Considerate Code Selection

Robert Giegerich
Universität Bielefeld
Technische Fakultät
Postfach 10 01 31
W-4800 Bielefeld 1
Germany
robert@techfak.uni-bielefeld.de

Abstract

Considerate code selection is not another code selection technique. It is concerned with the integration of code selection with other subtasks of code generation, such as register allocation and scheduling. Considerate code selection allows to defer decisions between alternative encodings. This is achieved by means of a shared representation of the overall solution space. Subsequent phases are adapted to process all solutions simultaneously, again producing results in a shared representation. Decisions may be interspersed in this process whenever desired.

The paper introduces this technique in a framework where code selection in done by tree parsing, and later phases are described by attribute coupled grammars. Being a general technique rather than an algorithm, considerate code selection can be used with any of the current, pattern based approaches to code selection.

1 Motivation

The pattern matching approach to code selection, when implemented by bottom-up tree parsing, allows three ways to deal with the fact that there are many alternative encodings for a given source program:

- The *maximal-munch-heuristic* favours encodings with machine instructions that implement serveral source operators at a time. Precaution must be taken, as this strategy may lead into blind alleys. Also, the maximal size of munches does not strictly imply minimal target program costs.

- A cost driven heuristic such as *dynamic programming* tries to achieve good overall solutions composed from locally optimal subsolutions.

Both strategies have proved to be practical. Both, however, simplify code selection at the price of a negative effect on the modularity of the code generation problem as a whole: As alternative solutions are discarded at each stage, all other considerations such as register usage must be interleaved with the code selection process. In this paper, we propose a third approach:

- The set of all solutions is explicitly constructed. Later phases of code generation may apply register requirements analysis, instruction cost considerations and other criteria to successively reduce the solution set, eventually to a single target program. This approach will be called *considerate code selection*, and the present paper is a first exploration of this idea. The name comes from the fact that for code selection proper, no heuristic is applied.

At the first glance, considerate code selection looks like a tantalizing idea. As the number of overall encodings for a given source program grows combinatorially with the number of encodings for its immediate subprograms, the set of all solutions is exponential in the size of the input. Hence, it should be excessively expensive to construct this solution set, as well as to process it further. At a second thought, however, we note the following:

- Since tree parsing can produce any encoding, it may as well produce all encodings without extra cost. In the pattern matching terminology, the bottom-up pattern matcher implicitly constructs all covers of the intermediate program tree in $\mathcal{O}(n)$, where n is the number of its nodes. What we need is a compact representation of this solution set of size $\mathcal{O}(2^n)$ in $\mathcal{O}(n)$ space, which can be achieved by an appropriate kind of sharing contexts.
- Later tasks of code generation can be described as analyses or transformations of a particular encoding. What we need is a mechanism to apply such operations to the compact representation of all solutions simultaneously, hopefully retaining the amount of sharing.

The data structure used for this purpose, introduced below, is called *shared forest*. It is named after a related approach that has evolved independently in the area of natural language parsing [15]. Although intuitively, the problems to be solved and the data structures used are quite similar in that work and ours, the formalizations are different and their relationship has not been explored yet in any depth.

2 The Model of Code Generation

For the formal development of our approach, we need the following notations: A signature $\Sigma = (S, F)$ is given by a set of sorts S and a set of F of operators together with their arity. The set of Σ-Terms is denoted $T(\Sigma)$, while the set of Σ-terms of sort s is denoted $T(\Sigma) : s$. $T(\Sigma, X)$ denotes terms with variables from a variable set X. For $t \in T(\Sigma, X), var(t)$ denotes the set of variables occuring in t, $t\sigma$ denotes application of a substitution (of terms for variables) to t, yielding a term t' called an instance of t. Given a confluent and terminating term rewrite system $R, t\downarrow_R$ denotes the normal form of t with respect to R. For further terminology about term rewrite systems, see e.g. [3]. Following common usage, we will also use the words *trees* and *forests* for terms and sets of terms, respectively.

We use an algebraic model of code generation, as in [12, 8, 9]. Source and target programs are represented as terms of a source signature IL and a target signature TL. Target
formedness predicates. Th
be expressed syntactically.
TL-homomorphism, specifie
proper means inverting this
$m(t) = p$ for the given sour
furthermore to find a t' tha
well as $m(t') = p$).

For the presentation, cor
tion problem: Let there be
$ld(r, c)$, denoting addition c
load constant. The first arg
ber of the target register use
p like $(1 + 2) + (3 + 4)$. Th
specified by the TL-homom

Example 2.1 :

signature TL =

sorts *R, C, Regno*

ops *ld: Regno, C →R*
addi: Regno, R, C
add: Regno, R, R

$$m(ld(r,c)) =$$
$$m(addi(r,x,c)) =$$
$$m(add(r,x,y)) =$$

□

Code selection means sol

$$m(z) = p$$

where z is a target progr

Here, the overall transla
operator of p and choices of t
2.1, the code selection proble
4), shown in Example 3.2 b
to further constraints regard

3 Shared Forests

A Σ-forest of sort s is a sub
representation of a Σ-forest
(up to) logarithmic space re

target signature TL. Target programs are further restricted by certain well-formedness predicates. These model target machine properties that cannot be expressed syntactically. The target is related to the source language by a TL-homomorphism, specified as a derivor (denoted m below). Code selection proper means inverting this derivor, i.e. constructing (one or all) t satisfying $m(t) = p$ for the given source program p. Code generation as a whole means furthermore to find a t' that satisfies all further well-formedness criteria (as well as $m(t') = p$).

