Go to Summary - - - - - - - - Índice de artigos

XI SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO

Natal — RN
23 a 25 de Maio de 2007

ANAIS
2007

Promoção
Comissão Especial de Linguagens de Programação
SBC - Sociedade Brasileira de Computação

Organização
Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte

Edição
Roberto S. Bigonha
Martin A. Musicante

XI SYMPOSIUM ON PROGRAMMING LANGUAGES

Natal — RN
May, 23 - 25, 2007

PROCEEDINGS

2007

Promotion
Special Committee on Programming Languages
SBC - Brazilian Computer Society

Organization
Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte

Edição
Roberto S. Bigonha
Martin A. Musicante

Cópias adicionais:

DIMAp
Univ. Federal do Rio Grande do Norte
Natal, RN, CEP 59.072-970
Brazil, e-mail: sblp2007@dimap.ufrn.br

XI Simpósio Brasileiro de Linguagens de Programação (1.; 2007; Natal,RN)

Anais do XI Simpósio Brasileiro de Linguagens de Programação, Natal, de 23 a 25 de maio de 2007. Editado por Roberto S. Bigonha (UFMG) e Martin A. Musicante(UFRN). 2007.

xx, 248 p
ISBN 978-85-7669-109-9

Conhecido também como SBLP 2007.

I. Linguagens de Programação. I. Bigonha, Roberto S. II. Musicante, Martin A. III. SBLP (11. 2007 : Natal) Sociedade Brasileira de Computação.

Esta obra foi impressa a partir de originais fornecidos pelos autores.

Foreword

The Brazilian Symposium on Programming Languages (SBLP) is a series of annual conferences promoted by the Brazilian Computer Science Society. SBLP 2007 was organized by the Computer Science Department (DIMAp) of the Federal University of Rio Grande do Norte (UFRN), in Natal. It was supported in part by CNPq, UFRN and Microsoft.

SBLP 2007 is the 11th edition in the series of SBLP events. Previous symposia were held in Belo Horizonte (1996), Campinas (1997), Porto Alegre (1999), Recife (2000), Curitiba (2001), Rio de Janeiro (2002), Ouro Preto (2003), Niteroi (2004), Recife (2005) and Itatiaia (2006).

The Program Committee selected 15 papers from a total of 46 submissions. Submitted papers came from Brazil, Germany, Portugal, France and India. Almost all papers were reviewed by at least five Program Committee members, who were often assisted by other referees. This program contains the time table for all accepted papers and two invited papers, corresponding to the two invited talks. Besides the scientific program, SBLP 2007 included one tutorial, dedicated to technological topics.

As in previous editions of the event, selected papers from the proceedings are eligible to be published in a Special Issue of the Journal of Universal Computer Science - JUCS.

We would like to thank everyone who contributed to the success of the symposium and to its scientific merit. In particular, we are grateful the Program Committee members for their demanding and responsible work, the referees for their careful reading of all the submissions, the invited speakers for accepting our invitation to share with us their knowledge, the authors of the submitted papers, the tutorial instructors for preparing their contributions, the sponsors and the Organizing Committee for their efforts in making the venue a success.

Natal may 23th, 2007

Roberto S. Bigonha
Program Committee Chair

Martin A. Musicante
Organizing Committee Chair

Prefácio

O 1º Simpósio Brasileiro de Linguagens de Programação (SBLP) foi proposto e coordenado por Roberto S. Bigonha em 1996, de 4 a 6 de setembro, em Belo Horizonte, no Campus da Universidade Federal de Minas Gerais, com o objetivo de criar um fórum específico para discussão e divulgação dos avanços da área de Linguagens de Programação. Especificamente, nesta primeira edição do simpósio, discutiram-se questões relacionadas à especificação, análise e implementação de linguagens de programação e sistemas. Esse Simpósio recebeu 41 submissões de artigos completos, dos quais foram selecionados 29 para apresentação e publicação nos anais do evento. O Comitê de Programa deste evento, coordenado por Roberto S. Bigonha, foi composto de 12 membros, todos pesquisadores brasileiros, e contou com mais 24 avaliadores para análise das submissões recebidas. O evento teve a participação do prof. Simon Peyton Jones e prof. Philip Wadler como conferencistas convidados.

O 2º SBLP, coordenado por Cecília Rubira e Luiz E. Buzato, foi realizado no Campus da Universidade Estadual de Campinas, SP, no período de 3 a 5 de setembro de 1997, e recebeu 42 submissões de artigos completos, dos quais 21 foram aceitos para apresentação e publicação nos anais daquele ano. O Comitê de Programa, coordenado por Cecília Rubira, foi composto por 12 pesquisadores brasileiros. Para a avaliação dos trabalhos, o Comitê contou com 39 avaliadores externos. Nesta edição, o SBLP trouxe como novidades, a submissão de artigos vindos de Portugal, Cuba e Inglaterra; o oferecimento de dois mini-cursos e a presença dos conferencistas convidados, prof. Jürg Gutknecht, prof. Kristen Nygaard, prof. Mehdi Jazayeri e prof. Stefano Levialdi.

O 3º SBLP coordenado por Maria Lúcia B. Lisboa e Vera Lúcia S. de Lima aconteceu no período de 5 a 7 de maio de 1999 em Porto Alegre, RS, resultado da cooperação entre a Universidade Federal do Rio Grande do Sul e a Faculdade de Informática da Pontifícia Universidade Católica do Rio Grande do Sul. Nesta terceira edição do SBLP, foram aceitos para apresentação e publicação 13 artigos completos, 5 artigos resumos, 4 tutoriais. O Comitê de Programa foi também coordenado por Maria Lúcia B. Lisboa e era composto de 11 pesquisadores brasileiros. Cooperaram na análise dos trabalhos mais 10 revisores externos. Prof. Egon Boerger participou como conferencista convidado, apresentando uma palestra e um tutorial, cujos resumos foram publicados nos Anais do III SBLP. Neste simpósio destacou-se a importância

dada à cooperação da área de Linguagens de Programação com áreas afins de pesquisa, incentivou-se a identificação de novos nichos de estudo e de projeto de pesquisa.

O 4º SBLP sob a coordenação geral de Hermano P. de Moura e André Luís de Medeiro Santos teve ênfase na integração entre a teoria e a prática de linguagens de programação. O Comitê de Programa foi composto por 18 pesquisadores brasileiros e seu coordenador foi Paulo Borba. Além do Comitê de Programa, 16 colaboradores externos auxiliaram na seleção dos artigos. Foram publicados nos Anais do IV SBLP artigos técnicos de pesquisa e relatos de experiências industriais utilizando-se de linguagens e tecnologias modernas. Ele foi realizado em Recife, PE, no Centro de Informática da Universidade Federal de Pernambuco, de 17 a 19 de maio de 2000. O IV SBLP publicou 19 artigos 2 resumos técnicos de pesquisa, 4 relatos de experiências industriais. Foram selecionados e apresentados também neste simpósio 3 tutoriais. Os conferencistas convidados foram o prof. David Watt, prof. Peter Mosses e o prof. Simon Peyton-Jones. Juntamente com o IV SBLP foram realizados o *Third International Workshop on Action Semantics* e o Java Brasil 2000.

O 5º SBLP foi coordenado por Martin Musicante e teve como Coordenador do Comitê de Programa, Edward Hermann Haeusler. O V SBLP foi realizado em Curitiba, PA, nas dependências da Universidade Federal do Paraná, no período de 23 e 25 de maio de 2001. A internacionalização do SBLP, meta definida desde seu início, com a participação de conferencistas internacionais, neste evento consolidou-se com a inclusão, em seu Comitê de Programa, composto por 25 membros, de 2 pesquisadores de outras nacionalidades. Dentre as 44 submissões foram selecionados 19 artigos técnicos completos. Nesta edição do evento destacaram-se como palestrantes convidados o prof. Uday Reddy, o prof. José L. Fiadeiro e o prof. Roberto Ierusalimschy.

O 6º SBLP foi sediado pela primeira vez no Rio de Janeiro, RJ, de 5 a 7 de junho de 2002, na Pontifícia Universidade Católica do Rio de Janeiro e teve como coordenador geral Edward Hermann Haeusler, e Carlos Camarão como coordenador do Comitê de Programa. O comitê foi composto de 27 membros, 23 pesquisadores brasileiros e 4 de outras nacionalidades. Para o VI SBLP foram submetidos 49 trabalhos, destes, selecionaram-se 18 artigos técnicos e 4 tutoriais. Três tutoriais foram ministrados por pesquisadores de Portugal, Espanha e França. Destacou-se neste simpósio a cooperação entre grupos de pesquisa, isto foi evidenciado em sua organização com a participação de professores dos Departamentos de Ciência da Computação do IME e da UFF. O

VI SBLP teve os professores Doug Lea e Simon Thompson como palestrantes convidados. A partir deste evento a Microsoft, por iniciativa de Roberto S. Bigonha, aparece como uma das patrocinadoras do SBLP.

O 7º SBLP sob a coordenação geral de Lucília Figueiredo e Marco Túlio Oliveira Valente foi sediado em Ouro Preto, MG, no período de 28 a 30 de maio de 2003, e teve como Coordenador do Comitê de Programa Roberto Ierusalimschy. O comitê foi composto de 25 membros. O VII SBLP foi organizado pela Universidade Federal de Ouro Preto e pela Pontifícia Universidade Católica de Minas Gerais. Foram submetidos para esta edição do evento 50 trabalhos técnicos, sendo 49 artigos e 1 tutorial. Destacou-se neste simpósio o número de submissões de artigos de outros países: foram submetidos 6 artigos de pesquisadores portugueses, 3 da Alemanha, 3 dos Estados Unidos, 1 da França e 1 tutorial do Uruguai. O Comitê de Programa selecionou 16 artigos e o tutorial para apresentação e publicação em seus Anais. Os palestrantes convidados do VII SBLP foram os professores Nick Benton, Johan Jeuring e Simon Thompson. Inovações deste ano: Mini-Escola de Linguagens de Programação, composta de seis minicursos dirigidos a alunos de graduação; e a publicação dos artigos em inglês em um número especial do *Journal of Universal Computer Science (JUCS)*, publicado pela *Springer*.

O 8º SBLP teve como coordenador geral Christiano de Oliveira Braga, e como coordenador do Comitê de Programa, Rafael Dueire Lins. O simpósio foi realizado em Niterói, RJ, nas dependências do Instituto de Computação da Universidade Federal Fluminense, de 26 a 28 de maio de 2004. Este foi o ano em que o evento recebeu mais submissões de trabalhos, 53, sendo 49 artigos técnicos e 4 propostas de tutorial. Foram selecionados 17 artigos completos e 1 tutorial. O Comitê de Programa foi formado por 42 pesquisadores, dos quais, 17 de outras nacionalidades. Mais 22 avaliadores externos ao comitê foram responsáveis pela seleção dos artigos. A maior parte dos artigos tiveram 4 ou mais avaliações e nenhum deles menos que 3 análises. Foram palestrantes convidados do VIII SBLP o prof. José Fiadeiro e o prof. David Turner. Dando continuidade a iniciativa do SBLP 2003, todos os artigos escritos em inglês foram publicados em uma edição especial do JUCS 2004.

Uma novidade deste evento foi criação do Comitê Diretor (*Steering Committee*), com mandato correspondente ao período de um simpósio até o seguinte. Definiu-se que o **Comitê Diretor deve ser composto pelos três últimos coordenadores de comitê de programa e pelos dois últimos organizadores de SBLP. Definiu-se ainda que o coordenador da Comissão Especial de Linguagens de Programação, que é também o presi-**

dente do Comitê Diretor, é o coordenador de comitê de programa do último simpósio realizado.

O 9º SBLP coordenado por Ricardo Massa Lima teve como coordenador do Comitê de Programa, com 42 membros, Martin Musicante. O evento aconteceu no Recife, PE, de 23 a 25 de maio de 2005. Foram submetidos ao IX SBLP 52 artigos provenientes do Brasil e de 14 diferentes países. Destes, foram selecionados 18 artigos para serem apresentados e publicados nos anais do evento. Cada artigo foi avaliado por pelo menos três membros do Comitê de Programa e por avaliadores externos ao comitê. Os palestrantes convidados foram o prof. Peter Mosses e prof. Mary Sheeran. A novidade do IX SBLP foi a inclusão de 2 tutoriais dedicados a tópicos tecnológicos, nesta ocasião apresentados por Alisson Sol.

O 10º SBLP, coordenado por Alex de Vasconcellos Garcia, teve como coordenador do Comitê de Programa, com 48 membros, Mariza Andrade da Silva Bigonha. O evento aconteceu em Itatiaia, RJ, de 15 a 17 de maio de 2006. O Comitê de Programa do 10º SBLP recebeu 50 submissões, sendo 4 tutoriais e 46 artigos técnicos, vindos do Brasil, Holanda, Austria, Portugal, Estados Unidos, Alemanha e Grã-Bretanha. Destes, foram selecionados pelos 48 membros do Comitê de Programa 2 tutoriais e 16 artigos completos, com 4 artigos de pesquisadores estrangeiros, perfazendo uma taxa de aceitação inferior a 35% dos artigos submetidos. Cada uma das submissões teve cinco avaliações realizadas pelo Comitê de Programa e mais 28 revisores externos. Os palestrantes convidados foram os professores Jens Palsberg, Guido de Araújo, Diana Santos e Roberto S. Bigonha.

O Comitê de Programa do 11º SBLP recebeu 47 submissões, sendo 2 tutoriais e 46 artigos técnicos, vindos do Brasil, Portugal, Alemanha, França e Índia. Destes, foram selecionados pelos 43 membros do Comitê de Programa, 1 tutorial e 15 artigos completos, perfazendo uma taxa de aceitação de artigos completos de 33,33% dos artigos submetidos. Cada uma das submissões, exceto três, teve pelo menos cinco avaliações realizadas pelo Comitê de Programa e revisores externos, listados nestes anais. Os palestrantes convidados desta edição do SBLP são os professores Paulo Borba (UFPE) e Guiseppe Castagna (CNRS, Universidade de Paris 7). Agradecemos aos autores, aos membros do Comitê de Programa e aos revisores externos o excelente trabalho, aos conferencistas convidados e a todos os pesquisadores que submeteram artigos para o XI SBLP.

Por fim, temos o prazer de registrar que, no fim de 2006, o Simpósio Brasileiro

de Linguagens de Programação foi classificado pela CAPES como um evento **QUALIS A**. Agradecemos aos autores, organizadores e aos comitês de programa esta conquista.

Natal, 23 de março de 2007

Roberto S. Bigonha	Martin A. Musicante	Mariza A. S. Bigonha
Comitê de Programa	Comitê Organizador	Comissão Especial de LP

COMITÊ DE PROGRAMA
PROGRAM COMMITTEE

Coordenador/Chair: Roberto S. Bigonha, UFMG

Alberto Pardo	Univ. de La Republica	Uruguay
Alex Garcia	IME	Brazil
Alfio Martini	PUC-RS	Brazil
Álvaro Freitas Moreira	UFRGS	Brazil
Ana Cristina Vieira de Melo	USP	Brazil
André R. Du Bois	UCPel	Brazil
André Santos	UFPE	Brazil
Carlos Camarão	UFMG	Brazil
Cecília Rubira	UNICAMP	Brazil
Christiano Braga	UFF	Brazil
David Naumann	Stevens Tech	USA
Edward Hermann Haeusler	PUC-Rio	Brazil
Francisco Heron de Carvalho-Junior	UFC	Brazil
Isabel Cafezeiro	UFF	Brazil
Jens Palsberg	University of California	USA
Johan Jeuring	Utrecht University	Netherlands
João Saraiva	Univ. do Minho	Portugal
José Guimarães	UFSCAR	Brazil
Jose Labra	Univ. of Oviedo	Spain
José Luiz Fiadeiro	U. of Leicester	UK
Lucília Figueiredo	UFOP	Brazil
Luis Soares Barbosa	Univ. do Minho	Portugal
Luiz Carlos Menezes	UFPE	Brazil
Marcelo Maia	UFU	Brazil
Marco Túlio de Oliveira Valente	PUC-MG	Brazil
Mariza A. S. Bigonha	UFMG	Brazil
Martin A. Musicante	UFRN	Brazil
Nick Benton	Microsoft Research	England
Noemi Rodriguez	PUC-Rio	Brazil
Paulo Borba	UFPE	Brazil
Peter D. Mosses	University Wales Swansea	UK

Rafael Dueire Lins	UFPE	Brazil
Renato Cerqueira	PUC-Rio	Brazil
Ricardo M. Lima	UPE	Brazil
Roberto Ierusalimschy	PUC-Rio	Brazil
Sandro Rigo	UNICAMP	Brazil
Sérgio de Mello Schneider	UFU	Brazil
Sérgio Soares	UPE	Brazil
Sergiu Dascalu	University of Nevada	USA
Simon Thompson	University of Kent	England
Varmo Vene	University de Tartu	Estonia
Vitor Costa	UFRJ	Brazil
Vladimir Di Iorio	UFV	Brazil

COMITÊ DIRETOR
STEERING COMMITTEE

Presidente/President: Mariza A. S. Bigonha, UFMG

Alex Garcia	IME	X SBLP organizer
Martin A. Musicante	UFRN	XI SBLP organizer and IX PC chair
Roberto S. Bigonha	UFMG	XI SBLP PC chair

COMITÊ ORGANIZADOR
ORGANIZING COMMITTEE

Coordenador Geral/Chair: Martin A. Musicante, UFRN/DIMAp

David Déharbe	UFRN/DIMAp
Marcel Oliveira	UFRN/DIMAp
João Marcos	UFRN/DIMAp
Selan Rodrigues dos Santos	UFRN/DIMAp
Umberto Souza da Costa	UFRN/DIMAp

REVISORES
REFEREES

Adriano Oliveira	Universidade de Pernambuco	Brazil
Alberto Pardo	Universidad de La República	Uruguay
Alex Garcia	Instituto Militar de Engenharia	Brazil
Alfio Martini	PUC do Rio Grande do Sul	Brazil
Alvaro Moreira	Univ. Federal do Rio Grande do Sul	Brazil
Ana Bove	Chalmers University of Technology	Sweden
Ana C. V. de Melo	Universidade de São Paulo	Brazil
Anderson Faustino	Universidade Federal do Rio de Janeiro	Brazil
André Du Bois	Universidade Católica de Pelotas	Brazil
André Furtado	Universidade Federal de Pernambuco	Brazil
André Luis Santos	Universidade Federal de Pernambuco	Brazil
Antônio Rocha Costa	Universidade Católica de Pelotas	Brazil
Arnaldo Mandel	Universidade de São Paulo	Brazil
Carlos Camarão	Universidade Federal de Minas Gerais	Brazil
Carlos Luna	Universidad de la Republica	Uruguay
Cecília Rubira	Universidade Estadual de Campinas	Brazil
Christiano Braga	Universidad Complutense de Madrid	Spain
Cláudia Justel	Instituto Militar de Engenharia	Brazil
Claudio Geyer	Univ. Federal do Rio Grande do Sul	Brazil
Claudio Naoto	Univ. Federal do Rio Grande do Sul	Brazil
David Naumann	Stevens Institute of Technology	USA
Dorgival Guedes	Universidade Federal de Minas Gerais	Brazil
Eduardo Almeida	Universidade Federal de Pernambuco	Brazil
Edward Hermann Haeusler	PUC do Rio de Janeiro	Brazil
Emilio Tuosto	University of Leicester	Great Britain
Flávio S. Corrêa da Silva	Universidade de São Paulo	Brazil
Francisco Carvalho-Junior	Universidade Federal do Ceará	Brazil
Irek Ulidowski	University of Leicester	Great Britain
Isabel Cafezeiro	Universidade Federal Fluminense	Brazil
Jens Palsberg	University of California	USA
João Dantas	Universidade Federal de Minas Gerais	Brazil
João Saraiva	Universidade do Minho	Portugal
Johan Jeuring	Utrecht University	The Netherlands

José Guimarães	Universidade Federal de São Carlos	Brazil
Jose Labra Gayo	University of Oviedo	Spain
Juan Quiroz	University of Nevada, Reno	USA
Jussara Almeida	Universidade Federal de Minas Gerais	Brazil
Leila Silva	Universidade Federal de Sergipe	Brazil
Lucília Figueiredo	Universidade Federal de Ouro Preto	Brazil
Luis Barbosa	Universidade do Minho	Portugal
Luis Menezes	Universidade de Pernambuco	Brazil
Luis Sierra	Universidad de La Republica	Uruguay
Marcelo Maia	Universidade Federal de Uberlândia	Brazil
Marco Túlio Valente	PUC de Minas Gerais	Brazil
Mariza Bigonha	Universidade Federal de Minas Gerais	Brazil
Martin Musicante	Univ. Federal do Rio Grande do Norte	Brazil
Mauricio Pilla	Universidade Católica de Pelotas	Brazil
Márcio Cornélio	Universidade Federal de Pernambuco	Brazil
Michael Hoffmann	University of Leicester	Great Britain
Mike McMahon	University of Nevada Reno	USA
Newton Vieira	Universidade Federal de Minas Gerais	Brazil
Nick Benton	Microsoft Research	Great Britain
Noemi Rodriguez	PUC do Rio de Janeiro	Brazil
Paulo Borba	Universidade Federal de Pernambuco	Brazil
Paulo Pires	Univ. Federal do Rio Grande do Norte	Brazil
Peter Mosses	Swansea University	Great Britain
Rafael Lins	Universidade Federal de Pernambuco	Brazil
Renata Reiser	Universidade Católica de Pelotas	Brazil
Renato Cerqueira	PUC do Rio de Janeiro	Brazil
Renato Ferreira	Universidade Federal de Minas Gerais	Brazil
Ricardo Lima	Universidade de Pernambuco	Brazil
Roberto Ierusalimschy	PUC do Rio de Janeiro	Brazil
Sandro Rigo	Universidade Estadual de Campinas	Brazil
Sérgio Schneider	Universidade Federal de Uberlandia	Brazil
Sérgio Soares	Universidade de Pernambuco	Brazil
Sergiu Dascalu	University of Nevada, Reno	USA
Simon Thompson	University of Kent	Great Britain
Tiago Massoni	Universidade Federal de Pernambuco	Brazil
Varmo Vene	University of Tartu	Estonia
Vitor Costa	Universidade Federal do Rio de Janeiro	Brazil
Vladimir Di Iorio	Universidade Federal de Viçosa	Brazil

SOCIEDADE BRASILEIRA DE COMPUTAÇÃO (SBC)

Diretoress

Cláudia Maria Bauzer Medeiros, UNICAMP	Presidente
José Carlos Maldonado, ICMC-USP	Vice-Presidente
Carla Maria Dal Sasso Freitas, UFRGS	Administrativa and Finanças
Karin Breitmann, PUC-Rio	Eventos e Comissões Especiais
Edson Norberto Cáceres, UFMS	Educação
Marta Lima de Queiros Mattoso, UFRJ	Publicação
Virgílio Augusto F. Almeida, UFMG	Planejamento e Programas Especiais
Aline dos Santos Andrade, UFBA	Secretarias Regionais
Altigran Soares da Silva, UFAM	Divulgação e Marketing
Roberto da Silva Bigonha, UFMG	Regulamentação da Profissão
Carlos Eduardo Ferreira, USP	Eventos Especiais

Conselho

Flávio Rech Wagner, UFRGS
Siang Wu Song, USP
Luiz Fernando Gomes Soares, PUC-Rio
Ariadne Maria B. Carvalho, UNICAMP
Taisy Silva Weber, UFRGS Ana Carolina Salgado, UFPE
Ricardo de Oliveira Anido, UNICAMP
Jaime Simão Sichman, USP
Daniel Schwabe, PUC-Rio
Marcelo Walter

Suplentes

Robert Carlisle Burnett, PUC-PR
Ricardo Reis, UFRGS
José Valdeni de Lima, UFRGS
Raul Sidnei Wazlawick, UFSC

Coordenadora da Comissão Especial de Linguagens de Programação
Chair of the Special Committee on Programming Languages

Mariza Andrade da Silva Bigonha, UFMG

Sumário

Modularity, Information Hiding, and Interfaces for Aspect-oriented Languages

Paulo Borba
Informatics Center
Federal University of Pernambuco

Aspect-oriented languages have been proposed with the aim of supporting the modularization of crosscutting concerns, which cannot be properly modularized by object-oriented constructs because the realization of such concerns affects the behavior of several methods, possibly in several classes, cutting across class structures. Despite being able to localize the implementation of crosscutting concerns, aspects and other related aspect-oriented constructs often do not support true modularization, which should enable independent development, understanding, and maintenance of modules. In fact, by having access to details on how classes are implemented, aspects break modular reasoning, requiring modifications to a class to be fully aware of the aspects that affect that class. So constructs aimed to support *crosscutting modularity* might actually break *class modularity*.

