Synthesis of Fast Programs for Maximum Segment Sum Problems

Srinivas Nedunuri William R. Cook

University of Texas at Austin

nedunuri@cs.utexas.edu, wcook@cs.utexas.edu

Abstract

It is well-known that a naive algorithm can often be turned into an efficient program by applying appropriate semantics-preserving transformations. This technique has been used to derive programs to solve a variety of maximum-sum programs. One problem with this approach is that each problem variation requires a new set of transformations to be derived. An alternative approach to synthesis combines problem specifications with flexible algorithm theories to derive efficient algorithms. We show how this approach can be implemented in Haskell and applied to solve constraint satisfaction problems. We illustrate this technique by deriving programs for three varieties of maximum-weightsum problem. The derivations of the different programs are similar, and the resulting programs are asymptotically faster in practice than the programs created by transformation.

1. Introduction

The idea of deriving or synthesizing an efficient algorithm from a specification is well-known in the functional programming community. One common technique is to start with a simple but inefficient program, which acts as a specification, and then apply program transformations to improve efficiency. An example of this technique is the derivation of algorithms for variations on the maximum segment sum problem, (SHT00; SHT01; SOH05). A drawback of this approach is that it requires the development of increasingly complicated program transforms for each problem variation; there is little reuse of steps in the derivation even for similar problems. Although the three problems have similar structure, deriving linear algorithms by transformation requires a completely different transformation for each solution.

An alternative approach to program synthesis is based on specifications and generic algorithm theories. This approach to algorithm synthesis is combines a pre/post condition specification of a problem with a high-level algorithm theory that captures generic problem-solving knowledge. The algorithm theory contains abstract operations that must be defined to satisfy the requirements of the particular problem. The synthesis is performed in the context of theory morphisms, ensuring that the resulting algorithm is correct by construction (after generated proof obligations are discharged). Smith and his colleagues have successfully synthesized a number of practical algorithms, including a scheduler that ran several orders of magnitude faster than comparable hand-written ones(SPW95)

In this paper we apply and extend Smith's approach to derive efficient Haskell algorithms for the three segment sum problems mentioned above. The paper has three main contributions:

- An implementation of the Global Search Optimization (GSO) theory in Haskell. The search algorithm is defined over a type class that defines the search space and appropriate pruning and filtering operations.
- Development of a Constraint Satisfaction Optimization theory which specializes GSO, introducing dominance relations (Iba77) into the abstract program.
- Derivation of efficient algorithms for solving the three problems listed above. The algorithms are significantly faster than those derived by Sasano et al. At least two of their algorithms are linear in theory, but did not perform linearly in Haskell. The algorithms derived here are all theoretically and experimentally linear. The same basic derivation steps are used for synthesizing all the algorithms.

2. Global Search Optimization in Haskell

In this section we present a version of the Global Search Optimization (GSO) theory (Smi88) in Haskell. A specification of a global search optimization problem is a 5-tuple $\langle D, R, C, o, p\rangle$, where D is an input type, R an output type, C a cost type, $o : D \times R \rightarrow Boolean$ is an *output* or *correctness condition* that any result must satisfy, and $p : D \times R \rightarrow C$ is a *profit criterion* to be maximized (analogously, minimized). Not all problems have an optimality requirement. For example, the following is a specification of

```
class       (Eq r, Ord c) =>
   GlobalSearchOpt d r c
      | d -> r c where
   precondition :: d -> Bool
   ok :: (d, r) -> Bool
   profit :: (d, r) -> c
   initial :: d -> r
   subspaces :: (d, r) -> [r]
   possible :: (d, r) -> Bool
   tighten :: (d, r) -> r
   dominates :: (d, r, r) -> Bool
   initBound :: d -> c
   upperBound :: (d, r) -> c
   extract :: (d, r) -> Maybe r

   -- default defs
   precondition _ = True
   ok _ = True
   possible (_, _) = True
   tighten (_, r) = r
   dominates (_, _, _) = False
```

Figure 1. GlobalSearchOpt class methods

sorting ($\mapsto$is to be read as "is instantiated as")

$$\begin{array}{lcl} D & \mapsto & [Integer] \\ R & \mapsto & [Integer] \\ o & \mapsto & \lambda(x,z).\ asBag(x) = asBag(z) \\ & & \quad \wedge \forall i,j.\, 0 \le i,j < \#x.\, i \le j \Rightarrow z_i \le z_j \end{array}$$

Global Search Optimization theory consists of two parts: a general purpose search algorithm and an abstract data type defining a search space. The search space includes operations for generating new search nodes, testing for success, and more advanced operators to prune and guide the search. Algorithm 1 defines the generic search optimization algorithm as a Haskell function, `findOpt`, contained in a type class (See Fig. 1); the non-default class methods are the functions the developer must instantiate. In this paper, we show how to calculate the required instantiations.

The function `findOpt` takes an input $x : D$ and returns a (possibly empty) list of solutions satisfying the output condition and optimizing the profit criterion. The search optimization function is classic branch-and-bound search, augmented with operators to help prune the search. It works by taking an initial *space* (also called a *partial solution*) of possible solutions (corresponding to the root node of a search tree), and if that space passes the propagation and bounds test, initializes a list of active spaces with that space and begins the search. The main search function selects an active space (s) from the list it is given. If it can, it immediately extract a feasible solution (the space corresponds to a leaf node) and, if it is optimal, adds it to the collection of optimum solutions and returns. Otherwise, the revised collection of active spaces is formed by first partitioning the current active space into sub-

