Optimizing a Geomodeling Domain Specific Language

Bruno Morais Ferreira, Fernando Magno Quintão Pereira,
Hermann Rodrigues and Britaldo Silveira Soares-Filho

Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil
{brunomf,fernando}@dcc.ufmg.br,{hermann,britaldo}@csr.ufmg.br

Abstract. In this paper we describe Dinamica EGO, a domain specific languages (DSL) for geomodeling. Dinamica EGO provides users with a rich suite of operators available in a script language and in a graphical interface, which they can use to process information extracted from geographic entities, such as maps and tables. We analyze this language through the lens of compiler writers. Under this perspective we describe a key optimization that we have implemented on top of the Dinamica EGO execution environment. This optimization consists in the systematic elimination of memory copies that Dinamica EGO uses to ensure referential transparency. Our algorithm is currently part of the official distribution of this framework. We show, via a real-life case study, that our optimization can speedup geomodeling applications by almost 100x.

1 Introduction

Domain Specific Languages (DSLs) are used in the most varied domains, and have been shown to be effective to increase the productivity of programmers [10]. In particular, DSLs enjoy remarkable success in the geomodeling domain [2]. In this case, DSLs help non-professional programmers to extract information from maps, to blend and modify this information in different ways, and to infer new knowledge from this processing. A tool that stands out in this area is *Dinamica EGO* [11, 14, 13, 15]. This application, a Brazilian home-brew created in the Centro de Sensoriamento Remoto of the Federal University of Minas Gerais (UFMG), today enjoys international recognition as an effective and useful framework for geographic modeling. Applications of it include, for instance, carbon emissions and deforestation [3], assessment of biodiversity loss [12], urbanization and climate change [9], emission reduction(REDD) [8] and urban growth [17].

Users interact with this framework via a graphical programming language which allows them to describe how information is extracted from maps, possibly modified, and written back into the knowledge base. Henceforth we will use the term EGO Script to describe the graphical programming language that is used as the scripting language in the Dinamica EGO framework. This language, and its underlying execution environment, has been designed to fulfill two main goals. Firstly, it must be easy to use; hence, requiring a minimum of programming

skill. Users manipulate maps and other geographic entities via graphical symbols which can be connected through different data-flow channels to build patterns for branches and loops. Secondly, it must be efficient. To achieve this goal, it provides a 64-bit native version written in C++ and java with multithreading and dynamic compilation. Being a dynamic execution environment, EGO Script relies heavily on Just-in-time compilation to achieve the much necessary speed.

Dinamica's current success is the result of a long development process, which includes engineering decisions that have not been formally documented. In this paper we partially rectify this omission, describing a key compiler optimization that has been implemented in the Dinamica EGO execution environment. In order to provide users with a high-level programming environment, one of the key aspects of Dinamica's semantics is referential transparency, as we explain in Section 3. Scripts are formed by components, and these components must not modify the data that they receive as inputs. This semantics imposes on the tool a heavy burden, because tables containing data to be processed must be copied before been passed from one component to the other. Removing these copies is a non-trivial endeavor, inasmuch as minimizing such copies is a NP-complete problem, as we show in Section 3. Because this problem is NP-complete, we must recourse to heuristics to eliminate redundant copies. We discuss these heuristics in Section 4. Although we have discussed this algorithm in the context of Dinamica EGO, we believe that it can also be applied in other data-flow based systems, such as programs built on top of the filter-stream paradigm [16].

We provide empirical evidence that supports our design decisions in Section 5, by analyzing the runtime behavior of a complex application in face of our optimization. This application divides an altitude map into slices of same height. In order to get more precise ground information, we must decrease the height of each slice; hence, increasing the amount of slices in the overall database. In this case, for highly accurate simulations our copy elimination algorithm boosts the performance of Dinamica EGO by almost 100x.

2 A Bird's Eye View of Dinamica EGO

We illustrate EGO Script through an example that, although artificial, contains some of the key elements that we will discuss in the rest of this paper. Consider the following problem: "what is the mean slope of the cities from a given region?" We can answer this query by combining data from two maps encompassing the same geographic area. The first map contains the slope of each area. We can assume that each cell of this matrix represents a region of a few hectares, and that the value stored in it is the average inclination of that region. The second map is a matrix that associates with each region a number that identifies the municipality where that region is located. Regions that are part of the same city jurisdiction have the same identifier. The EGO script that solves this problem is shown in Figure 1. An EGO Script program is an ensemble of components, which are linked together by data channels. Some components encode data, others computation. Components in this last category are called *functors*. The

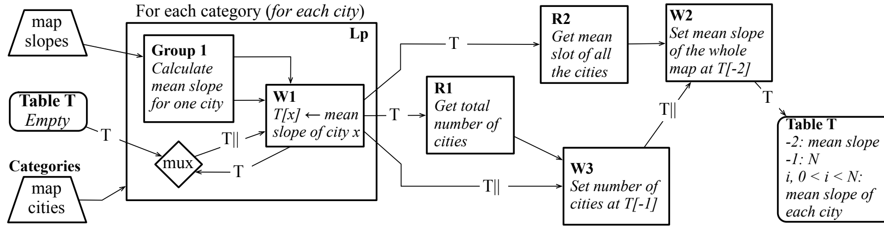

Fig. 1. An EGO Script program that finds the average slope of the cities that form a certain region. The parallel bars (||) denote places where the original implementation of Dinamica EGO replicates the table T.

order in which functors must execute is determined by the runtime environment, and should obey the dependencies created by the data channels.

