DINAMICA VIRTUAL MACHINE PARA GEOCIÊNCIAS

BRUNO MORAIS FERREIRA

DINAMICA VIRTUAL MACHINE PARA GEOCIÊNCIAS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Fernando Magno Quintão Pereira

Belo Horizonte
Janeiro de 2015

BRUNO MORAIS FERREIRA

THE DINAMICA VIRTUAL MACHINE FOR GEOSCIENCES

Dissertation presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais. Departamento de Ciência da Computação. in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Fernando Magno Quintão Pereira

Belo Horizonte

January 2015

Ferreira, Bruno Morais

F383d The dinamica virtual machine for geosciences / Bruno Morais Ferreira. — Belo Horizonte, 2015
xxii, 66 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.
Orientador: Fernando Magno Quintão Pereira

1. Computação — Teses. 2. Modelagem ambiental — Teses. 3. Linguagens de programação funcional — Teses. I. Orientador. II. Título.

CDU 519.6*31(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

The dinamica virtual machine for geosciences

BRUNO MORAIS FERREIRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]

PROF. BRITALDO SILVEIRA SOARES FILHO
Departamento de Cartografia - UFMG

[illegible]

PROF. RENATO ANTÔNIO CELSO FERREIRA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 12 de fevereiro de 2015.

Dedico este trabalho aos meus pais, avós, esposa e amigos.

Acknowledgments

Agradeço aos meus pais por investirem na minha educação, aos meus avós pelo duro trabalho e infinita inspiração, à minha esposa pelo amor e apoio constantes e aos meus amigos pela alegria e suporte. Agradeço ao meu orientador pela ajuda no trabalho e pela inspiração que sua atitude proporciona. Agradeço ao Centro de Sensoriamento Remoto da UFMG e a todos os colegas que ali trabalham. Por fim, agradeço a Deus por tudo.

Resumo

Este trabalho descreve a DinamicaVM, a máquina virtual para execução de aplicações desenvolvidas em Dinamica EGO. Dinamica EGO é uma plataforma utilizada em modelagem e simulação ambiental. Por detrás da sua biblioteca de elementos visuais em modo gráfico, Dinamica EGO roda em cima de uma máquina virtual. Esta máquina - DinamicaVM - oferece aos desenvolvedores um rico conjunto de instruções, com elementos como o "map" e "reduce", que são típicos no mundo de linguagens funcionais e paralelismo. Garantir que estes componentes, muito expressivos, trabalhem juntos de forma eficiente é uma tarefa desafiadora. O ambiente de execução do Dinamica vence este desafio através de um conjunto de otimizações, emprestando ideias de linguagens de programação funcional, levando ao comportamento específico esperado em programas de alta performance para modelagem ambiental. Como mostramos neste trabalho algumas dessas otimizações levam speedups de quase 100 vezes, e são fundamentais para o desempenho e qualidade de uma das ferramentas de modelagem ambiental mais utilizadas do mundo.

Palavras-chave: Fluxo de Dados, Linguagem de Domínio Específico, Modelagem Ambiental.

Abstract

This work describes DinamicaVM, the virtual machine that runs applications developed in Dinamica EGO. Dinamica EGO is a framework used in the development of geomodeling applications. Behind its multitude of visual modes and graphic elements, Dinamica EGO runs on top of a virtual machine. This machine - DinamicaVM - offers developers a rich instruction set architecture, featuring elements such as map and reduce, which are typical in the functional/parallel world. Ensuring that these very expressive components work together efficiently is a challenging endeavour. Dinamica's runtime addresses this challenge through a suite of optimizations, which borrows ideas from functional programming languages, and leverages specific behavior expected in geo-scientific programs. As we show in this work some of these optimizations deliver speedups of almost 100x, and are key to the industrial-quality performance of one of the world's most widely used geomodeling tools.

Palavras-chave: Dataflow, Domain Specific Languages, Geomodeling.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Computer Science have been playing an important role in several fields of science and this is not different with environmental sciences. The need to understand how the planet Earth works is the greatest motivation to the development of new methods, techniques, algorithms. With the advent of remote sensing a great amount of data about the Earth's surface became available. Apply these methods and techniques in models and simulations that use this great amount of data is a challenging endeavour and is in this context that Dinamica EGO emerges.

Dinamica EGO is a framework that supports the development of geomodeling applications [Soares-Filho et al., 2002, 2009, 2013]. It was first released in 1998, and since then it has grown to enjoy international recognition as an effective and useful framework for geomodeling. It has been used to model carbon emission and deforestation [Carlson et al., 2012; Nepstad et al., 2009], biodiversity loss [Pérez-Vega et al., 2012], urbanization and climate change [Huong and Pathirana, 2011], emission reduction [Hajek et al., 2011] and urban growth [Thapa and Murayama, 2011]. Testimony of Dinamica's maturity are the intergovernmental collaborations where it is used. Among its application to public policies in collaboration with governmental institutions in Brazil and abroad we cite the World Bank and the United Nations Development Programme. For instance, the REDD project, which integrates state departments from Bolivia, Peru and Brazil, is using Dinamica to map the southwestern Amazon[1]. As another example of relevant use, Dinamica's simulation of the environmental impact of the Santarém-Cuiabá Interstate (BR 163) has been key to lead the Brazilian government to create a national preservation area along this highway[2]. Finally, SimAmazonia, a large effort to model climate change in the Amazon Basin using Dinamica EGO [Soares-Filho et al.,

[1]`http://csr.ufmg.br/map/`

[2]`http://www.csr.ufmg.br/dinamica/applications/cuiaba- santarem.html`

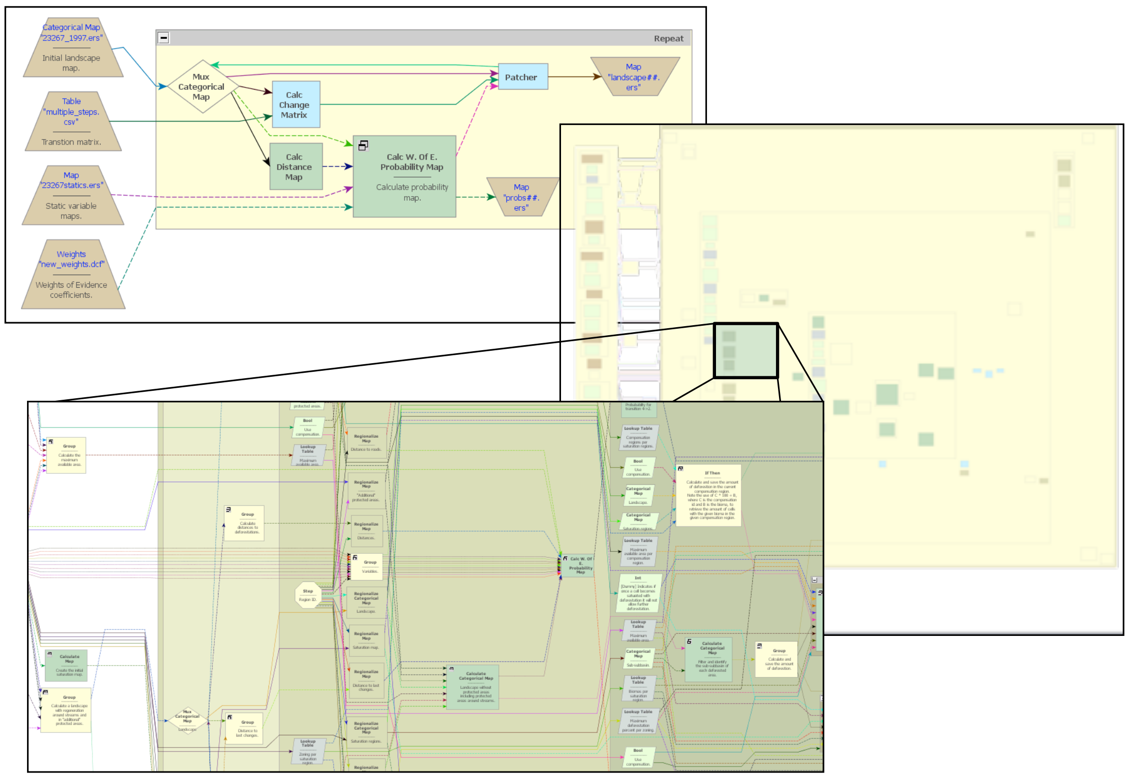


Figure 1.1. A typical screenshot of an EGO Script program. Components can be expanded in more complex views by the user.

2006][3].

Dinamica EGO - EGO stands for Environment for Geoprocessing Objects - was created as an assemblage of components implemented in C++, called *functors*, which represent typical *map algebra operations* [Tomlin, 1990] and other other specific algorithms. Figure 1.1 shows a screenshot of an application implemented in Dinamica. What this application does is immaterial for this presentation. Each functor has a number of inputs, and produces a number of outputs. The edges that interconnect these ports determine how data flows in a Dinamica's application. The original Dinamica's design had one fundamental disadvantage: functors were complex components implementing complete algorithms. Given this coarse granularity, whenever Dinamica's users needed to implement new behaviors, they had to ask the developers of that framework to code new functors.

To circumvent this shortcoming, we have implemented, from 2012 and 2014, DinamicaVM, a virtual machine designed to make the Dinamica framework more flexible.

[3]`www.ipcc.ch/publications_and_data/ar4/wg3/en/ch9.html`

The goal of this work is to describe this virtual machine and its companion programming environment. DinamicaVM contains an instruction set, a library of external components, a scheduler, a garbage collector and an optimizer. The instruction set is built around four functors: *map*, Reduce, Window and While, plus functors that define simple operations such as And, Add, Mul, etc. *Map* and Reduce are typical functional-oriented patterns, today heavily used in parallel programming [Dean and Ghemawat, 2008]. Henceforth, to avoid confusion with the maps used as Dinamica's main data type, we shall call the *map* functor Apply. Window returns a neighbourhood within a map. While receives a map, and a state, and return zero or more new states which are a function of the input map.

Dinamica's main data type is the *raster map*. Raster maps (or *rasters*) are regular grids of pixels called cells. A raster has a cell type wich can be float or integer with variable bit length. Map files may provide a variety of metadata that can describe map projection, coordinate systems, ellipsoids, datums and other geographic information. It can be seen as a 2D matrix of data along with a set of georeferencing information.

Tomlin [1990] described an algebra for the manipulation of geographic data and with a set of algebraic operations it is possible to produce a new raster map from other input rasters. The transformations proposed by this map algebra are separated into the groups: local, focal, global, and zonal. Each one differs from the other by the number of the cells a transformation works over: individual cells, neighbourhoods, entire maps or all cells with the same value.

It is not the goal of this work to describe the relation of Dinamica EGO and Tomlin's algebra, but to present our virtual machine for the current Dinamica EGO environment. As we state in Chapter 3, Dinamica provides general operators, but also specific components that are not present in map algebra. Indeed, it is possible to describe all map algebra transformations using Dinamica's operators, but the opposite is not true. We do not provide a proof for this statement since our focus is on the DinamicaVM components and its optimizations.

Applications built on top of DinamicaEGO manipulate very large data: maps having $70K \times 70K$ cells are not uncommon [Soares-Filho et al., 2014]. However, Dinamica's programming environment has been designed with focus on expressivity, not efficiency. *EGO Script*, the graphical programming language that embodies this environment, ensures referential transparency; hence, fostering a functional, side effect free, user experience. In order to ensure that such abstractions can be implemented efficiently, Dinamica applies a number of optimizations onto chains of primitive components, after these elements are linked together, but before they are deployed in the runtime system. These optimizations boost Dinamica's ability to deal with large

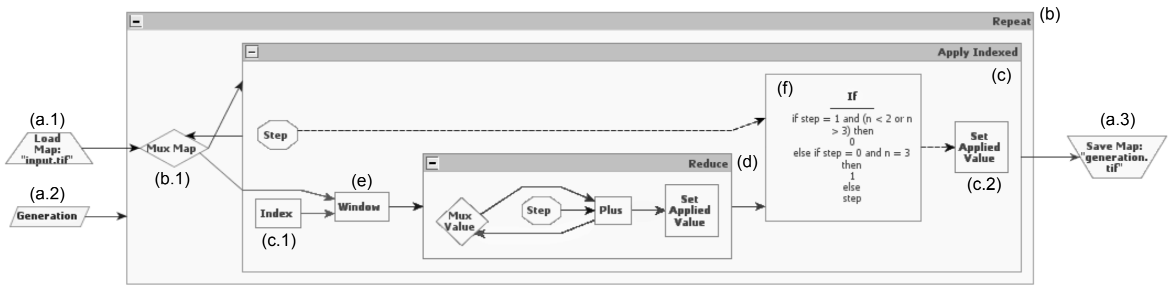


Figure 1.2. The implementation of Conway's Game of Life in Dinamica EGO. Letters in parentheses are not part of the original screenshot.

datasets dramatically, and in some cases, can deliver speedups of over 100x.

In Chapter 3 we introduce, the semantics of each of the four core building-blocks of Dinamica's applications. The architecture that these four components define finds no equal in other systems built with similar purpose, as we explain in Chapter 2. In Chapter 4 we describe the optimizations that ensure that these components run efficiently. Some of these optimizations are not new: they have already being implemented in functional languages [Wadler, 1988]. Nevertheless, we revisit them under the light of a virtual machine customized to handle maps and tables that represent geographic entities, i.e. a domain specific language. For instance, even though cache optimizations are well studied, we claim the cache-related transformations from Section 4.2 as original contributions of this work. A paper about these techniques and the virtual machine description has just been submitted. Furthermore, our techniques to address the tension between parallelism and memory allocation, which we address in Chapter 5, have been described for the first time in our paper [Ferreira et al., 2012] and improved since it was published.

1.1 Dinamica in One Example

In this section we present Dinamica EGO through an example application. This example uses the core components of our virtual machine that we describe in Chapter 3. It is possible to note how these components can be combined to build a program in the Dinamica EGO environment. More examples can be found in the Dinamica EGO Guidebook [4].

We illustrate the basic elements of Dinamica EGO via the implementation of Conway's Game of Life [Gardner, 1970]. Figure 1.2 shows a screenshot of this imple-

[4]`http://www.csr.ufmg.br/dinamica/tutorial/tutorial.html`

mentation. The game happens on a two-dimensional grid of square cells. Each cell can be either active or inactive. The state of all the cells in a grid determine a *generation* of the game. Generation $g + 1$ is a function of generation g. The state of cell i at generation $g + 1$ is determined by the state of this cell's neighbours, at generation g. The neighbourhood of a cell i is the 3×3 grid centered at i, excluding i itself. If this neighbourhood contains 2 or 3 active cells, i will be active in the next generation, otherwise, it will be inactive. Conway's Game of life is the canonical example of cellular automaton. Dinamica uses, among other techniques, different cellular automata to model land evolution due to human occupation [Soares-Filho et al., 2002].

The application in Figure 1.2 reads two inputs: an original map (a.1), plus an integer indicating how many generations of the game will be produced (a.2). Its output is the map after the final generation (a.3). A Repeat functor (b) produces the successive generations of the game. Repeat is a specialization of a more general functor called While, which we describe in Section 3.4. Repeat may either read a new map, or work on data that it sends back to its input port. This feedbacking is implemented by a functor called Multiplexer (b.1). Multiplexers are equivalent to the ϕ-functions so ubiquitous in compiler analyses and optimizations [Cytron et al., 1991].