For the presentation, consider the following trivial instance of a code selection problem: Let there be machine instructions $add(r, x, y)$, $addi(r, x, c)$ and $ld(r, c)$, denoting addition of two registers, add-immediate to a register, and load constant. The first argument within each instruction is the register number of the target register used. We want to generate code for source expressions p like $(1 + 2) + (3 + 4)$. The relation between target and source language is specified by the TL-homomorphism m in Example 2.1:

Example 2.1 :

signature $TL =$

sorts R, C, $Regno$

ops ld: $Regno, C \rightarrow R$
$addi$: $Regno, R, C \rightarrow R$
add: $Regno, R, R \rightarrow R$

$$\begin{aligned} m(ld(r,c)) &= c \\ m(addi(r,x,c)) &= m(x) + c \\ m(add(r,x,y)) &= m(x) + m(y) \end{aligned}$$

□

Code selection means solving the equation

$$m(z) = p \tag{1}$$

where z is a target program variable, and p a source program.

Here, the overall translation of a source program p depends on the root operator of p and choices of translations for certain subexpressions. In Example 2.1, the code selection problem $m(z) = p$ has four solutions for $p = (1+2)+(3+4)$, shown in Example 3.2 below. From them, we need to select one according to further constraints regarding register allocation and instruction costs.

3 Shared Forests

A Σ-forest of sort s is a subset of $T(\Sigma) : s$. A shared Σ-forest is a particular representation of a Σ-forest which exploits sharing of contexts and allows an (up to) logarithmic space reduction.

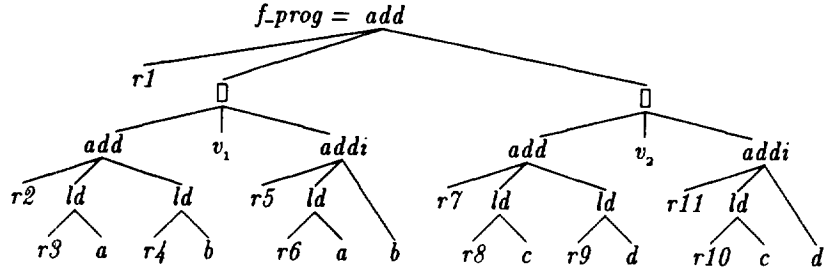
Definition 3.1 : Shared forests

1. *For a given $\Sigma = (S, F)$, and an S-sorted set of variables X let $d\Sigma = (S \cup \{choices\}, F \cup \{\square_s : s, choices, s \rightarrow s | s \in S\} \cup \{l :\rightarrow choices, r :\rightarrow choices\})$. The operators $\square_s$ are called choice operators.*
2. *Let V : choices be a set of variables of sort choices.*
3. *The set of shared Σ-forests is $T(d\Sigma, X \cup V)$.*

□

Of course, we assume that the symbols added by the extension are not already present in Σ and X. Mostly, we shall omit the subscript s with $\square$.

Example 3.2 : Shared forest solution *f_prog* to $m(z) = (a + b) + (c + d)$



□

Two kinds of variables occur in the shared target forest of Example 3.2: $r1, r2, \cdots$ are variables for register numbers, yet to be instantiated. v_1 and v_2 are choice variables, whose purpose will become clear shortly.

A shared forest represents a forest in an obvious way:

Definition 3.3 : Semantics of shared forests

1. *Let A be the rewrite system given by $\{\square_s(x, l, y) \rightarrow x, \square_s(x, r, y) \rightarrow y \mid s \in S\}$*
2. *For all $s \in S$, the interpretation $I : T(d\Sigma, V) : s \rightarrow 2^{T(\Sigma):s}$ is given by $I(w) = \{w\sigma\downarrow_A \mid \sigma$ is a ground substitution for the choice variables in $w\}$.*

□

A shared Σ-forest w denotes a set of Σ-terms $I(w)$. Instantiating the choice variables in w by l or r and normalizing with A yields a particular element of $I(w)$. Note that we can represent neither empty nor infinite forests.

The sharing provided by shared forests is complementary to that provided by the dag-representation of trees. While dags share identical subterms within a term, shared forests share *identical contexts of different subterms*. Both kinds of sharing can be combined to a certain extent. For simplicity, we avoid dags in this paper.

The potential compactification in representing $I(w)$ by w is measured as follows:

Lemma 3.4 :

Let w be a shared for
Then we have $1 \leq |I($

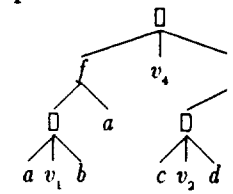
Proof:

To obtain $|I(W)|$,
$a \rightarrow 1$ for each co
$x \rightarrow 1$ for each va
$f(x_1, \cdots, x_n) \rightarrow *$
for each n-ary
$\square(x_1, v, x_2) \rightarrow x_1 +$

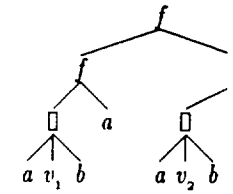
□

Evaluating the resulti
achieved when w is linea
all n $\square$-operators are ind
See Example 3.5.

Example 3.5 :



a) $\left|I(w)\right| = \left|\{f(a,\right.$



b.1) Maximum for n =

□

Note the way in which
of w. If we substitute all
Considering space red
of operators) to the sum
of context sharing presen
The following axioms

Lemma 3.4 :

Let w be a shared forest containing n $\square$-operators.
Then we have $1 \leq |I(w)| \leq 2^n$, and this bound is sharp.