EGO Script uses the trapezoid symbol to describe data to be processed, which is usually loaded from files. In our example, this data are the two maps. We call the map of cities a *categorical* map, as it divides a matrix into equivalence classes. Each equivalence class contains the cells that belong to the same city administration. The large rectangle named **Lp** with smaller components inside it is a *container*, which represents a loop. It will cause some processing to be executed for each different category in the map of cities. The results produced by this script will be accumulated in the table T. Some functors can write into T. We use names starting with **W** to refer to them. Others only read the table. Their names start with **R**. In our example, the positive indices of T represent city entries. Once the script terminates, T will map each city to its slope. Additionally, this accumulator will have in its -1 index the total number of cities that have been processed, and in its -2 index the average slope of the entire map of slopes.

The functor called **W1** is responsible for filling the table with the results obtained for each city. The element called mux works as a loop header: it passes the empty accumulator to the loop, and after the first iteration, it is in charge of merging the newly produced data with the old accumulator. **W1** always copies the table before updating it. We denote this copy by the double pipes after the table name in the input channel, e.g., T||. The attentive reader must be wondering: why is this copy necessary? Even more if we consider that it is performed inside a loop? The answer is pragmatic: before we had implemented the optimization described in this paper, each component that could update data should replicate this data. In this way, any component could be reused as a black box, without compromising the referential transparency that is a key characteristics of the language. We have departed from this original model by moving data replication to the channels, instead of the components, and using a whole program analysis to eliminate unnecessary copies.

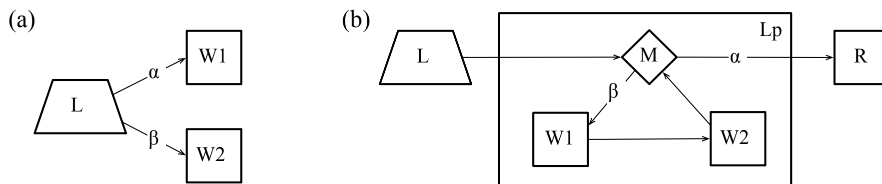

Fig. 2. Two examples in which copies are necessary.

The functor called **R1** counts the number of cities in the map, and gives this information to **W3**, which writes it in the index -1 of the table. Functor **R2** computes the mean slope of the entire map. This information is inserted into the index -2 of the table by **W2**. Even though the updates happen in different locations of T, the components still perform data replication to preserve referential transparency. The running example cannot trivially discover that updates happen at different locations of the data-structure. In this simple example, each of these indices, -1 and -2, are different constants. However, the locations to be written could have been derived, instead, from much more complicated expressions whose values could only be known at execution time.

As we will show later in this paper, we can remove all the three copies in the script from Figure 1. However, there are situations in which copies are necessary. Figure 2 provides two such examples. A copy is necessary in Figure 2(a), either in channel α or in channel β – but not in both – because of a write-write hazard. Both functors, **W1** and **W2** need to process the original data that comes out of the loader **L**. Without the copy, one of them would read a stained value. Data replication is necessary in Figure 2(b), either in channel α or β, because there is a read-write hazard between **R** and **W1**, and it is not possible to schedule **R** to run before **W1**. **W1** is part of a container, **Lp**, that precedes **R** in any scheduling, i.e., a topological ordering, of the script.

3 The Core Semantics

In order to formally state the copy minimization problem that we are interested, we will define a core language, which we call μ-Ego. A μ-Ego program is defined by a tuple (S, T, Σ), where S, a *scheduling*, is a list of processing elements to be evaluated, T is an output table, and Σ is a storage memory. Each processing element is either a *functor* or a *container*. Functors are three element tuples (N, I, P), where N is this component's unique identifier in T, I is the index of the storage area that the component owns inside Σ, and P is the list of *predecessors* of the component. We let $T : N \mapsto \mathbb{N}$, and $\Sigma : I \mapsto \mathbb{N}$. A container is a pair $(\mathbb{N}, S)$, where S is a scheduling of processing elements.

Figure 3 describes the operational semantics of μ-Ego. Rule Cont defines the evaluation of a container. Containers work like loops: the evaluation of (N, S_l)

$$[\textsc{Null}] \qquad ([], T, \Sigma) \to (T, \Sigma)$$

$$[\textsc{Cont}] \qquad \frac{S' = S_l^K @\ S \qquad (S', T, \Sigma) \to (T', \Sigma')}{((K, S_l) :: S, T, \Sigma) \to (T', \Sigma')}$$

$$[\textsc{Func}] \qquad \frac{\begin{array}{c} V = \max(P, \Sigma) \\ \Sigma' = \Sigma \setminus [I \mapsto V + 1] \qquad T' = T \setminus [N \mapsto V + 1] \qquad (S, T', \Sigma') \to (T'', \Sigma'') \end{array}}{((N, I, P) :: S, T, \Sigma) \to (T'', \Sigma'')}$$

Fig. 3. The operational semantics of μ-EGO.

consists in evaluating sequentially N copies of the scheduling S_l. We let the symbol @ denote list concatenation, like in the ML programming language. The expression $S_l^K @\ S$ denotes the concatenation of K copies of the list S_l in front of the list S. Rule FUNC describes the evaluation of functors. Each functor (N, I, P) produces a value V. If we let V_m be the maximum value produced by any predecessor of the component, i.e., some node in P, then $V = V_m + 1$. When processing the component (N, I, P), our interpreter binds V to N in T, and binds V to I in Σ.