We use an Apply functor (c) to produce generation $g + 1$ of the game, given generation g. This component applies some operation on each cell of the input map according to an iterating index (c.1). This operator does not work in-place, unless the runtime environment optimizes it, as we will see in Chapter 5. That is, Apply's output is a copy of the input map. In this particular example, we are using each Apply's index i to derive a Window (e) of 3×3 cells centered at i. A Reduce operator sums up the number of active cells on each neighbourhood. If this neighbourhood contains 2 or 3 active cells, then i will be active in the next generation, otherwise it will be inactive.

Some of the functors used in this example deal with data-structures. For instance, Window reads a map, plus an index, and returns a neighbourhood within the map. Other functors operate on individual data. For instance, the if container (f) implements a conditional expression made of several smaller components, which we have not shown for the sake of readability. These components, e.g., Equal, And, Or and LessThan, implement unary and binary operations. Users program applications in Dinamica EGO by combining these operators. In particular, the control flow of a Dinamica EGO program is determined by how the different instances of Apply, Reduce, Window and While are interconnected. The chapter 3 provides more details about each of these components.

Chapter 2

Literature Review

In this chapter we state related work present in the literature. We divide the relations in three areas. First we approach other geomodeling tools. Secondly, we give the reader an overview to other dataflow programming languages. Last, we present how the literature have been ensuring referential transparency in the presence of destructive updates. The latter is important due to our copy minimization optimization described in Chapter 5.

2.1 Other Geomodeling Tools

There exist many different tools that support the development of land use models. Some of them enjoy commercial success; others are popular in the academia. We are aware of four frameworks having a moderate to large user base that might compete with Dinamica in the geomodeling niche: ArcGIS [1], Idrisi [2], Metronamica [3] and PCRaster [4]. A direct comparison between all these tools is not possible, because they use different algorithms to perform simulations. Furthermore, there is not a common benchmark suite that they all can handle. Nevertheless, there exist a few limited studies comparing some of these tools. As an example, Pérez-Vega et al. [2012] have compared Dinamica and Idrisi, not from a performance perspective, but from the point of view of the accuracy of each modeling algorithm. Dinamica is substantially faster, and more accurate in some situations. Idrisi's algorithm, based on neural networks, is more accurate in others. Mas et al. [2014] compared Dinamica EGO in relation to CLUE-S and two Idrisi models. It presents the possibilities and limits of each software regarding the

[1] `http://www.esri.com/software/arcgis`

[2] `http://clarklabs.org/`

[3] `http://www.metronamica.nl/`

[4] `http://pcraster.geo.uu.nl/`

modeling process. Flexibility and user friendliness is also considered. Furthermore, a virtual case study is presented to illustrate the applications.

Metronamica [RIKS, 2012; van Delden et al., 2005] is a set of simulation models that run coupled to the Geonamica software environment. It is not as verstile as a programming language, but provides several functionalities: a land use model, a zoning tool, regional model, a set of spatial indicators, transport, macro-economics and demographics. Users have been developing these and other kind of models in Dinamica's framework.

PCRaster [Karssenberg et al., 2010] is a software framework used to build a variety of environmental models (geography, hydrology, ecology, rainfall-runoff, vegetation competition, slope stability, etc). It allows data assimilation in addition to process-based stochastic spatio-temporal modeling. It provides a script language to the development of models by the user and also building blocks and analytical functions for manipulating raster GIS maps. This script language is called PCRcalc [Wesselung et al., 1996] and give to the user a library of operations for model construction. A Python extension is also supported for a more advanced development. PCRcalc is similar to the EGO Script textual language, but PCRaster does not have a graphical user interface for interactive programming. Similarly to Dinamica's MapViewer, PCRaster deploys a data visualization tool called Aguila [Pebesma et al., 2007].

Idrisi and ArcGIS are geographic information systems software with several functionalities. Both have plugin support and can be extended with the Land Change Modeler - LCM - software [Gontier et al., 2010]. This application has been used for several environmetal models and simulations. Most of them REDD - Reducing emissions from deforestation and forest degradation. The Land Change Modeler is not a laguage, but a tool that allows model customization through parametrization in the software interface. Despite being powerful, it is not general as Dinamica EGO or PCRaster. Idrisi also have another model called CA-MARKOV that we describe later.

The Pérez-Vega et al. [2012] work compares Dinamica and the LCM modeling tools. This comparison points out the quality of the models built in each software and not the technical differences of the applications. The authors built two spatially explicit models in order to obtain reports on development and assessment of maps of change potential applied to a Tropical Deciduous Forest in western Mexico. They use Dinamica EGO weights of evidence method in the first model and neural networks in Idrisi's Land Change Modeler (LCM) in the second one. The evaluation of the models are via the Relative Operating Characteristic and Difference in Potential technique. Dinamica gives a better result in the "per transition" level, i.e., in the results of each time stamp in the model iterations. However, when an overall change potential map is obtained combining the information of each transition they have a more precise result with LCM. The authors attribute this better result to the neural networks. These "are able to express the simultaneous change potential to various land cover types more adequately than individual probabilities obtained through the weights of evidence method." On the other hand, it is hard to use if a variety of simulation scenarios is pursued. The Land Change Modeler is limited in this situation because it is hard to modify the relationship between explanatory variables and change potential in the model. Several variables are used to calibrate both models: slope, distance to settlements, distance to roads, distance to agriculture, and land tenure.

Mas et al. [2014] presents a broader comparison between Dinamica, CLUE-S and two models present in Idrisi: CA-MARKOV and the Land Change Modeler. CA-MARKOV [Paegelow and Olmedo, 2005] uses Markov chain matrices in the modeling process in the first step of the model: determine the quatity of change. To allocate these changes, that is the second step, it uses cellular automata and suitability maps. CLUE-S (Conversion of Land Use and Its Effects at Small regional extent) [Verburg et al., 2002; Verburg and Overmars, 2009] simulates the competition and interactions between the different land use changes types and uses mainly logistic regressions to make analysis of location suitability for this. The authors point LCM and CLUE have a more rigid structure with a fixed LUCC modeling flow procedures, while Dinamica and CA-MARKOV due to the large library of operators. Moreover, regarding ease of use the authos claim that "programming [Dinamica and CA-MARKOV] is easy even for users without previous programming experience because a graphical interface is provided in which operators can be dragged and linked dynamically to integrate feedback and iterative operations". Some user friendliness topics are presented describing available documentation, input files types, and OS support. Finally, a virtual case study shows each software modeling capabilities and accuracy.

2.2 Dataflow Programming Languages

Dinamica EGO implements a dataflow programming environment. Quoting Daniel Hils [Hils, 1992], "The central concept of the data flow model is that a program can be represented by a directed graph where nodes represent functions and where arcs represent the flow of data between functions." The idea of representing computation as graphs has been studied at least as early as 1966 [Bohm and Jacopini, 1966]. For a comprehensive survey about data-flow programming languages we recommend the following works: Johnston et al. [2004]; Whiting and Pascoe [1994]; Hils [1992]. We are not aware of any dataflow language that supports geomodeling and landuse simulation in particular. However, this paradigm has been employed in the most varied domains, ranging from music [Levitt, 1986] to image processing [Tanimoto, 1990], and has achieved commercial success [Johnson and Jennings, 2001].

Dataflow programming languages emerged due to the necessity of a language for dataflow hardware architechtures [Whiting and Pascoe, 1994; Johnston et al., 2004]. Conventional imperative programming languages was hard to compile for these hardwares. In turn, the first interest with dataflow architechtures was to explore massive parallelism. Researchers used to think that von Neumann hardware would not be able to achieve the same performance due to a global program counter and global updatable memory. Although, these dataflow architectures never surpassed the von Neumann hardware due to a too fine grain execution model, among other aspects. Nevertheless, the dataflow concept and the dataflow languages remained due to is powerful abstraction and sometimes enhanced productivity [Baroth and Hartsough, 1995]. There is a huge set of textual and visual dataflow languages. A selection of these is described next.

An MIT student, Weng [1975], has described one of the first dataflow languages: The Textual Data-Flow Language - TFDL. Its features are: compile time deadlock detection, translated into dataflow format, formalized semantics. No data structures were available and only two data types were present: Booleans and integers. Programs in TFDL are lists of modules that can be recursive, but iterations were not possible. Modules may have assignments, conditional statements or calls. Since it was the first effort to a textual description of dataflow graphs, TFDL become an important reference for the development of other dataflow languages, including discussions about communicating modules and streams implementations.

LAU [Gelly, 1976; Comte et al., 1978] was a language developed for the LAU computer architechture based on the static dataflow model. A LAU program, with its data-driven semantics, have statements for simple assignment, parallel assignment using the *expand* operator, decision using the *case* expression, and iteration with the *loop* operator (the *old* keyword was used to keep the single assingment rule). The ability to encapsulate data and operations was availabe, what is a feature present in object oriented languages. The *create* operator can be used to define an object attributes and operations.

An important set of the dataflow languages is the group of dataflow *visual* programming languages. Dinamica EGO is part of this set, providing a complete graphical user interface that allows the user to build models with no restrictions compared to Dinamica's textual language. Visual languages are those possible to program using visual components in a software. Usually, drawing the graph that represents the program where nodes are operations and edges are the paths for the data. We describe some dataflow visual programming languages next.

Davis [Davis, 1974, 1979] described the Data-Driven Nets (DDNs), a graphical language to program the DDM1 computer [Davis, 1978]. The programs consist of a collection of cells and a set of directed data paths interconnecting these cells within a cyclic graph structures. Data is typed and travels along these paths: message, vector, and scalar. DDN cells can be of different types in the DDMl computer machine language. Possible constructs are: procedure calls, conditionals, arbitration, iteration, synchronization, distribution of data items, and deterministic merging. Although a visual programming language, DDN is very low level as pointed by the author himself, and anyone should program directly in this language. However, key concepts could be presented without the use of a textual language (iteration, procedure calls, and conditional execution, e.g.).

Davis [Davis and Lowder, 1981] also described a higher level language called GPL. The intention was to create a more practical higher-level version of DDN. The authors tried to provide a system with intuitive clarity where the program is a graph itself and not just a representation of it. GPL programs have two types of functions, atomic and macrofunctions. The former are simpler indivisible functions and the latter can be expanded to subgraphs allowing structured programming. Macrofunctions also allows recursion. The GPL environment provide facilities for debugging, and visualization. If desired, text-based programming can be used. In these graphs, arcs are typed and handle tuples, files, and functions as values. It is believed that GPL was the first fully functional high-level graphical programming system to be implemented [Whiting and Pascoe, 1994].

LabView is probably the most famous dataflow visual programming language and has achieved commercial success [Johnson and Jennings, 2001; Santori, 1990]. It provides a virtual workbench interface used for data collection from lab instruments and data analysis in laboratories. This interface is suited for no professional programmers. The user builds a program connecting predefined functions that are represented by boxes with icons, using arcs for data paths. Program modules called virtual instruments are formed by a set of functions and connections. The interface also provides iterative constructs and a form of stepwise refinement for the programmer. LabView possess a sequential execution construct that allows the user to set a list of functions to run sequentially instead of obeying the normal dataflow firing rules. We are not aware of any other dataflow language with this feature.

ProGraph [Matwin and Pietrzykowski, 1985; Cox et al., 1989; Cox and Smedley, 1996; Green and Petre, 1996; Mosconi and Porta, 2000] is a general purpose dataflow visual programming language. It keeps the principles of the dataflow concept but also provides object-oriented programming functionalities. The user can define functions (called methods) using dataflow diagrams and these methods belong to classes. Modular abstraction is available: graphs can be condensed into single nodes and expanded if needed. Iteration is available in two forms. The first one is similar to a *map* function and applies an operation to all elements in a list, and return a new list. The second one is a *while* construct allowing iteration until a condition is satisfied.

Another general purpose language is Show and Tell [Kimura et al., 1986; McLain and Kimura, 1986]. It was created to teach programming for school children. As all dataflow visual programming languages, Show and Tell programs are made of boxes. These boxes can be of different types and represent functions constants, variables, iterators, and records. The data flows in the edges between boxes and the types available are real numbers, integers and files. The if-then statements obey to a special concept when combined with a Boolean value. This combination is called inconsistency. Such structure does not allow the data flow to continue after the inconsistecy box. In Dinamica EGO it does not happen explicitly, but a flow can be hindered in a conditional functor or in a iteration structure that iterates zero times. This situation may lead to an error if not properly treated by the user in the program code. Differently from Dinamica's ports, Show and Tell's ports are not suited for the operators inputs and outputs. Ports here are only for iteration and file boxes. These ports are similar to the *mux* functors in Dinamica EGO and are used to accumulate the last computed value in a iteration and transfer it to the next one. In Show and Tell sequential ports accumulate and transfer one value and parallel ports are used to accumulate and transfer a collection of values.

Another programming world related to dataflow is the stream programming. Also, several works have been described in this area. As example there is the StreamIt programming language [Lamb et al., 2003]. This is a domain specific language for the digital signal processing world [Oppenheim and Schafer, 1975]. Amarasinghe et al. [2005] describes it as a high-level, architecture independent programming language for stream programming. The authors claim improved programmer's productivity and program robustness within the streaming domain. Also, it is intended to serve as a common machine language for tile-based processors. Its compiler aims to perform novel stream-specific optimizations to achieve the performance of an expert programmer. They present the language syntax with some examples. *Filter* is the main type and it behaves as a functor in Dinamica EGO. Different stream structures are supported such as pipeline, splitjoins and feedbackloop. These structures are obtained with connections between filters.

Some optimizations for the StreamIt environment are specific for linear filters, i.e., filters whose outputs can be expressed as an affine combination of their inputs [Lamb et al., 2003]. In StreamIt each filter has its own address space and communicates with its neighbors using FIFO queues. The authors have implemented interesting optimizations in the environment. One of them is similar to the fusion optimizations we describe in Section 4.1. It works in a set of steps: first, with a linear dataflow analysis that extracts an abstract linear representation from imperative C-like code. It detects automatically if a user-defined filter is linear. Second, linear filter combination and linear pipeline collapsing. Finally, a domain specific optimization that transforms filters from time domain to frequency domain, converts data through a FFT, apply the new filter and inverse the FFT to get the data back. The idea of this optimization is that some filters are more efficient in the frequency domain and require less operations to compute.

Sermulins et al. [2005] presents three cache aware optimizations for stream programs. The work is in the context of the synchronous dataflow model where a program is a graph with actors (functions) are the vertices that communicate over channels (edges). These actors are independent (parallel processing) and the communication rates between actors are known at compile time (synchronous). These are features of a system that implement the pipelining parallelization model. It is not the case for Dinamica EGO, and we can set this as a future work. The three optimizations are (i) execution scaling, (ii) cache aware fusion, and (iii) scalar replacement. Execution scaling is a transformation that improves instruction locality by executing each actor in the stream graph multiple times before moving on to the next actor. This improves data locality and does not allow a too small task for an actor reducing the amount of communication or number of data transfer operations in the execution time. Cache aware fusion combines adjacent actors into a single function. This allows the compiler to optimize across actor boundaries (this is the kind of interprocedural optimization we were looking for in Dinamica EGO when we implemented the fusion optimizations described in Section 4.1). The goal here is to never fuse a pair of actors that will result in an overflow of the instruction cache. Scalar replacement is a buffer management strategy and serves to replace an array with a series of local scalar variables. These can be register allocated, unlike array references, leading to large performance gains. This latter optimization can not be implemented in our environment since it is a virtual machine with high-level instructions described in Chapter 3. Although, as future work, native code can be generated with these characteristics.