Algorithm 1 An abstract program for GSO

```
-- returns 0 or more optimal  solutions
findOpt :: d -> [r]
findOpt x =
  if not(precondition x) then error "i/p cond" else
  case propagate x (initial x) of
    Just s -> if upperBound(x, s) >= initBound x
              then globalSearchOpt x [s] [] 0
              else []
    Nothing -> []

globalSearchOpt :: d -> [s] -> [r] -> Integer -> [r]
globalSearchOpt x [] solutions = solutions
globalSearchOpt x (s : rest) solutions =
  let solutions' = case extract(x, s) of
        Just z -> if ok(x, z)
                  then optima x z solutions
                  else solutions
        Nothing -> solutions
      active' = rest_undom ++ ok_subspaces_undom
        where dom s s' = dominates(x, s, s')
              ok_subspaces =
                ok_subspaces_of
                   x (listToMaybe solutions) s
              undom_siblings =
                rmv_dominated_siblings
                   dom ok_subspaces
              (ok_subspaces_undom,rest_undom)=
                 rmv_dominated
                   dom undom_siblings rest
  in globalSearchOpt x active' solutions'

ok_subspaces_of :: d -> Maybe r -> s -> [s]
ok_subspaces_of x zs s =
    [ s' | s' <- mapMaybe (propagate x)
                   (subspaces(x, s)),
          zs /= Nothing ==>
              upperBound(x, s') >=
                 profit(x, fromJust zs)]

propagate :: d -> s -> Maybe s
propagate x s =
    if possible(x, s)
    then Just (iterate2FP tighten x s)
    else Nothing

iterate2FP :: ((d, s) -> s) -> d -> s -> s
iterate2FP f x z =
    let fz = f(x, z) in
    if fz == z then fz else iterate2FP f x fz

optima :: d -> r -> [r] -> [r]
optima x z [] = [z]
optima x z (z':zs) =
    if profit(x, z) > profit(x, z') then [z]
    else if profit(x, z) == profit(x, z')
         then (z:z':zs) else (z':zs)
```

Algorithm 2 Dominance testing

```
rmv_dominated_siblings f siblings =
    foldr (chkSibling4Dominance f) [] siblings

rmv_dominated f [] active = ([], active)
rmv_dominated f (s:siblings) active =
    let undom = chkSibling4Dominance2
                f s active
        active' = snd undom
        (ok_sibs, ok_active) =
          rmv_dominated f siblings active' in
    if not $ fst undom
    then (s:ok_sibs, ok_active)
    else (ok_sibs, ok_active)

chkSibling4Dominance f s ss =
    let undom_ss = filter (not.(f s)) ss in
    if and (map (not.(flip f s)) undom_ss)
    then (s:undom_ss)
    else undom_ss
chkSibling4Dominance2 f s ss =
    let undom_ss = filter (not.(f s)) ss in
    if and (map (not.(flip f s)) ss)
    then (False, undom_ss)
    else (True, ss)
```

$$\begin{array}{rcl} initial & :: & D \to R \\ extract & :: & D \times R \to R \\ subspaces & :: & D \times R \to [R] \\ upperBound & :: & D \times R \to C \\ initBound & :: & D \to C \\ possible & :: & D \times R \to Boolean \\ tighten & :: & D \times R \to R \\ dominates & :: & D \times R \times R \to Boolean \end{array}$$

Figure 2. Search space interface

spaces (corresponding to child nodes), selecting only those (`ok_subspaces`) that aren't emptied out by propagation and also pass the bounds test. Then a dominance test (see Algorithm 2, see also Section 4.1) is performed between every pair of subspaces (`rmv_dominated_siblings`) (those spaces that are dominated can be eliminated) followed by a dominance test between every active space and every subspace (`rmv_dominated`). Only those that survive these tests (`rest_undominated` and `ok_subspaces_undominated`) are retained and passed to the next level of search.

The interface of operations that define the search space is given in Figure 2. The `GlobalSearchOpt` class combines this interface with the specification $\langle D, R, C, o, p\rangle$, in which D, R, C become type variables `d,r,c` resp. and o, p are renamed `ok,profit` resp. The search-space operations are uncurried to distinguish them from other operations.

$initial$ returns a descriptor of the initial search space. $extract$ determines whether the given space is terminal and if so, returns a solution (otherwise Nothing, denoting an empty space). $subspaces$ returns a set of subspaces of the current space. $dominates$ is a predicate that determines whether one space dominates another. The dominated space can be pruned. *upperBound* is a function that returns an upper bound on the best solution possible in the given space. *initBound* is an initial bound. *possible* is a necessary filter - those spaces that do not pass *possible* need not be examined. Ideally, we want only those spaces r that contain feasible solutions, i.e. satisfy $\exists z.\ r \sqsubseteq z \wedge o(x, z)$ but finding exactly those spaces often is not feasible so we settle for a weaker test, namely some predicate *possible* satisfying $\forall z.\ r \sqsubseteq z \wedge o(x, z) \Rightarrow possible(r)$. $\sqsubseteq$ is a refinement relation over R. The intent of $\sqsubseteq$ is that if $r \sqsubseteq s$ then s is is a subspace of r (any solution contained in s is contained in r) and is "more defined" than r . *tighten* is called a *necessary propagator*. It "tightens" a given space to eliminate infeasible solutions and can be any function (of the appropriate type) satisfying $\forall z. r \sqsubseteq z \wedge O(x, z) \Rightarrow tighten(r) \sqsubseteq z$. When $\langle R, \sqsubseteq\rangle$ forms a lattice, a monotone inflationary *tighten* can be iterated from any starting space to a fixpoint which is the tightest possible space that still preserves all the original feasible solutions, (SPW95). The *propagate* function in the abstract program above does that, by comparing the space before applying *tighten* with the result after. An axiomatic definition of GSO theory and proof of correctness of the abstract program without dominance relations can be found in (Smi88). The proof when dominance relations are included is analogous.

3. A Theory of Constraint Satisfaction Optimization

Many of the problems we have looked at can be solved by constraint satisfaction (Dec03). In this paper we define a special subclass of GSO for constraint satisfaction optimization problems, which we call CSO theory. We then show how fast solutions for a number of variations of the MSS problem can be systematically derived within this theory. Constraint satisfaction operates as follows: Given a set of variables, $vars$, assign a value, drawn from some domain D_v, to each variable, in a manner that satisfies some set of constraints. Fig. 3 shows an instantiation of `GlobalSearchOpt` which carries this out. The intent is that specific CSO problems will monotonically extend this theory. We will just copy and paste the necessary definitions into problem specific code.