Figure 4 illustrates the evaluation of a simple μ-EGO program. The digraph in Figure 4(a) denotes a program with five functors and a container. We represent each functor as a box, with a natural number on its upper-right corner, and a letter on its lower-left corner. The number is the component's name N, and the letter is its index I in the store. The edges in Figure 4(a) determine the predecessor relations among the components. Figure 4(b) shows the scheduling that we are using to evaluate this program. We use the notation $(p_1, \ldots, p_n)^k$ to denote a container with k iterations over the processing elements $p_1, \ldots, p_n$. Figure 4(c) shows the store Σ, and Figure 4(d) shows the output table T, after each time the Rule FUNC is evaluated. In this example, Σ and T have the same number of indices. Whenever this is the case, these two tables will contain the same data, as one can check in Rule FUNC. We use gray boxes to mark the value that is updated at each iteration of the interpreter. These boxes, in Figure 5(d), also identify which component is been evaluated at each iteration.

We say that a μ-EGO program is *canonical* if it assigns a unique index I in the domain of Σ to each component. We call the evaluation of such a program a *canonical evaluation*. The canonical evaluation provides an upper bound on the number of storage cells that a μ-EGO program requires to execute correctly. Given that each component has its own storage area, data is copied whenever it reaches a new component. In this case, there is no possibility of data races. However, there is a clear waste of memory in a canonical evaluation. It is possible to re-use storage indices, and still to reach the same final configuration of the output table. This observation brings us to Definition 1, which formally states the storage minimization problem.

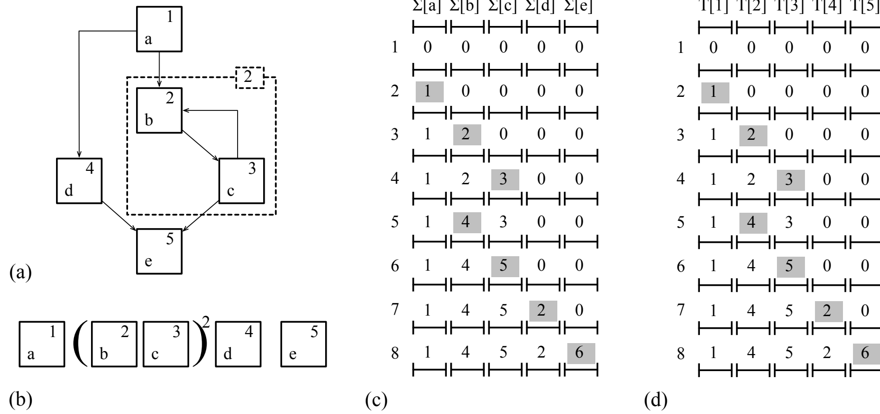

Fig. 4. Canonical evaluation of an μ-Ego program. (a) The graph formed by the components. (b) The scheduling. (c) The final configuration of Σ. (d) The final configuration of T.

Definition 1. Storage Minimization with Fixed Scheduling [Smfs]

Instance: *a scheduling S of the components in a μ-*Ego *program, plus a natural K, the number of storage cells that any evaluation can use.*

Problem: *find an assignment of storage indices to the components in S with K or less indices that produces the same T as a canonical evaluation of S.*

For instance, the program in Figure 5 produces the same result as the canonical evaluation given in Figure 4; however, it uses only 3 storage cells. In this example, the smallest number of storage indices that we can use to simulate a canonical evaluation is three. Figure 6 illustrates an evaluation that does not lead to a canonical result. In this case, we are using only two storage cells to keep the values of the components. In order to obtain a canonical result, when evaluating component 4 we need to remember the value of components 2 and 3. However, this is not possible in the configuration seen in Figure 6, because these two different components reuse the same storage unit.

3.1 SMFS has polynomial solution for schedulings with no back-edges

If a scheduling S has a component c" scheduled to execute after a component $c' = (N, I, P)$, and $c" \in P$, then we say that the scheduling has a back-edge $\overrightarrow{c"c'}$. Smfs has a polynomial time - exact - solution for programs without back-edges, even if they contain loops. We solve instances of Smfs that have this restriction by reducing them to interval graph coloring. Interval graph coloring has an $O(N)$ exact solution, where N is the number of lines in the interval [7]. The reduction

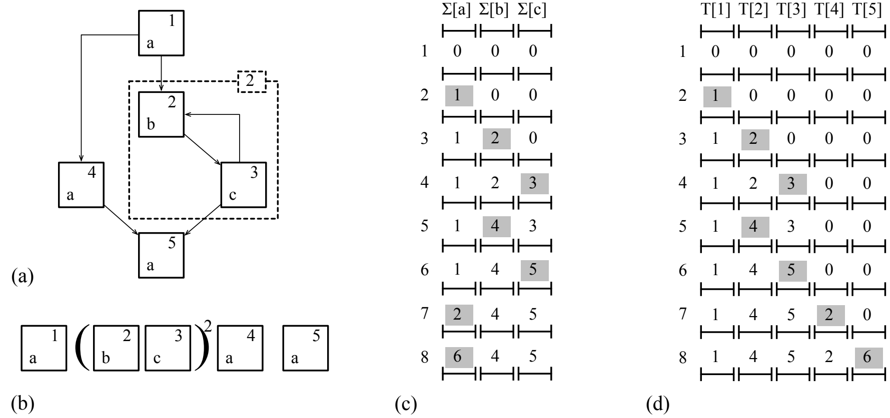

Fig. 5. Evaluation of an optimized μ-EGO program.