Spring et al. [2007] presents an extension for Java called StreamFlex. It is a stream programming language that runs in a Java Virtual Machine combining stream processing code with traditional object-oriented components. This language was inspired by StreamIt, and by the Real-time Specification for Java. It was implemented in a RTSJ - Real-Time Specification for Java - version of a JVM called Ovm. They have different data-types that implement filters, channels, messages and other artifacts. The authors also shows collaborations in parallelism, zero-copy message passing and innovative implementation.

2.3 Ensuring referential transparency in the presence of destructive updates

In Chapter 5 we present an algorithm that allows the compiler to generate data overwrites in a Dinamica EGO program while keeping the referential transparency from the user point of view. There exists a large body of work related to destructive updates of aggregate data structures in functional programming languages [Coutts et al., 2007; Hartel and Vree, 1994; Korfiatis et al., 2011; Leshchinskiy, 2009; Odersky, 1991; Vries et al., 2008; Wadler, 1990]. All these works are based on the fact that an update is always safe if the updated data structure is never accessed after being overwritten. In general, the optimizer receives a sequence of function applications and tries to recycle storage locations between actual and formal parameters. There are many ways to ensure referential transparency in face of destructive updates. Some of these methods are implemented automatically, by the compiler [Korfiatis et al., 2011; Leshchinskiy, 2009], whereas others require the direct intervention of the programmer [Wadler, 1990], or the use of specific libraries [Coutts et al., 2007]. For instance, linear types admit destructive updates, but require every variable to be used exactly once [Vries et al., 2008]. The main difference between the algorithm that we present in Chapter 5, and these previous works is the fact that we solve a simpler problem: our entire control flow graph is known before a program executes. This is often not the case in functional languages, due to the possibility of passing functions as parameters. Consequently, our optimizer is allowed to change the order in which each functional component is evaluated. There is another difference: some algorithms, as Hartel *et al.*'s [Hartel and Vree, 1994], might allow the same data structure to be updated by different writers, as long as they write at different places. We, like the majority of the other works, do not attempt to carry out this type of optimization. We provide here a review of works that relates to our algorithm presented in Chapter 5.

Deforestation is one basic technique used in functional programming languages to eliminate data copies [Wadler, 1988]. Although limited, this algorithm ignited several researches leading to better results than this attempt. The algorithm is based on a set of transformation rules. Implemented for the Haskell programming language, it focus mainly in the *case* construct. These rules can transform some *case* commands in other declarations and may eliminate data copies. The author address problems with termination [Burstall and Darlington, 1977] that could only be solved with the use of a restrictive form of function definition [Ferguson and Wadler, 1988]. Chin [1990] and Marlow and Wadler [1993] tried to lift some of these restrictions, but several problems still present.

Wadler [1989] presents a theory that allows us to derive theorems from function types. The author points that parametricity is very important for this theory. Proposed theorems wouldn't hold for an specific typed function like $Int{*}{\rightarrow}Int{*}$, but just for general ones ($\forall\ X.X{*}{\rightarrow}X{*}$). The usefulness of the theorems is multifold. In general it is useful for algebraic manipulation of functions. The author presents some examples where it is possible to "push map through a function" that is an opportunity for optimizations. It happens since the earlier maps are solved in a program, the less lists will be kept alive. Consequently, less copies of a large data as a list will happen. Wadler could use some of this kind of theorems to formulate an algorithm for compiling pattern matching in functional languages, as he points.

Following this idea of transformations in a functional program, Meijer et al. [1991] presents a set of recursion operators associated with data types definitions. Some algebraic laws are derived to manipulate programs written using these operators leading to optimizations. The main data types used are lists and the operators defined are catamorphisms, anamorphisms, hylomorphisms and paramorphisms. These are very related to the basic functional programming operations specially map and fold. Other algebraic data types are presented such as functors in several forms (these are not Dinamicas' functors). Using the laws presented in the paper the authors shows some possible optimizations mainly for recursive functions. Other general transformations for *map* and $fold$ like operations are also shown. These algebraic manipulation techniques from Meijer et al. [1991] and Wadler [1989] inspired our copy minimization algorithm shown in Chapter 5 an also optimizations from Section 4.

Gill et al. [1993] claim to have a better technique than the deforestation [Wadler, 1988] to eliminate intermediate lists in functional programs. They introduce their technique that is more general and still simple to implement. This technique applies to most standard list consuming functions such as *sum*, *and*, *map*, $filter$, ++, etc. It is also applied for list comprehensions and may be applied to more complicated data structures like trees. The deforestation technique makes use of a set of rules to transform the program. They give an example with the map fusion optimization to show how their technique perform algebraic transformations can eliminate an intermediate list. The point is that instead of having a huge set of rules for all possible pairs of functions, they use only one "standardising the way in which lists are consumed, and standardising the way in which they are produced". This proposed standardization is to always use $fold$ to consume the lists and a certain *build* function to produce it. The algorithm was implemented in the Glagow Haskell Compiler.

This consumer producer model using $fold$ and *build* functions is restrictive. Hence, the optimizations are only available for functions written using these constructs. Fegaras et al. [1994] has generalized this to arbitrary data structures. The authors take the generic recursion schemes from Meijer et al. [1991] and generalize it to induct over any number of structures at the same time. These structures do not need to be of the same type. This new induction scheme lifts a theorem that allows generic promotion. With this theorem is possible to use a normalization algorithm to optimize programs through deforestation, partial evaluation, loop fusion, and intermediate data structures elimination. These optimizations are applied in a term language defined by the authors.

Takano and Meijer [1995] goes even further. In this work they use the Acid Rain Theorem [Hu et al., 1996] and generalizes the $fold$ and *build* restrictions to any algebraic data types. Their approach is based on hylomorphisms [Meijer et al., 1991]. The contribution of this work is to show that the acid rain theorem can be stated as fusion rules for hylomorphisms. Hence, with these rules, they provide a new algorithm to eliminate intermediate data structures based on deforestation, but more powerful than previous works [Fegaras et al., 1994; Gill et al., 1993; Wadler, 1988]. As an example, this method deforests *zip* functions succesfully, what used to be an exception in other methods.

A different technique is used by Gopinath and Hennessy [1989]. In this approach, is computed the location where an expression is evaluated. This location is called *target*. This *targeting* method comes from code generation techniques where it is used to reduce the number of moves between registers. In this case, by computing some values in certain registers overwriting previous values. In their approach they perform a liveness analysis to compute the targets of functions in a intra-procedural manner. After, if it is safe, call-by-value array parameters are converted to call-by-reference parameters, requiring inter-procedural analysis. Finally, a *synthetic* and a *inherited* targets are used to decorate each program expression. These two targes are used to decide when it is necessary to allocate a new storage or if a storage can be shared. We use a similar idea with a storage memory with slots as described in Chapter 5. We need to decide if a new slot is necessary or if we can overwrite an existing data in a previously allocated slot.

Here we close our literature review. We present a set of other geomodeling tools and its features comparing to Dinamica EGO. We also considered other dataflow programming languages serving different purposes. Last, we discuss copy minimization techniques in languages with referential transparency.

Chapter 3

The Dinamica Virtual Machine

In this chapter we describe the new Dinamica EGO virtual machine and its core components. Dinamica EGO runs on top of a virtual machine called DinamicaVM. Figure 3.1 shows a schematic view of this virtual machine, including its programming environment. Dinamica provides its users with a Graphical User Interface, which is implemented in Java. It is also possible to load and run applications via a suite of command line tools. These applications are ensembles of functors. This virtual machine uses a set of functors, which include Apply, Reduce, While and Window. It also provides a suite of library components, which exist either due to efficiency reasons, or to keep compatibility with applications built prior to Dinamica 2.4.

The runtime environment of Dinamcia EGO consists of an optimizer, and ahead-of-time compiler, a scheduler, an interpreter and a garbage collector. The ahead-of-time compiler converts shading expressions, i.e., expressions that will be applied on every cell of a map or table, into binary code. These expressions are defined by Dinamica's user through a syntax that we call *EGO Script*. Translation to binary works in two steps: first EGO Script commands are converted to C++ instructions. Then, these instructions are compiled into binary code by `gcc`. The scheduler sorts the functors topologically, and forwards this information to the interpreter. If the target machine has multiple cores, then the scheduler parallelizes the execution of functors according to their dependences. The memory occupied by data structures that are no longer used are reclaimed by a garbage collector, which is based on reference counting. Cyclic dependences are not a problem in Dinamica, as it is not possible to create circular structures in it. In this section, we shall explore the four key components that this virtual machine interprets: Apply, Reduce, Window and While. We also present a formal semantics model for EGO script.

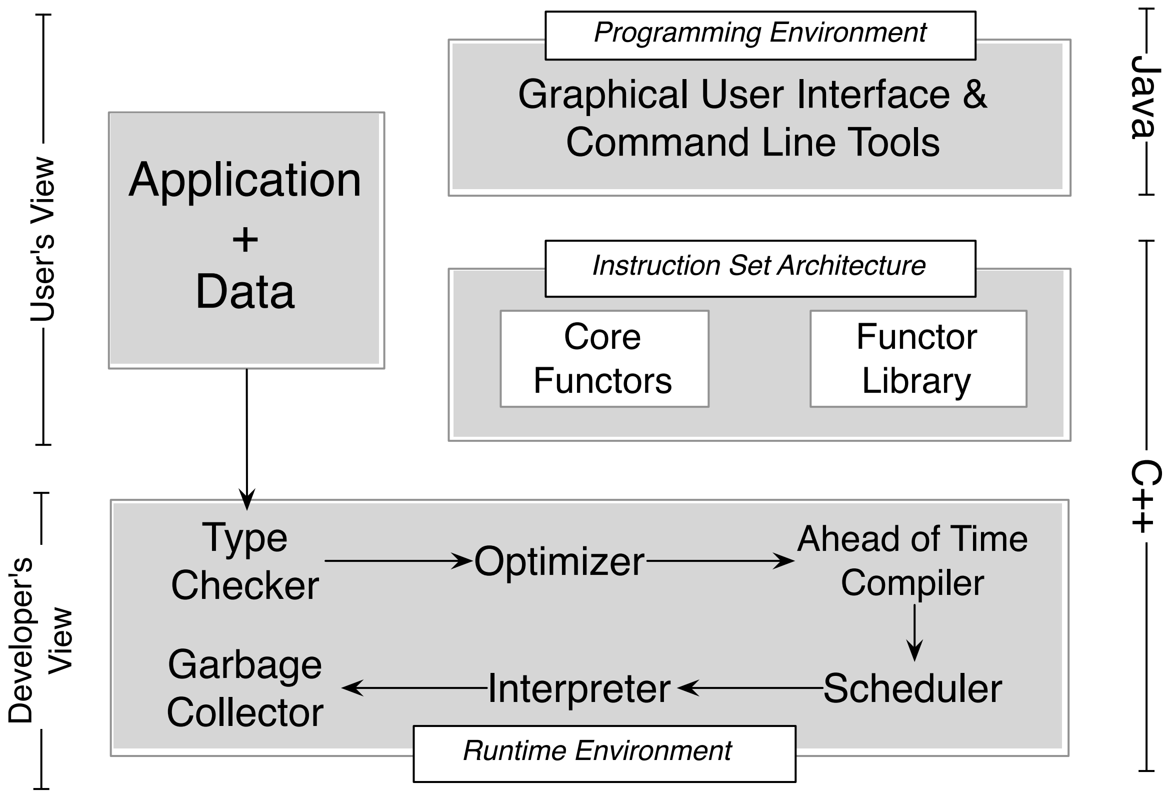


Figure 3.1. A schematic view of Dinamica EGO's Virtual Machine.

3.1 Apply

The Apply functor receives two inputs: (i) a map m, whose each cell has type t, e.g., $m : Map\langle t\rangle$, and (ii) a function $f : t \mapsto t$, that transforms the contents of each cell. The functor then applies f onto each cell of m, yielding a new map m'. Figure 3.2 (a) provides a visual representation of Apply when used in a program that increments every cell of a matrix of integers.

In Dinamica's Jargon, Apply is a Container. A Container is a functor that may incorporate other functors. In terms of implementation, Containers follow the "composite" design pattern [Gamma et al., 1995]. Containers, like every other functor, may have input and output ports. An Apply has only one input port, which receives the target map, and only one output port, which yields the new version of the map. However, contrary to regular functors, Containers have also *internal* ports, which are used to communicate with their components. Apply has only two internal ports. The first is Step, which returns the contents of a cell of the input map. This element keeps an internal state, in such a way that successive invocations of it return always different

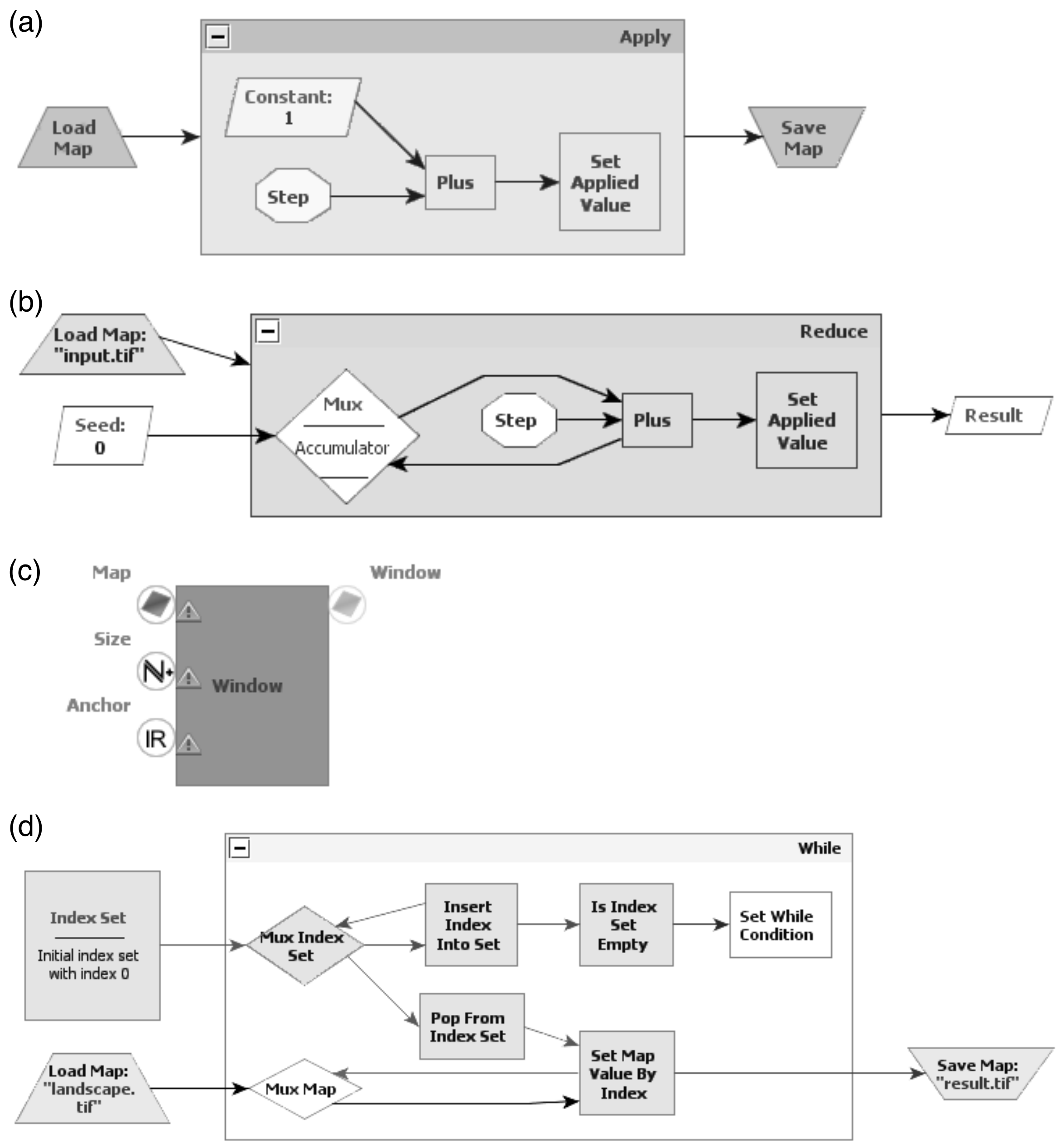


Figure 3.2. The four high-level functors in DinamicaVM's instruction set architecture. (a) Apply; (b) Reduce; (c) Window; (d) While.

elements, which come from contiguous positions in a column-major traversal of the map. The second internal port is Set, which causes a value to be written in a position of the output map that corresponds to the last index visited by Step in the input map.