In this talk, we use Design Structure Matrixes to analyze this issue in real systems and software product lines. Our analysis is based on a semantic notion of module dependence, going beyond usual coupling metrics, and reflecting more closely modularity dimensions. We also apply the concept of Design Rules, which generalizes Parnas notion of information hiding interfaces, to eliminate class and aspect dependence in those case studies. We discuss how this reconciles different dimensions of modularity and modular reasoning, and helps to better characterize a proper notion of interface for aspect-oriented languages. Finally, we discuss language extensions to support such a notion, and report our experience on applying them to the design and refactoring of aspect-oriented software product lines.

CDuce, an XML Processing Programming LanguageFrom Theory to Practice

Giuseppe Castagna

CNRS
Université Paris 7 - France

CDuce is a modern functional programming language for processing XML documents. Distinctive and innovative features of CDuce are a powerful pattern matching, first class functions, overloaded functions, a very rich type system (arrows, sequences, pairs, records, intersections, unions, differences), precise type inference for patterns, precise error localization with informative error messages, and a natural interpretation of types as sets of values.

This results into very concise and expressive programs in which most of the errors are detected at compile time thanks to the precision of the type system. This allows for fast prototyping and reduced development cycles. CDuce handles several standards (XML, DTD, XMLSchema, Namespaces, Unicode, ...) and is tightly coupled with the OCaml Programming Language (OCaml programs and libraries can be called from CDuce and vice-versa). Last but not least CDuce uses a highly efficient run-time and includes an optimizable query language which is programmable by a graphical interface. For all these reasons CDuce is used in production both by open source projects and industrial users.

The main advantages of CDuce were possible only thanks to its formal foundation. The CDuce project started from the formal study of semantic subtyping and yield a programming language in which all the gory details of semantic interpretation of types are completely hidden to the programmer.

In this talk we follow the inverse paths for our presentation (a more faithful subtitle would be "from Practice to Theory"). We start from the practice and then show the theory hidden under the hood. More precisely the talk is articulated in three parts.

In the first part we give an overview CDuce, discussing its most characteristic features such as the type system, the pattern matching, the query language, the error messages. To that end we comment some CDuce programs, possibly demonstrating the use of CDuce code embedded in XML documents and/or the definition of queries via a graphical interface. We also outline some important implementation issues; in particular, a dispatch algo-

rithm that demonstrates how static type information can be used to obtain very efficient compilation schemas.

In the second part we discuss the theory underlying CDuce. In particular the "semantic subtyping"approach and the set-theoretic interpretation of union, intersection and negation types.

In the last part we will discuss some outcomes of this research that go beyond XML. Among these are applications to concurrency theory and the pi-calculus, to polymorphic type systems and to polymorphic iterators.

CML: The C Modeling Language

Frederico de Oliveira Jr.[1], Ricardo Lima[1], Márcio Cornélio[1], Sérgio Soares[1], Paulo Maciel[2], Raimundo Barreto[2], Meuse Oliveira Jr.[2], Eduardo Tavares[2]

[1]Computing Systems Department, Pernambuco State University (DSC-UPE)
Recife – PE — Brazil

[2]Informatics Center, Federal University of Pernambuco (CIn-UFPE)
Recife – PE — Brazil

{fgaoj,ricardo,marcio,sergio}@dsc.upe.br

{prmm,rsb,mnoj,eagt}@cin.ufpe.br

Abstract. *Non-functional requirements such as performance, program size, and energy consumption significantly affect the quality of software systems. Small devices like PDAs and mobile phones have little memory, slow processors, and energy constraints. The C programming language has been the choice of many programmers when developing application for small devices. On the other hand, the need for functional software correctness has derived several specification languages that adopt the Design by Contract (DBC) technique. In this work we propose a specification language for C, called CML (C Modeling Language), focused on non-functional requirements. CML follows the Design By Contract technique. An additional contribution is a verification tool for hard real-time systems. The tool is the first application developed for CML. The practical usage of CML is presented through a case study, which is a real application for a vehicle monitoring system.*

1. Introduction

A software specification is intended to describe the structure and functionality required for a system[Gannon et al. 1994]. The specification is useful to understand the system and to eliminate errors in the later phases of the development cycle.

A number of specification languages have been designed to be annotated directly in the source code. Some of these languages adopts the Design by Contract (DBC) [Meyer 1992, Meyer 1997] technique. Contracts are a breakthrough technique to reduce the programming effort for large projects. Contracts are the concept of preconditions, postconditions, errors, and invariants. The idea of a contract is just an expression that must evaluate to true. If it does not, the contract is broken, and by definition, the program has a bug in it. Contracts form part of the specification for a program, moving it from the documentation to the code itself. And as every programmer knows, documentation tends to be incomplete, out of date, wrong, or non-existent. Moving the contracts into the code makes them verifiable against the program.

The concepts of DBC were introduced in Eiffel [Meyer 1997]. The Java Modeling Language (JML) follows the design by contract paradigm. It is a specification language for Java programs, using Hoare style pre- and postconditions and invariants [Hoare 1969]. The specifications are added as annotation comments to the Java program, which hence can be compiled with any Java compiler. There are various verification tools for JML, such as a runtime assertion checker [Cheon and Leavens 2002] and the Extended Static Checker (ESC/Java)[Flanagan et al. 2002].

Small devices, such as PDAs and mobile phones, have constrained resources, like memory, processor power, and energy. Therefore, applications developed targeting such devices cannot ignore these resources limitations. The C programming language has been the choice of many programmers

when developing application for small devices. The capacity for manipulating low level resources and the existence of efficient compilers justify the popularity of C for these applications.

Due to side effects caused by pointer manipulation is difficult to proof the functional correctness of C programs. Thus, although most C programmers like the idea of DBC, they abandon DBC because it is too inelegant, relies too much on macros, and is too weak without language support. On the other hand, C is widely adopted to implement application with stringent resource constraints (memory, time processing, communication cost, and energy consumption, for example). Therefore, it is intuitive to define a specification language to describe non-functional requirements of C programs. Indeed, most C programmers invent their own strategy to define such requirements, for instance, in form of comments, informally included in the source code. Additionally, the specification language should be associated with verification tools. It is important to automatically check if the non-functional requirement was fulfilled.

This work proposes a specification language for C focused on non-functional requirements, inspired in the DBC paradigm, called C Modeling Language (CML). Moreover, the paper contributes with a tool for hard real-time systems based on CML. The tool receives a C program, composed of several tasks, annotated with time restrictions, scheduling method (preemptive and non-preemptive), arbitrary inter-task relations (precedence and exclusion), task-to-processor allocation. It automatically looks for a feasible schedule. If a schedule is found, the tool generates a scheduler to control the tasks' execution. It is worth observing that this is pre-runtime scheduling policy, which is fundamental to satisfy timing requirements established in the CML specification.

The main contributions of this work are: the CML specification language focused on non-functional requirements; a tool for hard real-time systems based on CML that analyzes C programs according to the defined specification; and a case study that validates the proposed language (CML) and the analysis tool in a real application.

2. CML: The C Modeling Language

The C Modeling Language (CML) is a specification language developed to describe non-functional requirements of applications implemented through the C programming language. CML is particularly useful for applications with stringent constraints in terms of time, memory, area, power, and other limited resources.

Figure 1 depicts an overview of the CML environment. Similar to JML, programmer includes annotations in the C source code in form of comments. The CML compiler translates the annotated C code into the file format of the verification tool employed to check the system against the specified non-functional requirements.

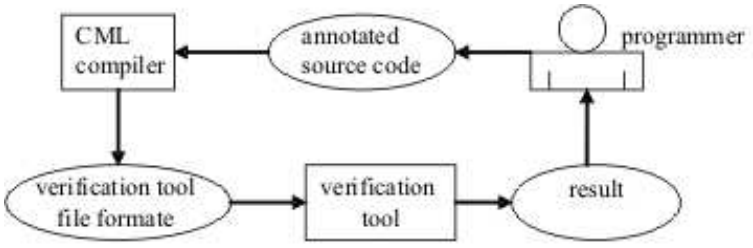


Figure 1. An overview of the CML environment

The CML specification is placed into comment blocks, between /*! and */ patterns. The pattern /*! indicates the beginning of the specification. The pattern */ establishes the end of a specification block. Figure 2 presents a simple example of a CML specification.

The complete set of annotation elements proposed for CML is presented in Appendix 7. Table

```
/*!
 * @attribute value
 */
```

Figure 2. A simple CML specification

1 lists a subset of CML defined for specifying hard-real time systems. We will focus on this subset to illustrate the practical usage of CML.

Constructor	Description	Format
@task	task name	String
@processor	processor where the task will be executed	String
@scheduling	task scheduling model	NP (Non-preemptive) or P (Preemptive)
@phase	task phase time	Integer
@release	task release time	Integer
@wcet	task worst case execution time	Integer
@deadline	task deadline time	Integer
@period	task period time	Integer
@precedes	tasks preceded by this task	List of tasks between the tokens { and } separated by comma
@excludes	tasks excluded by this task	List of tasks between the tokens { and } separated by comma
@sends	message sent by this task	This attribute is followed by: 1. Message name 2. Bus name 3. Worst case communication time 4. Receiver Task

Table 1. Subset of CML for hard real-time systems

Figure 3 presents a CML specification for task T1 (`@task T1`), which belongs to a hard real-time system. According to the specification, T1 cannot be preempted (`@scheduling NP`) and must execute in processor P1 (`@processor P1`). The attributes `@release`, `@period`, `@phase`, `@deadline`, and `@wcet` are related to the task timing constraints and expressed in Time Task Units (TTUs). A TTU is the smallest indivisible granule of a task, during which a task cannot be preempted by any other task. The attributes `@precedes` and `@excludes` specifies the relation between tasks. In the specification example, T1 precedes tasks T2 and T5. Consequently, T2 and T5 can only start executing after T1 has finished (`@precedes {T2,T5}`). `@excludes {T3}` indicates that task T1 excludes T3. Therefore, no execution of T3 can start while T1 is executing. Eventually, message M1 is sent by T1 to the task T4 through the communication bus B1. The communication should take, in the worst case, 17 TTUs (`@sends M1 T4 17 B1`). Section 3 provides a detailed discussion about each attribute included in the CML specification for hard real-time systems.

The CML compiler developed in this work translates the CML specification into a XML file. This file is read by an application developed in this work, which is a software synthesis tool for embedded hard real time systems described in Section 3. The XML file equivalent to the CML specification in Figure 3 is presented in Figure 4.

3. A CML Based Software Synthesis for Embedded Hard Real-Time Applications

The practical usage of CML depends on the implementation of verification tools integrated with a C programming environment. An additional contribution of this work is the development of a software synthesis tool for embedded hard real-time applications.

Embedded hard real-time systems are dedicated computer applications having to satisfy stringent timing constraints, or rather, they must guarantee that critical tasks finish before their deadlines.

```
/*!
 @task T1
 @scheduling NP
 @processor P1
 @release 1
 @period 9
 @phase 1
 @deadline 9
 @wcet 1
 @precedes {T2,T5}
 @excludes {T3}
 @sends M1 T4 17 B1
*/
 void T1(){ //task code }
```

Figure 3. Example of a task specification

A failure to meet deadlines may have serious consequences such as resources damage or even loss of human life. Software synthesis has become a key problem in design of embedded hard real-time systems, since the software is responsible for more than 70% of functions in such systems [Su and Hsiung 2002].

Scheduling is very important in embedded real-time systems. There are two general approaches for scheduling tasks: runtime and pre-runtime scheduling. The former approach computes schedules on-line as tasks arrive, through a priority-driven strategy. However, in some cases the runtime scheduler is unable to find a feasible schedule, even if such schedule exists [Xu and Parnas 1993]. On the other hand, a pre-runtime scheduler computes the schedule entirely off-line. This strategy improves processor utilization, reduces context switching, makes execution predictable, and excludes the need for complex operating systems.

This work focuses on embedded *hard* real-time systems. We decided to adopt a pre-runtime scheduling approach. To find a feasible schedule, we perform a state space exploration, since it presents a complete automatic strategy for verifying finite-state systems [Godefroid 1996]. The scheduled code is generated by traversing the timed Labelled Transition System (LTS), which represents a feasible schedule. Transition's instances visited are substituted by the respective code segments. Tasks can be distributed in several processors in order to achieve the time restrictions.

3.1. A Method for Software Synthesis

This section describes a method for software synthesis considering embedded hard real-time applications. Our method comprises four main steps:

- *Specification:* describes the properties of each task in the system, including time restriction (phase time, release time, worst execution time, deadline, and period), scheduling method (preemptive and non-preemptive), arbitrary inter-task relations (precedence and exclusion), task-to-processor allocation, and task source code; tasks allocated in different processors communicate through a special task, called communication task; such a task is described by the worst communication time, communication channel, sender and receiver;
- *Modelling:* the specification is translated into a Time Petri Net (TPN) model [Merlin and Faber 1976]; each specification element is modeled through a TPN building block; these blocks are composed to form the complete model [Tavares et al. 2005, Tavares 2006];
- *Pre-runtime Scheduler:* the next step searches for a feasible scheduling using the TPN model; the proposed scheduling algorithm performs a depth-first search on a finite timed Labeled Transition System (LTS) derived from a TPN model;

```
<realtime-table>
  <task release="1" period="9" phase="1"  processor="P2" schedulingModel="NP"
   oid="1088076" name="T1">
  <time>
     <computing value="1"/>
     <deadline value="9"/>
  </time>
  <precedes>
       <task-ref name="T2"/>
       <task-ref name="T5"/>
  </precedes>
  <excludes>
       <task-ref name="T3"/>
  </excludes>
  </task>
     ⋮
  <message bus="B1" oid="30377347" name="M1">
     <time>
        <communication value="17"/>
     </time>
     <precedes>
       <task-ref name="T4"/>
     </precedes>
  </message>
</realtime-table>
```

Figure 4. Specification in XML format

- *Software Synthesis:* the scheduled code is generated by traversing the timed LTS that represents a feasible schedule, if it exists, and substituting transition's instances by the respective code segments.

This work concentrates on the specification and software synthesis phases. More details about the modeling and pre-runtime scheduler phases may be found elsewhere [Tavares et al. 2005, Tavares 2006].

3.2. Specification of Hard Real-Time Systems

This subsection describes the specification elements included in CML for modelling hard real-time system.

A task is the basic element in the system. The specification is given in terms of temporal restrictions on task; the scheduling method adopted; inter-task relations; and inter-processor communications, which is modeled by a special task, called *communication task*.

Temporal Restrictions In real-time systems, there are, generally, three types of tasks:

- *periodic tasks* perform a computation that are executed once in each fixed period of time;
- *aperiodic tasks* are activated randomly;
- *sporadic tasks* are executed randomly, but the minimum interval between two consecutive activations is known a priori.

Pre-runtime method performs scheduling decisions at compile time. It aims at generating a schedule table for a runtime component, namely, dispatcher, which is responsible for controlling the tasks during system execution. In order to adopt such method, the major characteristics of the tasks must be known in advance. This approach can only be used to schedule *periodic tasks*.

Definition 3.1 *(Periodic Task) Let T_i be a periodic task defined by T_i =(ph_i, r_i, c_i, d_i, p_i), where ph_i is the initial phase; r_i is the release time; c_i is the worst case computation time; d_i is the deadline; and p_i is the period. A periodic task samples objects of interest at a fixed rate. The phase (ph_i) is the delay associated to the first time request of task T_i after the system starting. ph_i = 0 whenever not specified. The release time (r_i) is the time interval between the beginning of a period and the earliest time to start the execution of task T_i. The computation time (c_i) is the worst case computation time required for executing task Ti. The deadline (d_i) is the time interval between the beginning of a period and the time instant at which task T_i must be completed (in each period). The period (p_i) is the time interval in which T_i must be executed.*

The initial phase (ph_i) defines the point in time, after the system starts executing, when the task period can be allocated. The definition of ph_i is important, since non schedulable system may become schedulable when an initial phase is specified. For instance, considering two tasks, T_1 and T_2, having the same timing constraints (ph_1, r_1, c_1, d_1, p_1) = (ph_2, r_2, c_2, d_2, p_2) = (0, 0, 5, 5, 10). This system is not schedulable, since both takes 5 time units to execute and should finish at time unit 5. However, if an initial phase is specified, i.e. ph_2 = 5, the system becomes schedulable, because the period of T_2 is allowed to start 5 time units after the beginning of system execution. It is enough for task T_1 finishes. It is worth notice that the deadline is relative to the period and not to the entire system. Figure 5 presents a feasible schedule for the system.

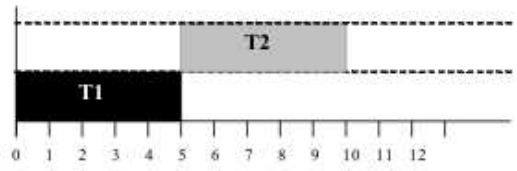


Figure 5. A feasible schedule for the system

Scheduling Method The scheduling methods are all-preemptive and all-non-preemptive. In the all-preemptive scheduling method tasks are implicitly split into all possible subtasks. This scheduling method permits running other conflicting tasks, implying that one task could preempt another task. In turn, with the all-non-preemptive scheduling method processor is just released after finishing the entire computation.

Arbitrary Inter-Task Relations A task T_i precedes task T_j, if T_j can only start executing after T_i has finished. In general, this kind of relation is suitable whenever a task (successor) needs information that is produced by another task (predecessor). A task T_i excludes task T_j, if no execution of T_j can start while task T_i is executing. If it is considered a single processor, then task T_i could not be preempted by task T_j. Exclusion relations may prevent simultaneous access to shared resources. In this work it is considered that the exclusion relation is not symmetrical, that is, when A EXCLUDES B, not necessarily implies that B EXCLUDES A.

Inter-Processor Communication When adopting a multiprocessing environment, all inter-processor communications have to be taken into account, since these communications affect the system predictability. An inter-processor communication is represented by a special task, namely, communication task, which is described as follows.

Definition 3.2 *(Communication Task) Let $\mu_m \in M$ be a communication task defined by $\mu = (T_i, T_j, ct_m, bus_m)$, where $T_i \in T$ is the sending task, $T_j \in T$ is the receiving task, ct_m is the worst case communication time, and $bus_m \in B$ is the bus, where B is the set of buses.*

It is worth observing that the bus is an abstraction for a communication channel used for providing communication between tasks from different processors.

3.3. Scheduled Code Generation

The proposed method for code generation includes not only tasks' code (implemented through C functions), but also a timer interrupt handler, and a small dispatcher. Such dispatcher automates several control mechanisms required during the execution of tasks. Timer programming, context saving, context restoring, and tasks' calling are examples of such additional controls. The timer interrupt handler always transfers the control to the dispatcher, which evaluates the need for performing either context saving or restoring, and calling a specific task.