Proof:

To obtain $|I(W)|$, we can translate w into a term over $(N, +, *)$ by

$a \rightarrow 1$ for each constant $a \in F$

$x \rightarrow 1$ for each variable $x \in X$

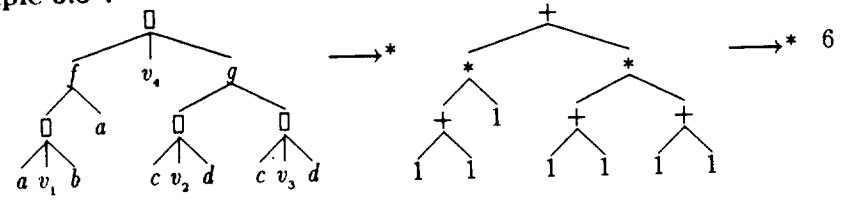
$f(x_1, \cdots, x_n) \rightarrow *(x_1, \cdots, x_n)$
for each n-ary operator $f \in F, n \geq 1$

$\square(x_1, v, x_2) \rightarrow x_1 + x_2$

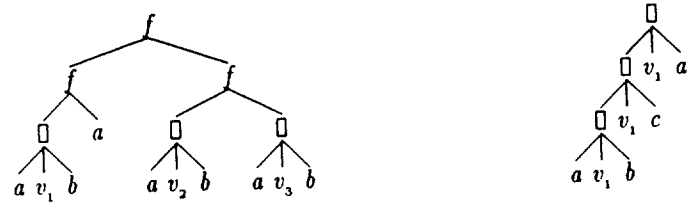
□

Evaluating the resulting expression yields an upper bound for $|I(w)|$. It is achieved when w is linear (i.e. no choice variable occurs more than once in w), all n $\square$-operators are independent, and when $\sigma \neq \sigma'$ implies $w\sigma \downarrow_A \neq w\sigma' \downarrow_A$. See Example 3.5.

Example 3.5 :



a) $\left|I(w)\right| = \left|\{f(a,a),\ f(b,a),\ g(c,c),\ g(d,c),\ g(c,d),\ g(d,d)\}\right| = 6$



b.1) Maximum for $n = 3$: $\left|I(w)\right| = 8$ *b.2) Minimum for* $n = 3$: $\left|I(w)\right| = 1$

□

Note the way in which $|I(w)|$ in example 3.5b.1) depends on (non-)linearity of w. If we substitute all choice variables by the same variable v, $|I(w)| = 2$.

Considering space reduction, we must relate the $|w|$ (the size of w in terms of operators) to the sum over $|w\sigma|$ for all σ. This ratio depends on the amount of context sharing present in w.

The following axioms are consistent with the interpretation I:

Definition 3.6 : Condensation/Expansion Axioms

For all $s \in S$

1. $\square(x, v, x) = x$
2. $\square(f(x_1, \cdots, x_{i-1}, x, x_{i+1}, \cdots, x_n), v, f(x_1, \cdots, x_{i-1}, y, x_{i+1}, \cdots, x_n))$
$= f(x_1, \cdots, x_{i-1}, \square(x, v, y), x_{i+1}, \cdots, x_n)$
$\forall f \in F \wedge 1 \leq i \leq n$

□

The condensation rewrite rule system C is obtained by orienting these equations left-to-right. The expansion rewrite rule system E is obtained by orienting 2. right-to-left.

Each w has a condensed normal form $w\downarrow_c$. Clearly $I(w) = I(w\downarrow_c)$, but $w\downarrow_c$ is neither the smallest, nor a unique representation of $I(w)$, as we do not consider the commutative, associative and idempotent properties, which $\square$ has under the interpretation I.

Any $w \in T(d\Sigma, V)$ can be fully expanded by E, such that all $t \in I(w)$, including duplicates, show up separately under a root portion consisting of $\square$-operators only. Conversely, such a "shared" forest can be condensed by C. Pragmatically, neither of these should ever happen in a computation where w is a logarithmic reduction of $I(w)$! Our interest in shared forests arises from the fact that we know algorithms (e.g. for the code selection problem) that directly construct shared representations of the desired solution space.

4 Construction of Shared Forests by Tree Parsing

Let us recall the approach of [12] and [8]: The equation $m(t) = p$ is solved by first converting the derivor m into a regular tree grammar. The productions are labeled by the corresponding target operators. The grammar for our example is untypically simple, as our target signature is one-sorted (there is only one address mode, R). This leads to a grammar with a single nonterminal symbol R and three productions corresponding to the target operators *ld*, *addi* and *add*. The terminal symbol c matches arbitrary numeric constants

(ld)		$R \rightarrow c$
$(addi)$	(2)	$R \rightarrow R + c$
(add)	(3)	$R \rightarrow R + R$

For $m(t) = a + b$ (where a, b are constants), the tree parser detects parses

$R \rightarrow R + b \rightarrow a + b$, corresponding to $addi(r_{11}, ld(r_{12}, a), b)$, and

$R \rightarrow R + R \rightarrow R + b \rightarrow a + b$, corresponding to $add(r_{13}, ld(r_{14}, a), ld(r_{15}, b))$.

As both start from the same nonterminal, the corresponding target terms are of the same sort and can share contexts. Hence, the result is

$$t = \square(addi(r_{11}, ld(r_{12}, a), b), v_1, add(r_{13}, ld(r_{14}, a), ld(r_{15}, b)))$$

When reducing by productio
addi(r, x, c) is applied to argu
corresponding to r. Thus, a f
number yet to be chosen. r
forest constructed this way.
the shared forest of Example

This informal description
Technical details of the const
parse of the input can be fou
in the tree grammar and inf
to infinitely many target ter
be represented finitely. A co
parsing is in preparation [11]

Shared forests that arise f
Later phases, as we shall see

5 Translations of

In our approach to code gen
lations between appropriate
translations is given by the u
modularity, it is wise to requ
of specifications is possible w
show how a category of tran
translations between the cor
morphisms between term alg
$h \in H$ some morphism dh
diagram commutes:

$2^{T(\Sigma}$

I

$T(d\Sigma,$

Whether and how this ca
shall consider the class of m
[6].