An Apply is used to obtain a map that is a function of another map. Examples of its use are:

- Given an input map substitute all cells by the correspondents values in a table.

- Apply a normalization function producing a map where each cell is the result of a linear operation over the correspondent cell in the input map.

- Generate a map where each cell is replaced by the maximum adjacent neighbour.

Indeed, the applications are limitless and the programmer can use it to obtain great number of results.

Apply is one of the most used components in the Dinamica's ecosystem. Examples of its use include mapping coordinates into administrative regions such as countries, states and municipalities; mapping altitude into costs; mapping cells into slope values, which are calculated given these cells's neighbours, etc. Thus, it is very important that this component be implemented efficiently. Each iteration of Apply uses data that is completely independent from the data used by the other iterations. In the PRAM (*parallel random-access machine*) model, Apply can be implemented to run in $O(1)$. Thus, this functor is implemented to run in parallel.

3.2 Reduce

Reduce takes a map m of type $Map\langle t \rangle$, a binary operator $\oplus$, of type $t' \times t \mapsto t'$, and a seed s of type t'. It then produces a single value v of type t', such that $v = s \oplus m[0] \oplus m[1] \oplus \ldots \oplus m[n-1]$. In this case, $m[0], \ldots m[n-1]$ are all the cells in m, assuming that m has n cells. Figure 3.2 (b) shows an application that sums up all the elements in a map of integers, thus producing an integer as its result.

A user can utilize a Reduce in a variety of ways. The idea of the Reduce operation is to obtain a value that is a function of all values from a given input map. Therefore, this operator can be used, for instance, to obtain the maximum or minimum values from a map, to calculate the sum of all values or to retrieve the nearest point from a given coordinate.

Like Apply, Reduce is also a Container. It has one internal input, Set, which bears the same semantics as the component of same name in Apply. It has one internal output port, Step, which delivers to the internal functors the current value of the iteration. A functor Mux performs the function of the accumulator used to keep track of the current value of a reduction. This functor, if applied on a $n_1 \times n_2$ map, runs sequentially in $O(n_1 \times n_2)$. We can parallelize it for a few operations, which are commutative and associative, such as summation, multiplication, minimum and maximum. In this case, it runs in $O(\ln(n_1 \times n_2))$ in the PRAM model.

3.3 Window

Several applications implemented in Dinamica use small neighbourhoods within a map: finding the average slope of a coordinate, with regard to its neighbours; detecting borders, smoothing images, applying convolutions, finding minimum/maximal cost paths, etc. Therefore, Dinamica provides users with an operator to find neighbourhoods in maps: the Window functor, whose inputs and output are represented in Figure 3.2 (c).

Window has three inputs ports, which receive a map, the size of a neighbourhood's side and an anchor, e.g., the coordinate that is the center of the neighbourhood. It outputs a set of cells that constitute the neighbourhood. Windows can be used to build filters along with the Apply and Reduce operators. Besides filters it is possible, e.g., to check for each cell of a given map if a certain feature is present in a defined neighbourhood or to produce a statistical analysis such as a *kernel map*. The vast majority of all the algorithms built in Dinamica use squared neighbourhoods whose sides contain an odd number of elements, and whose center point is the anchor. Because this setup is so common, it is heavily optimized, as we explain in Section 4.2.

3.4 While

Most of Dinamica EGO components are stateless. Data structures are usually copies, instead of being modified in place, for instance. However, there are cases when keeping track of state is desirable for efficiency reason. For instance, a stateless functor to model the movement of a ball, under the force of gravity only, when let loose onto an elevation map, could lead to a formidable number of copies of the target map. Dinamica avoids such situations by providing users with a statefull functor – the While iterator. The graphical representation of this element can be seen in Figure 3.2 (d).

An While has one input port, which receives an *index set*. An index set is a collection of sortable elements that index a data-structure: coordinates on a 2D or 3D map, points on a line, rows in a table, etc. The While has an internal Step port, which keeps track of the elements in the index set still to be processed. While also has an internal Set port, which may update the index set with new elements. Thus, in practice, While implements worklists: as long as the worklist is not empty, this functor perform an action. DinamicaVM uses While, for instance, to implement searches by depth and breadth in maps. A very common index set consists in contiguous sequences of integer numbers. This case is so common that we have a specialization of While – the Repeat functor – optimized to use it.

The While operator is used to produce repetitions in programs or to process a map that is a function of an input map and a point within this map. An example would be to calculate a path for a road to be built given a cost map as input. Each iteration *draws* a new cell for the road that starts in a input point value and moves to the minimum neighbour in a 3x3 neighbourhood until a local minima is reached.

3.5 Specific Components

The While functor seen in Section 3.4, plus the binary and unary operators of Dinamica EGO define a Turing complete language. Turing completeness comes from the fact that these functors subsume the **While** formalism, typically used to illustrate programming language semantics [Nielson and Nielson, 1992]. Nevertheless, there are applications that do not translate easily into amalgamations of these few elements. In particular, there exist behaviors that our optimizations from Section 4 do not derive automatically. Thus, Dinamica EGO provides a few specific – higher-level – components which are not implemented as combinations of the four previously described functors. These components are also necessary to keep compatibility with applications developed prior to Dinamica v2.4, which did not use the virtual machine that we describe here. In this section we describe one of these components as an example of a high-level operator.

For instance, Dinamica EGO contains a functor called CalcCostMap, which constructs cost-surface maps out of raster images [Eastman, 1989]. The cost calculation problem is very common in land use simulations. The problem has two inputs: a friction map, and a map of source points. The outcome of a cost calculation is a map that tells us, for each cell, the minimum cost to reach one of the source cells. This problem emerges, for instance, whenever it is necessary to determine the paths that roads must traverse to link each interior city to a given set of harbours. The cost calculation problem is usually solved via chaotic iterations. We start with a solution map in which each cell is mapped to an infinitely large cost. Then, we iterate successive applications of the operator below, until a fixed point is reached:

$$\text{cost}(x) = min \begin{cases} \text{cost}(x) \\ \text{cost}(y) + \text{friction}(x) \\ \text{sqrt}(2) \times (\text{cost}(z) + \text{friction}(x)) \end{cases}$$

z	y	z
y	x	y
z	y	z

We have implemented the chaotic iterations as successive applications of four loops, whose iteration space is given in Figure 3.3(a). Each of these loops is parallelized

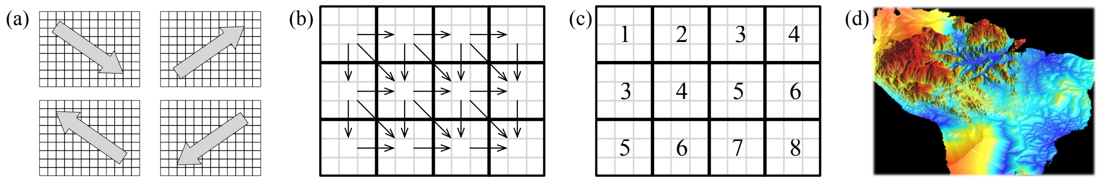


Figure 3.3. (a) The four loops that implement the chaotic iterations of the CalcCostMap functor. (b) Dependencies between tiles in the first loop: upper-left to lower-right. (c) Tiles with the same number can be processed together in the first loop. (d) Example of cost map that Dinamica produces.

independently. Figure 3.3(b) shows the pattern of dependences in the first loop, which traverses the map from the upper-left corner towards the lower-right corner. The execution runtime has a predefined number of available workers. Each worker has a task queue and can run a single task at a time. Tiles that must be processed are organized as a digraph of pending tasks. Tasks become eligible to run after all their dependencies have been processed. If a thread is idle, then it reclaims a tile that has no pending dependencies. This pattern continues until all the tiles have been processed. If the task queue of a processor becomes empty, then it might steal work from the queue of other processor. If a thread cannot steal any task, then it votes for the end of the computation. The computation terminates when a consensus is achieved among all the workers.

In this chapter we have presented our virtual machine and its components: Apply, Reduce, Window, While and other specific elements. With these operators is possible to write programs in Dinamica EGO and take advantage of the optimizations we describe in the next chapter.

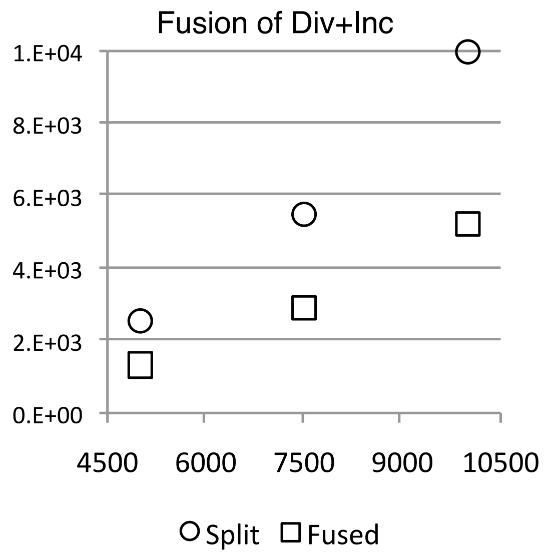
Chapter 4

Optimizations

In this chapter we present the optimizations implemented in the new DinamicaVM. These optimizations are divided in two groups: fusion optimizations and window optimizations. We also show the performace improvement through some case studies.

In order to be accepted by its users, the Dinamica Virtual Machine had to be at least as efficient as the original implementation of Dinamica's runtime, which was used until Dinamica v2.4, last released in 2014. The key to achieve this efficiency are optimizations. Not only the implementations of Apply, Reduce, While and Window are highly engineered, but also the way that these components interact is optimized. All the optimizations that we describe here, except the prefetching from Section 4.2 [1], are applied after an application has been type checked, but before its modules start to run. EGO Script's type system is static, i.e., types are known before an application starts running. Furthermore, this language does not support the dynamic loading of components, like PHP or JavaScript do. Therefore, we know the size of each map cell that is manipulated within an application, and we have a complete view of the dependence graph between components. This knowledge is important to generate code for the routines that read data, and move data between different functors. In this section we briefly touch the most important transformations that DinamicaVM applies onto its building blocks before an application runs. All the numbers that we show alongside the description of the optimization have been obtained in an Intel Core i5 with clock of 2.67GHz and 8GB of RAM.

[1]Prefetching is part of the implementation of Window; it does not requires any program transformation.



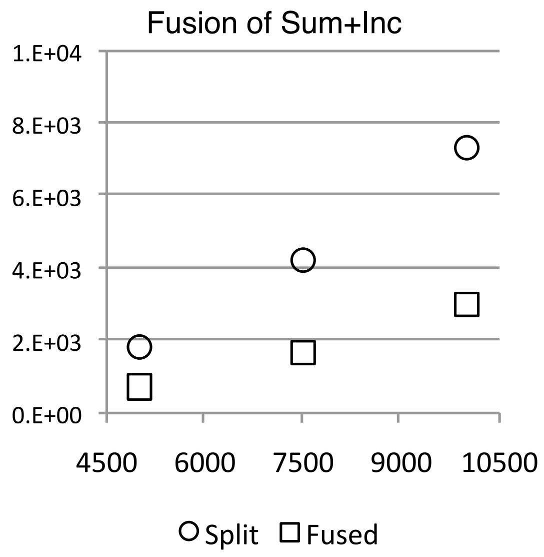


Figure 4.1. Example of fusion. The Apply operator is always the increment function, and the Reduce operator is the sum of integers.

4.1 Fusion

Fusion is a transformation that we implement onto combinations of Apply + Apply, and Apply + Reduce. This optimization is common in functional languages [Wadler, 1988]. It consists in combining the operators used by different functors in the following way:

$$\begin{aligned}\text{Apply } f \ (\text{Apply } g\ m) &= \text{Apply } (f \circ g)\ m \\ \\ \text{Reduce } s\ f\ (\text{Apply } g\ m) &= \text{Reduce } s\ f'\ m \\ \text{where } f &= \lambda(x, y)\ .\ x \oplus y \\ \text{and } f' &= \lambda(x, y)\ .\ g(x) \oplus y\end{aligned}$$

Function fusion is not a new idea of ours. If fact, we are using a very limited form of fusion, as we only apply it to two combinations of functions. More extensive implementations have been described, for instance, by Jones *et al.* [Gill et al., 1993]. Nevertheless, our simple implementation of function fusion is enough to speed up some of Dinamica's applications dramatically.

Figure 4.1 illustrates some of these performance gains. In this example we are using three very simple instances of Apply and Reduce:

$$\begin{aligned} \mathsf{Inc}\ m &= \mathsf{Apply}\ (\lambda x\ .\ x+1)\ m \\ \mathsf{Div}\ m &= \mathsf{Apply}\ (\lambda x\ .\ x/2.17)\ m \\ \mathsf{Sum}\ m &= \mathsf{Reduce}\ 0\ (\lambda(x,y)\ .\ x+y)\ m \end{aligned}$$

In the figure we use random square matrixes of integers having sides of 5.0K, 7.5K and 10.0K cells. Without fusion DinamicaVM takes 9.940 seconds to $\mathsf{Div} \circ \mathsf{Inc}$ every cell of the $10^3 \times 10^3$ matrix. Once fusion is activated, this time drops to 5.222 seconds. In the case of Reduce, gains are of similar nature. It takes us 7.294 seconds to $\mathsf{Sum} \circ \mathsf{inc}$ the matrix with 10K rows without fusion, and 2.998 seconds if we use fusion. These gains are due to two factors: the elimination of intermediate data structures, and the improved locality. Concerning the first factor, fusion automatically eliminates the need to copy the map that the leftmost Apply produces. As for locality, the input map will be traversed only once instead of twice. Indeed, only one iteration is necessary for any sequence of applications of the Apply functor, e.g.:

$$\mathsf{apply}\ f_1\ (\ldots(\mathsf{apply}\ f_n\ m)\ldots) = \mathsf{apply}\ (f_1 \circ \ldots \circ f_n)\ m$$

Fusion's improvements are proportional to the complexity of the kernel operator used in Apply or Reduce. The more complex is the computation used inside these functors, less performance gains we shall observe. The composition below illustrates this trend:

$$\mathsf{Apply\ Normalize}\ (\mathsf{apply\ calcSlope}\ m)$$

Normalize is a simple linear function of the input value, but the calcSlope operation is a substantially more complex functor present in the Dinamica EGO library. It applies a Reduce over the output of a Window for each index in the input map. For a 7500×7500 input map, fusion gives us 6% of speedup in this example.