An array of registers (`struct ScheduleItem`) is created to store the schedule table. Each input represents the *execution part* of a task instance. In case of preemption, a task instance may have more than one *execution part*. The register `struct ScheduleItem` contains the following information: (i) start time; (ii) flag, indicating if the task was preempted before; (iii) task id; and (iv) a pointer to a function (the respective task code). Figure 6 depicts the schedule table for a preemptive application. It includes two instances of `TaskA`, two instances of `TaskB`, two instances of `TaskC`, and one instance of `TaskD`. `TaskA1` and `TaskA2` are preempted in time 4 and 20, respectively. `TaskB1` is preempted twice: first in time 6 and, then, in time 10. Therefore, the schedule table contains 11 entries. Figure 7 presents the respective timing diagram.

```
struct ScheduleItem scheduleTable [SCHEDULE_SIZE] =
{{ 1, false, 1, (int *)TaskA}, TaskA1 starts
 { 4, false, 2, (int *)TaskB},  TaskB1 starts and preempts TaskA1
 { 6, false, 3, (int *)TaskC},  TaskC1 starts and preempts TaskB1
 { 8, true,  2, (int *)TaskB},  TaskB1 resumes executing
 {10, false, 4, (int *)TaskD},  TaskD1 starts and preempts TaskB1
 {11, true,  2, (int *)TaskB},  TaskB1 resumes executing
 {13, true,  1, (int *)TaskA},  TaskA1 resumes executing
 {18, false, 1, (int *)TaskA},  TaskA2 starts
 {20, false, 3, (int *)TaskC},  TaskC2 starts and preempts TaskA2
 {22, false, 2, (int *)TaskB},  TaskB2 starts
 {28, true,  1, (int *)TaskA}   TaskA2 resumes executing
};
```

Figure 6. Example of a Schedule Table

Figure 7. Timing Diagram for Schedule Table in Figure 6

A brief explanation of the dispatcher (Figure 8) is as follows: before calling a task, the dispatcher check some situations: (a) If the current task was preempted, the dispatcher saves its context (line 4); (b) If the next task has been preempted, and now it is being resumed, the dispatcher restores the context (line 5); and (c) If it is a new task instance, the dispatcher just stores the function pointer (line 7) in the variable `taskFunction` that will be called by the interrupt handler. Additionally, the table representing the feasible schedule is accessed as a circular list (line 9). The timer is automatically programmed using the start time of the next task instance to be called (line 10). After all these activities, the timer is activated to interrupt at the start time of the next task (line 11).

```
1 void dispatcher() {
2   struct ScheduleItem item = scheduleTable[scheduleIndex];
3   globalClock = item.clock;
4   if(currentTaskPreempted) { // context saving }
5   if(item.isPreemptionReturn) { // context restoring }
6   else {
7    taskFunction = item.functionPointer;
8   }
9   scheduleIndex = ((++scheduleIndex)%SCHEDULE_SIZE);
10  programTimer(scheduleTable[scheduleIndex].clock);
11  activateTimer();
12}
```

Figure 8. Simplified Version of the Dispatcher

Whenever considering hard real-time embedded system design based on multiple processor platforms, the mechanism for processors synchronization is an important concern. A specific architecture was designed for this purpose. A *master processor* performs the time counting. It periodically sends synchronization messages to *slave processors*. There is no runtime scheduler running in each processor, but a runtime dispatcher. The scheduling is performed using a pre-runtime approach, as described before. Master processor is only responsible for performing the time counting and for notifying slave processors. It does not execute any task, but only a dispatcher. In addition, slave processors do not have their own real-time clocks. The real-time clock only resides in the master processor.

In the proposed approach, all communication tasks are also taken into account in the code generation. Each communication task (μ_m) is translated into two special tasks: `sendMm` and `receiveMm`. Both tasks are executed at the same moment for guaranteeing the correct data transmission. `sendMm` and `receiveMm` are considered in the pre-runtime schedule table and both cannot be preempted. Section 4 presents an example of a system considering inter-processor communication.

4. A Case Study: Vehicle Monitoring System

This section presents a real application case study to illustrate the practical usage of CML for modeling a vehicle monitoring system. The example is particularly useful to demonstrate the application of CML for modeling systems executing in a multiple processing environment where inter-processor communication is required.

The system is composed of a set of sensors employed to verify whether the car components are working properly. If a component fails or works erroneously, the system notifies the driver through the dashboard. The vehicle monitoring system relies on multiple processors, since several sensors are considered and the microcontroller adopted (8051) contains only four 8-Bit I/O ports. In this way, two processors are used for interfacing with the sensors.

4.1. The specification model

A set of tasks were defined to check the status of the engine (TV and TR), breaks (TB), water (TW), gearing (TG), and temperature (TT). Finally, the data processed is sent to the task TRA, which is responsible for notifying the driver. Table 2 details the system specification, which is composed of 14 tasks, including the communication task M1. It implements the communication between processors P1 and P2. The implementation splits task TV into two subtasks: one (TV0) reads the sensor; and the other (TV1) processes the information. The same is done for tasks TR, TB, TW, TG and TT.

Figures 9 and 10 presents the CML specification for the vehicle monitoring system. The communication between processors P1 and P2 is implemented through the communication task M1. Two functions are generated to implement M1: `receiveM1`, in the receiving side; and `sendM1`,

TaskID	Task Name	Release	Comp.	Deadline	Period	Proc./Bus	From	To
TV0	ReadVelocity	0	231	20000	120000	P1	-	-
TV1	ProcVelocity	20000	5487	40000	120000	P1	-	-
TB0	ReadBreaks	20000	221	40000	120000	P1	-	-
TB1	ProcBreaks	40000	236	60000	120000	P1	-	-
TR0	ReadRPM	40000	232	60000	120000	P1	-	-
TR1	ProcRPM	60000	238	80000	120000	P1	-	-
TRA	Notifier	80000	2444	120000	120000	P1	-	-
TW0	ReadWater	0	237	20000	120000	P2	-	-
TW1	ProcWater	20000	241	40000	120000	P2	-	-
TT0	ReadTemperature	20000	259	40000	120000	P2	-	-
TT1	ProcTemperature	40000	234	60000	120000	P2	-	-
TG0	ReadGearing	40000	224	60000	120000	P2	-	-
TG1	ProcGearing	60000	236	80000	120000	P2	-	-
M1	-	-	1700	-	B1	TG1	TRA	-

Table 2. Task Timing Specification

in the sending side. B1 is the bus used to transmit the message, which takes 1700 Time Task Units (TTUs) in the worst case. Task TG1 is responsible for sending the message from processor P2 to processor P1. The message sent to the processor P1 is received by the task TRA, which notifies the driver about the vehicle status.

```
/*!
* @task TV0
* @processor P1
* @scheduling NP
* @phase 0
* @release 0
* @wcet 231
* @deadline 20000
* @period 120000
* @precedes {TV1}
*/
void TV0() {...}
```

```
/*!
* @task TV1
* @processor P1
* @scheduling NP
* @phase 0
* @release 20000
* @wcet 5487
* @deadline 40000
* @period 120000
*/
void TV1() {...}
```

```
/*!
* @task TB0
* @processor P1
* @scheduling NP
* @phase 0
* @release 20000
* @wcet 221
* @deadline 40000
* @period 120000
* @precedes {TB1}
*/
void TB0() {...}
```

```
/*!
* @task TB1
* @processor P1
* @scheduling NP
* @phase 0
* @release 40000
* @wcet 236
* @deadline 60000
* @period 120000
*/
void TB1(){...}
```

```
/*!
* @task TR0
* @processor P1
* @scheduling NP
* @phase 0
* @release 40000
* @wcet 232
* @deadline 60000
* @period 120000
* @precedes {TR1}
*/
void TR0(){...}
```

```
/*!
* @task TR1
* @processor P1
* @scheduling NP
* @phase 0
* @release 60000
* @wcet 238
* @deadline 80000
* @period 120000
*/
void TR1(){...}
```

```
/*!
* @task TRA
* @processor P1
* @scheduling NP
* @phase 0
* @release 80000
* @wcet 2444
* @deadline 120000
* @period 120000
*/
void TRA(){...}
```

```
void receiveM1(){...}
```

Figure 9. CML specification for tasks in processor P1

The system was verified using the tool presented in Section 3. A feasible schedule was found after visiting 78 states. Figure 11 presents the timing diagram with the schedule for the vehicle monitoring system. The schedule was automatically generated by the tool.

More applications, such as Pulse Oximeter and Heated-Humidifier, have already been developed with CML support. These applications are described in [Barreto 2005].

/*! * @task TW0 * @processor P2 * @scheduling NP * @phase 0 * @release 0 * @wcet 227 * @deadline 20000 * @period 120000 * @precedes {TW1} */ void TW0() {...}	/*! * @task TW1 * @processor P2 * @scheduling NP * @phase 0 * @release 20000 * @wcet 241 * @deadline 40000 * @period 120000 */ void TW1(){...}	/*! * @task TT0 * @processor P2 * @scheduling NP * @phase 0 * @release 20000 * @wcet 259 * @deadline 40000 * @period 120000 * @precedes {TT1} */ void TT0(){...}	/*! * @task TT1 * @processor P2 * @scheduling NP * @phase 0 * @release 40000 * @wcet 234 * @deadline 60000 * @period 120000 */ void TT1(){...}
/*! * @task TG0 * @processor P2 * @scheduling NP * @phase 0 * @release 40000 * @wcet 224 * @deadline 60000 * @period 120000 * @precedes {TG1} */ void TG0(){...}	/*! * @task TG1 * @processor P2 * @scheduling NP * @phase 0 * @release 60000 * @wcet 236 * @deadline 80000 * @period 120000 * @sends M1 B1 1700 TRA */ void TG1(){...}	void sendM1(){...}	—

Figure 10. CML specification for tasks in processor P2

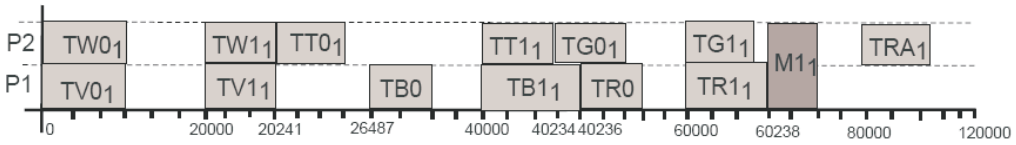


Figure 11. The schedule for the vehicle monitoring system

5. Related Work

Meyer introduced the concept of *Design by Contract* [Meyer 1992, Meyer 1997], a lightweight formal method that allows for dynamic runtime checks of specification violation. Design by Contract establishes that a relationship between a class and its clients is viewed as a formal agreement, which expresses each party's right and obligations. A precondition states the properties that must hold when a routine is called; the postcondition states the properties that the routine guarantees when it returns. As a consequence, pre and postconditions enforce behavior with contracts, which is also expressed by the term *software contract*.

JML [Burdy et al. 2005, Leavens et al. 2006] is a notation used to formally specify the behaviour and interfaces of classes and methods written in Java [K. Arnold and J. Gosling 1996], which follows the ”software contract” concept introduced in the Eiffel language. By using JML, one can specify both the interface of methods and classes as well as their behavior. Interface specification usually includes type information and visibility modifiers, for instance. We can specify the interface of methods (including name, visibility, number of arguments, return type, and so on), attributes (name, type and modifiers), and types (name, modifiers, whether it is a class or interface, its supertypes and so on). In fact, JML uses Java syntax to specify all such interface specification.

The behavior of a method or type specifies the transformations that are performed by them. The style of specification of JML is usually called model-oriented [Wing 1990]. Indeed, specifications written in JML follow the style of refinement calculus [Morgan 1994]. The attributes to which we can assign in a method are described by the method's *frame-axiom*. States to which methods

are defined are formally described by means of assertions (the method's *precondition*); states that may result from the normal execution of methods are described by logical assertions, are called the method's *normal postcondition*. The relationship between the state in which a method is called and the states that may result from throwing an exception are described by the method's *exceptional postcondition*. We can also specify class invariants in JML, describing properties that hold in all visible states [Leavens et al. 2006]. We can also deal with refinement in JML.

A distinguishing feature of JML is the range of tools available. We can check the correctness of JML specifications using the runtime assertion checker *jmlc*, the JML compiler. Unit test is partially automated by the *jmlunit* tool that generates test classes that rely on the JML runtime assertion checker. The *jmldoc* tool produces HTML browsable pages in the style of pages generated by *javadoc*. Other tools with distinct purposes are also available: *ESC/Java* (static checker) [Flanagan et al. 2002], *LOOP* tool [Bart Jacobs and Tews 1998] (compilation to PVS [S. Owre and Shankar 1992] theories), and *JACK* (program checker) [Lilian Burdy and Requet].

Although most of the JML annotations are directed to functional requirements, there are some annotations, which are related with expressions that can be used to express non-functional requirements. For instance, a duration expression describes the specified maximum number of virtual machine cycle times needed to execute a method call. Also, a space expression describes the amount of heap space allocated to an object given as argument. The CML handles this kind of assertions in a higher-level way, dealing not simply with time associated to machine cycles, but to the execution of tasks and scheduling. Besides that, we deal with memory use, and energy consumption.

The tool *Jass* [Bartetzko et al. 2001] translates Java annotated programs into pure Java programs in which compliance with the specification is dynamically tested during runtime. Assertions are written as comments into Java code; they are simply boolean expressions. Different kinds of assertions are allowed: method pre and postconditions; class invariants; loop invariants and variants; refinement checks; and trace assertions that specify the intended dynamic behavior of objects in time, describing allowed traces of events. In CML, we can also express the order in which tasks must be executed, but we go further since we can also express requirements on time, power and memory, for instance. On the other hand, in CML we cannot check tasks refinements.

The Bandera Specification Language [Corbett et al. 2000] is a source-level, model-checker independent language for expressing temporal properties of Java programs action and data. An assertion sublanguage allows programmers to define constraints on programs by writing pre- and postconditions. A temporal property sublanguage provides support for defining predicates on control points (method call and return) and data (object instantiation) present in Java programs. This sublanguage allows specifying temporal properties between system actions. The language provides *specification patterns* that describe properties like precedence. The patterns mentioned in [Corbett et al. 2000] can also be described in CML. However, we do not deal with pre and postconditions as in the assertion language of Bandera.

6. Conclusions

Non-functional requirements (NFR) has long been recognized as a fundamental issue in software development. The popularity of small devices and embedded systems increases the importance of NFR. Due to the limited amount of resources, the correctness and quality of software targeting these platforms relies equally on functional and non-functional aspects. This work presented CML (C Modeling Language), a specification language to describe non-functional requirements of C programs. We decided to focus on the C language because it is widely employed to develop applications with severe non-functional restrictions, such as performance, memory allocation, and energy consumption.

The development of CML was inspired in the Design By Contract (DBC) technique. Thus, specifications are added as annotation comments to the C program, which hence can be compiled

with any C compiler. We believe that this tight integration between specification and implementation languages contributes for motivating programmers to use the specification language in practice. It is worth notice that, CML does not follow the DBC technique. For instance, the concepts of pre-conditions, postconditions, and invariants have no interpretation in CML. On the other hand, a CML specification establishes a kind of contract, in the sense that: from the NFR perspective, if the specification was respected, the system will work properly. We believe that CML can contribute to improve the quality of software systems. In particular, those with stringent resource limitations and performance requirements.

In addition to the proposition of CML itself, this paper contributed with the first verification tool based on CML The tool receives a CML specification for embedded hard real-time systems and checks if exists a feasible schedule for the system. If the answer was yes, the schedule is automatically generated and the code for the system is synthesized considering a pre-runtime scheduling strategy.

In order to show the practical feasibility of the proposed software synthesis method, we specified a vehicle monitoring system. This is a multiple processing application, which was useful to demonstrate the modeling of inter-processor communication in CML. The system was then analyzed through the tool and the scale was found after visiting 78 states.

6.1. Future Works

The current CML implementation does not cover the whole language constructors. Through an incremental approach, we first implemented the elements that address hard-real time systems. The implementation was then validated using a real application. Other verification tools considering the remaining set of CML constructors (see Appendix 7) will be released in near future.

We are currently working to implement a verification tool considering two constructors (`@pesaccess` and `@optaccess`) related to concurrent access policies, in order to guarantee safe execution, and therefore, data consistency. The concurrent access policies are responsible to control a function execution in order to avoid undesirable interferences in a concurrent environment. There are two possible access policies: pessimistic (`@pesaccess`) and optimistic (`@optaccess`). In the pessimistic access policy the function access is made through a critical section using a default or a used-defined lock variable. This technique guarantees that while a thread/task is executing a function with this annotation no other thread/task can execute the same function. This is basically the synchronization of all functions execution. Since this synchronization might be too restrictive and decrease performance, an alternative access policy (`@optaccess`) considers the function semantics in order to decide when block and when allow concurrent access to a function. This optimistic policy considers that in some cases it is possible to define when two or more threads/tasks executions might conflict to each other [Soares and Borba 2001]. For instance, a function might not be concurrently executed if both executions are using the same arguments values.

We are constantly reviewing the language to include/remove constructors and constructors' parameters. New versions of CML will be released soon.

The tool presented in this work as well as more information about CML can be found at `www.dsc.upe.br/cml`.

References

Barreto, R. (2005). *A Time Petri Net-Based Methodology for Embedded Hard Real-Time Systems Software Synthesis*. PhD thesis.

Bart Jacobs, Joachim van den Berg, M. H. M. v. B. U. H. and Tews, H. (1998). Reasoning about java classes. In *OOPSLA'98*, pages 328-340.

Bartetzko, D., Fischer, C., Möller, M., and Wehrheim, H. (2001). Jass – Java with Assertions. *ENTCS*, 55(2):103-117. RV'2001 - Workshop on Runtime Verification at CAV'01.

Burdy, L. et al. (2005). An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212-232.

Cheon, Y. and Leavens, G. (2002). A runtime assertion checker for the java modeling language. Software Engineering Research and Practice (SERP'02), pages 322-328. CSREA Press.

Corbett, J. C., Dwyer, M. B., Hatcliff, J., and Robby (2000). A Language Framework for Expressing Checkable Properties of Dynamic Software. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 205-223. Springer-Verlag.

Cormac Flanagan, K. Rustan M. Leino, M. L. G. N. J. B. S. and Stata, R. (2002). Extended static checking for java. In *PLDI'2002*, pages 234-245.

Flanagan, C., , Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. (2002). Extended static checking for java. *SIGPLAN Not.*, 37(5):234-245.

Gannon, J. D., Purtilo, J. M., and Zelkowitz, M. V. (1994). *Software Specification: A Comparison of Formal Methods*. Ablex Publishing Co., 355 Chestnut Street, Norwood, NJ 07648.

Godefroid, P. (1996). *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *CACM*, 12(10):576-580.

K. Arnold and J. Gosling (1996). *The Java Programming Language*. Addison-Wesley.

Leavens, G. et al. (2006). *JML Reference Manual*.

Lilian Burdy, J.-L. L. and Requet, A. Jack (java applet correctness kit). Technical report.

Merlin, P. and Faber, D. J. (1976). Recoverability of communication protocols: Implications of a theoretical study. *IEEE Transactions on Communications*, 24(9):1036-1043.

Meyer, B. (1992). Applying "Design by Contract". *Computer*, 25(10):40-51.

Meyer, B. (1997). *Object-Oriented Software Construction*. Prentide-Hall, second edition.

Morgan, C. C. (1994). *Programming from Specifications*. Prentice Hall, second edition.

S. Owre, J. M. R. and Shankar, N. (1992). Pvs: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607, pages 748-752, Saratoga, NY.

Soares, S. and Borba, P. (2001). Concurrency Manager. In *First Latin American Conference on Pattern Languages of Programming – SugarLoafPLoP*, pages 221-231, Rio de Janeiro, Brazil.

Su, F.-S. and Hsiung, P.-A. (2002). Extended quase-static scheduling for formal syn- thesis and code generation of embedded software. In *International Symposium on Hardware/Software Codesign (CODES'02)*.

Tavares, E. (2006). A time petri net based approach for software synthesis in hard real-time embedded systems with multiple processors. Informatic Center - Federal University of Pernambuco.

Tavares, E., Maciel, P., Barreto, R., Meuse Oliveira, J., and Lima, R. (2005). A time petri net based approach for embedded hard real-time software synthesis with multiple operational modes. In *18th annual symposium on Integrated circuits and system design*, pages 98-103. ACM Press.

Wing, J. M. (1990). A Specifier's Introduction to Formal Methods. *Computer*, 23(9):8-24.

Xu, J. and Parnas, D. L. (1993). On satisfying timing constraints in hard-real-time systems. *IEEE Trans. Softw. Eng.*, 19(1):70-84.

