler construction.

References

[Cha87] *D.R.Chase:* An Improvement to Bottom-up Tree Pattern Matching, POPL87, Munich, Germany, January 1987

[ESL89] *H.Emmelmann, F-W.Schröer, R.Landwehr:* BEG – a Generator for Efficient Back Ends, Proc. of the Sigplan'89 Conference on Programming Language Design and Implementation, Sigplan Notices, Vol. 24, Number 7, July 1989

[FeSe92] *Ferdinand, Seidl and Wilhelm* Tree automata for code selection, Proceedings of the Code'91 workshop on code generation, Workshops in Computing, Springer 1992

[GiSchm88] *R.Giegerich, K.Schmal:* Code selection techniques: pattern matching, tree parsing and inversion of derivors, ESOP '88, LNCS 300, pp. 247-268, March 1988

[Gie90] *R.Giegerich:* Code Selection by Inversion of Order-Sorted Derivors, Theoretical Computer Science 73 (1990), pp. 177-211

[HaCh86] *P.J.Hatcher, T.W. Christopher:* High–Quality Code Generation Via Bottom–Up Tree Pattern Matching, Proceedings 13th ACM Symposium on Principles of Programming Languages pp. 119-130, 1986

[HoDo82] *C.Hoffmann, J.O'Donnell:* Pattern Matching in Trees, Journal of the ACM Vol. 29, No. 1, January 1982, pp. 68-95

[MöWe86] *Möncke, Weisgerber, R.Wilhelm* Generative support for transformational programming. ESPRIT: Status Report of Continuing Work, Elsevier Sc., 1986, Brussels