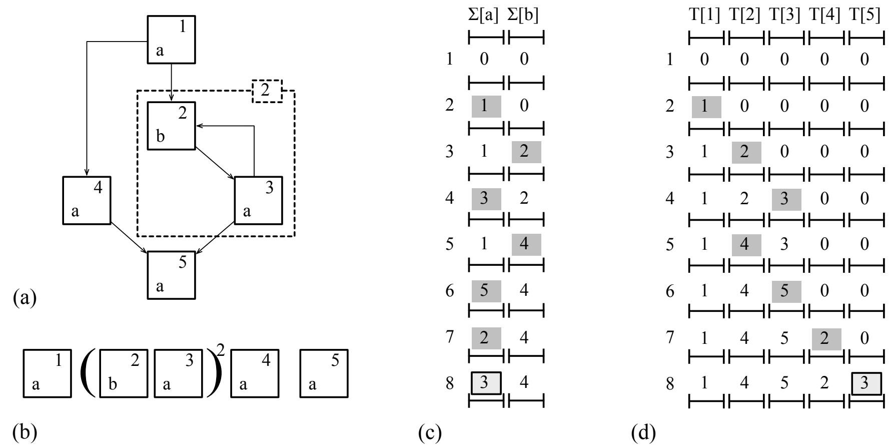

Fig. 6. Evaluation of an μ-EGO program that does not produce a canonical result.

is as follows: given a scheduling S, let s_c be the order of component c in S; that is, if component c appears after $n-1$ other components in S, then $s_c = n$. For each component c we create an interval that starts at s_c and ends at s_x, where s_x is the greatest element among:

- s'_c, where c' is a successor of c.
- s_{c_f}, where c_f is the first component after any component in a loop that contains a successor of c.

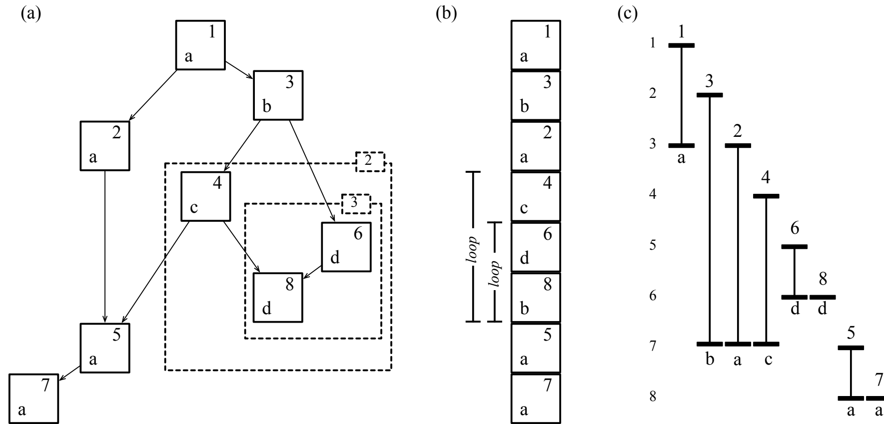

Fig. 7. Reducing SMFS to interval graph coloring for schedulings without back-edges. (a) The input μ-EGO program. (b) The input scheduling. (c) The corresponding intervals. The integers on the left are the orderings of each component in the scheduling.

Figure 7 illustrates this reduction. A coloring of the interval graph consists of an assignment of colors to the intervals, in such a way that two intervals may get the same color if, and only if, they have no common overlapping point, or they share only their extremities. Theorem 1 shows that a coloring of the interval graph can be converted to a valid index assignment to the program, and that this assignment is optimal. In the figure, notice that the interval associated to component three goes until component five, even though these components have no direct connection. This happens because component five is the leftmost element after any component in the two loops that contain successors of component three. Notice also that, by our definition of interval coloring, components six and eight, or five and seven, can be assigned the same colors, even though they have common extremities.

Theorem 1. *A tight coloring of the interval graph provides a tight index assignment in the μ-EGO program.*

proof: See [4]. □

3.2 SMFS is NP-complete for general programs with fixed scheduling.

We show that SMFS is NP-complete for general programs with fixed schedulings by reducing this problem to the coloring of Circular-Arc graphs. A circular-arc graph is the intersection graph formed by arcs on a circle. The problem of finding a minimum coloring of such graphs is NP-complete, as proved by Garey *et al* [6].