7. The Complete Set of CML Constructors

Constructor	**Description**	**Format**	**Program/ Function**†	**Non-functional requirement**
@task	task name	String	F	–
@processor	processor to task allocation	String	P and F	resource allocation
@scheduling	task scheduling strategy	NP (Non-preemptive) or P (Preemptive)	F	scheduling
@phase	task phase time	Integer	F	temporal restrictions
@release	task release time	Integer	F	temporal restrictions
@wcet	worst case execution time	Integer	P and F	temporal restrictions
@deadline	task deadline time	Integer	F	temporal restrictions
@period	task period time	Integer	F	temporal restrictions
@precedes	tasks preceded by this task	List of tasks between the tokens { and } separated by comma	F	inter-task relation
@excludes	tasks excluded by this task	List of tasks between the tokens { and } separated by comma	F	inter-task relation
@sends	message sent by this task	This attribute is followed by: 1-message name, 2-bus name, 3-worst case communication time, 4-receiver task	F	inter-task communication
@usedmemory ⋆	maximum amount of memory to use	Integer (Kb)	P and F	memory use
@codesize ⋆	generated binary	Integer (Kb)	P and F	memory use
@cachehit ⋆	maximum number of accesses to cache (L1 and L2 levels)	L1:Integer, L2:Integer	P and F	performance
@cachemiss ⋆	maximum number of cache misses (L1 and L2 levels)	L1:Integer, L2:Integer	P and F	performance
@pagefault ⋆	maximum number of page faults	Integer	P and F	performance
@power ⋆	maximum energy consumption	Real (Joules)	P and F	Power
@pesaccess ⋆	no concurrent access is allowed; optionally, specify a lock variable	String (var)	F	Safety/data consistency
@optaccess ⋆	some concurrent access is allowed; specify a variable or parameter list as the function semantics	List of variables between the tokens { and } separated by comma	F	Safety/data consistency

† *Constructor granularity; P means the constructor can be applied to a program and F to a function.*

⋆ *CML constructors not considered in the current implementation*

Constraint Programming Architectures: Review and a New Proposal

Jacques Robin[1], Jairson Vitorino[1], Armin Wolf[2]

[1]Centro de Informática – Universidade Federal de Pernambuco (CIn-UFPE)
Caixa Postal CDU 7851 – Recife – PE– Brazil

[2] Fraunhofer Institut - Rechnerarchitektur und Softwaretechnik (FIRST),
Kekuléstrasse 7 – Berlin – Germany

{robin.jacques,jairson}@gmail.com, Armin.Wolf@first.fraunhofer.de

Abstract. *Most automated reasoning tasks with practical applications can be automatically reformulated into a constraint solving task. A constraint programming platform can thus act as a unique, underlying engine to be reused for multiple automated reasoning tasks in intelligent agents and systems. We identify six key requirements for such platform: expressive task modeling language, rapid solving method customization and combination, adaptive solving method, user-friendly solution explanation, efficient execution, and seamless integration within larger systems and practical applications. We then propose a novel, model-driven, component and rule-based architecture for such a platform that better satisfies as a whole this set of requirements than those of currently available platforms*

1. Introduction

Over the last two decades, the practical inference needs of intelligent agents and systems have led the field of automated reasoning to vastly diversify from its roots in monotonic deduction. It now also encompasses abduction, default reasoning, inheritance, belief revision, belief update, planning, constraint solving, optimization, induction and analogy. Many specialized methods are now available to efficiently implement specific subclasses of these tasks in conjunction with some specific knowledge representation languages. These languages vary in terms of their two commitments, epistemological (*e.g.,* logical, plausibilistic, or probabilistic) and ontological (*e.g.,* propositional, first-order relational, first-order object-oriented, high-order relational or high-order object-oriented) (16). This diversity in tasks, methods and languages inhibits pervasive embedding of automated reasoning functionalities in applications. It confuses the development teams of most applications who generally have a sparse background in automated reasoning and it seems to prevent cost-cutting reuse of a single generic platform for many such purposes. However, a way out of this dilemma is suggested by the fact that whether using propositional (22), first-order relational (22) or object-oriented (2) representations with either logical (10), plausibilistic (11) or probabilistic (5) semantics, the reasoning tasks of deduction (22), abduction (1), default reasoning (1), inheritance (2), belief revision (11), belief update (19) and planning (19) have now all being reformulated as special cases of constraint solving. An adequate **Constraint Programming Platform** (CPP) could thus be successfully reused as an all subsuming

engine to implement and seamlessly integrate those diverse tasks, methods, and languages (15).

In this paper, after providing some background on constraint programming and relevant, cutting-edge software engineering approaches (Section 2), we first identify the key requirements of CPP platforms (Section 3). We then review the software architectures of the current CPP and evaluate them with respect to those requirements (Section 4). We then propose a new CPP architecture based on component, aspect and object models, as well as rewrite rules for constraint handling and model transformation, and we argue that it better fulfils the CPP requirements than current CPP architectures (Section 5). We conclude by quickly describing the current status and next steps in our implementation of this architecture (Section 6).

2. Background

Constraint programming is the cutting-edge IT to automate and optimize tasks such as resource allocation, scheduling, routing, layout and design verification in the most diverse industries. A **constraint program** models an application domain using a subset of first-order logic restricted to atom conjunctions. Each model consists of:

- A set of variables X_1, ..., X_n;
- For each variable X_i, a corresponding domain D_i of possible values (*i.e.,* constant symbols), *e.g.,* {red, green, blue} or floating point numbers;
- For each domain D_i a set of functions f_i^1, ... f_i^o with domain and range in D_i (*e.g.,* mix, +, **);
- A logical formula F of the form $C_1 \wedge ... \wedge C_p$ where each C_i is an atom that relates terms formed from variables, constants and functions (*e.g.,* $X_1 \neq X_2$, X_1 = blue, X_1 = mix(red,blue), $X_1 \geq 2.0000 * 3.1416 * X_2$).

Each C_i in F is called a **constraint** because it restricts the possible value combinations of the variables X_1, ... X_n that occur in it within their respective domains D_1, ... D_n. A solver takes as input a constraint problem instance P_i in the form of a conjunctive formula F_i. If P_i is exactly constrained (*e.g.,* $X,Y,Z \in N \wedge X+Y=Z \wedge 1<X \wedge X<Z \wedge X<Y \wedge Y<Z \wedge Z<6$) the solver returns as output a formula F_o in determined solved form (*i.e.,* of the form $X_1 = v_1 \wedge ... \wedge X_n = v_n$ with X_is variables and v_is constants) that is logically equivalent to F_i (*e.g.,* $X=2 \wedge Y=3 \wedge Z=5$)**.** If P_i is overconstrained, (*e.g.,* $X,Y,Z \in N \wedge X+Y=Z \wedge 1<X \wedge X<Z \wedge X<Y \wedge Y<Z \wedge Z<4$) the solver returns `false`. If P_i is underconstrained (*e.g.,* $X,Y,Z \in N \wedge X+Y=Z \wedge 1<X \wedge X<Z \wedge X<Y \wedge Y<Z \wedge Z<7$) the solver returns a formula F_o logically equivalent to F_i but syntactically *simpler* (*e.g.,* $(X=2 \wedge 3 \leq Y \wedge Y \leq 4 \wedge 5 \leq Z \wedge Z \leq 6)$, or $(X=2 \wedge ((Y=3 \wedge Z=5) \vee Y=4 \wedge Z=6)$. Key simplicity factors include fewer constraints, fewer variables, shorter atoms, higher proportion of atoms in determined solved form or (undetermined) solved form (*i.e.,* of the form $X_1 = t_1 \wedge ... \wedge X_n = t_n$ where each X_i is a variable and each t_i a term not containing occurrences of X_1, ..., X_{i-1}, X_{i+1}, ..., X_n). Figure 1 gives a simple illustrative example of a constraint solving task and solution. It is an instance of the classic map coloring problem: how to allocate variables representing regions on a map to values from a finite domain of colors, such that any two neighboring regions are allocated different colors? Figure 1 shows the problem as a constraint graph, with one node per variable/region and one arc per neighboring constraint. Above or below each node, the

variable domain is shown (with r, b, and g respectively abbreviating red, blue and green). In logic, this problem is represented as: (R1=r ∨ R1=b ∨ R1=g) ∧ (R2=r ∨ R2=b ∨ R2=g) ∧ (R3=r ∨ R3=b ∨ R3=g) ∧ (R4=r ∨ R4=b ∨ R4=g) ∧ (R5=r ∨ R5=b ∨ R5=g) ∧ (R6=r ∨ R6=b ∨ R6=g) ∧ (R7=r ∨ R7=b ∨ R7=g) ∧ ¬(R1=R2) ∧ ¬(R1=R3) ∧ ¬(R1=R4) ∧ ¬(R1=R7) ∧ ¬(R2=R6) ∧ ¬(R3=R7) ∧ ¬(R4=R5) ∧ ¬(R4=R7) ∧ ¬(R5=R6) ∧ ¬(R5=R7). In Figure 1, onecolor allocation solution is indicated by the grey boxes in each domain. Logically, is it represented by: R1=g ∧ R2=b ∧ R3=r ∧ R4=r ∧ R5=g ∧ R6=r ∧ R7=b. A CPP integrates the constraint task modeling syntax with that of a general purpose host programming language. It typically works by interleaving three main subtasks: **simplifying** a set of constraint into a simpler equivalent one (*i.e.,* leaving some of them implicit) **propagating** some logical consequences of a set of constraint (*i.e.,* making explicit some constraints they entail) and **searching** the space of possible variable combination values.

To achieve low-cost portability to multiple execution platforms and automate more development process sub-tasks, a **Model-Driven Architecture** (MDA) (18) switches the software engineering focus away from low-level source code towards high-level models, metamodels (*i.e.,* models of modeling languages) and model transformations that automatically map one kind of model into another. It prescribes the construction of a fully refined **Platform Independent Model (PIM)** together with two sets of model transformation rules to translate the PIM into source code via an intermediate **Platform Specific Model (PSM)**.

To achieve low-cost evolution, a **Component-Based** MDA (3) structures the PIM, PSM and source code as assemblies of reusable components, each one clearly separating the services interfaces that it provides to and requires from other components from its encapsulated realization of these services (itself potentially a recursive sub-assembly).

To achieve separation and reuse of **cross-cutting concerns** (*i.e.,* bits of processing that cannot be satisfactorily encapsulated in a single component following any possible assembly decomposition), **Aspect-Oriented** MDA (18) prescribes to model such concerns as PIM to PIM model transformations that weave the corresponding additional model bits at appropriate locations scattered throughout the main concern PIM.

Today, a component-based MDA can be fully specified using the UML2 standard (8) for it incorporates (a) a platform independent component metamodel, (b) the high level object-oriented functional constraint language OCL2 (21) to fully detail, constraint and query UML2 models, and (c) the **Profile** mechanism to define, in UML2 itself, UML2 extensions with platform specific constructs for diverse PSM. Model transformations for PIM to PIM aspect weaving and PIM to PSM to code translation can be specified using the rule-based, hybrid declarative-procedural Atlas Transformation Language (ATL) (4). It reuses OCL2 in the left-hand and right-hand sides of object-oriented model rewrite rules. These rules are applied by the ATL Development Tool (ATLDT), conveniently deployed as an Eclipse Plug-in (6). In a **Component-Based Aspect-Oriented Model-Driven Architecture** (CBAOMDA), only the core concern PIM and the model transformations are hand-coded. The other models and the code are automatically generated from the core concern PIM by applying the transformation pipeline to it.

3. Constraint Programming Platforms Requirements

A thoughtful engineering process chooses between various architectural designs based on a prior identification of the key functional and non-functional requirements to make the software under construction a most practical and useful tool. What are such requirements for a CPP?

The first requirement is to provide an **expressive input task modeling language**. This is clearly decisive for the CPP's versatility and its ability to fulfill the promise of an integrated platform for multiple automated reasoning needs of an application. Less obvious, is that it is also crucial for its overall efficiency, for the key of efficient solving often lays as much on the way the task in modeled as it does on the chosen solving method and how this method is implemented. A platform with a very limited task modeling language does not allow for many logically equivalent formulations of the same task. For many tasks, it thus risks of not supporting any of the formulations that lend themselves to efficient solving.

For the same versatility and overall efficiency reasons, the second key requirement is to provide many solving **method customization and combination facilities**. This allows exploiting the peculiarities of subtly different solving task sub-classes, which often results in dramatic efficiency gains for many task instances.

A third requirement in the overall efficiency puzzle is **efficient implementation** techniques for the available solving methods. It is somewhat conflicting with the two others since raising expressiveness often has the undesirable side-effect of increasing theoretical worst-case complexity, while method tweaking, mixing and matching facilities often brings some configuration and assembly run-time overhead.

A fourth requirement is **solution adaptation**. Consider two task instances T_1 and T_2 that differ only by a few more and/or a few less constraints. Once it has computed a solution S_i^1 for T_1, an adaptive solver is able to reuse S_i^1 so as to (a) find a solution S_a^2 for T_2 that is minimally distant from S_i^1 and (b) find it more quickly than a solution S_s^2 computed from scratch. A non-adaptive solver can only solve T_2 from scratch which may lead to a solution S_s^2 that has very few common components with S_i^1. In the practical application context, a large distance between S_s^2 and S_i^1 often makes S_s^2 unusable. For example, imagine than S_i^1 was an initial task schedule for a large engineering project that requires adjustment due to execution delays. A computed from scratch schedule S_s^2 for the uncompleted tasks may well promise a shorter revised delivery date estimate than that of an adaptive schedule S_a^2 which additionally strives for stability from the original S_i^1. But if S_s^2 allocates resources in a vastly different way than S_i^1 for the uncompleted tasks, the associated allocation reshuffling overhead cost generally far outweigh the gains of a shorter delivery date. Most available CPP do not provide adaptive solving.

The fifth key requirement of a practical CPP is a user-friendly, detailed **solution explanation** facility. Consider again the examples above, where a CPP propose alternative rescheduling plans for a late several billion dollars engineering project of a company flagship product. With so much at stake, adopting one of these alternatives requires the CPP to provide explanations that justify its discarding millions of possible others. It also requires to provide a concise summary of the contrasting trade-offs

embodied in two proposed alternative plans. The key challenge for such explanations and summaries is to be as directly understandable as possible by the executives with decision making power but no technical constraint solving background.

The sixth key requirement for a practical CPP is **seamless integration** within a variety of applications. Today, most of them are developed using **Component-based Object-oriented Imperative Platform** (COIP) such as EJB or .Net. There are two main reasons for this. The first is that these frameworks provide as consolidated built-ins the application independent services that constitute most of the running code of most information systems. These built-ins include high level API for database and GUI development, transparent secure persistence, scalable concurrent transaction handling and distributed deployment (including fully automated code generation to publish the various system functionalities as web services). The second is that the most advanced full life cycle software engineering methods and supporting CASE tools are based on the COI paradigm. This allows full integration of these CASE tools with COIP IDE and brings high gains in software productivity and quality. Thus, in today's practice, seamless integration of a CPP in applications, means encapsulating it as a Java or .Net component.

4. Constraint Programming Platform Architectures

Having defined the requirement of practical CPP, let us now review the various software architectures of available CPP and evaluate how they meet each of these requirements.

4.1. Component-based Object-oriented Imperative Platform API

The simplest CPP architecture is a library of classes or a component framework in a COIP, summarized in Figure 2. This is the case for example of the Java firstcs library (23) and the commercial C++ ILOG Solver (14). The first weak point of this architecture is to provide the least expressive constraint task modeling language: a restriction of the one described in Section 2, with the constraints, functions and constants all from the *closed, fixed set* implemented by the class library. Another drawback of the COIP architecture is its poor customizing facility, which is possible only through time-consuming low-level imperative code alterations. A COIP API provides some low-cost method combination facilities through component assembly. But the generation of solving step justifications for adaptation and explanations are cross-cutting concerns that cannot be encapsulated as separate components. To the best of our knowledge, there is neither an aspect-oriented nor comprehensively adaptive COIP API available to date. The main strengths of the COIP API architecture are implementation efficiency for the fixed set of available methods, seamless constraint solving services integration in application as components or classes, and the availability of COIP IDE and GUI development API.

The COIP API architecture:

- Represents the solving task declaratively by a conjunction of API operation calls;
- Structures the solving method declaratively as components and objects, but represents its details procedurally as operations;
- Deploys the code as compiled components and objects.
- Uses a compiling method that is generally structured declaratively as objects, but represented in details procedurally as operations.

4.2 Prolog + Procedural Libraries

The most widely used CPP architecture is the so-called parametric **Constraint Logic Programming** scheme CLP(D_1,...,D_q) (13). The task modeling language of CLP extends that of Section 2 with the equivalence and disjunction connectives and arbitrary symbolic atoms. A CLP(D_1, ..., D_n) program consists of Horn rules of the form: H :- G_1 ,..., G_r. The **head** H is an *arbitrary* first-order symbolic atom, and each goal $\mathbf{G_i}$ is either a symbolic atom (which appears as head in another rule) or a constraint atom (which does *not* appear as head in another rule) from a *fixed set* of built-ins that relate terms formed from functions and constants of a given domain D_i. The goal conjunction is called the **body** of the rule. Under CLP's closed-world assumption (16), the logical semantics of each rule subset sharing the same head, {H :- G_1^1,..., G_r^1., ..., H :- G_1^s,..., G_t^s.}, is $H \Leftrightarrow (G_1^1 \wedge ... \wedge G_r^1) \vee ... \vee (G_1^s \wedge ... \wedge G_t^s)$. The overall CLP program semantics is the conjunction of these equivalences. This extended expressivity allows modeling complex domain knowledge with high-level concepts declaratively defined from built-in constraints using Horn rules. Since these rules can be recursive and can contain function symbols, CLP provides a Turing-complete declarative constraint modeling language. It also provides a constraint query language: queries are simply headless rules. In CLP parlance, the task models and queries that a COIP API permits to express correspond to headless rules containing only built-in constraint atoms.

The first internal CPP architecture following the CLP(D_1,...,D_n) scheme is shown in Figure 3. It consists of n+1 components: a Prolog engine and n specialized library, L_1, ..., L_n, one for each domain D_i. Each library implements in a low-level imperative language (generally C or C++) fixed constraint simplification and propagation algorithms and heuristics, fine-tuned to the built-in constraints over D_i terms. The Prolog engine provides the solver for the arbitrary symbolic atoms, as well as a generic **Chronological Backtracking** (CBT) to search valid value combinations from underconstrained finite domains not reducible to singletons through simplification and propagation. Most CLP engines are simple extensions of backward chaining Prolog engines that upon encountering a constraint goal C_i of domain D_i adds it to a constraint store of the form $C_1 \wedge ... \wedge C_u$ and calls the imperative library for D_i to solve the updated store. If the store simplifies to `false`, the engine backtracks to the previous goal. Otherwise, it proceeds to the next goal. Data is exchanged between the solving library and the Prolog engine through instantiations of logical variables shared among the rule-defined goals and the built-in constraint goals.

Beyond task model expressiveness, the other strong point of the CLP Scheme is its efficient implementation, combining special purpose procedures with general purpose compiled Prolog code. As for weak points, the first is method customization and combination that requires changing low-level imperative library code that is not component-based and often not even object-oriented. The other weaknesses of the CLP scheme are inherited from Prolog: (a) no adaptation, (b) verbose, reasoning trace rarely presented in user-friendly GUIs supporting browsing at multiple abstraction levels, and (c) extremely difficult integration within applications for lack of components, interfaces, encapsulation, concurrent execution, and standard API for databases and GUI. More often than not, CLP engines only offer brittle bridges to C, C++ or more rarely Java as

sole mean of integration. The wide, conceptual impedance mismatch between the logic programming and COI paradigms constitutes in itself a serious integration barrier.

The Prolog + Library architecture:
- Represents the solving task declaratively by CLP Horn rules;
- Represents the solving method in part declaratively as CLP Horn rule and in part procedurally as imperative libraries;
- Deploys the code as rules to be applied by a CLP engine which is accessible via programming language bridges with no access to the embedded procedural solvers;
- Uses Prolog rules to declaratively compile the CLP rules into CLP virtual machine code and then from such intermediate code to native code.

4.3 Prolog + CHR

Constraint Handling Rules (CHR) (10) was initially conceived to bring rapid method customization and combination to the CLP scheme by making it fully rule-based. The idea is to substitute by a CHR base each procedural built-in solver of a CLP engine. The task modeling language of Prolog + CHR solver remains the same than in the Prolog + Library approach, since CHR are only used to declaratively define solving *methods* and not tasks. A CHR program is a set of constraint simplification rules, which are conditional rewrite rules of the form, S_1 ,..., S_a <=> G_1 ,.., G_b | B_1 ,..., B_c, and a set of constraint propagation rules, which are guarded production rules of the form: P_1 ,..., P_d ==> G_1 ,.., G_e | B_1 ,..., B_f. Each S_i, P_i, G_i and B_i is a constraint atom. The S_is and P_is are called **heads**, the G_is **guards** and the B_is **goals**. A goal conjunction is called a **body**. Constraint atoms that appear in a CHR head can only appear in other CHR heads and in CLP rule goals. They are called **Rule Defined Constraint** (RDC) atoms to distinguish them from **Built-In Constraint** (BIC) atoms that can appear only as guards and goals in CHR and only as head of Prolog rules that contain no constraint atoms in their bodies. Both kinds of constraint atoms can appear in CHR and CLP rule bodies. The logical semantics of simplification and propagation rules are $G_1 \wedge ... \wedge G_b \Rightarrow (S_1 \wedge ... \wedge S_a \Leftrightarrow B_1 \wedge ... \wedge B_c)$ and $G_1 \wedge ... \wedge G_e \Rightarrow (P_1 \wedge ... \wedge P_d \Rightarrow B_1 \wedge ... \wedge B_f)$ respectively. A CHR engine maintains a constraint store. Operationally, a rule fires when all its heads match against some RDC in the store, while its guards (together with the logical variable bindings resulting from the match) are entailed by the BIC in the store. A fired CHR rule adds its goals to the store. In addition, a fired simplification rule also deletes its heads from the store. A Prolog + CHR engine proceeds as a Prolog + Library engine, except that constraints are solved by forward chaining CHR rules instead of by calling a library procedure. CHR forward chaining stops when the store simplifies to `false` or when it reaches a fixed point, *i.e.,* when no firable rules can further simplify the store nor add new constraints to it.

As shown in Figure 4, Prolog plays multiple roles in the Prolog + CHR CPP architecture. First it solves the arbitrary symbolic constraints of the CLP model. Second, its built-in CBT search is reused to search finite domains that cannot be entirely reduced through CHR simplifications and propagations. Third, it is used as host platform to implement the CHR built-in constraints. Fourth, its built-in unification is reused to check the guard entailment precondition to the firing of each CHR rule.

Adding and altering CHR allows one to rapidly customize, extend, mix and match various solving methods. There are two main options for the CHR engine: it can

interpret the CHR, or it can compile them into imperative style Prolog rules (9). These Prolog rules (together with the application CLP rules and the Prolog rules that define the CHR built-ins) are then compiled to execution platform native code, through an intermediate level of CLP virtual machine code. The Prolog + CHR approach share all the adaptation, explanation and integration weaknesses of the Prolog + Library approach.

The Prolog + CHR architecture:
- Represents the solving task declaratively by CLP Horn rules;
- Represents the solving method declaratively by CHR conditional rewrite rules and Prolog Horn rules;
- Deploys the code as a hybrid CLP, CHR, Prolog rule base processed by a CLP-CHR engine which is accessible from an application via programming language bridges.
- Uses Prolog rules to declaratively compile the CLP, CHR and Prolog rules into a CLP virtual machine code and then from such intermediate code to native code.