Attribute coupled gramn
the underlying context free g
Attributes are associated wit
of an output signature Σ'.
$T(\Sigma', X)$. Attribute rules sp
attributes in the local conte
given $t \in T(\Sigma)$, the value of
called the translation $h(t)$ of
$T(\Sigma)$ to $T(\Sigma')$ specified in t
approach. A main appeal of

When reducing by production 2, its corresponding target operator $addi(r, x, c)$ is applied to arguments $ld(r_{12}, a)$ and b, while there is no argument corresponding to r. Thus, a free variable r_{11} is substituted, denoting a register number yet to be chosen. r_{11} must be unique in the overall target program forest constructed this way. Continuing this process for $p = (a + b) + (c + d)$, the shared forest of Example 3.2 is obtained.

This informal description of shared forests construction must suffice here. Technical details of the construction of a single target term from a single tree parse of the input can be found in [8], where also the possibilities of chain rules in the tree grammar and infinite derivations are considered. The latter lead to infinitely many target terms. According to recent results of [2], these can be represented finitely. A comprehensive treatment of derivor inversion by tree parsing is in preparation [11].

Shared forests that arise from tree parsing are linear in the choice variables. Later phases, as we shall see shortly, may well introduce non-linearities.

5 Translations of Shared Forests

In our approach to code generation, various subtasks are described as translations between appropriate representations. This class of representations and translations is given by the underlying specification technique. For the sake of modularity, it is wise to require that they form a category. Then, composition of specifications is possible where ever composition of translations is. We now show how a category of translations between terms gives rise to a category of translations between the corresponding shared forests. When H is a class of morphisms between term algebras $T(\Sigma)$ and $T(\Sigma')$, we want to derive for each $h \in H$ some morphism $dh : T(d\Sigma, V) \to T(d\Sigma', V)$ such that the following diagram commutes:

$$\begin{array}{ccc} 2^{T(\Sigma)} & \xrightarrow{\quad h \quad} & 2^{T(\Sigma')} \\ \uparrow I & & \uparrow I \\ T(d\Sigma, V) & \xrightarrow{\quad dh \quad} & T(d\Sigma', V) \end{array}$$

Whether and how this can be done depends on the way H is defined. We shall consider the class of morphisms defined by attribute coupled grammars [6].

Attribute coupled grammars are classical attribute grammars [14] where the underlying context free grammar is seen as an input signature $\Sigma = (S, F)$. Attributes are associated with the sorts s from S. Attribute values are terms of an output signature Σ'. "Semantic functions" are composite terms from $T(\Sigma', X)$. Attribute rules specify the values of attributes, depending on other attributes in the local context. Circular dependencies are forbidden. Thus, given $t \in T(\Sigma)$, the value of its designated root attribute is some $t' \in T(\Sigma')$, called the translation $h(t)$ of t. An attribute coupling is the translation from $T(\Sigma)$ to $T(\Sigma')$ specified in this way. Note that attributes are transient in this approach. A main appeal of this is that attribute couplings can be composed,

thus supporting modularity. An example of an attribute coupling is given after Theorem 5.2.

Attribute coupled grammars are used here because they are a very general (and well-understood) scheme of inductive definition. They encompass standard structural induction as the special case of a single, synthesized attribute. It is straightforward to transfer the construction to more restricted forms of structural induction.

Definition 5.1 : Lifting of attribute couplings to shared forests

Given an attribute coupling $h : T(\Sigma) \to T(\Sigma')$, *its "lifting" to shared forests is another attribute coupling* $dh : T(d\Sigma, V) \to T(d\Sigma', V)$, *obtained as follows:*
Take over all attribute declarations and rules of h. Add the following rules for each choice operator:

$$t = \square_s(x, v, y): \begin{array}{ll} x.i = t.i \\ y.i = t.i \end{array} \Big\} \quad \text{for each inherited attribute } i \text{ associated with } s$$
$$t.d = \square_{s'}(x.d, v, y.d) \quad \text{for each synthesized attribute } d \text{ of sort } s' \text{ associated with } s$$

□

The clue in this (otherwise straightforward) construction is that the choice-variable v, associated with the input choice-operator $\square_s$, is also associated with the output choice-operator $\square_{s'}$. We must now show that this construction is consistent with our interpretation of shared forests.

Theorem 5.2 :

Let $w \in T(d\Sigma, V), h : T(\Sigma, X) \to T(\Sigma', X), dh : T(d\Sigma, X \cup V) \to T(d\Sigma', X \cup V)$ *constructed according to Definition 5.1.*
Then, $\{h(t) | t \in I(w)\} = I(dh(w))$.

Proof:

We show that for an arbitrary ground substitution σ that substitutes all the choice variables in w, $h(w\sigma \downarrow_A) = (dh(w))\sigma \downarrow_A$. Consider the following synchronized A-reduction step of $w\sigma$ and $(dh(w))\sigma$: Let $w\sigma \to_A w_1\sigma$ be a reduction of some choice operator $\square_0(x_0, v\sigma, y_0)$.

Let s, the sort of $\square_0$, have n synthesized attributes. According to the definition of dh, their values have the form

$$\square_1(x_1, v\sigma, y_1), \cdots, \square_n(x_n, v\sigma, y_n).$$

(Note that $\square_0, \cdots \square_n$ use the same choice variable).