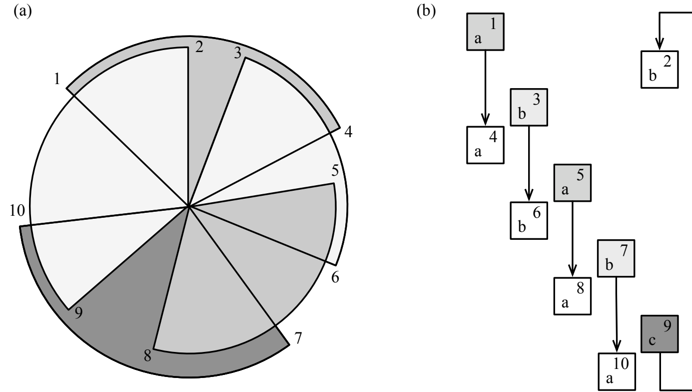

Fig. 8. Reducing SMFS to circular-arc graph coloring for general schedulings. (a) The input arcs. (b) The corresponding μ-EGO program.

Notice that if the number of colors k is fixed, then this problem has an exact solution in $O(n \times k! \times k \times \ln k)$, where n is the number of arcs [5].

We define a reduction R, such that, given an instance P_g of the coloring of arc-graphs, $R(P_g)$ produces an equivalent instance P_s of SMFS as follows: firstly, we associate an increasing sequence of integer number with each end point of an arc, in clockwise order, starting from any arc. If i and j are the integers associated with a given arc, then we create two functors, c_i and c_j. We let c_i be the single element in the predecessor set of c_j, and we let the predecessor set of c_i be empty. We define a fixed scheduling S that contains these components in the same order their corresponding integers appear in the input set of arcs. Figure 8 illustrates this reduction. We claim that solving SMFS to this μ-EGO program is equivalent to coloring the input graph.

Theorem 2. *Let P_s be an instance of* SMFS *produced by $R(P_g)$. P_s can be allocated with K indices, if, and only if, P_g can be colored with K colors.*

proof: See [4]. □

4 Copy Minimization

Data-Flow Analysis: We start the process of copy pruning with a backward-must data-flow analysis that determines which channels can lead to places where data is written. Our data-flow analysis is similar to the well-known *liveness analysis* used in compilers [1, p.608]. Figure 9 defines this data-flow analysis via four inference rules. If (N, I, P) is a component, and $N' \in P$, then the relation channel(N', N) is true. If N is a component that writes data, then the relation write(N) is true. Contrary to the original semantics of μ-EGO, given in Figure 3,

$$[\textsc{Df}1] \quad \frac{\text{channel}(N_1, N_2) \qquad \text{write}(N_2)}{\text{abs}(N_1, N_2, \{N_2\})} \qquad\qquad [\textsc{Df}2] \quad \frac{\text{abs}(N_1, N_2, A') \qquad \text{out}(N_1, A)}{A' \subseteq A}$$

$$[\textsc{Df}3] \quad \frac{\text{channel}(N_1, N_2) \qquad \neg\text{write}(N_2) \qquad \text{out}(N_2, A)}{\text{abs}(N_1, N_2, A)}$$

$$[\textsc{Df}4] \quad \frac{\text{channel}(N_1, N_2) \qquad \text{read}(N_2) \qquad \text{out}(N_2, A)}{\text{abs}(N_1, N_2, A \cup \{r\})}$$

Fig. 9. Inference rules that define our data-flow analysis.

we also consider, for the sake of completeness, the existence of functors that only read the data. If N is such a functor, than the predicate $\text{read}(N)$ is true. This analysis uses the lattice constituted by the power-set of functor names, plus a special name "r", that is different from any functor name. We define the abstract state of each channel by the predicate $\text{abs}(N_1, N_2, P)$, which is true if P is the set of functors that can write data along any path that starts in the channel (N_1, N_2), or $P = \{r\}$. Rule DF1 states that if a functor N_2 updates the data, then the abstract state of any channel ending at N_2 is a singleton that contains only the name of this functor. We associate with each functor N a set (out) of all the functor names present in abstract states of channels that leave N. This set is defined by Rule DF2. According to Rule DF3, if a functor N does not write data, the abstract state of any channel that ends at N is formed by N's out set. Finally, Rule DF4 back propagates the information that a functor reads data.

Figure 10 shows the result of applying our data-flow analysis onto the example from Section 2. The channels that lead to functors where table T can be read or written have been labeled with the abstract states that the data-flow analysis computes, i.e., sets of functor names. In this example each of these sets is a singleton. There is no information on the dashed-channels, because T is not transmitted through them. Notice that we must run one data-flow analysis for each data whose copies we want to eliminate. In this sense, our data-flow problem is a *partitioned variable problem*, following Zadeck's taxonomy [18]. A partitioned variable problem can be decomposed into a set of data-flow problems – usually one per variable – each independent on the other.