4.2 Window Optimizations

Window is a heavily used functor; thus, it is natural that it be optimized. DinamicaVM applies two optimizations on Window: prefetching and unrolling. The latter is only applicable on 3×3 instances of Window. Prefetching avoids unnecessary trips to main memory in order to collect the pieces of a squared window view. Unrolling removes unnecessary control flow from the most common type of view that we have observed in Dinamica's applications.

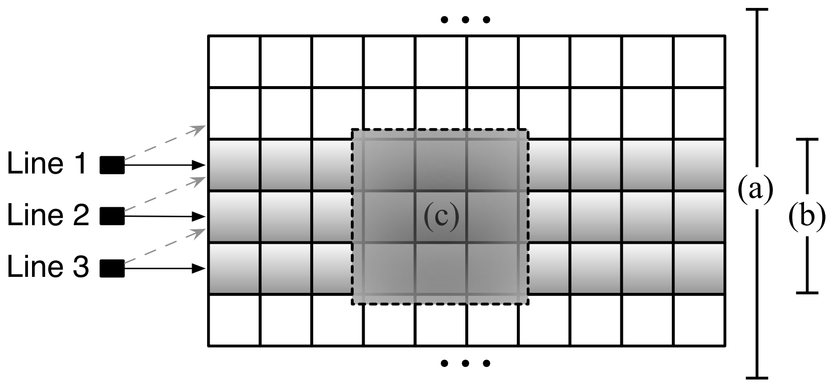


Figure 4.2. The three-lines cache. The dashed arrows show line pointers in the previous iteration of Window. The solid arrows show the pointers in the current iteration. (a) Input map. (b) Cached lines. (c) Center of 3×3 window.

4.2.1 Prefetching

Most of the applications that use Window slide it over an image in row-major order, that is, starting from the upper-left corner of an image, and going to its lower-right corner. This pattern is so common because it is the default order in which Apply and Reduce evaluate the elements of a map. Our optimizer ensures that each cell of a Window is read only once from main memory, if Window is used in that way. To ensure this property, we *pre-fetch* the lines that will be traversed by Window.

Figure 4.2 illustrates this approach for a 3×3 instance of Window. In this example, each time Window is called, it reads nine elements of the input map. Instead of fetching this data when Window is created, we pre-fetch three entire lines of the map, and let Window slide on these lines. Once Window reaches the rightmost border of the image, we discard the topmost line, and read one line more from main memory. If Window works with submatrices of n rows, then we should, in principle, keep n lines in cache. However, most of the applications available in the Dinamica's ecosystem work with 3×3 windows. Thus, we chose to work with only three lines at a time. Consequently, larger instances of Window may lead to multiple trips to the main memory.

The prefetching is only necessary for maps that cannot fit entirely in the L0 cache. This is usually the case in Dinamica, as maps are very large, and each of their cells contains a non-negligible amount of data, which include colour patterns and geographic information. In the absence of this optimization, a $n \times n$ Window causes each – non-border – map cell to be read n^2 times. Usually data in the same row of Window are fetched only once to the L0 cache; however, data may be fetched more times if it happens to be read as part of different rows of Window.

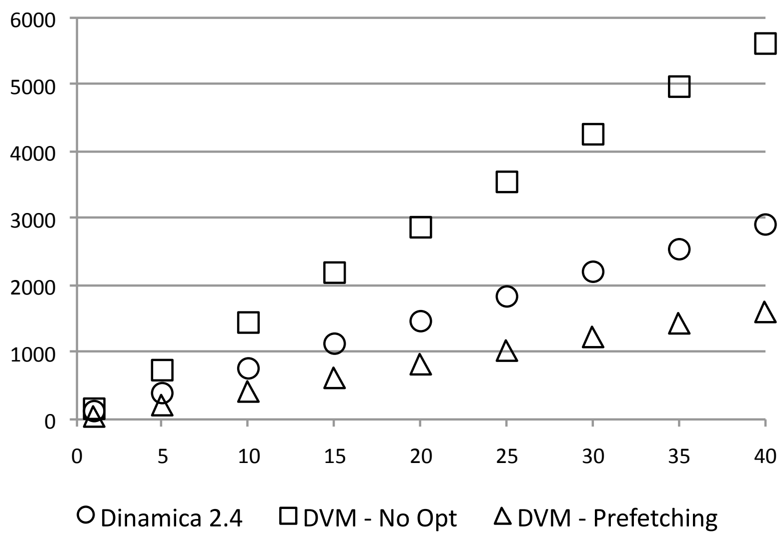


Figure 4.3. Performance gain due to prefetching in an image smoother.

4.2.2 Performance Improvement due to Pre-fetching

Figure 4.3 shows the performance of three different instances of an image smoothing algorithm. The algorithm uses a 3×3 convolution matrix that does simple average to implement smoothing. The smoothing filter returns, for a given cell i, the average of all the immediate neighbours of i plus i itself. The three instances of the algorithm are:

- **Library** – the algorithm was implemented using a monolithic smoothing filter available in Dinamica's library.
- **DVM - No Opt** – our algorithm, built as the following combination of functors: $\mathsf{Smooth}\ m = \mathsf{Apply}\ ((\mathsf{Reduce\ Average}) \circ \mathsf{Window})\ m$
- **DVM - Prefetching** – the previous implementation, with prefetching enabled in DinamicaVM.

Figure 4.3 varies the number of times that the image is smoothed. Each of these times requires one application of the smoothing algorithm. As we observe in Figure 4.3, our optimization speeded up Window by a factor that reached 3.9x for 40 successive applications of the smoothing algorithm. It even improved on the library component, which has a much more monolithic design. Our optimized version of image smoothing is 1.7x faster.

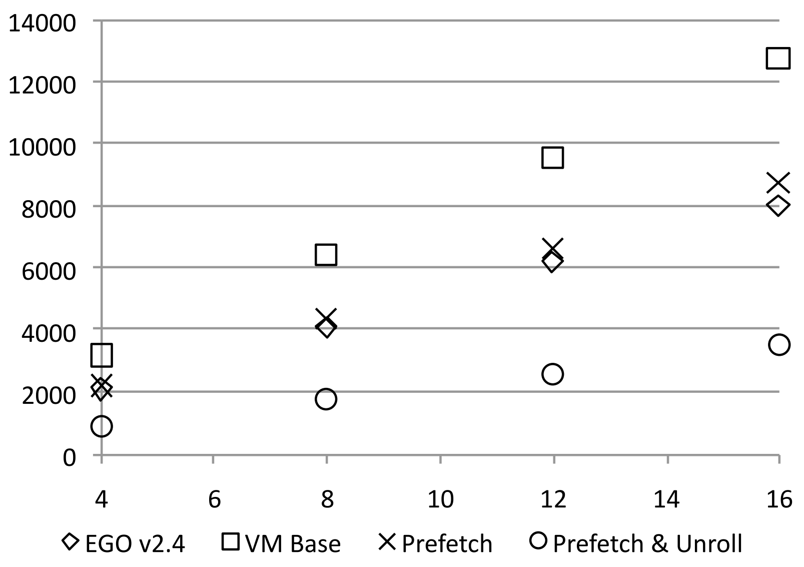


Figure 4.4. Performance gains in Conway's Game of Life (Section 1.1) due to unrolling and prefetching. X-axis is the number of generations of the automaton, and Y-axis is runtime, in msecs.

4.2.3 Unrolling

The most used type of Window is a 3×3 squared view of a map, with anchor in the center. Because this pattern is so common, we use a special implementation of it, which has no control flow. This implementation reads a chunk of memory that is large enough to fit each one of the nine indices to be processed. It then divides this memory into nine pieces, and fills up the positions in the map view with them. The size of memory that must be read is determined by the ahead-of-time compiler, before Window is invoked, but after the type of its input is already known.

Figure 4.4 shows the performance gains obtained due to unrolling and prefetching when applied on the implementation of Conway's Game of Life. This application was discussed in Section 1.1. To provide some perspective to the reader, we show the runtime of the implementation of Conway's automaton in Dinamica 2.4, before the virtual machine was released. In this case, the game is implemented with a set of functors from the library. The series "VM Base" shows our application running on the virtual machine without either prefetching or unrolling. In this case, DinamicaVM is 59% slower than Dinamica v2.4's implementation of Conway's game. However, once we turn on optimizations, we see substantial gains: Unrolling already puts DinamicaVM's times on pair with v2.4's results. And the combination of unrolling and prefetching makes us 57% faster than the old version of Dinamica. In other words, the two optimizations makes our virtual machine 3.6x faster.

The set of optimizations presented in this chapter are twofold: fusion and window optimizations. The former acts over sequences of operators Apply or sequences of Apply followed by a Reduce. The latter is divided in two items: prefetching and unrolling. These optimizations takes our virtual machine to a competitive level in relation to the previous version of the application.

Chapter 5

A Semantics Model of EGO Script and the Copy Elimination Optimization

In this chapter we describe a core optimization we have implemented in the Dinamica EGO evironment. This optimization eliminate data copies and allows the execution environment to overwrite data structures while keeping referential transparency that is a key feature in the language. To present this optimization we first define a formal semantics model for the EGO Script language. After, we present the copy elimination algorithm and two case studies to show the arising performance improvement.

In order to provide users with a high-level programming environment, one of the key aspects of Dinamica's semantics is referential transparency. This is a feature frequently found in dataflow programming languages [Johnston et al., 2004] The components in a Dinamica EGO script must not modify the data that they receive as inputs. A typical way to ensure referential transparency is to rely on immutable data structures [Sondergaard and Sestoft, 1990]. If the contents of a data structure must be updated, then the whole data is copied into a new memory location. However, this alternative is too expensive in Dinamica, because our data structures – maps and tables – tend to be very large. Therefore, a copy minimization algorithm is essential to allow these applications to scale. Removing these copies is a non-trivial endeavor, inasmuch as minimizing such copies is a NP-complete problem, as we show in Section 5.2.

We demonstrate the effectiveness of our optimization via two case studies. The first one 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

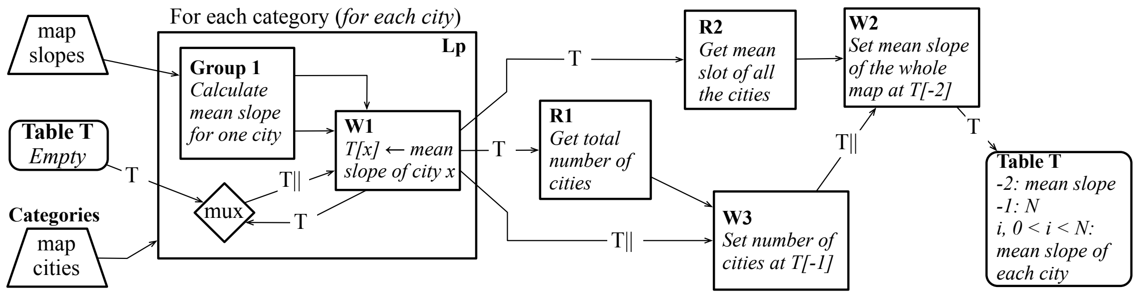

map slopes
For each category (for each city)
Lp
Group 1
Calculate mean slope for one city
W1
T[x] ← mean slope of city x
Table T
Empty
T
T||
mux
T
Categories
map cities
T
R1
Get total number of cities
R2
Get mean slot of all the cities
W2
Set mean slope of the whole map at T[-2]
T||
T
T||
W3
Set number of cities at T[-1]
Table T
-2: mean slope
-1: N
i, 0 < i < N: mean slope of each city

Figure 5.1. 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 data.

our copy elimination algorithm boosts the performance of Dinamica EGO by almost 100x. The second model - simpler - performs an statistical analysis of map values getting data such as sum of values, variance and standard deviation.

5.1 Dinamica EGO in Another Example

As we did in Section 1.1 we present here another application implemented in Dinamica EGO. This example will guide us to state the copy minimization algorithm in Section 5.4. Although simple, this application contains some of the key elements that we will discuss in the rest of this chapter. More examples can be found in the Dinamica EGO Guidebook [1].

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 slope 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 5.1. 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 order in which functors must execute is determined by the runtime environment, and should obey the dependencies created by the data channels.

[1]`http://www.csr.ufmg.br/dinamica/tutorial/tutorial.html`

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 have in each city entry the average slope of that city. 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 our optimization, each component that could update data should replicate this data before writing on it. 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 depart from this model by moving data replication to the channels, instead of the components, and using a whole program analysis to eliminate unnecessary copies.

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, we can remove all the three copies in the script from

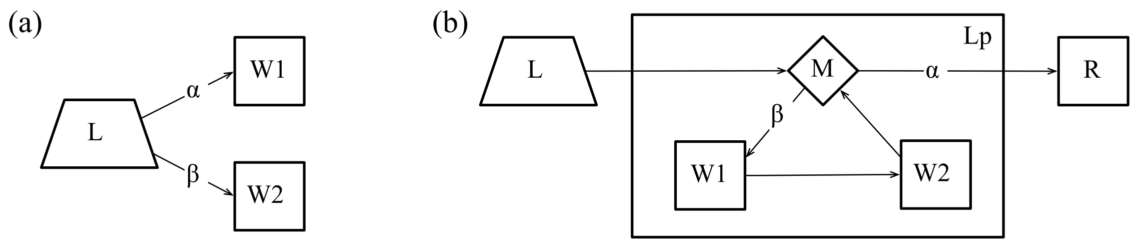


Figure 5.2. Two examples in which copies are necessary.

Figure 5.1. However, there are situations in which copies are necessary. Figure 5.2 provides two such examples. A copy is necessary in Figure 5.2(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 5.2(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.

5.2 The Core Semantics

In order to formally state the copy minimization problem that we are interested, we will define in this section a core language, which we call μ-Ego. We also present the operational semantics for this language and formally define the Storage Minimization problem we are solving. Futhermore, np-completeness proofs for the problem and variations are provided.