4.4 Prolog CHR$^{\vee}$

CHR$^{\vee}$ (1) extends CHR with disjunctive bodies, *i.e.,* allowing rules of the form: S_1 ,..., S_a <=> G_1 ,.., G_b | (B_1^1,..., B_c^1) ;...; (B_1^g ,..., B_h^g) and P_1 ,..., P_d ==> G_1 ,.., G_e | (B_1^1 ,..., B_i^1) ;...; (B_1^j ,..., B_k^j) with the expected corresponding logical semantics: $G_1 \wedge ... \wedge G_b \Rightarrow (S_1 \wedge ... \wedge S_a \Leftrightarrow (B_1^1 \wedge ... \wedge B_c^1) \vee ... \vee (B_1^g \wedge ... \wedge B_h^g)$, and $G_1 \wedge ... \wedge G_e \Rightarrow (P_1 \wedge ... \wedge P_d \Rightarrow (B_1^1 \wedge ... \wedge B_i^1) \vee ... \vee (B_1^j \wedge ... \wedge B_k^j)$. Operationally, disjunctive bodies introduce the need for backtracking search in the CHR engine. When a disjunctive rule R triggers, one of its alternative bodies (B_1^m ,..., B_1^n) is chosen to be added to the constraint store and the engine then continues CHR forward chaining. However, if at a latter point, the store simplifies to `false`, instead of terminating, the CHR$^{\vee}$ engine then backtracks to (B_1^m,...,B_1^n) and deletes it from store together with all the constraints that were subsequently added (directly or indirectly) based on the its presence in the store. It then adds to the store the next alternative body (B_o^p,...,B_o^q) in R and resumes rule forward chaining. CHR$^{\vee}$ not only extends CHR with disjunctive bodies, but it also extends Prolog with multiple heads (1). Recall from Section 4.2 that the semantics of Prolog rules that share the same head is: is H $\Leftrightarrow (G_1^1 \wedge ... \wedge G_r^1) \vee ... \vee (G_1^s \wedge ... \wedge G_t^s)$. This precisely matches the semantics of the single head, CHR$^{\vee}$ rule: H <=> (G_1^1,..., G_r^1) ;...; (G_1^s ,..., G_t^s). Thus, CHR$^{\vee}$ is a single language that is more expressive to model constraint *tasks* than CLP and more expressive to model constraint *methods* than CHR. Using CHR$^{\vee}$ instead of CLP to model constraint tasks provides the power of full first order logic to define complex constraints from a minimum set of very primitive ones like `true`, `false` and = (syntactic equality between first order logic terms). Any first order logic formula F can be converted to a semantically equivalent formula N in the implicative normal form $P_1 \wedge ... \wedge P_n \Rightarrow B_1 \vee ... \vee B_o$ which is precisely the semantics of a guardless CHR$^{\vee}$ propagation rule.

All currently available implementations of CHR$^{\vee}$ are Prolog-based. Their architecture is shown in Figure 5. As for the hybrid Prolog + CHR architecture, some implementations interpret the CHR$^{\vee}$ rules, while others compile them into imperative style Prolog rules. While this architecture uses CHR$^{\vee}$ instead of CLP rules to model the solving task and to implement the arbitrary symbolic constraints, it nevertheless still relies on Prolog rules to define the CHR$^{\vee}$ built-in constraints and on Prolog's naive CBT search to process

disjunctive bodies and underconstrained finite domains. This makes them significantly slower than procedural libraries that rely on more efficient methods for these tasks such as **Conflict-Directed Backjumping** (CDBJ) or **stochastic local search** (16). This also makes them inherit the other already mentioned weaknesses of the Prolog-based CPP architectures.

The Prolog CHR$^{\vee}$ architecture:
- Represents the solving task declaratively by CHR$^{\vee}$ disjunctive conditional rewrite rules;
- Represents the solving method declaratively by CHR$^{\vee}$ disjunctive conditional rewrite rules and Prolog Horn rules;
- Deploys the code as a hybrid CHR$^{\vee}$, Prolog rule base processed by a CLP-CHR engine which is accessible from an application via programming language bridges.
- Use Prolog rules to declaratively compile the CHR$^{\vee}$ and Prolog rules into a CLP virtual machine code and then from such intermediate code to native code.

Prolog engines SICStus (17) and ECLiPse (7) offer all three Prolog-based CPP architectures.

4.5 Compiling CHR to COIP API

This most recent CPP architecture, shown in Figure 6, attempts to combine the respective strengths of the COIP API and Prolog + CHR architectures by:
- Representing the solving task declaratively by CHR conditional rewrite rules;
- Representing the solving method in part declaratively by CHR conditional rewrite rules and in part procedurally by COIP class operations;
- Deploying the code as COIP objects.

These deployed objects result from a two stage compilation scheme: (a) compiling the CHR to COIP source code, followed by (b) compiling the resulting source code, together with the COIP classes that implement the CHR built-in constraint handlers, into deployable code. The compiling method is declaratively structured as COIP classes but its details are procedurally represented as operations. In this approach, a COIP such as Java substitutes Prolog as the underlying host language for the CHR (12), (24). This requires re-implementing from scratch in the COIP services that were provided by reusing Prolog built-ins: unification to check the CHR guard entailment condition and CBT for finite domain search. While requiring more work, this also creates new opportunities such as relying on more efficient specialized algorithms for these tasks *e.g.,* CDBJ instead of CBT. It also allows incorporating solution adaptation techniques such as **Justification-Based Truth-Maintenance** (JTMS) (25), where each constraint in the store is kept jointly with pointers to the rule which firing inserted the constraint in the store and the justification for such firing, namely the RDC that matched the rule heads and the matching equations and BIC that entailed the rule guards.

The Java CHR engines JCHR (12) and DJCHR (24) follow this architecture, with the latter incorporating a JTMS scheme to provide efficient solution adaptation. As interface, they only provide a Java API. The lack of an interactive GUI makes them unpractical for rapid testing and application-specific customization and combination of solving methods.

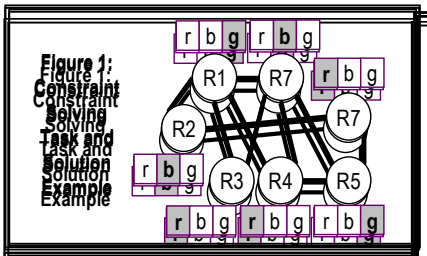


Figure 1: Constraint Solving Task and Solution Example

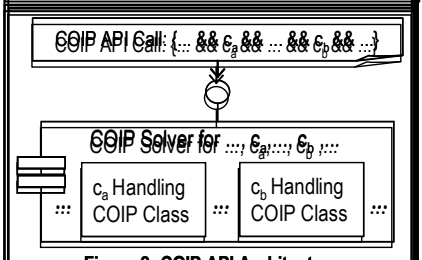


Figure 2: COIP API Architecture

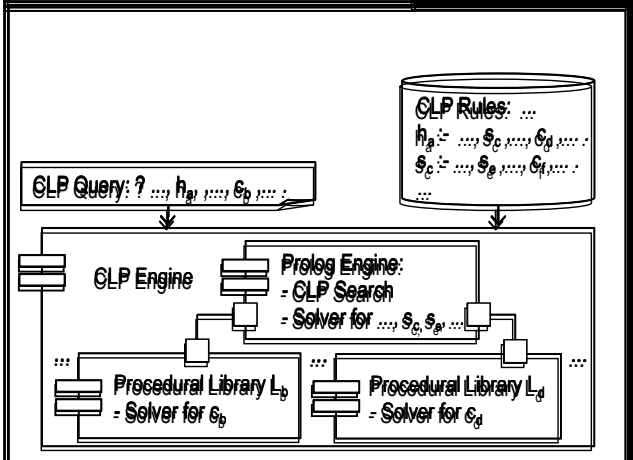


Figure 3: Prolog + Procedural Library Architecture

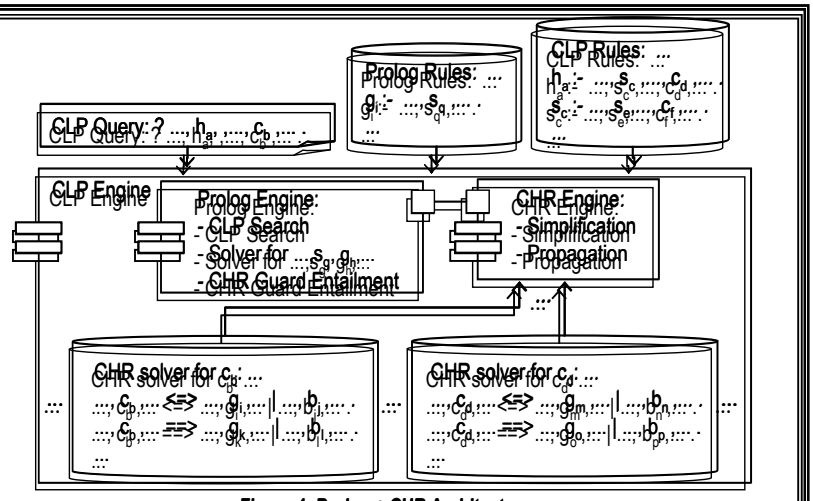


Figure 4: Prolog + CHR Architecture

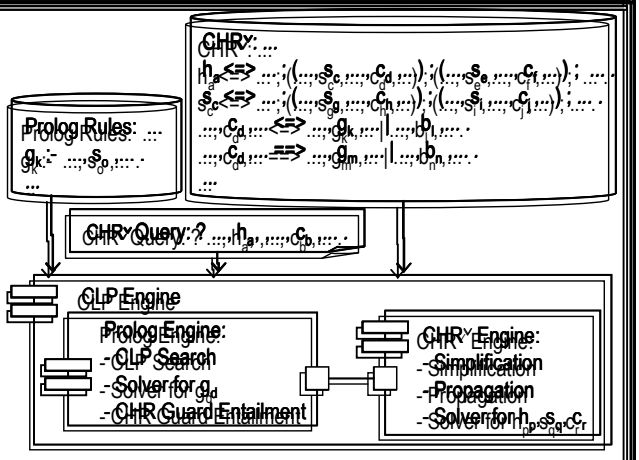


Figure 5: Prolog/CHR Architecture

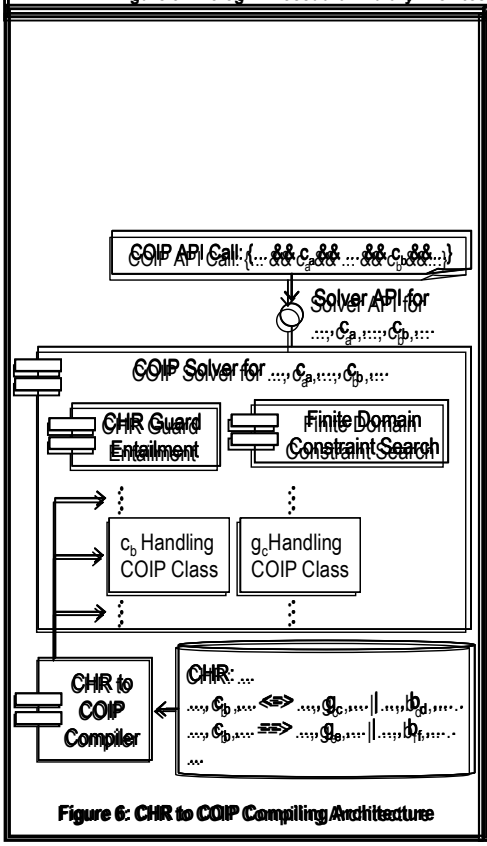


Figure 6: CHR to COIP Compiling Architecture

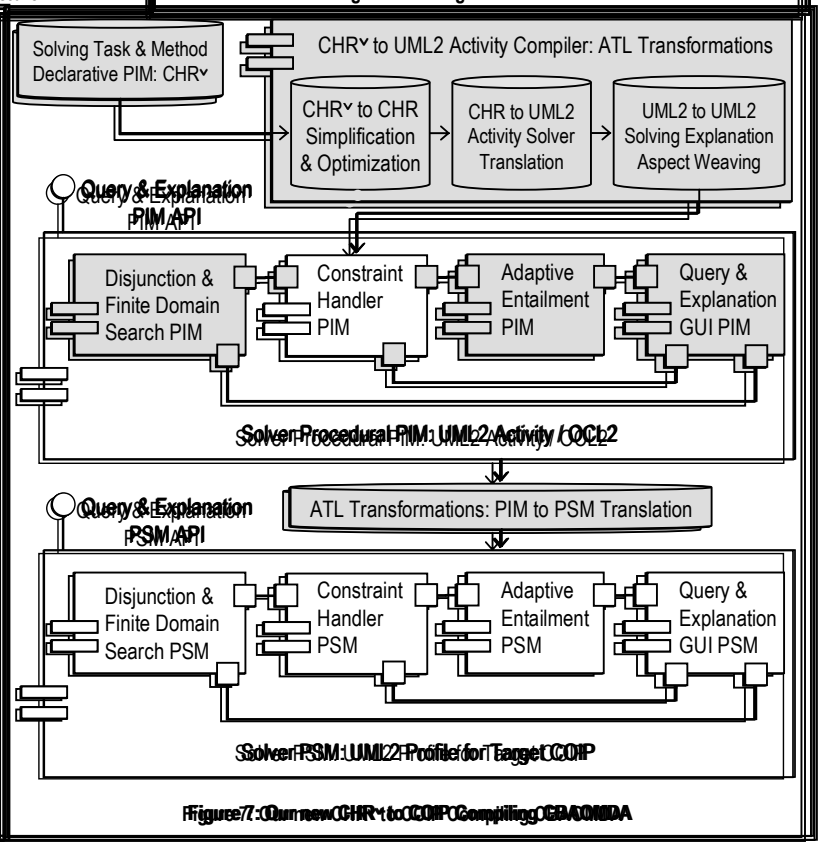


Figure 7: Our new CHR to COIP Compiling GRADUDA

5. A New Adaptive CHR$^\vee$ to COIP Compiling CBAOMDA for CPP

In the previous section, we have seen that all CPP proposed so far only very partially fulfill the six key requirements for a practical CPP. In this section, we propose a new CBAOMDA for CPP that aims to simultaneously fulfil them all. It is based on the following principles:

1. To combine very expressive solving task modeling with rapid method customization and combination, use **CHR$^\vee$** as uniform language to **declaratively represent both the solving task and the solving method** (except for a minimal set of CHR built-in constraints represented by Boolean operations of COIP classes);
2. To combine solution adaptation with solution explanation generation, incorporate the **JTMS** techniques of DJCHR and extend them to deal with **disjunctive CHR**;
3. To provide rapid prototyping deploy the CHR$^\vee$ engine as an **Eclipse plug-in** (6) with a GUI to **interactively submit queries and inspect solution explanations** at various levels of details;
4. To combine efficient solving implementation techniques with seamless integration in applications, build the first **CHR$^\vee$ to COIP compiler**;
5. To go one extra step towards reuse and extensibility, follow a an object-oriented, **component-based, model-driven architecture** for the overall CHR$^\vee$ engine;
6. To go one extra step towards easy to customize declarative code, define the CHR$^\vee$ compiler as a base of **object-oriented model transformation rewrite rules**;
7. To go one extra step towards separation of concerns, represent the **JTMS** and solving **explanation generation** processing models **as orthogonal aspects** separated from the core solving model and incorporate them by using weaving model transformations.

The high-level blueprint of the resulting architecture is shown in Figure 7, where the hand-coded elements are highlighted in gray while the automatically generated ones are in white. It has five layers: declarative PIM, procedural PIM, PSM and COIP source code and deployed COIP code (the last two omitted for conciseness). At the four lowest layers, the solver is an assembly of four components: (1) the constraint handler that encapsulates the constraint simplification and propagation techniques, (2) the search component needed to process CHR$^\vee$ disjunctions and finite domains not reducible to singletons by the handler, (3) the adaptive entailment component to determine which CHR$^\vee$ can be fired given the current state of the store, and (4) a GUI to type in queries with which to initialize the store and provide explanation for the solver answer and reasoning. The constraint handler is automatically generated by compiling the CHR$^\vee$ which represent relationally the solving task and method at the declarative PIM layer. The three other PIM components are modeled in an object-oriented procedural way as UML2 classes and activities with OCL2 constraints and expressions. The CHR$^\vee$ to UML2 compiler is a pipeline of three main ATL rule bases. The first transforms the input full syntax CHR$^\vee$ program into a semantically equivalent core syntax CHR program already optimized for fast procedural execution. The second translates this declarative core CHR program into an operationally equivalent generic object-oriented procedural representation as a UML2 class and activity diagram. The third weaves to this model, the additional classes and activities needed to generate the reasoning explanations to be displayed through the GUI. The resulting constraint handler is

incorporated in the procedural PIM assembly by connecting it to the manually modeled components. Applying an ATL rule base transforms this PIM assembly into a corresponding PSM assembly modeled using a UML2 profile for a given target COIP deployment platform. Applying another ATL rule base (omitted from Figure 7) then generates the COIP platform source code from this PSM. A target COIP platform IDE is then used to compile and deploy the source code for execution in overall application assembly.

This innovative architecture reconciles:

- Expressive solving task modeling in CHR$^\vee$;
- Rapid method customization and combination in CHR$^\vee$;
- Efficient method implementation by compiling CHR$^\vee$ onto a COIP classes which are then compiled to bytecode or native code;
- Solution adaptation by incorporating JTMS;
- Solution explanation by generating a solving trace exploiting the JTMS justifications and featuring a trace browsing GUI;
- Seamless integration in application by providing the solving services as a COIP interface.

It thus promises to be the first to fulfill all six key requirements of a CPP. Table 1 sums up how each CPP architecture fulfills these requirements.

	COIP API	Prolog + Handler Library	Prolog + CHR	Prolog CHR$^\vee$	CHR to COIP Compiler	CBAOMD CHR$^\vee$ to COIP Compiler
Task Expression	*	**	**	***	**	***
Method Customization	*	*	**	**	**	***
Efficient Implement.	***	**	*	*	*	**
Solution Adaptation	*	*	*	*	***	***
Solution Explanation	*	*	**	**	*	***
Seamless Integration	**	*	*	*	**	***

Table 1: Compared fulfillment of CPP requirements by CPP architectures.

6. Conclusion

In this paper, we identified six key requirements for a versatile and practical CPP. We used these requirements to critically review five prominent architectures among currently available CPP. We showed that each of them fails to adequately fulfill roughly half of these requirements. There is thus much room for improvement in the field of CPP architecture. To contribute to such improvement we proposed an innovative, CBAOMDA that (a) compiles CHR$^\vee$ rules into COIP source code, in several stages, using UML2 as an intermediate language, (b) incorporates JTMS techniques to support efficient solution adaptation and explanation, (c) includes a GUI to pass as query input a constraint task instance and browse the generated justification-based explanation at various levels of detail and (d) also includes an API to seamlessly provide the same query, solving and explanation facility to external software. We discussed why such architecture is the first one ever proposed with the potential to fulfil all key six requirements of a CPP. Naturally, confirming such potential will require

(a) implementing a running prototype of the architecture, (b) measuring its efficiency against state of the art solvers on benchmark tasks and (c) integrating it in within various practical applications for usability validation. We now stand midway towards the first of these goals. We have developed in UML2/OCL2 the fully refined PIM of an adaptive guard entailment component, as well as the ATL rules to compile a CHR$^{\vee}$ base into a constraint handler PIM in UML2/OCL2. The detail of this compiler is the object another publication currently in preparation. The pieces still missing from puzzle are the PIM for the search and GUI components, as well as the ATL rules for the PIM to PSM and PSM to source code translations. Together, our innovative architecture and its implementation will make practical contributions to several fields beyond constraint programming. To automated reasoning, it will provide the first scalable yet highly versatile base component on top of which to assemble and integrate deduction, abduction, default reasoning, inheritance, belief revision, belief update, planning and optimization services. To innovative programming language compiler and run-time system engineering, it will show the benefits of the component-based, aspect-oriented and model-driven approaches. It will also extend the scope of application of these approaches by showing that their practical benefits are even greater for cutting edge systems that perform intelligent processing with high aggregated value than for the straightforward GUI to database back to GUI translations performed by the standard web information systems for which these approaches were initially conceived.

7. References

(1) Abdennadher, S. 2001. Rule-based Constraint Programming: Theory and Practice. Habilitationsschrift, Institut für Informatik, Ludwig-Maximilians-Universität München.

(2) Ait-Kaci, H., Nasr, R. 1986. LOGIN: A Logic Programming Language with Built-in Inheritance. In Journal of Logic Programming, 3:185--215.

(3) Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wust, J. and Zettel, J. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.

(4) The ATL User Manual. http://www.eclipse.org/gmt/atl/doc/.

(5) Costa, V.S., Page, D., Qazi, M., Cussens, J. 2003. CLP(BN): Constraint Logic Programming for Probabilistic Knowledge. In Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence. Acapulco, Mexico.

(6) Eclipse: An Open Development Platform. http://www.eclipse.org/

(7) The ECLiPSe Constraint Programming System. http://eclipse.crosscoreop.com/.

(8) Eriksson, H.E., Penker, M., Lyons, B., Fado, D. 2004. UML 2 Toolkit. Wiley.

(9) Holzbaur, C., Frühwirth, T. 1998. Compiling Constraint Handling Rules. In Proceedings of the ERCIM/COMPULOG Workshop on Constraints, Amsterdam.

(10) Frühwirth, T. 1994. Theory and Practice of Constraint Handling Rules. In Journal of Logic Progamming. 19(20).

(11) Jin, Y., Thielscher, M. 2006. Iterated Belief Revision, Revised. In Artificial Intelligence. (to appear).

(12) KULeuven JCHR. http://www.cs.kuleuven.be/~petervw/JCHR/.

(13) Marriott, K. and Stuckey, P. 1998. Programming with Constraint: An Introduction. MIT Press.

(14) Puget, J.F. 1994. A C++ Implementation of CLP. In Proceedings of the Second Singapore International Conference on Intelligent Systems. Singapore.