The initial space $\widehat{z}_o$ makes all the values in `vals x` available to every variable. The `subspaces` function picks a variable from the set of variables not yet assigned a value (`tbd`) and returns the subspaces formed by assigning to `v` each of the possible values (drawn from `ch v`), adding each pair to the map `m`, and removing `v` from `tbd` and `a` from `ch v`. The choice of which variable to pick does not matter func-

```
data D = D {maxVar::Nat, vals::[Dv]}
data R = R {m::Map Nat Dv, tbd::[Nat]}
ok_CST(x, z) = map_keys(m z)==[0..(maxVar x)]
initial(x) = R{m=empty, tbd=[0..(maxVar x)]}
subspaces(x, r) =
  if (tbd r)==[] then [] else
     let var=chooseV(tbd r)
         mS x s var a = mkSubspace(x,s,var,a)
     in map (mS x r var) (cs r)
mkSubspace_CST s var val =
    s {m=map_insert var val (m s),
       tbd=delete var (tbd s)}
chooseVar [] = Nothing
chooseVar (v:vs) = Just v
extract(x, r) =
  if (tbd r)==[] then Just r else Nothing
upperBound(x, z_hat) = maxBound
⊑(z, z') = z.m ⊆ z'.m
```

Figure 3. CSOT: A partial instantiation of GSO theory for solving CSO problems

tionally, but can have a significant impact on the efficiency of the program. The expressions `m_lookup i (m z_hat)` and $\#\widehat{z} - 1$ are typically abbreviated as $\widehat{z}_i$ and `m_findMax (m z_hat)` resp. in derivations. The functions `m_findMax`, `m_lookup` and `m_insert` are library functions, defined in Fig. 5.

In order to get a working constraint satisfaction solver, the developer (the user of this theory) needs to instantiate C, c, D_v and whatever additional output conditions are appropriate. We give an example of this next.

4. Maximum Independent Segment Sum (MISS)

The maximum segment sum problem has been popular in the FP community after Bird (Bir89) showed how to calculate a linear-time algorithm. As (Mu08) reports, the subject is still going strong, and attracting attention, partly because it has important applications in filter design and bioinformatics, and the 2-D version has applications in image processing. So, the synthesis of fast solutions to the problem and its variants is also important. MISS is a variant of MSS in which the goal is to select some elements from a given array, with the restriction that no two adjacent elements can be selected, such that the sum of the elements in the selection is maximized. The specification of the problem is in Fig. 4. Because we are extending CSOT, only the elements that differ from CSOT are shown. A complete theory is obtained by combining this partial specification with CSOT.

`nonAdj` ensures that no two adjacent elements of the input (`x`) are present in the final solution (`z`) and `c'` calculates the profit of the final solution counting down from the highest allocated variable. These bindings instantiate the abstract program in Fig 1 into a working (albeit inefficient) solver for

```
data Dv = Bool
data D = D {maxVar::Nat, vals::[Dv], data::[Integer]}
data C = Integer
ok(x, z) = ok_CST(x,z) && nonAdj z (maxVar x)
profit(x, z) = c' (data x) (m z) ((map_findMax.m) z)
nonAdj z 0 = True
nonAdj z (i+1) =
    (selected z (i+1) ==> not (selected z i))
    && nonAdj z i
c' x z 0 = if selected z 0 then head x else 0
c' x z (i+1) =
    let contrib =
    if selected z (i+1) then x!!(i+1) else 0
    in contrib + c' x z i
```

Figure 4. A specification of the MISS problem

the MISS problem. The key to making it efficient are good definitions for the operators *dominates*, *possible*, *tighten* and *upperBound*. Often, further optimizations such as context-dependent simplification, finite differencing, and data structure selection have to be carried out before arriving at a final efficient algorithm. With the exception of finite differencing, these latter operations are not the focus of this paper. For details, please refer to one of the many detailed descriptions of the approach and other applications (e.g. (Smi90)). Each operator has the effect of drastically reducing the search space, until the combined effect of all the operators taken together is a highly efficient functional program. In the rest of this section, we will show how a developer can come up with such definitions, by way of a running example.

4.1 Dominance Relations

A dominance relation provides a way of comparing two spaces in order to show that one will always have a cheaper best solution than the other. The first one is said to *dominate* the second, and the second can be eliminated from the search. Dominance relations have been used in algorithm development in operations research for a long time and researchers from Baker [?] to Allahverdi et al. (ANCK08) report that careful use of dominance relations can considerably reduce the search space. Because dominance in its most general form is difficult to demonstrate, we restrict ourselves to demonstrating dominance between semi-congruent spaces (explained below). To simplify the presentation, we assume all spaces are reachable from the initial space $\widehat{z}_0$ by a finite sequence of calls to mkSubspace. Let $p^*(\widehat{z})$ be the profit of the best solution in a space $\widehat{z}$. Finally $\oplus$ denotes adding a pair to a map and is defined as $m \oplus (x, a) \triangleq m - \{(x, a')\} \cup \{(x, a)\}$. We also overload $\oplus$ in $\widehat{z} \oplus e$ to be a solution z for which z.f=$\widehat{z}.f \cup e.f$ (unless otherwise stated, we will always be assuming $dom(\widehat{z}.f) \cap dom(e.f) = \emptyset$). e is called an *extension* and z the *completion* of $\widehat{z}$. If z is feasible (satisfies `ok`), then e is called a feasible extension.