Let $(dh(w)\sigma) \to_A^n q_1$ by n-fold reduction of these choice operators in $(dh(w))\sigma$. Since the choice is consistently x_i or y_i in all cases $(0 \le i \le n)$, and the inherited attributes of x_0 and y_0 are copied from $\square_0$, we have $q_1 = dh(w_1)\sigma$. Iterating this step until all choice operators are eliminated, we obtain some w_k, q_k with $q_k = (dh(w_k))\sigma \downarrow_A = dh\ w_k = h\ w_k$, since dh and h coincide on terms without choice operators. Remembering that $w_k = w\sigma \downarrow_A$ and $q_k = (dh(w))\sigma \downarrow_A$, we have established $h(w\sigma \downarrow_A) = (dh(w))\sigma = \downarrow_A$, q.e.d.

□

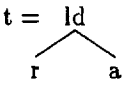
Even when w is in C-n
a condensation step by ref

$t.d =$ **if** $x.d = y.d$ **th**

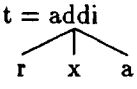
However, $x.d$ and $y.d$ m
be compared to achieve th
this step should be include

As an example, we desc
ules. The variables creat
shared forest now act as s
is obtained from TL by ac
quence [ ]. We assume th
all load-instructions before
attribute coupling that sh
attributes[1] involved is:

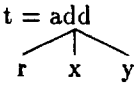
il, sl: inherited/synth
containing a se
ic, sc: attributes cont
n: number of regi
(this may be a



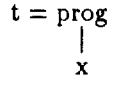
t.n =
t.sl =
t.sc =



t.n =
t.sl =
t.sc =



t.n =
t.sl =
t.sc =



t.code

According to Definition
forests. Applied to the sh
shared forest of schedules:

[1] An equivalent attribute cou
pler. This example was chosen
inherited attributes.

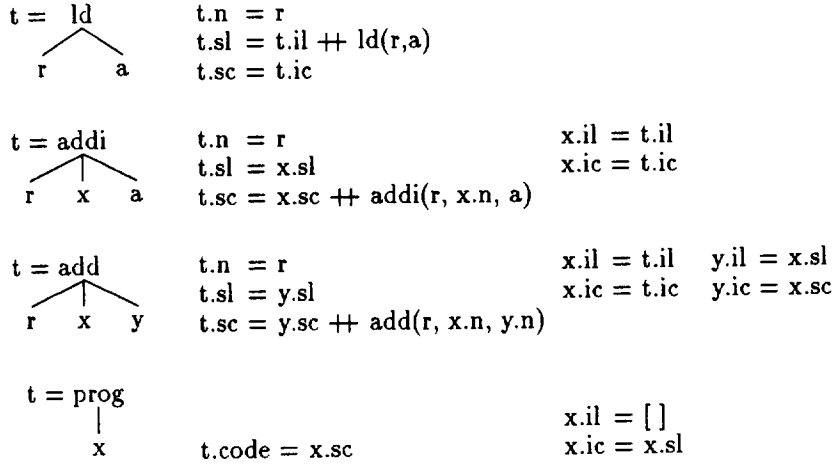
Even when w is in C-normal form, $dh(w)$ need not be so. One may include a condensation step by reformulating the equation for synthesized attributes to

$$t.d = \textbf{if } x.d = y.d \textbf{ then } x.d \textbf{ else } \square_{s'}(x.d, v, y.d).$$

However, $x.d$ and $y.d$ may be rather large terms from $T(d\Sigma', V)$, which must be compared to achieve the condensation. It is a pragmatic question whether this step should be included.

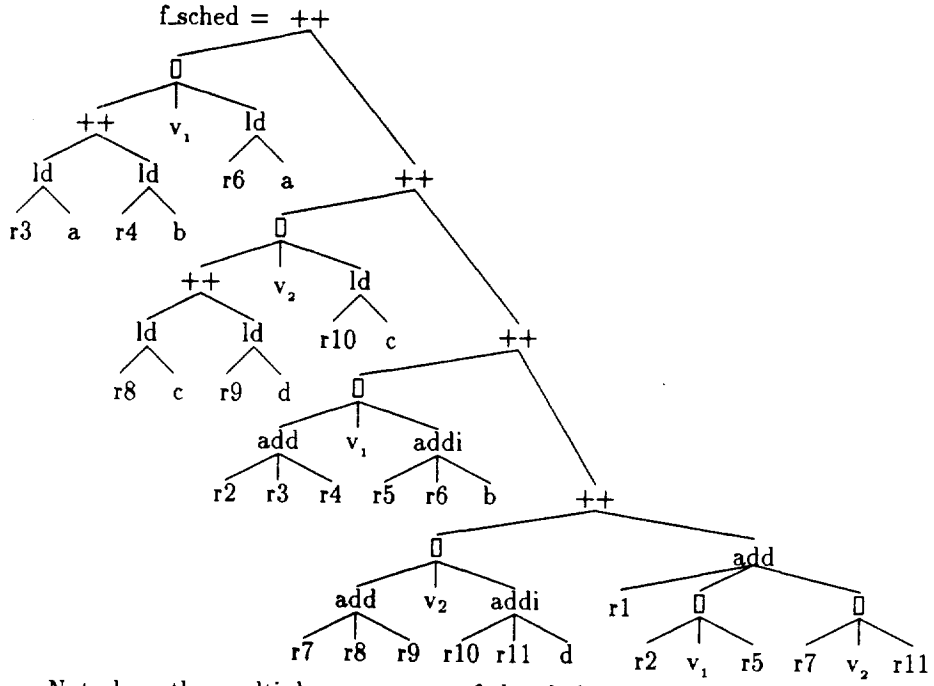
As an example, we describe the linearization of target programs into schedules. The variables created for register numbers during construction of the shared forest now act as symbolic register names. The signature of schedules is obtained from TL by adding a sequencing operator ($++$) and an empty sequence $[\,]$. We assume that the processor architecture suggests to schedule all load-instructions before any arithmetic instructions. We specify this by an attribute coupling that should be largely self-explanatory. The purpose of the attributes[1] involved is:

il, sl: inherited/synthesized attribute pair containing a sequence of load instructions,
ic, sc: attributes containing final schedule,
n: number of register that holds result of a subtree, (this may be a variable from X : *Regno*).