(15) Robin J., Vitorino, J. 2006. ORCAS: Towards a CHR-Based Model-Driven Framework of Reusable Reasoning Components. In Proceedings of the 20th Workshop on (Constraint) Logic Programming (WLP'06).Vienna, Austria.

(16) Russell, S. and Norvig, 2003. P. Artificial Intelligence: A Modern Approach (2nd Ed.). Prentice-Hall.

(17) SICStus Prolog. http://www.sics.se/isl/sicstuswww/site/index.html

(18) Stahl, T., Völter, M. 2006. Model-Driven Software Development: Technology, Engineering, Management. Wiley.

(19) Thielscher, M. 2002, Programming of Reasoning and Planning Agent with FLUX. In Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR), Toulouse, France.

(20) Vitorino, J., Robin, J., Frühwirth, T. Towards a Model Transformation Based Compiler for Disjunctive Constraint Handling Rules. Submitted to the 9th International Conference on Enterprise Information System, Madeira, Portugal.

(21) Warmer J., Kleppe A. 2003. The Object Constraint Language (2nd Ed.): Getting Your Models Ready for MDA, Addison-Wesley.

(22) WebCHR: http://www.cs.kuleuven.be/~dtai/projects/CHR/.

(23) Wolf, A. 2006. Object-Oriented Constraint Programming in Java Using the Library firstcs. In Proceedings of the 20th Workshop on (Constraint) Logic Programming (WLP'06).Vienna, Austria.

(24) Wolf, A. 2001. Adaptive Constraint Handling with CHR in Java. In Lecture Notes in Computer Science, 2239, Springer.

(25) Wolf, A., Gruenhagen, T., Geske, U. 2000. On Incremental Adaptation of CHR Derivations. In Journal of Applied Artificial Intelligence 14(4). Special Issue on Constraint Handling Rules.

Logic Programming for Verification of Object-Oriented Programming Law Conditions

Leandro de Freitas[1] , **Marcel Caraciolo**[1] , **Márcio Cornélio**[1]

[1]Departamento de Sistemas Computacionais
Escola Politécnica — Universidade de Pernambuco
50720-001 Recife, PE, Brazil

{ldf,mpc,mlc}@dsc.upe.br

Abstract. *Programming laws are a means of stating properties of programming constructs and resoning about programs. Also, they can be viewed as a program transformation tool, being useful to restructure object-oriented programs. Usually the application of a programming law is only allowed under the satisfaction of side-conditions. In this work, we present how the conditions associated to object-oriented programming laws are checked by using Prolog. This is a step towards a tool that allows user definable refactorings based on the application of programming laws.*

1. Introduction

Object-oriented programming has been acclaimed as a means to obtain software that is easier to modify than conventional software [Meyer 1997]. Restructuring an object-oriented program is an activity known as refactoring [Opdyke 1992, Fowler 1999], allowing us, for instance, to move attributes and methods between classes or to split a complex class into several ones. These modifications just change the internal software structure without affecting the software behaviour as perceived by users. Work on refactoring usually describes the steps used for program modification in a rather informal way [Fowler 1999, Opdyke 1992].

The refactoring practice usually relies on test and compilation cycles, based on small changes applied to a program. Works in the direction of formalising refactorings deal mainly with the identification of conditions that must be satisfied to guarantee that a change to a program is behaviour preserving [Opdyke 1992, Roberts 1999]. Opdyke [Opdyke 1992] proposes a set of conditions required for application of refactorings, whereas Roberts [Roberts 1999] introduces postconditions for refactorings, allowing the definition of refactoring chains. The possibility of defining conditions that must be satisfied to apply a chain of refactorings is a benefit of the introduction of postconditions.

In our approach, programming laws are the basis for the derivation of refactoring rules, along with laws that lead to data refinement of classes [Cornélio 2004]. These laws precisely indicate the modifications that can be done to a program, with corresponding proof obligations. Using laws, program development is justified and documented. In order to deal with the correctness of refactorings, in our approach the derivation of a refactoring rule is carried out by the use of laws that are themselves proved [Cornélio 2004] against the semantics [Cavalcanti and Naumann 1999, Cavalcanti and Naumann 2000] of our language. Here we consider sequential programs and we do not take into account programs with space and time issues.

```
class Account extends object
  pri balance : int;
  meth getBalance ≙ (res r : int • r := self.balance)
  meth setBalance ≙ (val s : int • self.balance := s)
  new ≙ self.balance := 0
end •
var acct : Account •
  acct := new Account();
  acct.setBalance(10);
end
```

Figure 1. Example of program in ROOL

In order to apply an object-oriented programming law it is necessary to check side-conditions to its application or to prove some proof obligations. In this work we present an application of logic programming to verify conditions of programming laws, determining whether a law can be applied to a program or not. Program transformations accomplished by the use of programming laws preserve program behaviour [Cornélio 2004]. The use of logic programming was inspired by the JTransformer framework [Rho et al. 2003], a query and transformation engine for Java source code.

This paper is organised as follows. In Section 2 we present our study language and object-oriented programming laws. In Section 3, we introduce the structure of the program fact databases that we use for representing program syntax trees. In Section4, we describe how we verify conditions of object-oriented programming laws. We discuss some related work in Section 5. Finally, in Section 6, we present our conclusions and suggestions for future work.

2. The Language and Laws of Programming

The language we use for our study is called ROOL (an acronym for Refinement Object-Oriented Language) and is a subset of sequential Java with classes, inheritance, visibility control for attributes, dynamic binding, and recursion [Cavalcanti and Naumann 2000]. It allows reasoning about object-oriented programs and specifications, as both kinds of constructs are mixed in the style of Morgan's refinement calculus [Morgan 1994]. The semantics of ROOL is based on weakest preconditions [Cavalcanti and Naumann 2000]. The imperative constructs of ROOL are based on the language of Morgan's refinement calculus [Morgan 1994], which is an extension of Dijkstra's language of guarded commands [Dijkstra 1976].

A program *cds* • *c* in ROOL is a sequence of classes *cds* followed by a main command *c*. Classes are declared as in Fig. 1, where we define a class *Account*. Classes are related by single inheritance, which is indicated by the clause **extends**. The class **object** is the default superclass of classes. So, the **extends** clause could have been omitted in declaration of *Account*. The class *Account* includes a private attribute named *balance*; this is indicated by the use of the **pri** qualifier. Attributes can also be protected (**prot**) or public (**pub**). ROOL allows only redefinition of methods which are public and can be recursive; they are defined using procedure abstraction in the form of Back's parameterized commands [Cavalcanti et al. 1999]. A parameterised command can have the form

$e \in Exp$	$::=$	**self** | **super**	special 'references'
		| **null** | **error**	
		| **new** N	object creation
		| x	variable
		| $f(e)$	application of built-in function
		| e **is** N	type test
		| $(N)e$	type cast
		| $e.x$	attribute selection
		| $(e;\ x : e)$	update of attribute
$\psi \in Pred$	$::=$	e	boolean expression
		| $\psi \Rightarrow \psi$	
		| $(\vee\ i \bullet \psi_i)$	
		| $\forall x : T \bullet \psi$	

Table 1. Grammar for expressions and predicates

val $x : T \bullet c$ or **res** $x : T \bullet c$, which correspond to the call-by-value and call-by-result parameter passing mechanisms, respectively. For instance, the method *getBalance* in Fig. 1 has a result parameter *r*, whereas *setBalance* has a value parameter *s*. Initialisers are declared by the **new** clause. In the main program in Fig. 1, we introduce variable *acct* of type *Account* to which we assign an object of such class.

For writing expressions, ROOL provides typical object-oriented constructs (Table 1). We assume that x stands for a variable identifier, and f for a built-in function; **self** and **super** have a similar semantics to `this` and `super` in Java, respectively. The type test e **is** N has the same meaning as in e `instanceof` N in Java: it checks whether non-null e has dynamic type N; when e is **null**, it evaluates to false. The expression $(N)e$ is a type cast; the result of evaluating such an expression is the object denoted by e if it belongs to the class N, otherwise it results in error. Attribute selection $e.x$ results in a run-time error when e denotes **null**. The update expression $(e_1;\ x : e_2)$ denotes a copy of the object denoted by e_1 with the attribute x mapped to a copy of e_2. If e_1 is **null**, the evaluation of $(e_1;\ x : e_2)$ yields **error**. Indeed, the update expression creates a new object rather than updating an existing one.

The expressions that can appear on the left-hand side of assignments, as the target of a method call, and as result arguments constitute a subset *Le* of *Exp*. They are called left-expressions.

$$
\begin{array}{lcl}
le \in Le & ::= & le1 \mid \textbf{self}.le1 \mid ((N)le).le1 \\
le1 \in Le1 & ::= & x \mid le1.x
\end{array}
$$

The predicates of ROOL (Table 1) include expressions of type **bool**, and formulas of the first-order predicate calculus.

The imperative constructs of ROOL, including those related to object-orientation concepts, are specified in the Table 2. In a specification statement $x : [\psi_1, \psi_2]$, x is the frame, and the predicates ψ_1 and ψ_2 are the precondition and postcondition, respectively. It concisely describes a program that, when executed in a state that satisfies the precondition, terminates in a state that satisfies the postcondition, modifying only the variables

$c \in Com$	::=	$le := e$	multiple assignment
		$\mid x : [\psi_1, \psi_2]$	specification statement
		$\mid pc(e)$	parameterised command application
		$\mid c;\ c$	sequential composition
		$\mid$ **if** $[]i \bullet \psi_i \rightarrow c_i$ **fi**	alternation
		$\mid$ **rec** $Y \bullet c$ **end** $\mid Y$	recursion, recursive call
		$\mid$ **var** $x : T \bullet c$ **end**	local variable block
		$\mid$ **avar** $x : T \bullet c$ **end**	angelic variable block
$pc \in PCom$	::=	$pds \bullet c$	parameterisation
		$\mid le.m \mid ((N)le).m$	method calls
		$\mid$ **self**.$m \mid$ **super**.m	
$pds \in Pds$	::=	$\varnothing \mid pd \mid pd;\ pds$	parameter declarations
$pd \in Pd$	::=	**val** $x : T \mid$ **res** $x : T$	

Table 2. Grammar for commands and parameterised commands

present in the frame. In a state that does not satisfy ψ_1, the program $x : [\psi_1, \psi_2]$ aborts: all behaviours are possible and nontermination too. The variable x is used to represent both a single variable and a list of variables; the context should make clear the case.

For alternation, we use an indexed notation for finite sets of guarded commands. A method in ROOL can use its name in calls to itself in its body. This is the traditional way to define a recursive method. ROOL also includes the construct **rec** $Y \bullet c$ **end**, which defines a recursive command using the local name Y. A (recursive) call Y to such a command is also considered to be a command. The iteration command can be defined using a recursive command. Blocks **var** $x : T \bullet c$ **end** and **avar** $x : T \bullet c$ **end** introduce local variables. The former introduces variables that are demonically initialised; their initial values are arbitrary. The latter introduces variables that are angelically chosen [Back and von Wright 1998]. This kind of variable is also known as logical constant, logic variable, or abstract variable. In practice, angelic variables only appear in specification statements.

As methods are seen as parameterised commands, which can be applied to a list of arguments, yielding a command, a method call is regarded as the application of a parameterised command. A call $le.m$ refers to a method associated to the object denoted by le. In a method call $e_1.m(e_2)$, e_1 must be a left expression.

As we intend to apply object-oriented programming laws as a means to prove more complex transformation like refactorings, we consider the application of an object-oriented programming laws as two-fold: verification of conditions and program transformation itself. Below we present an example of a law. We write '$(\rightarrow)$' when some conditions must be satisfied for the application of the law from left to right. We also use '$(\leftarrow)$' to indicate the conditions that are necessary for applying a law from right to left. We use '$(\leftrightarrow)$' to indicate conditions necessary in both directions. Conditions are described in the **provided** clause of laws.

To eliminate a class to which there are no references in a program, we apply Law 1 $\langle$*class elimination*$\rangle$ from left to right. This application requires that the name of the class declared in cd_1 must not be referred to in the whole program. In order to apply

this law from right to left, the name of the class declared in cd_1 must be distinct from the name of all existing classes; the superclass that appears in the declaration cd_1 is **object** or is declared in *cds*. Finally, only method redefinition is allowed for the class declared in cd_1. direction introduces a new class in the program.

Law 1 ⟨*class elimination*⟩

$cds\ cd_1 \bullet c = cds \bullet c$

provided

($\rightarrow$) The class declared in cd_1 is not referred to in *cds* or *c*;

($\leftarrow$) (1) The name of the class declared in cd_1 is distinct from those of all classes declared in *cds*; (2) the superclass appearing in cd_1 is either **object** or declared in *cds*; (3) and the attribute and method names declared by cd_1 are not declared by its superclasses in *cds*, except in the case of method redefinitions.

□

Using Law 2 ⟨*attribute elimination*⟩, from left to right, we can remove an attribute from a class, since this attribute is not read or written inside such a class. The notation $B.a$ refers to access to an attribute a via expressions whose static type is exactly B. Applying this law from right to left, we introduce an attribute in a class, since this attribute is new, not declared by such class, nor is declared by any superclass or subclass.

Law 2 ⟨*attribute elimination*⟩

class B **extends** A
  pri $a : T$; ads
  ops
end
	$=_{cds,c}$	**class** B **extends** A
		  ads
		  ops
		end

provided

($\rightarrow$) $B.a$ does not appear in *ops*;

($\leftarrow$) a does not appear in *ads* and is not declared as an attribute by a superclass or subclass of B in *cds*.

□

To eliminate a method from a class we use Law 3 ⟨*method elimination*⟩. We can eliminate a method from a class if it is not called by any class in *cds*, in the main command *c*, nor inside class *C*. For applying this law from right to left, the method m cannot be already declared in *C* nor in any of its superclasses or subclasses, so that we can introduce a new method in a class. The notation $B.m$ refers to calls to a method m via expressions whose static type is exactly B. We write $B \leq A$ to denote that a class B is a subclass of a class A.

Law 3 ⟨*method elimination*⟩

class C **extends** D
  ads
  meth $m \mathrel{\widehat{=}} pc$ **end**; ops
end
	$=_{cds,c}$	**class** C **extends** D
		  ads
		  ops
		end

provided

```
package(PackageId, MainId).
package(PackageId, ClassId).
class(ClassId, ClassName).
extends(ClassId, SuperClassId).
main(MainId, ``Main'').
mainCommand(MainId, MainCmdId).
```

Figure 2. Class and main command program facts

($\rightarrow$) $B.m$ does not appear in cds, c nor in ops, for any B such that $B \leq C$.
($\leftarrow$) m is not declared in ops nor in any superclass or subclass of C in cds.

□

There are also laws to deal with moving attributes and methods to superclasses, changing types of attributes and parameters, eliminating method calls, for instance, and other features [Borba et al. 2004, Cornélio 2004]. Some programming laws presented in [Borba et al. 2004, Cornélio 2004] can be considered basic refactorings when compared to the classification of refactorings presented by Opdyke [Opdyke 1992].

3. Program Facts Database Structure

Our aim is to check side-conditions of programming laws relying on a Prolog factbase that represents the abstract syntax tree of a ROOL program. The main reason to use Prolog is due to its declarative nature, allowing us to concentrate in the solution, not in the process to describe a solution. We have implemented a compiler to translate a ROOL program to a fact base. In fact, the input to our compiler is a ROOL program that is enriched with tokens that are recognized by the rewriting system Maude [Clavel et al. 2005]. This is a consequence of a decision of using Prolog as a means to verify conditions stated by programming laws, whereas Maude would be used for the implementation of the transformations described by the programming laws.

The compilation process is constituted by two phases. As already said, the compiler receives as input a ROOL program enriched with tokens that can be read by the Maude rewriting system. In the first phase, the compiler reads such an input program and generates an abstract syntactic tree. In the second phase, the compiler scans the syntactic tree and generates Prolog clauses that represent the program syntactic tree.

For every syntactic element of the language we define a fact in Prolog. For instance, we describe facts for classes and the main command (Fig.2). Although we do not have packages in ROOL, we decided to represent such a concept in program fact bases. We consider that a program is written inside a single package that is introduced in the fact `package` whose first element is the package identifier; the second element is a class identifier. We consider the main command as a particular case: the fact package also lists the identifier of the main command (`main`). Classes are introduced in the fact `class` whose first element is a class identifier and the second is a class name. Attributes and methods of a class have specific facts that take into account identifier of the class in which they are declared.

For attribute declaration, the fact `attribute` introduces an identifier for an attribute, the identifier of the class in which the attribute is declared, and the attribute name.

```
attribute(AttributeId, ClassId, 'AttributeName').
attributeType(AttributeId, 'Type', 'Visibility').
method(MethodId, ClassId, 'MethodName').
methodVal(MethodId, ValParamId).
methodRes(MethodId, ResParamId).
methodStat(MethodId, StatementId).
```

Figure 3. Attribute and method facts

```
assign(CmdId, ParentId, MethodId)
assignExp(CmdId, LeftExpId, ExpId)
```

Figure 4. Assignment facts

The type and visibility of an attribute are introduced by the fact `attributeType`. As already discussed, methods in ROOL are declared as parameterised commands. We separate facts about parameter declarations from facts about the method body itself, the method statement (command). Method parameters are described by distinguishing them according to the parameter passing mechanism. The declaration of a result parameter is introduced by the fact `methodRes`, we use `methodVal` for a value parameter. To introduce the parameter identifier, we use fact `varDec` for variable declaration (see Appendix A). The method body is introduced by the fact `methodStat` (for method commmand). The first element of this fact is the identifier of the method, and a list of identifiers of commands that appear in the method body.

As an example of facts about a command, we present facts related to assignment (Fig. 4) The fact `assign` introduces an assignment identifier, the parent command in which the assignment appears (a guarded command, for instance), and the identifier of the method in which the assignment is introduced. The assignment itself is constituted by a left-expression (the assignment target), and the expression that is assigned to the left-expression, the right-hand side of the assignment. More examples of facts can be found in Appendix A.

In Fig. 5, we present the program that appears in Fig. 1 in the syntax that is also read by the Maude rewriting system. After compiling the program presented in Fig.5, we obtain program facts as those presented in Fig. 6. Here we present just a subset of the facts for the program presented in Fig. 6.

4. Verifying Side-Conditions of Programming Laws

In this section, we present how we have implemented the verification of conditions for application of programming laws. Here we deal just with the activity of verifying conditions for the application of programming laws, not with the program transformation that is a consequence of a law application. We encoded in a Prolog program the conditions that appear in the object-oriented programming laws presented in [Borba et al. 2004, Cornélio 2004]. Our implementation can be seen as constituted by layers. The bottom layer is constituted by the facts that represent the abstract syntax tree of a program; the second layer is composed by clauses that express conditions associated to programming

```
class CLID 'Account extends CLID 'object {
pri 'balance:  Int;
meth MtID 'getBalance ^ = res 'r:  int .# 'r := self.'balance; # end
meth MtID 'setBalance ^ = val 's:  int .# self.'balance := 's; # end
new ^ = self.'balance := 0;
}
main <
var 'acct :  Account .#
    'acct := new 'Account();
    'acct.'setBalance(10);
# end >
```

Figure 5. Input example program

```
package(1000, 1001).
class(1001, "'Account").
extends(1001, 0000).  //Object  0000
attribute(1002, 1001, "'balance").
attributeType(1002, "int", "pri").

method(1003, 1001, "'getBalance").
methodRes(1003, 1004).                varDec(1004, 1003, 1003, "'r").
methodStat(1003, 1005).               VarDecType(1004, "int").

assign(1005, 1003, 1003).             exp(1006, 1005, 1003, "'r", null).
assignExp(1005, 1006, 1007).          exp(1007, 1005, 1003, "self", 1008).
                                      exp(1008, 1007, 1003, 'balance, null).
package(1000, 1020).
main(1020, "Main").                   mainCommand(1020, 1021).
varDec(1020, 1001, null, "'acct").    assign(1021, 1020, null).
varDecType(1020, "'Acoount").         assignExp(1021, 1022, 1023).
                                      exp(1022, 1021, null, "'acct", null).
new(1023, 1021, null).
newExp(1023, 1024, 1021).             exp(1024, 1023, null, "'Account", null).
```

Figure 6. Program facts for the example program of Fig.5

laws. The third layer is constituted by conjunction of clauses expressing the conditions of programming laws.

Some clauses appear in the implementation of conditions of distinct laws. For instance, the clause `superClass` (Fig. 7) checks whether a class whose identifier is `classId` has class `Ancestor` as superclass. Notice that this clause is based on the fact `extends` of program factbases. We use clause `attributeNameClass` (Fig. 8) to check whether an attribute name appears in a given class. In this clause, we use clause `returnAttributeID`, besides other clauses, that returns the attribute identifier. We consider clause `returnAttributeID` as basic.