Definition 1. *Semi-Congruence* is a relation $\rightsquigarrow \subseteq R^2$ such that

$$\forall z, z', e \in R, \widehat{z}, \widehat{z}' \in \widehat{R}: \ \widehat{z} \rightsquigarrow \widehat{z}' \Rightarrow O(\widehat{z}' \oplus e) \Rightarrow O(\widehat{z} \oplus e)$$

That is, semi-congruence ensures that any feasible extension of $\widehat{z}'$ is also a feasible extension of $\widehat{z}$. In Haskell, $\rightsquigarrow$ is written `semi-congruent`.

Definition 2. Given a profit function p, *Weak Dominance* is a relation $\widehat{\delta} \subseteq R^2$ such that

$$\forall z, z', e \in R, \widehat{z}, \widehat{z}' \in \widehat{R}: \\ \widehat{z}\delta\widehat{z}' \ \Rightarrow O(\widehat{z} \oplus e) \wedge O(\widehat{z}' \oplus e) \Rightarrow p(\widehat{z} \oplus e) \geq p(\widehat{z}' \oplus e)$$

That is, weak dominance ensures that one feasible completion of a partial solution is at least as beneficial as the same feasible completion of another partial solution. The following theorem and proposition show how to combine the two concepts.

Theorem 3. *If $\rightsquigarrow$ is a semi-congruence relation, and $\widehat{\delta}$ is a weak dominance relation, then*

$$\forall \widehat{z}, \widehat{z'} \in R: \ \widehat{z}\widehat{\delta}\widehat{z}' \wedge \widehat{z} \rightsquigarrow \widehat{z}' \Rightarrow p^*(\widehat{z}) \geq p^*(\widehat{z}')$$

Proof. By contradiction. Suppose that $\widehat{z}\widehat{\delta}\widehat{z}' \ \wedge \ \widehat{z}' \rightsquigarrow \widehat{z}$ but $\exists z'^* \in \widehat{z}', O(z'^*) \wedge p(z'^*) > p^*(\widehat{z})$, that is $p(z'^*) > p(z)$ for any feasible $z \in \widehat{z}$. We can write z'^* as $\widehat{z}' + e$ for some e. Since z'^* is more profitable than any feasible $z \in \widehat{z}$, specifically it is more profitable than $z = \widehat{z} + e$, which by the semi-congruence assumption and Definition 1, is feasible. But by the weak dominance assumption, and Definition 2, this means $p(z) \geq p(z'^*)$, contradicting the initial assumption. □

When $p^*(\widehat{z}) \geq p^*(\widehat{z}')$ we say $\widehat{z}$ *dominates* $\widehat{z}'$, written $\widehat{z}\delta\widehat{z}'$.

We can also prove a useful property that applies to all instances of CSOT. Note that in the following proposition we assume we can apply the profit function to partial solutions.

Proposition 4. *If p distributes over $\oplus$ then $p(\widehat{z}) \geq p(\widehat{z}')$ is a weak dominance relation*

Proof. We will show that Definition 2 is satisfied

$$\begin{aligned}
& p(\widehat{z} \oplus e) \ \geq p(\widehat{z}' \oplus e) \\
& = \{\text{p distributes over } \oplus\} \\
& p(\widehat{z}) + p(e) \geq p(\widehat{z}') + p(e) \\
& = \{\text{algebra}\} \\
& p(\widehat{z}) \geq p(\widehat{z}')
\end{aligned}$$

□

We now show how to calculate a very useful dominance relation for MISS. First we calculate the required semi-congruence condition $\rightsquigarrow$ between $\widehat{z}$ and $\widehat{z}'$ by using the method of derived preconditions. Starting with the conclusion of Def. 1, and working backwards

$$\begin{aligned}
& O(\widehat{z} \oplus e) \\
& = \{\text{unfold defn, let } L = \#\widehat{z} - 1, L' = \#\widehat{z}' - 1\} \\
& m\_keys\,(m\,\widehat{z}) + m\_keys\,(m\,e) = [0..(maxVar\,x)] \\
& \wedge nonAdj(\widehat{z}) \wedge nonAdj(e) \wedge (\widehat{z}_L \Rightarrow \neg e_0) \\
& \Leftarrow \{m\_keys\,(m\,\widehat{z}') + m\_keys(m\,e) = [0..(maxVar\,x)] \\
& \quad \wedge nonAdj(\widehat{z}') \wedge nonAdj(e) \wedge (\widehat{z}'_L \Rightarrow \neg e_0)\text{ie. } O(\widehat{z}' \oplus e)\} \\
& L = L' \ \wedge \ nonAdj(\widehat{z}) \ \wedge \ (\neg\widehat{z}'_L \Rightarrow \neg\widehat{z}_L) \\
& = \{\text{simplification}\} \\
& L = L' \ \wedge \ nonAdj(\widehat{z}) \wedge (\widehat{z}_L \Rightarrow \widehat{z}'_L)
\end{aligned}$$

That is, in Haskell:

```
semiCongruent(z_hat, z_hat') =
  let highest_var = (map_findMax.m) z_hat in
      highest_var == (map_findMax.m) z_hat'
      && nonAdj z_hat
      && (selected z_hat highest_var
         ==> selected z_hat' highest_var)
```

Since p is a distributive profit function, by Proposition 4 the definition for dominates follows immediately. Dominance has the effect of reducing the complexity from exponential to polynomial (see Theorem 5). However, the evaluation of nonAdj still makes the algorithm nonlinear (quadratic). The next section shows how propagation eliminates this expensive computation.

4.2 Necessary Propagator (*tighten*)

The *tighten* is calculated in the same way $\rightsquigarrow$ was calculated. However, not all calculations are as obvious. Sometimes the calculation requires a key insight, and deciding where to start can be a bit challenging. To ease this process, we have been investigating the use of tactics that not only package up the insight in the form of a pattern but also bypass the actual calculation through the use of pattern matching rules (analogous in intent to pattern matching rules such as “integration by parts", “integration by change of variable", etc. used in calculus for evaluating indefinite integrals). We have described some of these tactics in (NSC09), where the following tactic was introduced:

If one of the conjuncts of O matches the form $\forall j \in N_i.\, z_i \neq z_j$ where N_i is some neighborhood of points around i then a possible tighten *is* one in which the choice of values available to variable j excludes the value assigned to variable i.