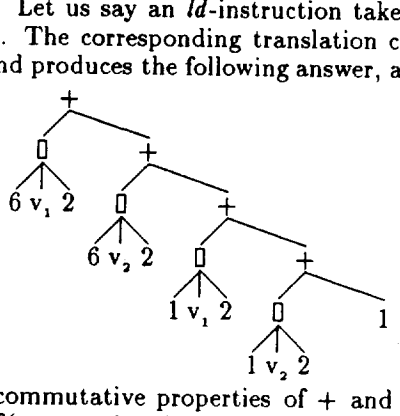
According to Definition 5.1, this attribute coupling can be lifted to shared forests. Applied to the shared forest of Example 3.2, we obtain the following shared forest of schedules:

[1] An equivalent attribute coupling using synthesized attributes only may appear even simpler. This example was chosen to illustrate the role of choice variables in the presence of inherited attributes.

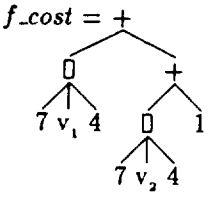


Note how the multiple occurences of the choice variables now control the effects of possible choices in quite different parts of the schedule, while retaining the sharing.

Another phase might measure schedules by (say) machine cycles. Again this measure could be described as an attribute coupling translating a single schedule into its cost. Let us say an *ld*-instruction takes 3, *addi* takes 2, and *add* takes 1 unit cost. The corresponding translation can be lifted to shared forests of schedules, and produces the following answer, a shared forest of costs:



Using associative-commutative properties of + and distributive laws like $\square(a,v,b)+\square(c,v,d)=\square(a+c,v,b+d)$, we obtain the shared cost forest:



6 Selections fr

Solution sets of potentiall
of a particular computatic
even unitary. So we must
by a shared forest. Even
some point.

One possibility is to a
enumerates solutions fro
the transparency of our i
tion. A second possibility
handling of choice operat
first approach views each
more ad-hoc, allows for e
choice based on a preferer

From the shared cost f
determine that the substi
of 9 units. Now we finally
schedule, $(f\_prog)\sigma\downarrow_A$ is t

There is one thing ad
a chance to avoid it alto
different alternatives. Hen
operators.

We now study a diffe
the setting of Example 1.
by the morphism m. W
operators, it does not car
Example 6.1 defines a pr
target program are assign
architectures. Thus, the fi
is a list of registers (i.e.
first argument. We borro

Example 6.1 :

$alloc(add(i,x,y),[i|u])$
$alloc(add(i,x,y),[\,])=$
$alloc(addi(i,x,c),[i|u]$
$alloc(addi(i,x,c),[\,])=$
$alloc(load(i,c),[i|u])=$
$alloc(load(i,c),[\,])=f$

□

Note that the equation
canonical rewrite system.

Given a source progra
at hand is to generate wel

[2] In general, this is by no m
an exponential solution space
not avoid it. Their advantage
suitable data structure.

6 Selections from Shared Forests

Solution sets of potentially exponential size are likely to be intermediate results of a particular computation, while the final solution set often is of linear size, or even unitary. So we must provide ways to restrict the solution set represented by a shared forest. Even considerate code selection must make selections at some point.

One possibility is to adopt a general, nondeterministic method that (lazily) enumerates solutions from the shared representation. This approach retains the transparency of our implementation technique for the writer of a specification. A second possibility is to give up this transparency and allow an explicit handling of choice operators, to be provided by the specification writer. The first approach views each solution independently, the second approach, while more ad-hoc, allows for explicitly relating different sub-solutions and making choice based on a preference relation. We will sketch both approaches here.

From the shared cost forest *f_cost* obtained above, it is straightforward[2] to determine that the substitution $\sigma = [v_1 \leftarrow r, v_2 \leftarrow r]$ yields the minimal cost of 9 units. Now we finally do code selection: $(f\_sched)\sigma\downarrow_A$ is the minimal-cost schedule, $(f\_prog)\sigma\downarrow_A$ is the minimal-cost target program t with $m(t) = p$.

There is one thing ad-hoc with this way of selection (but we do not see a chance to avoid it altogether): Determining *min* $I(f\_cost)$ needs to relate different alternatives. Hence the specifier must make explicit reference to choice operators.

We now study a different way to restrict the solution space. Reconsider the setting of Example 1. Machine programs are related to source expressions by the morphism m. While m associates machine instructions with source operators, it does not care for register requirements. The equation system in Example 6.1 defines a predicate *alloc*, which checks if register numbers in a target program are assigned in the stack-like fashion typical for non-pipelined architectures. Thus, the first argument to *alloc* is a target program, the second is a list of registers (i.e. register numbers) available for allocation within the first argument. We borrow Prolog list notation in Example 6.1.

Example 6.1 :

$$
\begin{aligned}
&alloc(add(i,x,y),[i|u]) = alloc(x,[i|u]) \wedge alloc(y,u)\\
&alloc(add(i,x,y),[\,]) = false\\
&alloc(addi(i,x,c),[i|u]) = alloc(x,[i|u])\\
&alloc(addi(i,x,c),[\,]) = false\\
&alloc(load(i,c),[i|u]) = true\\
&alloc(load(i,c),[\,]) = false
\end{aligned}
$$

□

Note that the equations in Example 6.1, when oriented left-to-right, form a canonical rewrite system.