Criteria to Eliminate Copies: Once we are done with the data-flow analysis, we proceed to determine which data copies are necessary, and which can be eliminated without compromising the semantics of the script. Figure 11 shows the two rules that we use to eliminate copies: (CP1) write-write race, and (CP2) write-read race. Before explaining each of these rules, we introduce a number of relations used in Figure 11. The rules PT1 and PT2 denote a path between two components. Rule DOM defines a predicate $\text{dom}(C, N)$, which is true whenever the component N names a functor that is scheduled to execute inside a container C. We say that C *dominates* N, because N will be evaluated if, and only if, C

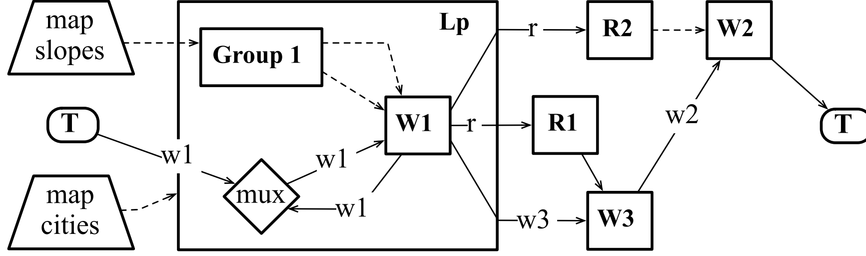

Fig. 10. The result of the data-flow analysis on the program seen in Figure 1.

$$[\textsc{Pt1}]\quad \frac{\text{channel}(N_1, N_2)}{\text{path}(N_1, N_2)} \qquad [\textsc{Pt2}]\quad \frac{\text{channel}(N_1, N) \qquad \text{path}(N, N_2)}{\text{path}(N_1, N_2)}$$

$$[\textsc{Dom}]\quad \frac{C = (\mathbb{N}, S) \qquad N \in S}{\text{dom}(C, N)} \qquad [\textsc{Ori}]\quad \frac{\text{channel}(N, N_1) \qquad \text{channel}(N, N_2) \qquad N_1 \neq N_2}{\text{orig}(N, N_1, N_2)}$$

$$[\textsc{Dp1}]\quad \frac{\text{path}(N_1, N_2)}{\text{dep}(N_1, N_2)} \qquad [\textsc{Dp2}]\quad \frac{\text{path}(N_1, N) \qquad \text{dom}(N_2, N)}{\text{dep}(N_1, N_2)}$$

$$[\textsc{Dp3}]\quad \frac{\text{dom}(N_1, N) \qquad \text{path}(N, N_2)}{\text{dep}(N_1, N_2)} \qquad [\textsc{Dp4}]\quad \frac{\text{dom}(N_1, N_1') \qquad \text{dom}(N_2, N_2') \qquad \text{path}(N_1', N_2')}{\text{dep}(N_1, N_2)}$$

$$[\textsc{Lcd}]\quad \frac{\text{dom}(N, N_1) \qquad \text{dom}(N, N_2) \qquad \nexists N', \text{dom}(N', N_1), \text{dom}(N', N_2), \text{dom}(N, N')}{\text{lcd}(N_1, N_2, N)}$$

$$[\textsc{Prd}]\quad \frac{\text{orig}(O, N_1, N_2) \qquad \text{lcd}(N_1, N_2, D) \qquad \text{dom}(D, O) \qquad \neg\text{dep}(N_2, N_1)}{\text{pred}(N_1, N_2)}$$

$$[\textsc{Cp1}]\quad \frac{\text{orig}(N, N_1, N_2) \qquad \text{out}(N_1, \{\ldots, f_1, \ldots, \}) \qquad \text{out}(N_2, \{\ldots, f_2, \ldots\}) \qquad f_1 \neq f_2 \neq N}{\text{need_copy}(N, N_1)}$$

$$[\textsc{Cp2}]\quad \frac{\text{orig}(N, N_1, N_2) \qquad \text{out}(N_1, \{\ldots, f_1, \ldots\}) \qquad \text{out}(N_2, \{r\}) \qquad f_1 \neq N \qquad \neg\text{pred}(N_2, N_1)}{\text{need_copy}(N, N_1)}$$

Fig. 11. Criteria to replicate data in Ego Script programs.

is evaluated. Rule Lcd defines the concept of *least common dominator*. The predicate $\text{lcd}(N_1, N_2, N)$ is true if N dominates both N_1 and N_2, and for any other component N' that also dominates these two components, we have that N' dominates N. The relation $\text{orig}(N, N_1, N_2)$ is true whenever the functor N is linked through channels to two different components N_1 and N_2. As an example, in Figure 7(a) we have $\text{orig}(3, 4, 6)$.

Rules Dp1 through Dp4 define the concept of *data dependence* between components. A component N_2 depends on a component N_1 if a canonical evaluation of the script requires N_1 to be evaluated before N_2. The relation $\text{pred}(N_1, N_2)$

indicates that N_1 can always precede N_2 in a canonical evaluation of the Ego Script program, where components N_1 and N_2 have a common origin. In order for this predicate to be true, N_1 and N_2 cannot be part of a loop that does not contain O. Going back to Figure 7(a), we have that pred$(4, 6)$ is not true, because 3, the common origin of components 4 and 6, is located outside the loop that dominates these two components. Furthermore, N_1 should not depend on N_2 for pred(N_1, N_2) to be true.

By using the predicates that we have introduced, we can determine which copies need to be performed in the flow chart of the Ego Script program. The first rule, CP1, states that if there exist two channels leaving a functor f, and these channels lead to other functors different than f where the data can be overwritten, then it is necessary to replicate the data in one of these channels. Going back to the example in Figure 10, we do not need a copy between the components **W1** and mux, because this channel is bound to the name of **W1** itself. This saving is possible because any path from mux to all the other functors that can update the data must go across **W1**. Otherwise, we would have also the names of these functors along the **W1**-mux channel. On the other hand, by this very Rule CP1, a copy is necessary between one of the channels that leave **L** in Figure 2(a). The second rule, CP2, is more elaborated. If two components, N_1 and N_2 are reached from a common component N, N_2 only reads data, and N_1 writes it, it might be possible to avoid the data replication. This saving is legal if it is possible to schedule N_2 to be executed before N_1. In this case, once the data is written by N_1, it will have already been read by N_2. If that is not the case, e.g., pred(N_2, N_1) is false, then a copy is necessary along one of the channels that leave out N. This rule lets us avoid the data replication in the channel that links **W1** and **W3** in Figure 1. In this case, there is no data-hazard between **W3** and either **R1** or **R2**. These components that only read data can be scheduled to execute before **W3**.