For our problem, let N_i be the left and right neighbors of i (for $0 < i < \#z - 1$), i.e. $i - 1$ and $i + 1$, if z_i and $\{\}$ otherwise. Then in the case where z_i holds, applying the above tactic, $tighten(\widehat{z}) = \widehat{z}\{cs(i+1) = cs(i+1) - \{True\}\}$ which is just $\widehat{z}\{cs(i+1) = \{False\}\}$. Since the choice set reduces to a singleton, we can just set the value directly for the $i+1^{th}$ place, and dispense with `cs`, as shown in the definition for `tighten` in Algorithm 3.

This propagator ensures that every partial solution automatically satisfies `nonAdj` so it need no longer be checked in `semi-congruent` or in `ok`.

4.3 Necessary Filter (*possible*)

When the propagator *tighten* is very efficient it can eliminate the need for a filter . That appears to be the case for this example so we use the default definition we inherit from the class `GlobalSearchOpt`.

4.4 Upper Bound Function (*upperBound*)

An upper bound is a value associated with a partial solution that puts an upper limit on the value of the best possible solution that can be obtained from that partial solution. Bounds calculation is an integral part of branch-and-bound algorithms. A good tactic for determining a bound is similar to Tactic 1 for deriving a *possible* in (NSC09): obtained by combining the profit of the current partial solution with the best possible values for the remaining variables. Applying such a tactic gives us the following upper bound function:

$$upperBound(x, \hat{z}) = p(x, \hat{z}) + \sum_{i=\#\hat{z}}^{\#x.data} max(x.data(i), 0)$$

written in Haskell as

```
profit(x, z_hat)
+ c' (drop (((map_findMax.m) z_hat)+1) (data x))
    (max 0) (maxVar x)
```

All operators combined

The table below shows the cumulative effect of the operators on the size of the search space for the input `[1..10]`. The "Operator Added" column refers to the introduction of a non-default definition for the corresponding operator.

Operator Added	# of nodes in search tree
None	2047
+ *dominates*	486
+ *tighten*	12
+ *upperBound*	12

As the table shows, dominance and propagation are significant in eliminating large swathes of the search space.

4.5 Finite Differencing

Finite differencing is a program optimization technique due to Paige and Koenig (PK82). Smith (Smi90) contains an extensive discussion of finite differencing and its use in KIDS. The basic idea behind finite differencing is as follows. Suppose a program fragment contains the following recursive definition :

```
f(x) = P(E(x), f(U(x)))
```

where `E(x)` is some expression dependent on `x`, and *U* is some update of `x` (e.g. `x+k`). Finite differencing replaces the definition with the following one

```
getValueOfVar z_hat v =
  fromJust (map_lookup v (m z_hat))

type Map a b = [(a, b)]
map_empty = []
map_null xs = ([] == xs)
map_lookup = lookup
map_insert x v m = ((x, v):m)
map_keys = sort.(map fst)
map_findMax = fst.head

infix 1 ==>
(==>) :: Bool -> Bool -> Bool
x ==> y = (not x) || y
```

Figure 5. Basic library functions

```
f(x, Ex) = P(Ex, f(U(x), E'(Ex, U(x))))
```

where *E'*`(Ex,U(x))` is a direct update of the expression *E* of U`(x)` which is cheaper to compute than *E*`(`U`(x))` Typically if `x` is of some product type *T*, the update expression *E'*(x) is stored as an extra field of an augmented type, *T'* $= T \times T_E$ so `f` does not take any additional arguments. Of course, all existing code that uses *T* will need to be updated to use *T'*. Given an isomorphism from *T* to *T'* (and in inverse from *T'* to *T*), the Specware tool (S) from Kestrel can automatically carry out the necessary program modifications. Since we are using Haskell, we have implemented the finite differencing transformations by hand. As an example, consider the calculation of the profit function `profit`. The z_hat argument to the profit function changes whenever there is an update (by `updateSpace)`. So we extend the type `R` to accomodate an extra field, accum: `data R = R {m::Map Nat Dv, tbd::[Nat], accum:C}`. Next we change `updateSpace(x, z_hat, var, val)` from `z_hat{m = ..., tbd = ...}` to `z_hat{m = ..., tbd= ..., accum = (accum z_hat) + (if val then (head (data z_hat)) else 0)}`. Finally, the calculation of `c` is replaced by `c(x, z_hat) = accum z_hat`. The result of carrying out this and other finite differencing optimizations is certain tests (like `ok`) reduce to `True` and can be replaced with their defaults. The final set of definitions shown in Algorithm 3.

The definitions rely on a set of library functions defined in Fig. 5.

We refer to the combination of the abstract program for GSO in Algorithm 1 with the above bindings in Algorithm 3 as Algorithm MISS.

Lemma 5. *Algorithm MISS gives rise to a search tree of width at most 2*

Proof. By induction on the height of the search tree. Since the map `m` is numerically indexed in order, we will represent the state of a partial solution as a list e.g. $[FTF]$ denotes a