Given a source program p and a list *regs* of register numbers, the problem at hand is to generate well-allocated target programs. We have to solve

[2] In general, this is by no means straightforward. It may still involve minimalisation over an exponential solution space. Shared forests only defer combinatorial explosion, they do not avoid it. Their advantage is that heuristics may be applied after translation to a more suitable data structure.

1. $m(z) = p$

2. $alloc(z, regs) = true$

Given a canonical rewrite system for m, *alloc*, etc, we can solve both equations simultaneously using a narrowing procedure [13]. However, this only works in principle, due to the large solution space.

More efficiently than applying narrowing to (1) and (2), we may first solve (1) by pattern matching according to Section 4, obtaining a shared forest w representing the solution space $I(w)$ (w contains variables $r_1, r_2, \cdots$ for the register numbers yet to be assigned). A narrowing derivation from $alloc(t, regs) = true$, separately for each $t \in I(w)$, will either construct a register assignment (i.e. a ground substitution for $i_1, i_2, \cdots$), if t can be evaluated with the given list *regs* of registers, or else it will fail.

But by combining the selection axioms A with the rules for *alloc*, we obtain a canonical rewrite system again. Hence, the narrowing procedure applies to shared forests w as well as to individual members $t \in I(w)$. The narrowing tree issuing from $alloc(w, regs) = true$ now shares prefixes of paths for different $t \in I(w)$. The calculated substitution of a successful narrowing derivation not only instantiates register variables, but also the choice variables, and hence indicates the selected element from $I(w)$.

Solving $alloc(f\_prog, [1, 2]) = true$ rejects the solutions that load d into a register, and returns calculated substitutions $\sigma_1 = [v_1 \leftarrow l, v_2 \leftarrow r]$ and $\sigma_2 = [v_1 \leftarrow r, v_2 \leftarrow r]$ with

1. $w\sigma_1\downarrow_A = add(1, add(1, ld(1, a), ld(2, b)), addi(2, ld(2, c), d))$, and

2. $w\sigma_2\downarrow_A = add(1, addi(1, ld(1, a), b), addi(2, ld(2, c), d))$.

Note that under both substitutions, $v_2 = r$. Hence we may form $f\_prog2 := f\_prog[v_2 \leftarrow r]\downarrow_A$, thus restricting the solution space to all encodings that do not need more than 2 registers. Now the scheduling and cost phases can just as well be applied to f_prog2, yielding a reduced f_sched2 and f_cost2. Of course, these are identical to $f\_sched[v_2 \leftarrow r]\downarrow_A$ and $f\_cost[v_2 \leftarrow r]\downarrow_A$.

7 Conclusion

7.1 Relation to other work

State-of-the-art techniques for retargetable code generation, based on tree parsing [1], [5], "BURS-theory" [16], or regularly controlled rewriting [4] combine pattern matching and cost analysis. A maximum of efficiency is achieved by encoding cost information into the states of the generated pattern matcher. These approaches provide no formalism to deal with further machine specific aspects of code generation, such as pipeline optimization, register allocation, machine data type coercions, peephole optimization, and maybe others. But these approaches can easily be extended to produce code as terms over some target program signature.

Given this extension, the pure tree parsing approach of [1], [5] is an implementation of our approach when we restrict it to perform cost analysis immediately subsequent to code selection. Since the two phases both work bottom-up,

the y can be interleaved, a
constructed in this case. T
and can detect even more
the intermediate or target
matcher. If certain decisio
to represent the result.

The specific virtue of th
of code generation, in par
specification adheres to th
ferent ways of implementi
indicated with scheduling,
shared forest of target prog
applied independently, and

7.2 Implementatio

The lifting operation on sig
into the compiler-writing s
essential construction is st
plication. Translations by
MARVIN involve a "semar
uated in a particular Σ_0'-al
must be incorporated. Ev
This extension is nontrivia
writer. Although this impl
here have not yet been app

The MARVIN system
on shared forests, mainly b
scribe a rather general clas
the Definition 5.1, howeve
formation systems. The ke
forests are a general progr
functional or logic program

7.3 Future Work

While it is conceptually p
order to select from it cor
problems that so far have

Some machine architec
As temporaries are not rep
derivor equations such as

$$m(tmove\ \ r) = m(r),$$

and hence to chain produc

$$R \rightarrow R.$$

This means that the gra
on the target side, to circul
isters (maybe of different r

the y can be interleaved, and no shared forests of target programs need to be constructed in this case. The approach of [16] and that of [4] are more flexible and can detect even more encodings, as they are able to perform rewriting on the intermediate or target term. Cost considerations are built into the pattern matcher. If certain decisions were to be delayed, shared forests could be used to represent the result.

The specific virtue of the approach presented here is the gain in modularity of code generation, in particular on the specification level. While the overall specification adheres to the structure recommended in [9] for verifiabiliy, different ways of implementing overall code generation may be studied. As was indicated with scheduling, cost analysis and register allocation above, once a shared forest of target programs has been constructed, several subtasks may be applied independently, and even in parallel.

7.2 Implementation Status

The lifting operation on signatures and attribute couplings has been integrated into the compiler-writing system MARVIN [7] by M. Reinold [17]. While the essential construction is straight-forward to implement, there is a severe complication. Translations by attribute couplings according to [6] as well as in MARVIN involve a "semantic" subsignature Σ'_0 of Σ' where in terms are evaluated in a particular Σ'_0-algebra. Evaluation in the corresponding $d\Sigma'_0$-algebra must be incorporated. Evaluation of $1 + \square(2, v, 3)$ to $\square(3, v, 4)$ is an example. This extension is nontrivial, as this Σ'_0-algebra is implemented by the compiler writer. Although this implementation is operational, the techniques described here have not yet been applied to a realistic code generator specification.