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, and I is the index of the storage area 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 5.3 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) consists in evaluating sequentially N copies of the scheduling S_l. We let the symbol

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

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

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

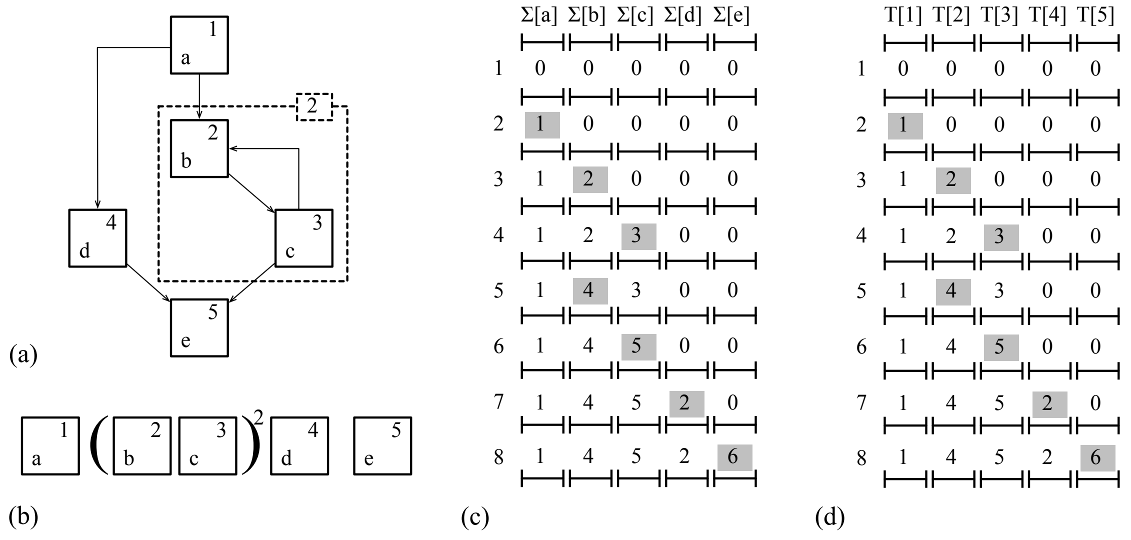
Figure 5.3. The operational semantics of μ-Ego.

@ 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 a 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 5.4 illustrates the evaluation of a simple μ-Ego program. The digraph in Figure 5.4(a) denotes a program with five components and a container. We represent each component 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 5.4(a) determine the predecessor relations among the components. Figure 5.4(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 5.4(c) shows the store Σ, and Figure 5.4(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 5.5(d) also identify which component is been evaluated at each iteration.

5.3 The Copy Elimination Problem

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



	Σ[a]	Σ[b]	Σ[c]	Σ[d]	Σ[e]
1	0	0	0	0	0
2	1	0	0	0	0
3	1	2	0	0	0
4	1	2	3	0	0
5	1	4	3	0	0
6	1	4	5	0	0
7	1	4	5	2	0
8	1	4	5	2	6

	T[1]	T[2]	T[3]	T[4]	T[5]
1	0	0	0	0	0
2	1	0	0	0	0
3	1	2	0	0	0
4	1	2	3	0	0
5	1	4	3	0	0
6	1	4	5	0	0
7	1	4	5	2	0
8	1	4	5	2	6

Figure 5.4. 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.

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 5.3.1, which formally states the storage minimization problem.

Definition 5.3.1 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 5.5 produces the same result as the canonical evaluation given in Figure 5.4; 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 5.6 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 5.6, because these two different components reuse the same storage unit.

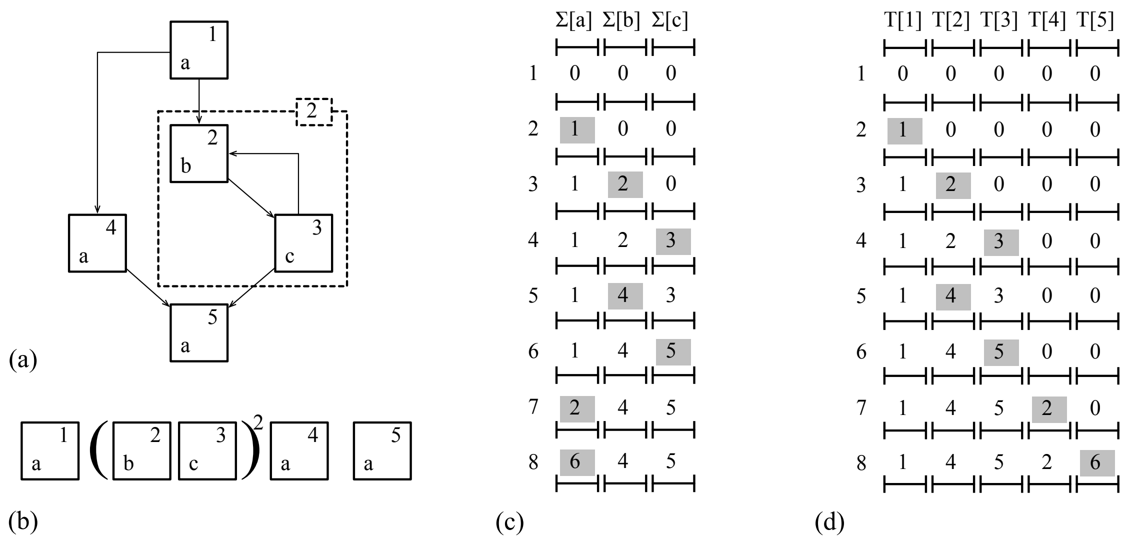


Figure 5.5. Evaluation of an optimized μ-Ego program.

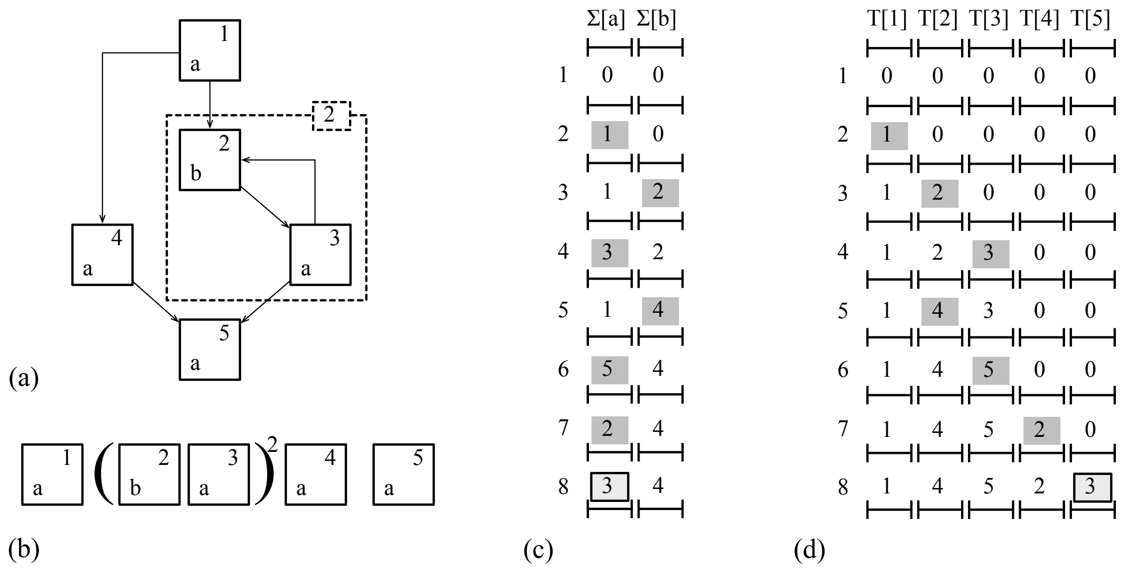


Figure 5.6. Evaluation of an μ-Ego program that does not produce a canonical result.

5.3.1 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 [Golumbic, 2004]. The reduction 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.

Figure 5.7 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 5.3.2 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 5.3.2 *A tight coloring of the interval graph provides a tight index assignment to the μ-EGO program.*

proof: first, we show that a coloring of the interval graph gives a valid index assignment. Let c' and c" be two components in S, such that c" comes after c'. If their corresponding intervals overlap, then these components cannot be given the same index, as the store of c' will be read after the store of c" is written. On the other hand, if these intervals do not overlap, then every component that reads the value of c' is scheduled to run before c". Second, we show that this assignment is optimal. A tight coloring of an interval graph, which we denote by χ, is bounded by the size of its largest clique [Golumbic, 2004]. Let s_c be a common point to every interval in this clique. We know that there exist χ components scheduled to execute after c, by construction. Hence, χ also represents a tight index assignment to S. $\square$

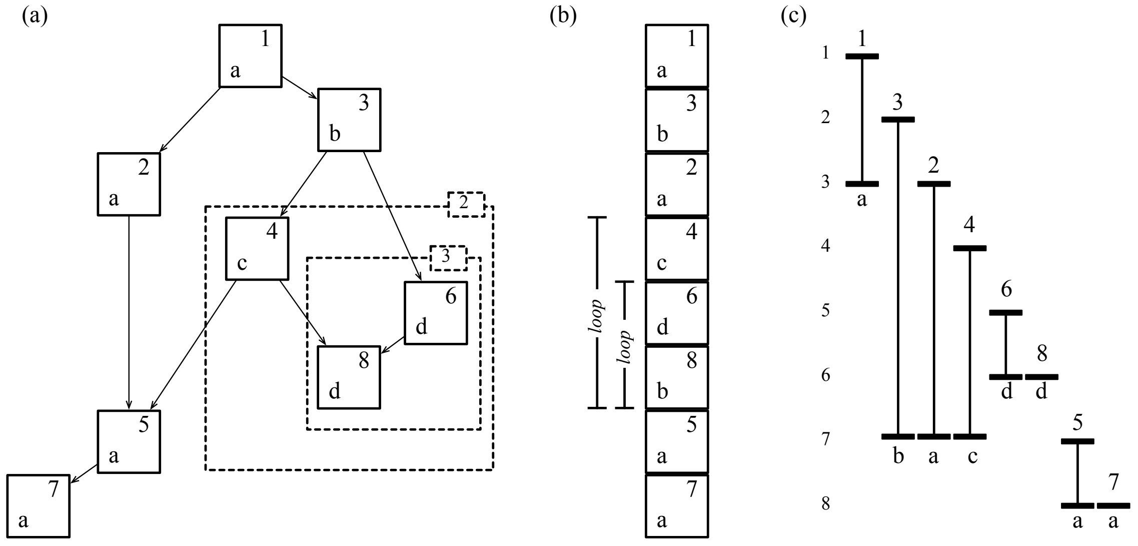


Figure 5.7. Reducing Smfs to interval graph coloring for schedulings without back-edges. (a) The input μ-Ego(program. (b) The input scheduling. (c) The corresponding intervals. The integers on the left are the orderings of each component in the scheduling.

5.3.2 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* [Garey et al., 1976]. 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 [Garey et al., 1980].

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 5.8 illustrates this reduction. We claim that solving Smfs to this μ-Ego program is equivalent to coloring the input graph.

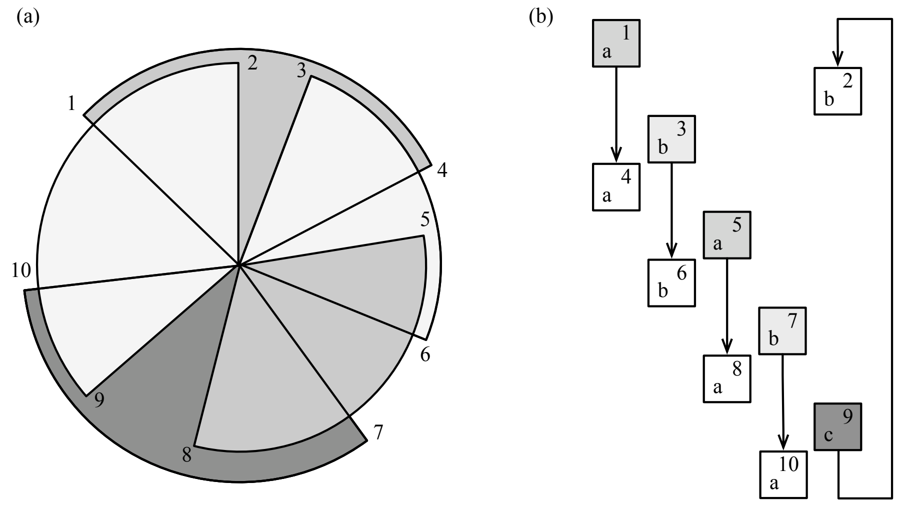


Figure 5.8. Reducing SMFS to circular-arc graph coloring for general schedulings. (a) The input arcs. (b) The corresponding μ-EGO program.

Theorem 5.3.3 *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: Let $a_i = [w, x]$ and $a_j = [y, z]$ be two arcs from P_g, in which w, x, y, z are the integers that R associates with them. R defines four functors c_w, c_x, c_y and c_z. If C is a map of arcs to colors that solves P_g, then we let $\Sigma[c_w] = C(a_i)$, and $\Sigma[c_x] = C(a_j)$. If $w < y < x$, then a solution of SMFS must assign the components c_w and c_y different indices, and the equivalent arcs get different colors because they overlap. Similarly, if a_i and a_j overlap, then a solution of P_g must assign them different colors. By construction, we have that $w < y < x$; hence, a solution of SMFS must also assign c_w and c_y different indices. $\square$

5.3.3 Copy replication is NP-complete with open scheduling on DAGs.

Another problem of our interest is the storage minimization with an open scheduling on directed acyclic graphs. Definition 5.3.4 states this problem.

Definition 5.3.4 STORAGE MINIMIZATION WITH OPEN SCHEDULING [SMOS]

Instance: *the components in a μ-*EGO *program forming a directed acyclic graph, plus a natural K, the number of storage cells that any evaluation can use.*

Problem: *find an scheduling S, plus a map Σ from storage indices to components in S. The scheduling must respect the topological ordering of the DAG formed by the functors, and Σ must use K or less indices. The evaluation of S must be canonical.*

There are situations when by relaxing the restrictions on a problem we can make it easier. For instance, register allocation is NP-complete in general [Chaitin et al., 1981]. On the other hand, if we relax the restriction that each variable must stay always in the same register, than we can solve it in polynomial time [Pereira and Palsberg, 2005]. This observation could lead one to believe that by not fixing the scheduling we could have a polynomial-time exact solution to Smos. Unfortunately, this is not the case: Smos is NP-complete. The reduction, in this case, is to the formulation of the register allocation problem, as defined by Ravi Sethi [Sethi, 1973].

Sethi defines register allocation over the so called *Expression Directed Acyclic Graphs* (e-DAGs). Each node of such a structure represents a program variable; that is, a virtual location that must be assigned a physical storage space. There is an edge from node v to node u if the computation of v depends on the computation of u. Sethi has defined the register allocation problem as an instance of the following game, over an e-DAG, and an infinite supply of stones:

1. Place a stone on a leaf of the DAG.

2. Pick up a stone from a node

3. If there are stones on every predecessor of a node x:

    a) place a stone on x – or –

    b) move a stone to x from one of its predecessors.

The following problem, as proved by Sethi [Sethi, 1973], is NP-complete: given an e-DAG, and an integer K, is there a computation over the game that starts with no stones on the nodes, and ends with a stone on every root, using less than K stones? From this observation we provide a straightforward reduction from Sethi's register allocation to Smos in Theorem 5.3.5. Given an e-DAG G, we create an μ-Ego P program whose scheduling follows any topological ordering of G. For any node $n \in G$ we define a functor $f_n \in P$, and for each edge in G from n to m we create an edge in P from f_n to f_m.