Algorithm 3 Instantiation of typeclass methods to solve MISS

```
data D = D{maxVar::Nat,vals::[Dv],data::[Integer]}
type Dv = Bool
data R = R {m::Map Nat Dv, tbd::[Nat],
            accum::C,sum_tbds::C,remain::[Integer]}
         deriving Show
type C = Integer

selected = getValueOfVar
cs _ = error "cs not defined"

instance ConstraintSat D R Integer Dv where
mkSubspace(x, s, var, val) =
    --NB: Must always call the parent fn
    let s' = mkSubspace_CST s var val
        varth_elt = head (remain s)
    in s'{accum =
            accum s +
            (if val then varth_elt else 0),
          sum_tbds =
              (sum_tbds s) -
                 if varth_elt > 0
                 then varth_elt else 0,
          remain = tail (remain s)}
semiCongruent(x, s, s') =
    let highest_var = (map_findMax.m) s in
    highest_var == (map_findMax.m) s' &&
    selected s highest_var ==>
        selected s' highest_var)

instance Eq R where
s' == s =
    (map_null (m s') && map_null (m s'))
    ||
    let highest_var = map_findMax (m s)
    in  highest_var == map_findMax (m s')

instance GlobalSearchOpt D R Integer where
initial(x) = R{m=map_empty, tbd=[0..(maxVar x)],
               accum=0, remain=data x,
               sum_tbds=sum $ (filter (0<))(data x)}
precondition x = length (data x) == (maxVar x)+1
tighten (x, s) =
    if (map_null (m s)) then s
    else
        let highest_var = map_findMax (m s)
            next_element = head (remain s) in
        if highest_var < (maxVar x) &&
           selected s highest_var
        then
            mkSubspace(x,s,(highest_var+1),False)
        else s
profit(x, z) = accum z
upperBound(x, s) = profit(x, s) + (sum_tbds s)
initBound _ = 0
dominates(x, s, s') =
    semiCongruent(x, s, s') &&
    profit(x, s) >= profit(x, s')
```

partial solution in which the variable 0 has the value False, variable 1 is True, and variable 2 is False.

Base case: A tree of height 1. There are at most 2 possible leaves: [F] and [T].

Inductive case: Assume the theorem holds for trees of height h. That is there are at most 2 frontier nodes at level h. Each such node gives rise to at most 2 children. Therefore, after a call to subspaces, there are 4 possible configurations: [...FF], [..FT], [...TF], and [...TT]. [...TT] will never be generated because of *tighten*. [...FF] and [...TF] are congruent ($[...FF] \rightsquigarrow [...TF] \wedge [...TF] \rightsquigarrow [...FF]$), and since our profit function produces a total order, one must dominate the other. Therefore we are left with at most 2 children again. □

Theorem 6. *Algorithm MISS runs in linear time*

Proof. The height of the search tree is at most n, the number of elements in the input list. At each level, by Lemma 5 the dominance testing examines at most 3 pairs. All user-defined functions are constant time. Therefore, the running time of the program is $O(n)$. □

The results of comparing the run-time (in seconds) of our program with that generated by Sasano et al. on sequences of randomly generated numbers of varying lengths is shown in the table below. All times were obtained by compiling under GHC 6.10.1 with full optimization and run on an Intel Dual Core 1.66MHz machine.

Input Length	Ours (s)	Sasano et al. (s)
1000	0.00	0.00
10,000	0.12	0.14
20,000	0.22	0.28
40,000	0.43	0.72
80,000	0.75	1.8
100,000	1.1	2.8
200,000	2.2	8.9
400,000	4.6	stack overflow

As can be seen our program outperforms theirs by a factor of two on inputs with length over 20,000. We attribute this difference to our propagation step. Furthermore, it is not obvious how to incorporate such an improvement into their program transformation. We are not certain why their program execution time suddenly spikes on large inputs.

5. Maximum Multi Marking Problem (MMM)

While a constant factor improvement of our synthesized code over transformation produced code is nice, we believe the real benefit of our approach is the flexibility it provides over program transformation. For example, the program transformation used in (SHT00) has some shortcomings. It can only handle problems in which the value set is binary, and the property p (equivalent to our `ok`) cannot include accumulating parameters. An example of where this requirement does not hold is a variation of MSS called the Maximum Multi Marking problem which is similar to the MISS problem except that instead of an element just being included (+) or excluded (0) from the result list, it can also be negated (-). The problem now is to find a result sublist of elements in which no two adjacent elements in the *result* have the same sign (+/-/0). To handle the MMM problem, in (SHT01), Sasano et al, introduce a much more complicated program transform. In contrast, we need change nothing in the theory or the supporting tools. The developer just follows the same steps outlined earlier. In fact, the revised definitions can be obtained by small modifications to what was already done for MISS, as shown below. First, the problem specification is now

```
data Dv = Pos | Neg | Zero
data D = D {maxVar::Nat, vals::[Dv], data::[Integer]}
data C = Integer
ok(x, z) = o_CSOT(x, z) && nonAdj z (maxVar x)
profit(x, z) = c' (data x) (m z) ((map_findMax.m) z)
nonAdj z 0 = True
nonAdj z (i+1) =
    let ith_val = getValueOfVar z i
        i_plus_1th_val = getValueOfVar z (i+1)
    in i_plus_1th_val /= ith_val && nonAdj z i
c' x z 0 = wt (x!!0)
c' x z (i+1) = c' x z i + wt (x!!(i+1))
wt s x = case s of Pos -> x | Neg -> -x | Zero -> 0
```

Next, the revised semi-congruence condition $\rightsquigarrow$ between $\widehat{z}$ and $\widehat{z}'$ calculated in a similar manner to that for MISS, is :

$$L == L' \wedge nonAdj(\widehat{z}) \wedge (\widehat{z}_L = \widehat{z}'_L)$$

There does not appear to be any interesting tightener so this time we just use the default from the typeclass.

Finally, we change `upperBound` to reflect the fact that in the best possible case a positive number will be un-negated, and a negative number negated. Calculation yields:

$$upperBound(x, \widehat{z}) = c(x, \widehat{z}) + \sum_{i=\#\widehat{z}}^{\#x.data} abs(x.data(i))$$

The final program instantiations, after finite differencing, are shown in Algorithm 4. To make it easier to see what the changes are we only show the functions which are different from those in the MISS solution.

It can be shown that this program is linear-time by using a very similar idea to that used earlier (the width of the tree will be slightly different but still constant) The results of comparing the run-time (in seconds) of our program with that generated by (SHT01) on sequences of randomly generated numbers of varying lengths is shown in the following table.