4.1 Correctness

In order to show that the rules in Figure 11 correctly determine the copies that must be performed in the program, we define a correctness criterion in Theorem 3. A μ-EGO program is correct if its evaluation produces the same output table as a canonical evaluation. The condition in Theorem 3 provides us with a practical way to check if the execution of a program is canonical. Given a scheduling of components S, we define a *dynamic scheduling* $\boldsymbol{S}$ as the complete trace of component names observed in an execution of S. For instance, in Figure 5, we have $S = 1, (2, 3)^2, 4, 5$, and we have $\boldsymbol{S} = 1, 2, 3, 2, 3, 4, 5$. We let $\boldsymbol{S}[i]$ be the i-th functor in the trace $\boldsymbol{S}$, and we let $|\boldsymbol{S}|$ be the number of elements in this trace. In our example, we have $\boldsymbol{S}[1] = 1$, $\boldsymbol{S}[7] = 5$, and $|\boldsymbol{S}| = 7$. Finally, if $p = \boldsymbol{S}[j]$ is a predecessor of the functor $\boldsymbol{S}[i]$, and for any $k, j < k < i$, we have that $\boldsymbol{S}[k] \neq \boldsymbol{S}[j]$, then we say that $\boldsymbol{S}[j]$ is an immediate dynamic predecessor of $\boldsymbol{S}[i]$.

Theorem 3. *The execution of an* μ-EGO *program* (S, T, Σ) *is canonical if, for any* $n, 1 \leq n \leq |\boldsymbol{S}|$*, we have that, for any predecessor* p *of* $\boldsymbol{S}[n]$*, if* $\boldsymbol{S}[i] = p$ *and* $i, 1 \leq i < n$ *is an immediate dynamic predecessor of* $\boldsymbol{S}[i]$*, then for any* $j, i < j < n$*, we have that* $\Sigma[\boldsymbol{S}[j]] \neq \Sigma[p]$*.*

proof: See [4]. □

We prove that the algorithm to place copies is correct by showing that each copy that it eliminates preserves the condition in Theorem 3. There is a technical inconvenient that must be circumvented: the Rules CP1 and CP2 from Figure 11 determine which copies *cannot* be eliminated. We want to show that the *elimination* of a copy is safe. Thus, we proceed by negating the conditions in each of these rules, and deriving the correctness criterion from Theorem 3.

Theorem 4. *The elimination of copies via the algorithm in Figure 11 preserves the correctness criterion from Theorem 3.*

proof: See [4]. □

5 Experiments

We show how our optimization speeds up Dinamica EGO via a case study. This case study comes from a model used to detect hilltops in protected areas, which is available in Dinamica's webpage. Figure 12 gives a visual overview of this application. This model receives two inputs: an elevation map and a vertical resolution value. The EGO script divides the elevation map vertically into slices of equal height. This height is defined by the vertical resolution. Then the map is normalized and divided in discrete regions, as we see in Figure 12(b) and (c). Before running the functor that finds hilltops, this script performs other analyses to calculate average slopes, to compute the area of each region and to find the average elevation of each region. The model outputs relative height, plateau identifiers, hilltops, plus coordinates of local minima and local maxima. This EGO script uses tables intensively; hence, data replication was a bottleneck serious enough to prevent it from scaling to higher resolutions before the deployment of our optimization.

Figure 13 shows the speedup that we obtain via our copy elimination algorithm. These numbers were obtained in an Intel Core2Duo with a 3.00 GHz processor and 4.00 GB RAM. We have run this model for several different vertical resolution values. The smaller this value, more slices the map will have and, therefore more regions and more table inputs. This model has three operators that perform data replication, but given that they happen inside loops, the dynamic number of copies is much greater. Figure 13 shows the number of *dynamic copies* in the unoptimized program. High resolution, plus the excessive number of copies, hinders scalability, as we can deduce from the execution times given in Figure 13. This model has three components that copy data, and our optimization has been able to eliminate all of them. The end result is an improvement of almost 100x in execution speed, as we observe in the fourth column of Figure 13.

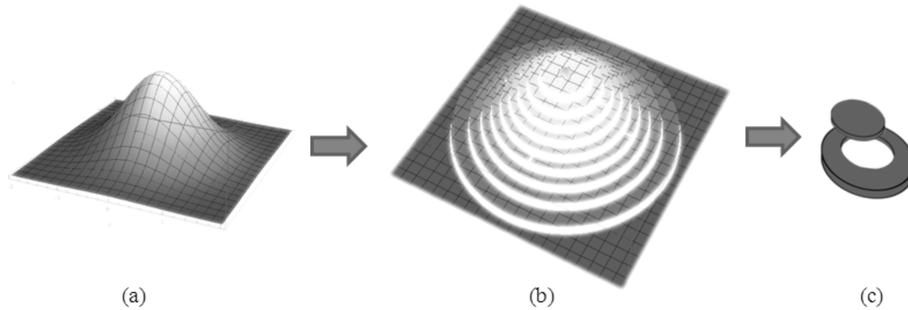

Fig. 12. Hilltop detection. (a) Height map. (b) Normalized map. (c) Extracted discrete regions.