Here we describe how we verify the conditions associated to the laws we presented in Section 2. Let us first consider the conditions for applying Law 1 ⟨*class elimination*⟩, from left to right. To eliminate a class, it cannot be referred to in the entire program. In other words, such a class cannot be type of attributes (`verifyClassAttType`), variables (`verifyClassVariableType`), and parameters (`verifyClassParamType`). Also, this class is not superclass of any class (`verifyClassIsSuperclass`), it is not used in type casts and tests (`verifyClassIsCast` and `verifyClassTypeTest`), and does not appear in **new** expressions (`verifyClassInNewExp`). The negation of

```
superClass(ClassID, Ancestor) :-
  extends(ClassID, Ancestor).
superClass(ClassID, Ancestor) :-
  extends(ClassID,Y),
  superClass(Y,Ancestor).
```

Figure 7. Clause for checking superclass relation

```
attributeNameClass(AttributeName,ClassName):-
   returnAttributeID(AttributeName, AttributeId),
   returnClassID(ClassName,ClassId),
   atributeIDClass(AttributeId, ClassId).
returnAttributeID(AttributeName, AttributeId):-
   attribute(AttributeId, _, AttributeName, _, _).
```

Figure 8. Clauses for obtaining attribute identifier and checking attribute name in class

these clause define clause `verifyClassEliminationLR` (see (1)).

```
verifyClassEliminationLR(ClassName,PackageId):-
   verifyClass(ClassName,ProgramID),
   not(verifyClassAttType(ClassName)),
   not(verifyClassParamType(ClassName)),
   not(verifyClassVariableType(ClassName)),          (1)
   not(verifyClassIsSuperclass(ClassName)),
   not(verifyClassIsCast(ClassName)),
   not(verifyClassTypeTest(ClassName)),
   not(verifyClassInNewExp(ClassName)).
```

Clause `verifyClassIsSuperclass` (Fig. 9) verifies whether a class identified by its class name is superclass of any class in a program. For this, it is necessary to retrieve the class identifier from the class name. We use the identifier to check if any other class in a program is a subclass of a class with such an identifier. On the other hand, to introduce a class in a program we have to check that the name of the class we are introducing is not already in the program—we negate clause `verifyClass` that checks if a class is already in a program. Moreover, we check if the superclass of the class we introduce is already in the program or is **object** (clause `verifySuperclasAlreadyDeclared`). We have also to guarantee that attributes of the new class are not present in the superclass by negating clause `verifyAttSuperlasses`. We have to be more careful with methods. If the new class we are introducing declares a method with a name that is already a name of a method of any superclass of the class being introduced, we require that this method has the same parameters, since in ROOL overloading is not allowed. These conditions are conjoined in the clause `verifyClassEliminationRL` (see (2)).

```
verifyClassIsSuperclass(ClassName):-
   returnClassID(ClassName, ClassID) ,
   package(PackageId,ClassID),
   subClass(ClassID, Descendant),
   package(PackageId,Descendant).
```

Figure 9. Verifying whether a class has subclasses

```
verifyClassEliminationRL(ClassName,PackageId):-
   not(verifyClass(ClassName,PackageId)),
   verifySuperclasAlreadyDeclared(ClassName),
   not(verifyAttSuperlasses(ClassName)),                    (2)
   ((verifyMethNameInSuperclasses(ClassName),
   verifyMethParamSuperclasses(ClassName));
   (not(verifyMethNameInSuperclasses(ClassName))).
```

The condition to remove a private attribute by using Law2 ⟨*attribute elimination*⟩, from left to right, is expressed by the clause `attElimLR` (see (3)) that requires the attribute is not read or written inside a class, which is expressed by clause`priAttAccess`.

```
attElimLR(AttributeName,ClassName) :-
   not(priAttAccess(AttributeName, ClassName)).             (3)
```

On the other hand, to introduce an attribute, we require that is not already present in a class—we negate `attDecClass`—nor it is declared in any superclass or subclass—we negate `attDecHierarchy`. This is expressed by the clause `attElimRL` (see (4)).

```
attElimRL(AttributeName,ClassName) :-
   not(attDecClass(AttributeName, ClassName)),              (4)
   not(attDecHierarchy(AttributeName, ClassName)).
```

The condition for applying Law3 ⟨*method elimination*⟩, from left to right, is expressed in condition `methElimLR` (see (5)). We require there are no calls to the method in the entire program. For this moment, we deal with the static type of attributes, variables, and parameters to check the type of method call targets to a method. We require that there are no calls on attributes whose type is the class from which we eliminate the method or any of its superclasses. This is also applied to variables and parameters. We have also to check object chains. For instance, consider a method call like $x.y.z.m()$, in which m is the method we want to eliminate. A method call as the one given above does not necessarily implies that we cannot eliminate method m, we have to check the declared type of z. If it is the class from which we intend to eliminate m or any of its subclasses, we cannot eliminate m, otherwise we can.

```
methElimLR(MethodName, ClassName) :-
  not(methodCallOnAttribute(MethodName, ClassName)),
  not(methodCallOnVar(MethodName, ClassName)),
  not(methodCallOnParam(MethodName, SubClassName)),
  not(methodCallOnObjectChains(MethodName, ClassName)).
```
(5)

To introduce a method in a class, we require the method to be new in the hierarchy (not declared in the class, nor in any superclass or subclass (see (6)).

```
methElimRL(MethodName, ClassName) :-
 newMethodInHierarchy(MethodName, ClassName).
```
(6)

These are examples of verification of conditions of some object-oriented programming laws using logic programming. Besides these laws, we have implemented verification of conditions of other 19 programming laws. We have 25 laws altogether.

5. Related Work

Opdyke [Opdyke 1992] formally describes conditions that must be satisfied to apply a refactoring. Some "low-level" refactorings [Opdyke 1992, Chapter 5] proposed by Opdyke are, in fact, programming laws of our language. This is the case of the refactoring that deals with the introduction of attributes, for instance. In fact, some conditions of programming laws are similar to conditions of Opdyke's "low-level" refactorings.

Roberts [Roberts 1999] goes a step further than Opdyke and describes both preconditions and postcondition of refactorings, allowing support for refactoring chains. The definition of postconditions allows for the elimination of program analysis that are required within a chain of refactorings. This comes from the observation that refactorings are typically applied in sequences intended to set up preconditions for later refactorings. Pre- and postconditions are all described as first-order predicates; this allows the calculation of properties of sequences of refactorings. We have not defined postconditions of programming laws; we describe the transformations of such laws as meta-programs, not by means of properties they have.

Kniesel [Kniesel 2005] enables conditional transformations to be specified from a *minimal* set of built-in conditions and transformations. In our approach, conditions of a refactoring, for instance, are defined with basis on object-oriented programming laws conditions. In this way, applications of programming laws serve to derive more complex transformation, like refactorings, that can be applied to programs. We have not defined a minimal set of conditions for law application, but defined layers that are composed by Prolog clauses. The bottom layer is the program factbase itself. Upon the factabse, we define conditions that may be common to different laws—like condition`verifyClass` that can be considered basic—or specific to a law. Notice that these conditions are clearly stated by programming laws. Kniesel [Kniesel 2005, Kniesel and Koch 2004] defines *conditional transformation (CT)* to be a program transformation guarded by a precondition, such that the transformation is performed only if its precondition is true. We can also view a programming law as a transformation with condition that can be checked by

the existence of elements in a program. The JTransformer program transformation engine [Rho et al. 2003], which is used as a backend for conditional transformations, has inspired us in the definition of our logic factbase. The conditions we implemented, as already said, are based on programming law conditions.

Li [Li 2006] has defined refactoring for Haskell programs along with a refactorer called HaRe. Some functional refactorings have object-oriented counterparts like renaming. However, there are refactorings that are specific to functional programs like the one that deals with monadic computation of expression. The Haskell refactorer (HaRe) deals with structural, module, and data-oriented refactorings. HaRe is based on Strafunski [Lämmel and Visser 2003] which is a Haskell-centered software bundle for implementing language processing components and can be instantiated to different programming languages. Since programming languages of different paradigms have distinct program structures, they have their own program collection of refactorings. In our case, conditions and transformations are based on programming laws that were described and proved against the semantics of an object-oriented language [Cornélio 2004].

Tools that implement refactorings, like Eclipse [ecl], have a larger set of refactorings than ours, as we deal with a subset of sequential Java. Since our object-oriented programming laws deal with a language with a copy semantics, we can define refactorings that deal mainly with program structures, not involving sharing. On the other hand, we have identified some limitations of Eclipse when dealing with casts and when moving methods to a superclass [Cornélio 2004]. Also, differently from Eclipse, we intend to build a tool that allows programmers to define their own refactorings.

6. Conclusions

Changes in software are usually consequence of evolution or correction. However, some changes are performed to improve program structure, leading to a program that is easier to understand and to maintain. These changes modify the program structure without affecting the software behaviour as perceived by users.

A disciplined way to change a program without affecting its behaviour is to apply programming laws that guarantee correctness of program transformation by construction. This is based on proof of soundness of programming laws; in our case of laws that deal with imperative and object-oriented constructs [Borba et al. 2004, Cornélio 2004]. To apply an object-oriented programming law, conditions have to be satisfied. In this work, we presented an application of logic programming to check if the conditions of programming laws are satisfied, allowing us to strictly apply a law. Refactoring developers can take programming laws as a toolkit for the development of new refactorings. By using a tool that encodes programming laws, refactorings obtained are correct by construction.

We took advantage of using a declarative language like Prolog that facilitates the description of conditions. This can also be used to define preconditions for the application of a sequence composition of programming laws. In fact, this goes in the direction of the ConTraCT refactoring editor [Kniesel 2005]. It should also be necessary to describe the transformation defined by programming laws in a logic program. In fact, we have to deal with program transformations, we are considering the use of Constraint Handling Rules [Frühwirth 1998] or Transaction Logic [Bonner and Kifer 1994] in order to implement the transformations expressed by programming laws as direct changes in program

factbases.

We have already used rewriting systems for the mechanical proof of refactoring rules [Júnior et al. 2005]. However, we have not verified conditions for the application of programming laws. Our work here and the one presented in [Júnior et al. 2005] can be viewed as complementary. The conditions for a transformation would be checked by the implementation in logic programming, whereas the transformation would be realised by the rewriting system in which programming laws are encoded.

7. Acknowledgements

We would like to thank the anonymous referees for the comments that helped to improve the final version of this paper. The authors are supported by the Brazilian Research Agency, CNPq, grant 506483/2004-5.

References

[ecl] *Eclipse*. Eclipse.org. Available on-line http://www.eclipse.org/. Last accessed in March, 2007.

[Back and von Wright 1998] Back, R. J. R. and von Wright, J. (1998). *Refinement Calculus: A Systematic Introduction*. Springer-Verlag.

[Bonner and Kifer 1994] Bonner, A. J. and Kifer, M. (1994). An Overview of Transaction Logic. *Theoretical Computer Science*, 133(2):205–265.

[Borba et al. 2004] Borba, P., Sampaio, A., Cavalcanti, A., and Cornélio, M. (2004). Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, (52):53–100.

[Cavalcanti and Naumann 1999] Cavalcanti, A. L. C. and Naumann, D. (1999). A weakest precondition semantics for an object-oriented language of refinement. In *FM'99 - Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1439–1459.

[Cavalcanti and Naumann 2000] Cavalcanti, A. L. C. and Naumann, D. A. (2000). A Weakest Precondition Semantics for Refinement of Object-oriented Programs. *IEEE Transactions on Software Engineering*, 26(8):713–728.

[Cavalcanti et al. 1999] Cavalcanti, A. L. C., Sampaio, A., and Woodcock, J. C. P. (1999). An Inconsistency in Procedures, Parameters, and Substitution in the Refinement Calculus. *Science of Computer Programming*, 33(1):87–96.

[Clavel et al. 2005] Clavel, M. et al. (2005). *Maude Manual*. SRI International.

[Cornélio 2004] Cornélio, M. L. (2004). *Refactorings as Formal Refinements*. PhD thesis, Centro de Informática, Universidade Federal de Pernambuco. Also available at http://www.dsc.upe.br/∼mlc.

[Dijkstra 1976] Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

[Fowler 1999] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[Frühwirth 1998] Frühwirth, T. (1998). Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138.

[Júnior et al. 2005] Júnior, A. C., Silva, L., and Cornélio, M. (2005). Using CafeOBJ to Mechanise Refactoring Proofs and Application. In Sampaio, A., Moreira, A. F., and Ribeiro, L., editors, *Brazilian Symposium on Formal Methods*, pages 32—46.

[Kniesel 2005] Kniesel, G. (2005). ConTraCT - A Refactoring Editor Based on Composable Conditional Program Transformations. In Lämmel, R., Saraiva, J., and Visser, J., editors, *Generative and Transformational Techniques in Software Engineering*. Preproceedings of the International Summer School, GTTSE 2005.

[Kniesel and Koch 2004] Kniesel, G. and Koch, H. (2004). Static Composition of Refactorings. *Science of Computer Programming*, (52):9—51.

[Lämmel and Visser 2003] Lämmel, R. and Visser, J. (2003). A Strafunski Application Letter. In Dahl, V. and Wadler, P., editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag.

[Li 2006] Li, H. (2006). *Refactoring Haskell Programs*. PhD thesis, University of Kent.

[Meyer 1997] Meyer, B. (1997). *Object-Oriented Software Construction*. Prentide-Hall, second edition.

[Morgan 1994] Morgan, C. C. (1994). *Programming from Specifications*. Prentice Hall, second edition.

[Opdyke 1992] Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign.

[Rho et al. 2003] Rho, T. et al. (2003). *JTransformer Framework*. Computer Science Department III–University of Bonn. http://roots.iai.uni-bonn.de/research/jtransformer/.

[Roberts 1999] Roberts, D. B. (1999). *Practical Analysis for Refactoring*. PhD thesis, University of Illinois an Urbana-Champaign.

A. Facts for Commands and Expressions

A.1. Variable Declaration

```
varDec(VarDecId, ParentId, MethodId, 'VariableName')
varDecType(VarDecId, 'Type')
avarDec(AVarDecId, ParentId, MethodId, 'VariableName')
avarDecType(AVarDecId, 'Type')
```

A.2. Parameterized Command and Method Call

```
pCommand(CmdId, ParentId, MethodId)
methodCall(MethodCallId, ParentId, MethodId)
methodCallMeth(MethodCallId, ExpId, MethodName)
methodCallExp(MethodCallId, ExpCallId)
```

A.3. Types Test and Cast

```
is(IsId, ParentId, MethodId)
isExp(IsId, ExpId, 'ClassName')
cast(CastId, ParentId, MethodId)
castExp(CastId, ClassName, ExpId)
```

A Methodology for Removing LALR(k) Conflicts

Leonardo Teixeira Passos, Mariza A. S. Bigonha, Roberto S. Bigonha

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
CEP: 31270-010 – Belo Horizonte – MG – Brazil

{leonardo, mariza, bigonha}@dcc.ufmg.br

Abstract. *Despite all the advances brought by LALR parsing method by DeRemer in the late 60's, conflicts reported by LALR parser generators are still removed in an old fashion and primitive manner, based on analysis of a huge amount of textual and low-level data stored on a single log file. For the purpose of minimizing the effort and time consumed in LALR conflict removal, which is definitely a laborious task, a methodology is proposed, along with the set of operations necessary to its realization. We also present a tool and the ideas behind it to support the methodology, plus its plugin facility, which permits the interpretation of virtually any syntax specification, regardless of the specification language used.*

Resumo. *Apesar de todos os avanços obtidos pelo método de análise sintática LALR de DeRemer no final da década de 60, conflitos reportados por geradores de analisadores sintáticos LALR ainda são removidos de forma primitiva, baseada na análise de uma grande quantidade de dados de baixo nível, disponibilizados em um único arquivo de* log*. Com o propósito de minimizar o esforço e o tempo gasto na remoção de conflitos, que é definitivamente uma tarefa laboriosa, uma metodologia é proposta, juntamente com as operações necessárias à sua realização. Além disto, apresenta-se uma ferramenta e as idéias utilizadas em sua criação no suporte à metodologia, acrescido da descrição da facilidade oferecida pelo mecanismo de* plugins*, que permite virtualmente a interpretação de qualquer especificação sintática, sem a preocupação da linguagem de especificação utilizada.*

1. Introduction

The great advantage of working with LALR(k) grammars is the fact that they can be used by *parser generators* to automatically produce fully operational and efficient parsers, encoded in languages like C, C++, Java, Haskell, etc. Examples of LALR parser generators are YACC [Johnson 1979], CUP [CUP 2007], Frown [Frown 2007], among others. However, the specification of a grammar that is indeed LALR(k) is not a trivial task, specially when k is limited to one, which is often the case. This happens due to the recurrent existence of *conflicts*, i.e., non-deterministic points in the parser. It is quite common in a typical programming language grammar being designed to find hundreds, if not more than a thousand conflicts. To illustrate that, when implementing the LALR(1) parser for the Notus [Tirelo and Bigonha 2006] language, whose grammar has 236 productions, 575 conflicts were reported by the parser generator.

There exists many approaches to remove conflicts. Those based on ad hoc solutions, such as precedence and associativity settings, are not considered by the methodology proposed herein. We favor the method based on rewriting some rules of the grammar, without changing the defined language.

A usual way to remove conflicts is to analyse the output file created by the parser generator. This output consists of a considerable amount of textual data, from the numerical code associated to grammar symbols to the grammar and the LALR automaton itself. Using the Notus language as an example, the Bison parser generator (the GNU version of YACC) dumps a 54 Kb file, containing 6244 words and 2257 lines. The big amount of data and the fact that none of it is interrelated – hyperlinks are not possible in text files, make it very difficult to browse. The level of abstraction in these log files is also a problem, since non experts in LALR parsing may not interpret them accordingly. When facing these difficulties, these users often migrate to LL parser generators. Despite their simplified theory, this approach is not a real advantage, since LL languages are a proper subset of the LALR ones. Even for experts users, removing conflicts in such harsh environment causes a decrease of productivity. To face this scenario, in this paper we present a methodology for removing conflicts in non LALR(k) grammars. This methodology consists of a set of steps whose intention is to capture the natural way the compiler designer acts when handling conflicts: (i) understand the cause of the conflict; (ii) classify it according to known conflict categories; (iii) rewrite some rules of the grammar to remove the conflict; (iv) resubmit the specification to make sure the conflict has been eliminated. Each of these steps comprises a set of operations that must be supported. The realization of these operations are discussed when presenting SAIDE [1], a supporting tool for the proposed methodology.

This article is organized as follows: Section 2 gives the necessary background to understand the formulations used in later sections; Section 3 discusses conflicts in LR and LALR parsing; Section 4 presents the proposed methodology; Section 5 presents SAIDE, the mentioned tool to support the methodology, and Section 6 concludes this article.

2. Background

Before we present the methodology itself, it is necessary to establish some formal concepts, conventions, definitions and theorems. Most of the subject defined here is merely a reproduction or sometimes a slight variation of what is described in [Charles 1991], [DeRemer and Pennello 1982], [Aho and Ullman 1972] and [Kristensen and Madsen 1981]. It is assumed that the reader is familiar with LR and LALR parsing.

A context free grammar (CFG) is given by $G = (N, \Sigma, P, S)$. N is a finite set of nonterminals, Σ the finite set of terminals, P the set of rules in G and finally $S \in N$ is the start symbol. $V = N \cup \Sigma$ is said to be the vocabulary of G. When not mentioned the opposite, a given grammar is considered to be in its augmented form, given by (N', Σ', P', S'), where $N' = \{S'\} \cup N$, $\Sigma' = \{\$\} \cup \Sigma$, $P' = \{S' \rightarrow S\$\} \cup P$, considering that $S' \notin N$ and $\$ \notin \Sigma$.

The following conventions are adopted: lower case greek letters (α, β, ...) define strings in V^*; lower case roman letters from the beginning of the alphabet (a, b, ...) and

[1] Correct pronunciation: /saɪd/

t, bold strings and operator characters ($+, -, =, .$, etc) represent symbols in Σ, whereas letters from the end of the alphabet (except for t) denote elements in Σ^*; upper case letters from the beginning of the alphabet (A, B, ...) and italic strings represent nonterminals in N, while those near the end (X, Y, ...) denote symbols in V. The empty string is given by λ and the EOF marker by \$. The length of a string γ is denoted as $|\gamma|$. The symbol Ω stands for the "undefined constant".

An LR(k) automaton LRA_k is defined as a tuple $(M_k, V, P, IS, GOTO_k, RED_k)$, where M_k is the finite set of states, V and P are as in G, IS is the initial state, $GOTO_k : M_k \times V^* \rightarrow M_k$ is the transition function and $RED_k : M_k \times \Sigma_k^* \rightarrow \mathcal{P}(P)$ is the reduction function, where $\Sigma_k^* = \{w \mid w \in \Sigma^* \ \wedge 0 \leq |w| \leq k\}$.

A state, either a LR or LALR one, is a group of items. An item is an element in $N \times V^* \times V^*$ and denoted as $A \rightarrow \alpha \bullet \beta$.

The usual way to build the LALR(k) automaton is to calculate the LRA_0 automaton first. For such, let the components of LRA_0 be defined.

The set of states is generated by the following equation:

$$\begin{aligned} M_0 \ = \ & \{F^{-1}(CLOSURE(\{S' \rightarrow \bullet S\$\}))\} \cup \\ & \{F^{-1}(CLOSURE(F(q))) \mid q \in SUCC(p) \ \wedge \ p \in M_0\} \end{aligned}$$

where F is a bijective function that maps a state to a set of items (excluded the empty set) and

$$\begin{aligned} CLOSURE(is) &= is \cup \{B \rightarrow \bullet\beta \mid A \rightarrow \alpha \bullet B\omega \in is \ \wedge \ B \rightarrow \beta \in P\} \\ SUCC(p) &= \{F^{-1}(ADVANCE(p, X)) \mid X \in V\} \\ ADVANCE(p, X) &= \{A \rightarrow \alpha X \bullet \beta \mid A \rightarrow \alpha \bullet X\beta \in F(p)\} \end{aligned}$$

The initial state (IS) is obtained by $F^{-1}(CLOSURE(\{S' \rightarrow \bullet S\$\}))$. $RED_0(q, w)$ is stated as

$$RED_0(q, w) = \{A \ \rightarrow \ \gamma \mid A \rightarrow \gamma\bullet \in F(q)\}$$

$GOTO_k, \forall k \geq 0$, can be defined as:

$$\begin{aligned} GOTO_k(p, \lambda) &= p \\ GOTO_k(p, X) &= F^{-1}(CLOSURE(ADVANCE(p, X))) \\ GOTO_k(p, X\alpha) &= GOTO_k(GOTO_k(p, X), \alpha), \forall \alpha \neq \lambda \end{aligned}$$

From this point, when mentioning a state p, it will be known from the context whether it refers to the number or to the set of items of the state.