Algorithm 4 Instantiation of typeclass methods to solve MMM

```
data R = R {m::Map Nat Dv, tbd::[Nat], cs::[Dv],
           accum::C,sum_tbds::C,remain::[Integer]}
         deriving Show

wt :: Dv -> Integer -> Integer
wt Neg varth_elt = -varth_elt
wt Zero _ = 0
wt Pos varth_elt = varth_elt

instance ConstraintSat D R Integer Dv where
mkSubspace(x, s, var, val) =
    --NB: Must always call the parent fn
    let s' = mkSubspace_CST s var val
        varth_elt = head (remain s)
        highest_numbered_var_in_s = var-1
    in s'{cs = delete val (vals x),
          accum = (accum s)+ wt val varth_elt,
          sum_tbds = (sum_tbds s) - abs varth_elt
          remain = tail (remain s)}
semiCongruent(x, s, s') =
    let highest_var = (map_findMax.m) s in
    highest_var == (map_findMax.m) s' &&
    getValueOfVar s highest_var ==
        getValueOfVar s' highest_var)

instance GlobalSearchOpt D R Integer where
initial(x) = R{m=map_empty, tbd=[0..(maxVar x)],
              cs=(vals x), accum=0, remain=data x,
              sum_tbds=sum $ (filter (0<))(data x)}
```

Input Length	Ours	Sasano et al.
1000	0.02	0.02
10,000	0.14	0.36
20,000	0.42	0.89
40,000	0.86	2.91
80,000	1.66	9.67

Sasano's program appears to exhibit non-linear behaviour, but we are not certain why this is. In contrast, our synthesized program maintains its linear behavior.

6. Maximum Alternating Segment Sum Problem (MASS)

Even the more complex program transformation introduced in (SHT01) still falls short – it requires the weight function be in homomorphic form. An example of where this does not hold occurs in a version of the MSS Sasano et al. call the Maximum Alternating Segment Sum problem (MASS) which is identical to the MSS except that the profit of a solution is evaluated by alternately negating elements, and summing. To handle this kind of problem, in (SOH05), Sasano et al. introduced a revised transform and associated theory. On the other hand, we again need change nothing, except for the developer to revise their previous derivations. The problem specification is obviously different:

```
data Dv = Bool
data D = D {maxVar::Nat, vals::[Dv], data::[Integer]}
data C = Integer

ok(x, z) = o_CSOT(x, z) && contig z (maxVar x)

contig z 0 = True
contig z (i+1) =
  (selected z (i+1) && (not (selected z i)) && i>0
    ==> rest_unsel z (i-1))
  && contig z i
rest_unsel z 0 = not (selected z 0)
rest_unsel z (i+1) = not (selected z (i+1))
                   && rest_unsel z i

profit(x, z) =
  c' (select (FirstTrue (m z)) (LastTrue (m z))
    (data x)) False
c' [] _ = 0
c' (x:xs) False = x + c' xs True
c' (x:xs) True = -x + c' xs False
```

Next, the revised semi-congruence condition $\rightsquigarrow$ between $\widehat{z}$ and $\widehat{z}'$ calculated in a similar manner to that for MISS, is (again letting $L = \#\widehat{z} - 1$) :

$$L == L' \wedge contig(m(\widehat{z}))\, L \wedge \widehat{z}_L \wedge \widehat{z}'_L$$

The profit function, c, is no longer distributive over all $\widehat{z}$ and e , but we can calculate the necessary conditions under which it does distribute, namely $even(\#\widehat{z}) \Leftrightarrow even(\#\widehat{z}')$ which is implemented as

`flip_last_elt z_hat == flip_last_elt z_hat'` (seeAlgorithm 5). There does not appear to be any useful propagator so again we just use the default we inherit from the typeclass.

Finally, `upperBound` is as it was for MMM.

The final program instantiations, after finite differencing of `contig` and `c`, are shown in Algorithm 5 . Due to space limitations we only show the functions which are different from those in the MISS solution. As before, it is possible to show this algorithm is also linear-time using the same technique was used for MISS

The results of comparing our synthesized program with that generated by (SOH05) on a number of inputs of varying length is shown in the following table.

Input Length	Ours	Sasano et al.
1,000	0.02	0.02
10,000	0.33	0.45
20,000	0.7	1.1
40,000	1.4	3.0
80,000	3.0	9.8
160,000	4.5	24

Algorithm 5 Abstract program instantiations for MASS

```
data R = R {
  m :: Map Nat Dv,
  tbd :: [Nat],
  seg_started :: Bool,
  seg_ended :: Bool,
  flip_last_elt :: Bool,
  accum :: C,
  sum_tbds :: C,
  remain :: [Integer]
  } deriving Show

wt False _ _ = 0
wt True True e = -e
wt True False e = e

toFlipOrNotToFlip s =
  (seg_started s) && not (flip_last_elt s)

instance ConstraintSat D R Integer Dv where
mkSubspace(x, s, var, val) =
  let s' = mkSubspace_CSOT s var val
      varth_elt = head (remain s)
  in s'
    {
    accum = (accum s)
      + wt val (toFlipOrNotToFlip s) varth_elt,
    seg_started = seg_started s || val,
    seg_ended = seg_ended s
      || (not val && seg_started s)
    tbd = if seg_ended s
            || (not val && seg_started s)
          then [] else (tbd s')
    flip_last_elt = toFlipOrNotToFlip s,
    sum_tbds = sum_tbds s - abs varth_elt,
    remain = tail (remain s)
    }

semiCongruent(s, s') =
  (let highest_numv = (map_findMax.m) s in
    highest_numv == (map_findMax.m) s'
    && getValueOf s highest_numv
    && getValueOf s' highest_numv
    && flip_last_elt s == flip_last_elt s')
  || seg_ended s'

instance GlobalSearchOpt D R Integer where
initial(x) =
  R {
    m = map_empty,
    tbd = [0..(maxVar x)],
    seg_started = False,
    flip_last_elt = False,
    seg_ended = False,
    accum = 0,
    sum_tbds = sum $ (map abs) (data x),
    remain = data x
    }
```

Again, Sasano et al.'s program appears to exhibit non-linear behaviour but our synthesized program maintains its linear behavior.