Theorem 5.3.5 *The* Smos *problem is NP-complete.*

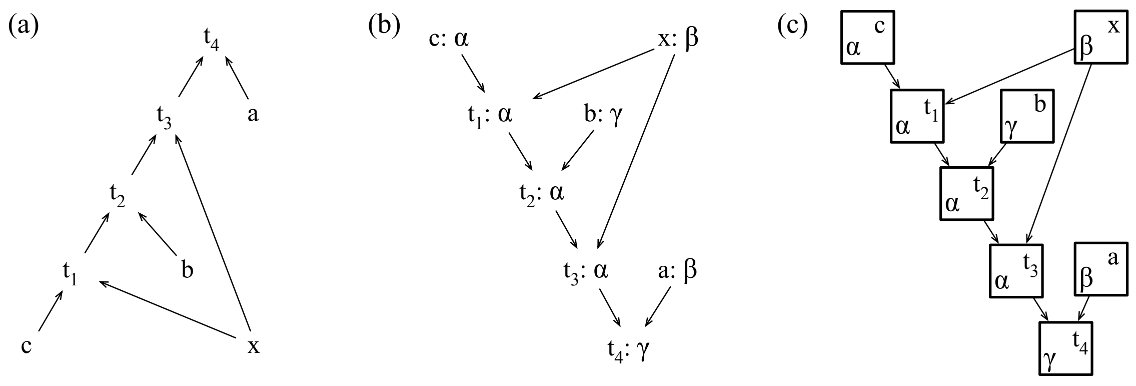


Figure 5.9. An example that illustrates how Sethi's programs can be translated to μ-Ego scripts. (a) The e-DAG that represents the expression $(c \times x + b) \times x + a$. (b) A solution to Sethi's game. (c) The corresponding μ-Ego program.

proof: From a reduction to Sethi's register allocation. First, we map a solution of Sethi's problem to Smos by using the name of each register as the index in Σ. We must show that an evaluation of this solution is canonical. We use induction on the position of each functor in S. By the time we apply the semantic Rule Func to evaluate a node n_f, we know that every predecessor of this node has already been evaluated, and this evaluation is canonical by induction. Lets assume that this functor has been assigned the index r, where r is the stone bound to the corresponding node in the e-DAG. If we placed r using Rule (1) of the game then this index has not been used so far, and writing on it will not overwrite the position used by any other functor. If we placed r using Rule (3), then every node that needs to read from r has already been evaluated, and we can reuse this location. Second, we map a solution from Smos to Sethi's allocation. Given that the evaluation of P follows a topological ordering of the graph formed by the predecessor relation, we have that it obeys the rules in Sethi's game. Finally, we can verify if a solution to Smos is canonical in time linear on the size of the μ-Ego script program. $\square$

5.4 Copy Minimization

In this section we present the algorithm that eliminates copies of data structures in a Dinamica EGO program. This algorithm is composed by a Data-Flow analysis and a criteria to eliminate copies. Finally, this section closes with a correctness proof for our criteria.

$$[\text{Df1}]\quad \frac{\text{channel}(N_1, N_2) \qquad \text{write}(N_2)}{\text{abs}(N_1, N_2, \{N_2\})} \qquad\qquad [\text{Df2}]\quad \frac{\text{abs}(N_1, N_2, A') \qquad \text{out}(N_1, A)}{A' \subseteq A}$$

$$[\text{Df3}]\quad \frac{\text{channel}(N_1, N_2) \qquad \neg\text{write}(N_2) \qquad \text{out}(N_2, A)}{\text{abs}(N_1, N_2, A)}$$

$$[\text{Df4}]\quad \frac{\text{channel}(N_1, N_2) \qquad \text{read}(N_2) \qquad \text{out}(N_2, A)}{\text{abs}(N_1, N_2, A \cup \{r\})}$$

Figure 5.10. Inference rules that define our data-flow analysis.

5.4.1 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 [Aho et al., 2006, p.608]. Figure 5.10 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 5.3, we also consider, for the sake of completeness, the existence of funtors that only read the data. If N is such a functor, than the predicate 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 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 5.11 shows the result of applying our data-flow analysis onto the example from Section 5.1. 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

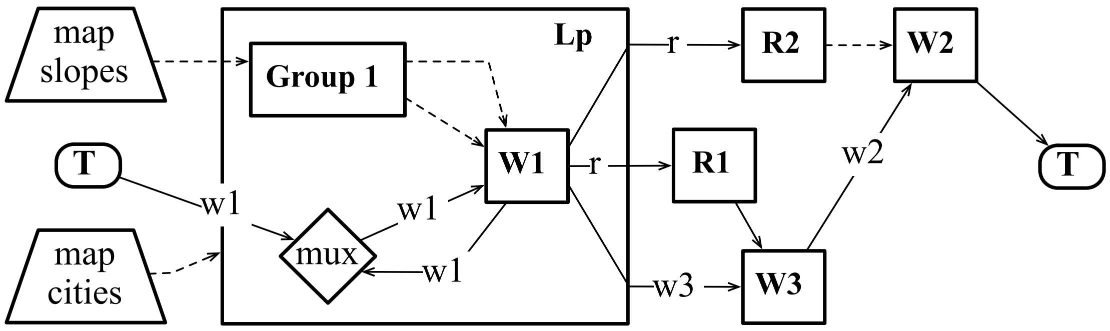


Figure 5.11. The result of the data-flow analysis on the program seen in Figure 5.1.

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 [Zadeck, 1984]. A partitioned variable problem can be decomposed into a set of data-flow problems – usually one per variable – each independent on the other.

5.4.2 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 5.12 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 5.12. 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 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 5.7(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

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

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

$$[\textsc{Dom}] \quad \frac{C = (\mathbb{N}, S) \qquad N \in S}{\text{dom}(C, N)}$$

$$[\textsc{Ori}] \quad \frac{\text{channel}(N, N_1) \qquad \text{channel}(N, N_2) \qquad N_1 \neq N_2}{\text{orig}(N, N_1, N_2)}$$

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

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

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

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

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

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

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

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

Figure 5.12. Criteria to replicate data in Ego Script programs.

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 5.7(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 5.11, 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 5.2(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., $\text{pred}(N_2, N_1)$ 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 5.1. 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**.

5.4.3 Correctness

In order to show that the rules in Figure 5.12 correctly determine the copies that must be performed in the program, we define a correctness criterion in Theorem 5.4.1. A μ-Ego program is correct if its evaluation produces the same output table as a canonical evaluation. The condition in Theorem 5.4.1 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* $\vec{S}$ as the complete trace of component names observed in an execution of S. For instance, in Figure 5.5, we have $S = 1, (2,3)^2, 4, 5$, and we have $\vec{S} = 1, 2, 3, 2, 3, 4, 5$. We let $\vec{S}[i]$ be the i-th functor in the trace $\vec{S}$, and we let $|\vec{S}|$ be the number of elements in this trace. In our example, we have $\vec{S}[1] = 1$, $\vec{S}[7] = 5$, and $|\vec{S}| = 7$. Finally, if $p = \vec{S}[j]$ is a predecessor of the functor $\vec{S}[i]$, and for any $k, j < k < i$, we have that $\vec{S}[k] \neq \vec{S}[j]$, then we say that $\vec{S}[j]$ is an immediate dynamic predecessor of $\vec{S}[i]$.

Theorem 5.4.1 *The execution of an μ-Ego program (S, T, Σ) is canonical if, for any $n, 1 \leq n \leq |\vec{S}|$, we have that, for any predecessor p of $\vec{S}[n]$, if $\vec{S}[i] = p$ and $i, 1 \leq i < n$ is an immediate dynamic predecessor of $\vec{S}[i]$, then for any $j, i < j < n$, we have that $\Sigma[\vec{S}[j]] \neq \Sigma[p]$.*

proof: the proof is an induction on n. In the base case $n = 1$, and a scheduling of one element is canonical by definition, because there exists in this case the same number of indices in the output table T, and in the store Σ. We assume that the theorem holds up to the $n-1$-th element of $\vec{S}$. By induction, every predecessor of $\vec{S}[n]$ holds a canonical value. And by the hypothesis of the theorem, this value has not been overwritten by any component that precedes $\vec{S}[n]$ in $\vec{S}$. $\square$

We prove that the algorithm to place copies is correct by showing that each copy that it eliminates preserves the condition in Theorem 5.4.1. There is a technical inconvenient that must be circumvented: the Rules Cp1 and Cp2 from Figure 5.12 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 5.4.1.

Theorem 5.4.2 *The elimination of copies via the algorithm in Figure 5.12 preserves the correctness criterion from Theorem 5.4.1.*

proof: we negate each premise used in Rules Cp1 and Cp2 and show that the correctness criterion is preserved.

- $\neg\text{orig}(N, N_1, N_2)$: if the entire script consists of a single line of components, then no copy is necessary, given that a race condition cannot exist in this scenario.
- $\text{out}(N, \{r\}) \vee \text{out}(N, \emptyset)$: if no data is ever updated from by any channel that is reachable from N, then the correctness criterion is trivially maintained.
- $f_1 = N \vee f_2 = N$ in Rule Cp1. Lets assume, without loss of generality, that $f_2 = N$. If there is no path from f_1 back to N, then we can schedule N to always run before f_1. If there is a back-path, then N shall be alive in the out set of f_1. In case there are other nodes alive there, a copy will be created by Rule Cp1.
- $\text{pred}(N_2, N_1)$ in Rule Cp2. If $\text{pred}(N_2, N_1)$ is true, then it is possible to schedule N_2 to always execute before N_1. Given that N_2 does not write data, only read it, the correctness criterion is maintained for N_1.

$\square$

5.5 Experiments

In this section we show how our optimization speeds up Dinamica EGO via two case studies. The first one is the Hilltop detection algorithm used by Soares-Filho et al.

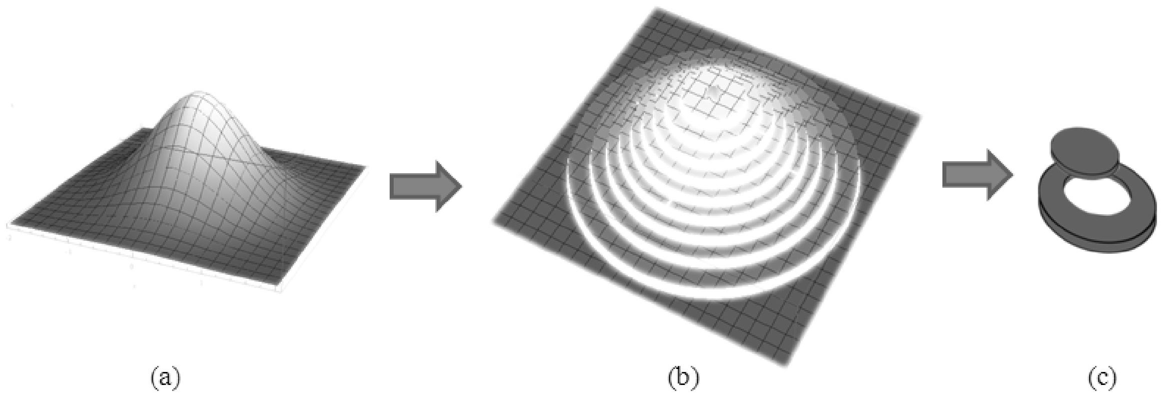


Figure 5.13. Hilltop detection. (a) Height map. (b) Normalized map. (c) Extracted discrete regions.

[2014] in a real world analysis. The second one, although simpler, still reflects real life uses of the application and state the importance of the optimization. The numbers that we present in this section were obtained in an Intel Core2Duo with a 3.00 GHz processor and 4.00 GB RAM.

5.5.1 First Case Study: Hilltop Calculation

The first case study comes from a model used to detect hilltops in protected areas, which is available in Dinamica's webpage. Figure 5.13 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, which is defined by the vertical resolution. Then the map is normalized and divided in discrete regions, as we see in Figure 5.13(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.

Table 5.1 shows the speedup that we obtain via our copy elimination algorithm. 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. Table 5.1

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

Table 5.1. V: Vertical resolution(m). D: Number of dynamic copies without optimization. T_u: Execution time without optimization (s). T_o: Execution time with optimization (s). R: Execution time ratio.

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 Table 5.1. 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 Table 5.1.

5.5.2 Second Case Study: Map Statistical Analysis

The second case study comes from a simpler model that derives statistics from a map, like a typical spreadsheet application does. Values from the map are first stored in a table. Then, the model iterates over this table, computing data such as number of non-null values, sum of these values, arithmetic mean, variance and standard deviation. From this table the model creates a new one that relates values to its frequency in the map.

Table 5.2 shows the behavior of this model in face of maps having different sizes. The figure shows that the higher the map dimension, the higher the number of copies, and the higher the execution time. This model had two operators performing data copies and both of them could be eliminated. For our largest example map, having 300x300 cells, we could improve execution time by more than 32x. Transposing maps into tables is a common programming pattern in Dinamica EGO today. However, before our optimization, only very small maps could be processed in this way.

In this chapter we presented a key optimization of the DinamicaVM that allows the environment to perform data overwriting while keeping the referential transparency. This optimization is presented in the company of a semantics model that

MS	D	T$_u$	T$_o$	R
100	20000	23	5	4.6
120	28800	44	7	6.2857
140	39200	77	10	7.7
160	51200	125	13	9.6153
180	64800	196	17	11.5294
200	80000	289	21	13.7619
220	96800	418	25	16.72
240	115200	580	29	20
260	135200	791	34	23.2647
280	156800	1083	39	27.7692
300	180000	1451	45	32.2444

Table 5.2. MS: size of map's side, in number of cells. Each map is a square matrix. D: Number of dynamic copies without optimization. T$_u$: Execution time without optimization (s). T$_o$: Execution time with optimization (s). R: Execution time ratio.

describes programs in our language as well as the copy minimization problem and its np-completeness.

Chapter 6

Conclusion

This work has presented DinamicaVM, the virtual machine that supports the execution of applications built on top of the Dinamica EGO geo-scientific framework. DinamicaVM differs from other virtual machines, such as Oracle's JVM or Microsoft's .NET, in a number of ways. In particular, DinamicaVM is not a general purpose virtual machine: it is tailored to run applications that process very large images representing cartographic maps. Furthermore, as we saw that DinamicaVM has a high-level instruction set, which includes components borrowed from functional programming, such as map (e.g., Apply), Reduce, that live among specific components for high performance algorithms.

We also formulated a formal semantics model for the EGO Script. This model were used to describe the copy minimization problem and its np-completeness. The copy minimization is important in order to allow the execution environment to perform destructive updates in the data structures, but preserve the referential transparency for the user point of view. We described an algorithm to solve this problem using compiler techniques such as data-flow analysis and stated the performance improvement through two case studies. Although simple, these case studies reflect real world situations and highlight the importance of the copy minimization optimization.

The optimizations presented could improve both individual components of the virtual machine and whole programs what makes this new VM is faster than the previous version of the software. This optimizations are, nowadays, part of the official distribution of Dinamica EGO, and are the key elements responsible for the high scalability of this framework. It also allows a more focused development of the system since the developers of the Dinamica EGO software can describe new optimizations concerning only a small set of instructions instead of a whole library of operations.

6.1 Limitations and Future Work

While the DinamicaVM is very powerful, it has limitations. The necessity of specific components as shown in Section 3.5 demonstrates that. In spite of the core components versatility, some operations are hard to describe using only these elements. Nevertheless, the use of high-level components allows the developers to implemen t particular optimizations and apply specific parallelization methods for the algorithms implemented in these operators. Regarding the copy minimization algorithm, the limitations are due to the np-completeness of the problem. Thereby, our method may not eliminate all possible data copies in a program.