The MARVIN system was used for the implementation of transformations on shared forests, mainly because it implements attribute couplings, which describe a rather general class of tree transformations. The construction used in the Definition 5.1, however, may as well be implemented in other tree transformation systems. The key is the proper handling of choice variables. Shared forests are a general programming language technique that is well-suited for functional or logic programming.

7.3 Future Work

While it is conceptually pleasing to represent the complete solution space in order to select from it considerately, the pragmatics of this aproach present problems that so far have only been dealt with in an ad-hoc way:

Some machine architectures require MOVEs between temporary registers. As temporaries are not represented in the intermediate language, they lead to derivor equations such as

$$m(tmove\ \ r) = m(r), \tag{2}$$

and hence to chain productions in the tree grammar like

$$R \rightarrow R. \tag{3}$$

This means that the grammar allows circular derivations, which correspond, on the target side, to circulating an intermediate result through temporary registers (maybe of different register classes). The tree parser can be modified to

cut off such circular derivations, allowing only a finite number of such moves (usually at most 1 or 2). They are actually needed in and sufficient for rare situations like achieving a register pair or inserting sign extensions. But theoretically, they are always possible, and hence blow up the solution space, even in a shared representation. A solution to this is given in [11].

On the conceptual side, there are several open questions. One is the following: Applying narrowing as explained in section 6 yields an explicit enumeration of the remaining solution set. It should be possible to modify the narrowing procedure such that these solutions are again represented as a shared forest.

Furthermore, shared forests may have applications outside code generation as well.

Acknowledgements

Thanks go to H. Hogenkamp for discussing these ideas, to M. Reinold who extended the MARVIN system to shared forests, and to A. Bodzin for preparing the manuscript.

References

[1] Balachandran A, Dhamdhere DM, Biswas S. Efficient Retargetable Code Generation Using Bottom-Up Tree Pattern Matching. Computer Languages, 15(3):127–140, 1990.

[2] Chen H, Hsiang J. Logic Programming with Recurrence Domains. In Proceedings 18th International Colloquium on Automata, Languages and Programming, vol 510 of Lecture Notes in Computer Science (LNCS), pp 20–34. Springer, 1991.

[3] Dershowitz N, Jouannaud JP. Rewrite Systems, vol B of Handbook of Theoretical Computer Science, chapter 15. North Holland, 1990.

[4] Emmelmann H. Code Selection by Regularly Controlled Rewriting. In [10], 1992.

[5] Ferdinand C, Seidl H, Wilhelm R. Tree Automata for Code Selection. In [10], 1992.

[6] Ganzinger H, Giegerich R. Attribute Coupled Grammars. In Proceedings of the International Symposium on Compiler Construction, pp 70–80. Association for Computing Machinery (ACM), 1984. Issue 19(6),1984 of SIGPLAN NOTICES.

[7] Ganzinger H, Giegerich R, Vach M. MARVIN – A Tool for Applicative and Modular Compiler Specifications. Technical Report 220, University Dortmund, 1986.

[8] Giegerich R. Code Selection by Inversion of Order-Sorted Derivors. TCS, 73:177–211, 1990.

[9] Giegerich R. On th
In Proceedings SIG
sign and Implemer
NOTICES.

[10] Giegerich R, Graha
niques. This vol o
1992.

[11] Giegerich R, Hoger
Development. Subr

[12] Giegerich R, Schm
Tree Parsing and I
Symposium on Pro
Science (LNCS), pp

[13] Hullot JM. Canoni
Conference on Auto
Science (LNCS), pp

[14] Knuth DE. Semant
Theory 2, pp 127–1

[15] Lang B. Towards a
Current issues in pa

[16] Pelegri-Llopart E. I
tion. PhD thesis, U

[17] Reinold M. Transfor
Dortmund, 1991. in

[9] Giegerich R. On the Structure of Verifiable Code Generator Specifications. In Proceedings SIGPLAN '90 Conference on Programming Language Design and Implementation, pp 1–8, 1990. Issue 25(6),1990 of SIGPLAN NOTICES.

[10] Giegerich R, Graham SL (eds). Code Generation – Concepts, Tools, Techniques. This vol of Workshops in Computing (WICS). Springer Verlag, 1992.

[11] Giegerich R, Hogenkamp H. Semi-Formal Validation in Code Generator Development. Submitted, 1992.

[12] Giegerich R, Schmal K. Code Selection Techniques: Pattern Matching, Tree Parsing and Inversion of Derivors. In Proceedings of the European Symposium on Programming 1988, vol 300 of Lecture Notes in Computer Science (LNCS), pp 247–268. Springer, 1988.

[13] Hullot JM. Canonical Forms and Unification. In Proceedings of the 5th Conference on Automated Deduction, vol 87 of Lecture Notes in Computer Science (LNCS), pp 318–334. Springer, 1980.

[14] Knuth DE. Semantics of Context-free Languages. Mathematical Systems Theory 2, pp 127–145, 1968.

[15] Lang B. Towards a Uniform Framework for Parsing. In Tomita M (ed), Current issues in parsing technologies. Kluver Academic Press, 1990.

[16] Pelegri-Llopart E. Rewrite Systems, Pattern Matching and Code Generation. PhD thesis, UC Berkeley, 1987. EECS-Report.

[17] Reinold M. Transformations in Shared Forests. Master's thesis, Universität Dortmund, 1991. in German.