7. Related Work

Sasano et al. in (SHT00) have a comparison with related work of Bird and de Moor (BM97), Jeuring (JP93), and others. Many of the same arguments carry over to our work so we will not repeat them here, except to mention that our work, like that of Bird and de Moor involves program calculation at design time, as opposed to a meta-level calculation of the program transformation itself. The difference is that we do not require the developer to calculate the entire program but only very specific operators. King and Launchbury (KL95) describe an elegant way of constructing depth-first graph search algorithms in Haskell. Our approach does not rely on the laziness of Haskell or on depth-first but exposes the relevant properties at the top level so they can be examined and alterered if necessary. Smith investigated a necessary form of dominance in (Smi88) and also (Smi87) synthesized a efficient 1-D and 2-D versions of MSS using a an algorithm class called Divide and Conquer. We have found the Global Search class to be more appropriate for the variations of MISS we have investigated

8. Conclusions

We have shown how to systematically synthesize fast solutions to a number of variants of the MSS problem. Our synthesized programs improve on the results of Sasano et al. who use program transformation to arrive at their programs. Perhaps more importantly, we wish to claim that our approach is simple enough to be used by a competent and skilled developer and flexible enough that the same strategy can be used with minor modifications to each of the variants of the problem. We have found dominance relations to be extremely crucial to the efficiency of the final algorithm. While bounds tests contribute important constant factor improvements, it is the dominance relation that reduces the complexity from exponential to polynomial. Propagation and finite differencing then further reduces it to linear. We are currently developing a theory of greedy algorithms based around dominance relations and hope to report on this work in the future.

It could be argued in favor of program transformation that a program transform is designed by a tool or library designer, ie. someone other than the developer. From the developer's point of view, the transformation of their specification into an efficient program is automatic. But this rests on the assumption that a suitable program transform is available, which is not always the case. Program transforms work best when they have the fewest number of conditions for their applicability (e.g. the fusion transformation used in GHC). Unfortunately, if the conditions for the transform are not satisfied, then it cannot be applied - it is all or nothing. In order to handle the new requirements, the program transform and its associated theory needs to be reworked. Since the skill and knowledge to do this is not generally with the developer (by our starting assumption), this step becomes a bottleneck. We prefer instead to start from a very general framework and give the developers a set of techniques by which they can construct and experiment with a variety of solutions to their problems.

There is still room for improvement of dominance testing in the abstract program. More efficient schemes are possible, in which finer-grained control over which spaces are tested for semi-congruence, as well as better data structures that speed up the search for possible semi-congruent spaces. Also, for simplicity, represent a map as an association list for the `m` field in `R`. In reality, an Array or similar $O(1)$ structure would be a better. Appropriate data structure refinement would effect this improvement.

References

[ANCK08] A Allahverdi, C T Ng, T C E Cheng, and M K Kovalyov. A survey of scheduling problems with setup times or costs. *European J. of Operational Res.*, 187:985–1032, 2008.

[Bir89] R. S. Bird. Algebraic identities for program calculation. *Comput. J.*, 32(2):122–126, 1989.

[BM97] Richard Bird and Oege De Moor. *Algebra of programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[Dec03] R Dechter. *Constraint Processing*. Morgan Kauffman, 2003.

[Iba77] T. Ibaraki. The power of dominance relations in branch-and-bound algorithms. *J. ACM*, 24(2):264–279, 1977.

[JP93] D. J. T. Jeuring and T. O. Pekela. Theories for algorithm calculation. Technical report, 1993.

[KL95] David J. King and John Launchbury. Structuring depth-first search algorithms in haskell. pages 344–354. ACM Press, 1995.

[Mu08] Shin-Cheng Mu. Maximum segment sum is back: deriving algorithms for two segment problems with bounded lengths. In *PEPM '08*, pages 31–39. ACM, 2008.

[NSC09] S Nedunuri, D R Smith, and W Cook. Tactical synthesis of global search algorithms. In *Submitted To: Proc. NASA Symposium on Formal Methods*, 2009.

[PK82] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM TOPLAS*, 4(3):402–454, 1982.

[S] Specware. http://www.specware.org.

[SHT00] Isao Sasano, Zhenjiang Hu, and Masato Takeichi. Make it practical: A generic linear-time algorithm for solving maximum-weightsum problems. In *Proc. Intl. Conf. on Functional Prog.(ICFP)*, 2000.

[SHT01] Isao Sasano, Zhenjiang Hu, and Masato Takeichi. Generation of efficient programs for solving maximum multi-marking problems. In *Proc. 2nd Intl. SAIG Workshop*, 2001.

[SL90] D. R. Smith and M. R. Lowry. Algorithm theories and design tactics. *Sci. Comput. Program.*, 14(2-3):305–321, 1990.

[Smi87] Douglas R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Sci. Comput. Program.*, 8(3):213–229, 1987.

[Smi88] D R Smith. Structure and design of global search algorithms. Technical Report Kes.U.87.12, Kestrel Institute, 1988.

[Smi90] D R Smith. Kids: A semi-automatic program development system. *IEEE Trans. on Soft. Eng., Spec. Issue on Formal Methods*, 16(9):1024–1043, September 1990.

[SOH05] Isao Sasano, Mizuhito Ogawa, and Zhenjiang Hu. Maximum marking problems with accumulative weight functions. In *Proc. ICTAC*. Springer-Verlag, 2005.

[SPW95] Douglas R. Smith, Eduardo A. Parra, and Stephen J. Westfold. Synthesis of high-performance transportation schedulers. Technical report, Kestrel Institute, 1995.