As future work, there are a number of things that could be done. Although Dinamica EGO was made to run in regular computers, it would be interesting to provide the possibility to run simulations in clusters of computers. To do not make supercomputers necessary is important in order to make this geomodeling tool handy. However, if a user have a set of computers available for clustering, that would be a good resource to explore. Something that is not possible nowadays.

Also, there are other optimizations that can be implemented as future work. An example would be the reuse of computations performed by reduces. It is possible when a window slides linearly in a image and at each position a reduce is computed over the values in this window. Lets say that for each position the sum of the neighbours is being calculated, e.g. The reader may notice that the sum computed for each column of the window (except for the first one) will be recalculated in the next window positioning. A cache could be created and these computations reused as the window moves. Of course this optimization is not available for every operation a reduce can perform. A more formal definition of the problem is necessary.

6.2 Closing Remarks

This work lets us draw a number of conclusions which are also valid to other dataflow, functional, programming environments and runtime systems. First, it is possible to design and implement a very expressive instruction set, and still ensure that it can be executed efficiently. Key to this efficiency are the optimizations that we apply after typechecking the application, but before loading it up. Second, it is possible to make the most of the specific nature of applications that will run on the virtual machine with very positive results. In our case, the virtual machine is customized to handle well large images and tables, which are typical in geosciences. We believe that the techniques currently available in DinamicaVM can be of use in other systems that process large images or large matrices of data.

Software: Dinamica EGO is available on-line at:
http://www.csr.ufmg.br/dinamica.

Bibliography

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

Amarasinghe, S., Gordon, M. I., Karczmarek, M., Lin, J., Maze, D., Rabbah, R. M., and Thies, W. (2005). Language and compiler design for streaming applications. *Int. J. Parallel Program.*, 33(2):261--278. ISSN 0885-7458.

Baroth, E. and Hartsough, C. (1995). Visual object-oriented programming. chapter Visual Programming in the Real World, pages 21--42. Manning Publications Co., Greenwich, CT, USA.

Bohm, C. and Jacopini, G. (1966). Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366--371.

Burstall, R. M. and Darlington, J. (1977). A transformation system for developing recursive programs. *J. ACM*, 24(1):44--67. ISSN 0004-5411.

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., and Rodrigues, H. O. (2012). Committed carbon emissions, deforestation, and community land conversion from oil palm plantation expansion in west kalimantan, indonesia. *Proceedings of the National Academy of Sciences.*

Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. (1981). Register allocation via coloring. *Computer Languages*, 6:47–57.

Chin, W. (1990). *Automatic methods for program transformation.* PhD thesis, Imperial College.

Comte, D., Durrieu, G., Gelly, O., Plas, A., and Syre, J. C. (1978). Parallelism, control and synchronization expression in a single assignment language. *SIGPLAN Not.*, 13(1):25--33. ISSN 0362-1340.

Coutts, D., Leshchinskiy, R., and Stewart, D. (2007). Stream fusion: from lists to streams to nothing at all. In *ICFP*, pages 315--326. ACM.

Cox, P., Giles, F., and Pietrzykowski, T. (1989). Prograph: a step towards liberating programming from textual conditioning. In *Visual Languages, 1989., IEEE Workshop on*, pages 150–156.

Cox, P. and Smedley, T. (1996). A visual language for the design of structured graphical objects. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 296–303. ISSN 1049-2615.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490.

Davis, A. and Lowder, S. (1981). A sample management application program in a graphical data-driven programming language. *Digest of Papers Compcon Spring*, 81:162--167.

Davis, A. L. (1974). Data driven nets - a class of maximally parallel, output-functional program schemata. Technical report IRC Report, Burroughs, San Diego, CA.

Davis, A. L. (1978). The architecture and system method of ddm1: A recursively structured data driven machine. In *Proceedings of the 5th Annual Symposium on Computer Architecture*, ISCA '78, pages 210--215, New York, NY, USA. ACM.

Davis, A. L. (1979). Ddn's - a low level program schema for fully distributed systems. In *Proceedings of the 1st European Conference on Parallel and Distributed Systems*, pages 1--7, Toulouse, France.

Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107--113.

Eastman, J. R. (1989). Pushbroom algorithms for calculating distances in raster grids. In *Auto-Carto*, pages 288--297. ASPRS and ACSM.

Fegaras, L., Sheard, T., and Zhou, T. (1994). Improving programs which recurse over multiple inductive structures. In *In ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'94*, pages 21--32.

Ferguson, A. B. and Wadler, P. (1988). When will deforestation stop? In *In 1988 Glasgow Workshop on Functional Programming*, pages 39--56.

Ferreira, B. M., ao Pereira, F. M. Q., Rodrigues, H., and Soares-Filho, B. S. (2012). Optimizing a geomodeling domain specific language. In *Simposio Brasileiro de Linguagens de Programacao*. Sociedade Brasileira de Computacao.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0-201-63361-2.

Gardner, M. (1970). Mathematical games - the fantastic combination of john conway's new solitaire game life. *Scientific American*, 1(223):120--123.

Garey, M. R., Johnson, D. S., Miller, G. L., and Papadimitriou, C. H. (1980). The complexity of coloring circular arcs and chords. *J. Algebraic Discrete Methods*, 1:216--227.

Garey, M. R., Johnson, D. S., and Sockmeyer, L. (1976). Some simplified NP-complete problems. *Theoretical Computer Science*, 1:193--267.

Gelly, O. (1976). Lau software system: A high-level data-driven language for parallel processing. In *Proceedings of the International Conference on Parallel Processing*, New York, NY.

Gill, A., Launchbury, J., and Peyton Jones, S. L. (1993). A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223--232, New York, NY, USA. ACM.

Golumbic, M. C. (2004). *Algorithmic Graph Theory and Perfect Graphs*. Elsevier, 1st edition.

Gontier, M., Mörtberg, U., and Balfors, B. (2010). Comparing gis-based habitat models for applications in eia and sea. *Environmental Impact Assessment Review*, 30(1):8--18.

Gopinath, K. and Hennessy, J. L. (1989). Copy elimination in functional languages. In *POPL*, pages 303--314. ACM.

Green, T. R. G. and Petre, M. (1996). Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131--174.

Hajek, F., Ventresca, M. J., Scriven, J., and Castro, A. (2011). Regime-building for redd+: Evidence from a cluster of local initiatives in south-eastern peru. *Environmental Science and Policy*, 14(2):201 – 215.

Hartel, P. H. and Vree, W. G. (1994). Experiments with destructive updates in a lazy functional language. *Comput. Lang.*, 20(3):177--197.

Hils, D. D. (1992). Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing*, 3:69--101.

Hu, Z., Iwasaki, H., and Takeichi, M. (1996). An extension of the acid rain theorem. In *In T Ida, A Ohori, and M Takeichi, eds, Proceedings 2nd Fuji Int Workshop on Functional and Logic Programming, Shonan Village*, pages 1--4. World Scienti.

Huong, H. T. L. and Pathirana, A. (2011). 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.

Johnson, G. and Jennings, R. (2001). *LabVIEW Graphical programming*. McGraw-Hill, 1st edition.

Johnston, W. M., Hanna, J. R. P., and Millar, R. J. (2004). Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1--34.

Karssenberg, D., Schmitz, O., Salamon, P., de Jong, K., and Bierkens, M. F. (2010). A software framework for construction of process-based stochastic spatio-temporal models and data assimilation. *Environmental Modelling & Software*, 25(4):489 – 502. ISSN 1364-8152.

Kimura, T. D., Choi, J. W., and Mack, J. M. (1986). *A visual language for keyboardless programming*. Washington University, Department of Computer Science.

Korfiatis, G., Papakyriakou, M. A., and Papaspyrou, N. (2011). A type and effect system for implementing functional arrays with destructive updates. In *FedCSIS*, pages 879–886.

Lamb, A. A., Thies, W., and Amarasinghe, S. (2003). Linear analysis and optimization of stream programs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 12--25, New York, NY, USA. ACM.

Leshchinskiy, R. (2009). Recycle your arrays! In *PADL*, pages 209--223. Springer-Verlag.

Levitt, D. (1986). Hookup: An iconic, real-time data-flow language for entertainment. Technical report, MIT.

Marlow, S. and Wadler, P. (1993). Deforestation for higher-order functions. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 154--165, London, UK, UK. Springer-Verlag.

Mas, J.-F., Kolb, M., Paegelow, M., Camacho Olmedo, M. T., and Houet, T. (2014). Inductive pattern-based land use/cover change models: A comparison of four software packages. *Environ. Model. Softw.*, 51:94--111. ISSN 1364-8152.

Matwin, S. and Pietrzykowski, T. (1985). Prograph: A preliminary report. *Comput. Lang.*, 10(2):91--126. ISSN 0096-0551.

McLain, P. and Kimura, T. D. (1986). *Show and Tell user's manual.* Washington University, Department of Computer Science.

Meijer, E., Fokkinga, M., and Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124--144, New York, NY, USA. Springer-Verlag New York, Inc.

Mosconi, M. and Porta, M. (2000). Iteration constructs in data-flow visual programming languages. *Computer Languages*, 26(2-4):67 – 104. ISSN 0096-0551.

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., and Stella, O. (2009). The end of deforestation in the brazilian amazon. *Science*, 326:1350--1351.

Nielson, H. R. and Nielson, F. (1992). *Semantics with Applications – A Formal Introduction.* John Wiley and Sons.

Odersky, M. (1991). How to make destructive updates less destructive. In *POPL*, pages 25--36. ACM.

Oppenheim, A. V. and Schafer, R. (1975). *Digital Signal Processing.* Prentice-Hall. ISBN 9780132146357.

Paegelow, M. and Olmedo, M. T. C. (2005). Possibilities and limits of prospective gis land cover modelling a compared case study: Garrotxes (france) and alta alpujarra granadina (spain). *International Journal of Geographical Information Science*, 19(6):697–722.

Pebesma, E. J., de Jong, K., and Briggs, D. (2007). Interactive visualization of uncertain spatial and spatio-temporal data under different scenarios: An air quality example. *Int. J. Geogr. Inf. Sci.*, 21(5):515--527. ISSN 1365-8816.

Pereira, F. M. Q. and Palsberg, J. (2005). Register allocation via coloring of chordal graphs. In *APLAS*, pages 315–329. Springer.

Pérez-Vega, A., Mas, J.-F., and Ligmann-Zielinska, A. (2012). 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.

Pérez-Vega, A., Mas, J.-F., and Ligmann-Zielinska, A. (2012). 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.

RIKS, B. (2012). Metronamica documentation.

Santori, M. (1990). An instrument that isn't really [laboratory virtual instrument engineering workbench]. *IEEE Spectr.*, 27(8):36--39. ISSN 0018-9235.

Sermulins, J., Thies, W., Rabbah, R., and Amarasinghe, S. (2005). Cache aware optimization of stream programs. *SIGPLAN Not.*, 40(7):115--126. ISSN 0362-1340.

Sethi, R. (1973). Complete register allocation problems. In *5th annual ACM symposium on Theory of computing*, pages 182–195. ACM Press.

Soares-Filho, B., Nepstad, D., Curran, L., Cerqueira, G., Garcia, R., Ramos, C., Voll, E., McDonald, A., Lefebvre, P., and Schlesinger, P. (2006). Modelling conservation in the amazon basin. *Nature*, 440:520--523.

Soares-Filho, B., Pennachin, C., and Cerqueira, G. (2002). Dinamica - a stochastic cellular automata model designed to simulate the landscape dynamics in an amazonian colonization frontier. *Ecological Modeling*, 154:217--235.

Soares-Filho, B., Rajão, R., Macedo, M., Carneiro, A., Costa, W., Coe, M., Rodrigues, H., and Alencar, A. (2014). Cracking brazil's forest code. *Science*, 344(6182):363–364.

Soares-Filho, B., Rodrigues, H., and Costa, W. (2009). *Modeling Environmental Dynamics with Dinamica EGO.* Centro de Sensoriamento Remoto (IGC/UFMG).

Soares-Filho, B., Rodrigues, H., and Follador, M. (2013). A hybrid analytical-heuristic method for calibrating land-use change models. *Environ. Model. Softw.*, 43:80--87.

Sondergaard, H. and Sestoft, P. (1990). Referential transparency, definiteness and unfoldability. *Acta Inf.*, 27(6):505--517.

Spring, J. H., Privat, J., Guerraoui, R., and Vitek, J. (2007). Streamflex: High-throughput stream programming in java. *SIGPLAN Not.*, 42(10):211--228. ISSN 0362-1340.

Takano, A. and Meijer, E. (1995). Shortcut deforestation in calculational form. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 306--313, New York, NY, USA. ACM.

Tanimoto, S. L. (1990). VIVA: a visual language for image processing. *Journal of Visual Languages and Computing*, 1(1):127--139.

Thapa, R. B. and Murayama, Y. (2011). Urban growth modeling of kathmandu metropolitan region, nepal. *Computers, Environment and Urban Systems*, 35(1):25 – 34.

Tomlin, C. D. (1990). *Geographic Information Systems and Cartographic Modelling.* Prentice-Hall.

van Delden, H., Escudero, J. C., Uljee, I., and Engelen, G. (2005). Metronamica: A dynamic spatial land use model applied to vitoria-gasteiz. In *Virtual Seminar of the MILES Project. Centro de Estudios Ambientales, Vitoria-Gasteiz.*

Verburg, P. and Overmars, K. (2009). Combining top-down and bottom-up dynamics in land use modeling: exploring the future of abandoned farmlands in europe with the dyna-clue model. *Landscape Ecology*, 24(9):1167–1181. ISSN 0921-2973.

Verburg, P. H., Soepboer, W., VELDKAMP, A., LIMPIADA, R., ESPALDON, V., and MASTURA, S. S. (2002). Modeling the spatial dynamics of regional land use: The clue-s model. *Environmental Management*, 30(3):391–405. ISSN 0364-152X.

Vries, E., Plasmeijer, R., and Abrahamson, D. M. (2008). Uniqueness typing simplified. In Chitil, O., Horváth, Z., and Zsók, V., editors, *Implementation and Application of Functional Languages*, pages 201--218. Springer-Verlag.

Wadler, P. (1988). Deforestation: transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231--248.

Wadler, P. (1989). Theorems for free! In *Functional Programming Languages and Computer Architechture*, pages 347--359. ACM Press.

Wadler, P. (1990). Comprehending monads. In *LFP*, pages 61--78. ACM.

Weng, K. S. (1975). Stream oriented computation in recursive data-flow schemas. Technical report 68, Laboratory for Computer Science, MIT, Cambridge, MA.

Wesselung, C. G., KARSSENBERG, D.-J., Burrough, P. A., and DEURSEN, W. (1996). Integrating dynamic environmental models in gis: the development of a dynamic modelling language. *Transactions in GIS*, 1(1):40--48.

Whiting, P. G. and Pascoe, R. S. (1994). A history of data-flow languages. *Annals of the History of Computing, IEEE*, 16(4):38--59.

Zadeck, F. K. (1984). *Incremental Data Flow Analysis in a Structured Program Editor*. PhD thesis, Rice University.