V	D	T_u	T_o	R
20	1,956	20	20	1
15	2,676	28	26	1.0769
13	3,270	30	29	1.0344
11	4,677	32	32	1
10	6,126	36	36	1
9	9,129	39	36	1.08333
8	15,150	49	39	1.25641
7	29,982	87	50	1.74
5	137,745	995	76	13.0921
4	279,495	4,817	116	41.5258
3	518,526	18,706	197	94.9543

Fig. 13. V: Vertical resolution(m). D: Number of dynamic copies without optimization. T_u: Execution time without optimization (sec). T_o: Execution time with optimization (sec). R: Execution time ratio: (time non optimized / time optimized).

6 Conclusion

This paper has described a compiler optimization that we have implemented on top of the Dinamica EGO domain specific language for geomodeling. This optimization is, nowadays, part of the official distribution of Dinamica EGO, and is one of the key elements responsible for the high scalability of this framework. Dinamica EGO is freeware, and its use is licensed only for educational or scientific purposes. The entire software, and accompanying documentation can be found in Dinamica's website at `http://www.csr.ufmg.br/dinamica/`.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley (2006)

2. Beven, K.: Towards a coherent philosophy for modelling the environment. Proceedings of the Royal Society 458, 2465–2484 (2002)
3. Carlson, K.M., Curran, L.M., Ratnasari, D., Pittman, A.M., Soares-Filho, B.S., Asner, G.P., Trigg, S.N., Gaveau, D.A., Lawrence, D., Rodrigues, H.O.: Committed carbon emissions, deforestation, and community land conversion from oil palm plantation expansion in west kalimantan, indonesia. Proceedings of the National Academy of Sciences (2012)
4. Ferreira, B.M., Pereira, F.M.Q., Rodrigues, H., Soares-Filho, B.S.: Optimizing a geomodeling domain specific language. Tech. Rep. LLP001/2012, Universidade Federal de Minas Gerais (2012)
5. Garey, M.R., Johnson, D.S., Miller, G.L., Papadimitriou, C.H.: The complexity of coloring circular arcs and chords. J. Algebraic Discrete Methods 1, 216–227 (1980)
6. Garey, M.R., Johnson, D.S., Sockmeyer, L.: Some simplified NP-complete problems. Theoretical Computer Science 1, 193–267 (1976)
7. Golumbic, M.C.: Algorithmic Graph Theory and Perfect Graphs. Elsevier, 1st edn. (2004)
8. Hajek, F., Ventresca, M.J., Scriven, J., Castro, A.: Regime-building for redd+: Evidence from a cluster of local initiatives in south-eastern peru. Environmental Science and Policy 14(2), 201 – 215 (2011)
9. Huong, H.T.L., Pathirana, A.: Urbanization and climate change impacts on future urban flood risk in can tho city, vietnam. Hydrology and Earth System Sciences Discussions 8(6), 10781–10824 (2011)
10. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (2005)
11. Nepstad, D., Soares-Filho, B. Merry, F., Lima, A., Moutinho, P., Carter, J., Bowman, M., Cattaneo, A., Rodrigues, H., Schwartzman, S., McGrath, D., Stickler, C., Lubowski, R., Piris-Cabeza, P., Rivero, S., Alencar, A., Almeida, O., Stella, O.: The end of deforestation in the brazilian amazon. Science 326, 1350–1351 (2009)
12. Pérez-Vega, A., Mas, J.F., Ligmann-Zielinska, A.: Comparing two approaches to land use/cover change modeling and their implications for the assessment of biodiversity loss in a deciduous tropical forest. Environmental Modelling and Software 29(1), 11–23 (2012)
13. Soares-Filho, B., Nepstad, D., Curran, L., Cerqueira, G., Garcia, R., Ramos, C., Voll, E., McDonald, A., Lefebvre, P., Schlesinger, P.: Modelling conservation in the amazon basin. Nature 440, 520–523 (2006)
14. Soares-Filho, B., Pennachin, C., Cerqueira, G.: Dinamica - a stochastic cellular automata model designed to simulate the landscape dynamics in an amazonian colonization frontier. Ecological Modeling 154, 217–235 (2002)
15. Soares-Filho, B., Rodrigues, H., Costa, W.: Modeling Environmental Dynamics with Dinamica EGO. Centro de Sensoriamento Remoto (IGC/UFMG) (2009)
16. Spring, J.H., Privat, J., Guerraoui, R., Vitek, J.: Streamflex: high-throughput stream programming in java. In: OOPSLA. pp. 211–228. ACM (2007)
17. Thapa, R.B., Murayama, Y.: Urban growth modeling of kathmandu metropolitan region, nepal. Computers, Environment and Urban Systems 35(1), 25 – 34 (2011)
18. Zadeck, F.K.: Incremental Data Flow Analysis in a Structured Program Editor. Ph.D. thesis, Rice University (1984)