The LALR(k) automaton, shortly $LALRA_k$, is a tuple $(M_0, V, P, IS, GOTO_k, RED_k)$, where except for RED_k, all components are as in LRA_0. Before considering RED_k, it is necessary to model a function to capture all predecessor states for a given state q, under a sentential form α. Let $PRED$ be such function:

$$PRED(q, \alpha) = \{p \mid GOTO_k(p, \alpha) = q\}$$

Then,

$$RED_k(q, w) = \{A \rightarrow \gamma \mid w \in LA_k(q, A \rightarrow \gamma\bullet)\}$$

where LA_k is the set of lookahead strings of length not greater than k that may follow a processed right hand side of a rule. It is given by

$$LA_k(q, A \rightarrow \gamma) = \{w \in FIRST_k(z) \mid S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A z \ \wedge \ \alpha\gamma \ \ access\ q\}$$

where

$$FIRST_k(\alpha) = \{x \mid (\alpha \underset{lm}{\overset{*}{\Rightarrow}} x\beta \ \wedge \ |x| = k) \ \vee \ (\alpha \overset{*}{\Rightarrow} x \ \wedge \ |x| < k)\}$$

and $\alpha\gamma$ *access* q iff $PRED(q, \alpha\gamma) \neq \emptyset$.

For $k = 1$, DeRemer and Pennello proposed an algorithm to calculate the lookaheads in LA_1 [DeRemer and Pennello 1982] and it still remains as the most efficient one [Charles 1991]. They define the computation of LA_1 in terms of $FOLLOW_1 : (M_0 \times N \times M_0) \rightarrow \mathcal{P}(\Sigma)$. The domain $(M_0 \times N \times M_0)$ is said to be the set of nonterminal transitions. The first component is the source state, the second the transition symbol and the last one the destination state. For presentation issues, transitions will be written as pairs if destination states are irrelevant. $FOLLOW_1(p, A)$ models the lookahead tokens that follow A when ω becomes the current handle, as long as $A \rightarrow \omega \in P$. These tokens arise in three possible situations [DeRemer and Pennello 1982]:

a) $\exists\ C \rightarrow \theta \bullet B\eta \in p$, such that $p \in PRED(q, \beta)$, $B \rightarrow \beta A\gamma \in P$ and $\gamma \overset{*}{\Rightarrow} \lambda$. In this case, $FOLLOW_1(p, B) \subseteq FOLLOW_1(q, A)$. This situation is captured by a relation named *includes*: $(q, A)\ includes\ (p, B)$ iff the previous conditions are respected;

b) given a transition (p, A), every token that is directly read by a state q, as long as $GOTO_0(p, A) = q$, is in $LA_1(p, A)$. This is modeled by the direct read function:

$$DR(p, A) = \{t \in \Sigma \mid GOTO_0(q, t) \neq \Omega \ \wedge \ GOTO_0(p, A) = q\}$$

c) given (p, A), every token that is read after a sequence of nullable nonterminal transitions is in $LA_1(p, A)$. To model the sequence of nullable transitions the *reads* relation is introduced: $(p, A)\ reads\ (q, B)$ iff $GOTO_0(p, A) = q\ e\ B \overset{*}{\Rightarrow} \lambda$.

The function $READ_1(p, A)$ comprises situations (b) and (c):

$$READ_1(p, A) = DR(p, A) \cup \bigcup \{READ_1(q, B) \mid (p, A)\ reads\ (q, B)\}$$

From this and (a), $FOLLOW_1$ is written as:

$$FOLLOW_1(p, A) = READ_1(p, A) \cup \bigcup \{FOLLOW_1(q, B) \mid (p, A)\ includes\ (q, B)\}$$

Finally,

$$LA_1(q, A \rightarrow \omega) = \bigcup \{FOLLOW_1(p, A) \mid p \in PRED(q, \omega)\} \qquad (1)$$

3. Conflicts in non LALR(k) grammars

Conflicts arise in grammars when, for a state q in the LALR(k) automaton and a lookahead string $w \in \Sigma^*$, such that $|w| \leq k$, at least one condition is satisfied:

a) $|RED_k(q, w)| \geq 2$: reduce/reduce conflict;
b) $|RED_k(q, w)| \geq 1 \wedge \exists\, A \rightarrow \alpha \bullet \beta \in q \wedge\ w \in FIRST_k(\beta)$: shift/reduce conflict.

If one of these conditions is true, q is said to be an inconsistent state. A grammar is LALR(k) if its correspondent LALR(k) automaton has no inconsistent states.

A conflict is caused either by ambiguity or lack of right context, resulting in four possible situations. Ambiguity conflicts are the class of conflicts caused by the use of grammar rules that result in at least two different parsing trees for a certain string. These conflicts cannot be solved by increasing the value of k; in fact there isn't a k (or $k = \infty$) such that the grammar is LALR(k). Some of these conflicts are solved by rewriting some grammar rules in order to make it LALR(k), according to the k used by the parser generator (situation (i)). As an example, consider the *dangling-else* conflict. It is well known that its syntax can be expressed by a non ambiguous LALR(1) set of rules, although is more probable that one will first write an ambiguous specification. Some ambiguity conflicts, on the other hand, simply cannot be removed from the grammar without altering the language in question (situation (ii)). These conflicts are due to the existence of inherently ambiguous syntax constructions. An example would be a set of rules to describe $\{a^m b^n c^k \mid m = n \vee n = k\}$.

The next class of conflicts are those that are caused by the lack of right context when no ambiguity is involved. These conflicts occur due to an insufficient quantity of lookaheads. A direct solution is to increase the value of k (situation (iii)). To illustrate this, consider the grammar fragment presented in Figure 1:

declaration	$\rightarrow$	*visibility exportable-declaration*
	|	*non-exportable-declaration*
non-exportable-declaration	$\rightarrow$	*function-definition*
visibility	$\rightarrow$	**public** | **private** | λ
exportable-declaration	$\rightarrow$	*syntatic-domain*
syntatic-domain	$\rightarrow$	**domain-id** $=$ *domain-exp*
function-definition	$\rightarrow$	*temp4*
temp4	$\rightarrow$	*temp5* **id**
temp5	$\rightarrow$	**domain-id .**

Figure 1. Notus grammar fragment.

An LALR(1) parser generator would flag a shift/reduce conflict between items

temp5	$\rightarrow$	$\bullet$**domain-id .**
visibility	$\rightarrow$	$\lambda\bullet$

in a state q, for **domain-id** $\in LA_1(q, \textit{visibility} \rightarrow \lambda)$. However, the grammar is LALR(2), because the tokens after **domain-id** are either the equals sign (=), from *syntatic-domain* $\rightarrow$ **domain-id** $=$ *domain-exp*, or dot (**.**), from *temp5* $\rightarrow$ **domain-id .**.

However, even when no ambiguities are involved, there might be cases in which an infinite amount of lookahead is required (situation (iv)). In these cases, the solution to be tried is to rewrite some rules of the grammar without changing the language. Consider, for instance, the regular language $L = (b^+a) \cup (b^+b)$. A possible grammar for L is

$$
\begin{array}{lcl}
S & \rightarrow & A \mid B \\
A & \rightarrow & B_1\ \mathbf{a} \\
B & \rightarrow & B_2\ \mathbf{b} \\
B_1 & \rightarrow & B_1\ \mathbf{b} \mid \mathbf{b} \\
B_2 & \rightarrow & B_2\ \mathbf{b} \mid \mathbf{b}
\end{array}
$$

From the given productions, it is not possible to find a k for which the given grammar is conflict free. The reason is that $B_2 \rightarrow \mathbf{b}\bullet$ can be followed by an indefinite number of **b**'s when a conflict involving the item $B_2 \rightarrow \mathbf{b}\bullet$ is reported. The only possible solution for this example is to rewrite the grammar. For this simple example, such rewrite definitely exists, because L is a regular language. Nevertheless, it should be pointed out that this kind of solution is not always possible.

The mentioned four situations exhaust all possibilities of causes of conflicts in LALR(k) parser construction. These situations of conflicts are also applicable to LR(k) parser generation. One type of reduce/reduce conflict is, however, *LALR specific*. It arises when calculating LA_k for reduction items in states in M_0. Such calculation can be seen as generating the LRA_1 automaton and merging states with the same item set; lookaheads of reduction items in the new state are given by the union of the lookaheads in each reduction item in each merged state. When performing the merge, reduce/reduce conflicts, not present in the LR(1) automaton, can emerge. Specific LALR reduce/reduce conflicts occur if the items involved in the conflict do not share the same left context, i.e., a sentential form obtained by concatenating each entry symbol of the states in the path from IS to q, the inconsistent state. As a consequence, these conflicts do not represent ambiguity, but do not imply in the existence of a k.

4. The proposed methodology

The proposed methodology consists of four phases performed in an iteration while conflicts continue to be reported by the parser generator. These phases are: (i) understanding; (ii) classification; (iii) conflict removal and (iv) testing.

4.1. Understanding

To overcome the difficulty in analysing the data recorded in the log file dumped by the parser generator, this phase presents the same data available in the log file, but divided in proper parts, interrelated as hyperlinks. For example, when observing a state in the LALR(k) automaton, the user is able to directly visit the destination states given by the transitions in the currently state under visualization. The opposite operation should be possible as well, i.e., from a current state, grab all predecessor states. A modularized linked visualization of this data provides a better and faster browsing.

One drawback in visualizing this content is due to the low abstraction level it provides. While desired by expert users, this situation is not acceptable nor suitable for

analysis by non proficient users in LALR(k) parser construction. Therefore, one important characteristic of this phase is to provide high level data in order to understand the cause of the conflict. Derivation trees do meet this requisite, putting the user in a more comfortable position, as they approximate him/her to the real object of study - the syntax of the language, while reducing the amount of LALR parsing knowledge one must have in order to remove conflicts.

4.2. Classification

This phase aims to find one of the four situations described in Section 3 that gave rise to a conflict. Before removing it, a strategy must be planned. To understand the cause of the conflict is the first step for this, but the knowledge of the conflict's category adds much more confidence, as we strongly believe that a strategy used in removing a past conflict can be applied many times to other conflicts in the same category.

4.3. Conflict removal

Conflicts due to situation (iii) can be automatically removed. In the case of situation (i) the user is assisted with examples of solutions for known cases that match the current conflict. The removal, however, is performed manually. The methodology does not define operations for conflicts in situations (ii) and (iv).

4.4. Testing

The last step in conflict removal is testing. This should only be made in the case of a manual removal performed in the previous phase. To test, the user resubmits the grammar to the parser generator. As a result, it lists all conflicts found, plus the total amount of conflicts. The user checks this list, browsing it to make sure the conflict has indeed been eliminated.

5. SAIDE

SAIDE (*Syntax Analyser Integrated Development Environment*) is a tool, currently under construction, that aims to support the proposed methodology when it is applied to non LALR(1) grammars. Its main window is shown in Figure 2. The upper left corner frame contains the text editor with the opened syntax specification. This editor supports syntax highlighting and other common operations, such as searching, line and column positioning, undo, redo, cut, copy, paste, etc. The left bottom frame is the compilation window, the place where messages from the compilation of the the syntax specification are printed. A possible message is the report of a conflict. In this case, there is a link to allow its debug. A debug trace for the dangling else conflict is shown in the window at the right bottom. Finally, the last window in this figure is the LALR(1) automaton. Note that whenever possible, data is always linked, as indicated by underlined strings.

5.1. Realizing the methodology

This section outlines the algorithms used to support each phase of the proposed methodology.

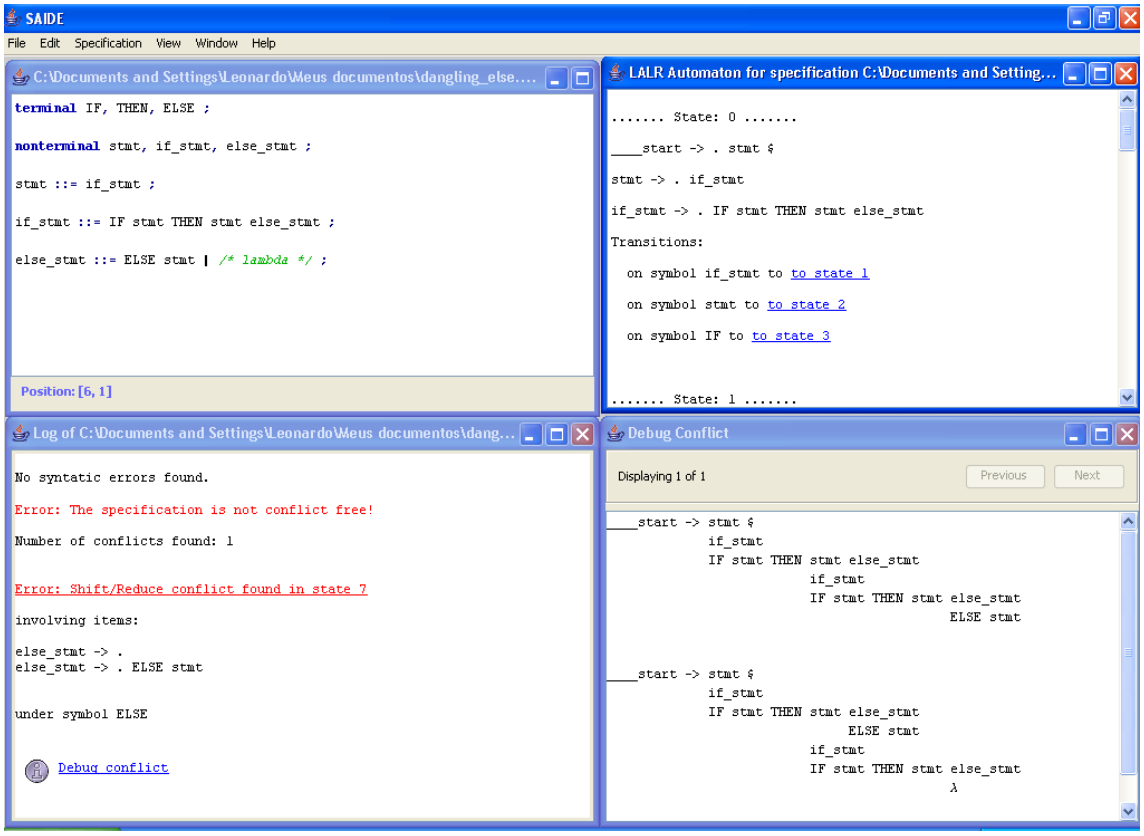


Figure 2. SAIDE's main window.

5.1.1. Understanding

To present the user with the LALR(1) automaton, first LRA_0 is obtained by the application of the $CLOSURE$ operation, as previously explained. The next step is to create the graphs corresponding to the *reads* and the *includes* relations. When there are two functions F and F' defined over a set of elements in X, and F is defined as $F(x) = F'(x) \cup \bigcup\{F(y) \mid xRy\}$, $\forall x, y \in X$, it follows that the nodes in a strongly connected component (SCC) found in the graph representing R have equal values for F [DeRemer and Pennello 1982]. Since $FOLLOW_1$ and $READ_1$ do match F's pattern, performing a search and identifying SCC's in the *reads* and *includes* graph permit the calculation of their value. The algorithm to efficiently perform this is presented in [DeRemer and Pennello 1982]. From the values in $READ_1$ and $FOLLOW_1$, the lookaheads of each reduction item are calculated using Equation 1, presented in Section 2.

To elucidate the cause of a conflict, SAIDE explains it in terms of derivation trees, as proposed by the methodology. Derivation trees are constructed for each reduction item. Their format is illustrated in Figure 3. The means to calculate parts (c), (b) and (a) are as follows [DeRemer and Pennello 1982]:

Part (c): given the item $A_{s-1} \rightarrow \alpha_s \bullet$ in an inconsistent state q, the traversal begins from the transitions in (q', A_{s-1}), where q' is in $PRED(q, \alpha_s)$, and then following some edges in the *includes* graph until a nonterminal transition (p, B) is found whose $READ_1$ set

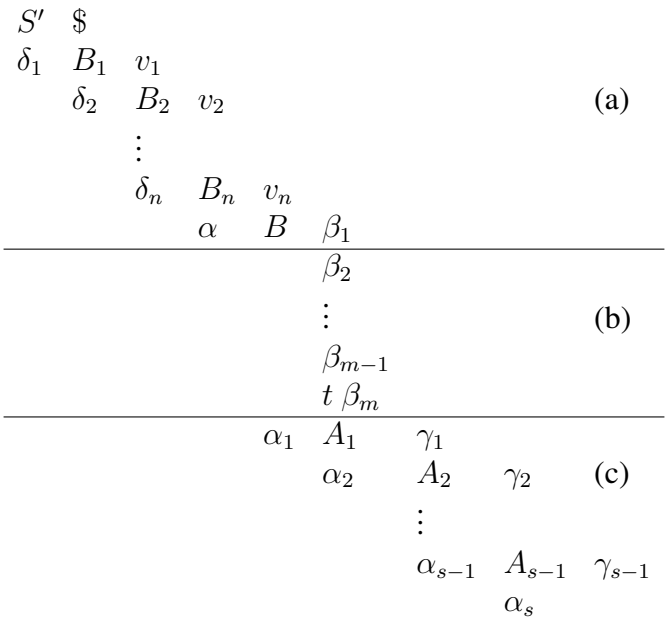


Figure 3. The format of a derivation tree.

contains the conflict symbol t. Every item $B_n \rightarrow \alpha \bullet B\beta_1 \in p$, such that $t \in FIRST_1(\beta_1)$, is considered a contributing one. For each of these, a debug should be printed separately.

To obtain the described path, a breadth-first search should be employed while traversing the *includes* graph. The production that induced an includes edge from (p, A) to (q, B) is rediscovered by following the automaton transitions from state q under the right parts of B productions.

Part (b): the next step is to get a derivation from β_1 until t appears as first token. To do this, the set

$$\begin{aligned} E \;=\; & \{B_n \rightarrow \alpha B \bullet \beta_1\} \\ & \cup \{A \rightarrow \delta X \bullet \eta \mid A \rightarrow \delta \bullet X\eta \in E \wedge X \stackrel{*}{\Rightarrow} \lambda\} \\ & \cup \{C \rightarrow \bullet\alpha \mid A \rightarrow \delta \bullet C\eta \in E \wedge C \rightarrow \alpha \in P\} \end{aligned} \tag{2}$$

is calculated. Each addition to E must be linked back to the items that generated it. Items of the form $C \rightarrow \bullet t\beta_m$, being t a conflict symbol, will be in E and are traced back to $B_n \rightarrow \alpha B \bullet \beta_1$ by following these links.

Part (a): calculate the derivation from S' that gave rise to $B_n \rightarrow \alpha \bullet B\beta_1$. First, it is necessary to find the shortest path from the start state to the contributing state, i.e., the state in which $B_n \rightarrow \alpha \bullet B\beta_1$ appeared. ξ is the sequence of transition symbols in this path. Then,

$$\begin{aligned} E' \;=\; & \{(S' \rightarrow \bullet S\$, 1)\} \\ & \cup \{(C \rightarrow \bullet\alpha, j) \mid (A \rightarrow \delta \bullet C\eta, j) \in E' \wedge C \rightarrow \alpha \in P)\} \\ & \cup \{(A \rightarrow \delta X \bullet \eta, j+1) \mid (A \rightarrow \delta \bullet X\eta, j) \in E' \wedge X = \xi_j \wedge j \leq |\xi|\} \end{aligned} \tag{3}$$

is calculated in a breadth first search, linking additions to E' back to pairs that generated

them. When $(B_n \rightarrow \alpha \bullet B\beta_1, |\xi|)$ appears, the computation stops. The derivation is given by the links created when elements were added.

To debug a reduce/reduce conflict, the sequence (c) $\rightarrow$ (b) $\rightarrow$ (a) is applied to each reduction item, and printed to the user. Sometimes, the left context formed by $\delta_1\delta_2...\delta_n\alpha\alpha_1...\alpha_s$ happens to be different from one reduction debug tree to the other. This indicates that the conflict in question is LALR specific, and is accordingly indicated by the tool.

When a conflict presents a shift, its corresponding tree is obtained by using an algorithm that implements Equation 3. The only difference is that ξ now corresponds to the symbols in $\delta_1\delta_2...\delta_n\alpha\alpha_1...\alpha_s$.

5.1.2. Classification

The classification aims to find the category to which the conflicts belongs. Each category represents one of the four situations explained in Section 3.

At this point, SAIDE first attempts to find a value of k, limited by a parameter value, say k_{max}, capable of giving enough right context to allow the removal of the conflict situation (iii). The value of k_{max} is read from a configuration file at SAIDE's start up and its default value is 3. Before presenting how the correct value of k is achieved, it follows a discussion of how a given lookahead w, such that $|w| \leq k$, is found.

Charles [Charles 1991] proposes Equation 4 to calculate the $FOLLOW_k$:

$$
\begin{array}{rcl}
FOLLOW_0(p,A) & = & \{\lambda\} \\
FOLLOW_k(p,A) & = & \quad READ_k(p,A) \\
 & & \cup \bigcup\{FOLLOW_k(p',B) \mid (p,A)\ includes\ (p',B)\} \\
 & & \cup \bigcup\{CONCAT(\{w\}, FOLLOW_{k-|w|}(q,B)) \mid \\
 & & \qquad w \in SHORT_k(p,A), \\
 & & \qquad B \rightarrow \alpha \bullet A\beta \in p, \\
 & & \qquad q \in PRED(p,\alpha), \\
 & & \qquad B \neq S\}
\end{array}
\tag{4}
$$

where

$$
\begin{array}{rcl}
SHORT_k(p,A) & = & \{w \mid w \in FIRST_k(\beta), 0 < |w| < k, B \rightarrow \alpha \bullet A\beta\ \in p\} \\
READ_k(p,A) & = & \{w \mid w \in FIRST_k(\beta), |w| = k, B \rightarrow \alpha \bullet A\beta \in p\}
\end{array}
$$

and $CONCAT(M,N)$ is defined as $\{mn \mid m \in M \wedge n \in N\}$.

Charles presents an algorithm to calculate $READ_k$ and $SHORT_k$ based on the simulation of the steps performed by the LRA_0 automaton. His algorithm is directly based on Equation 5 [Charles 1991]:

$$
\begin{array}{lcl}
READ_{0*}(stack, X) & = & \{\lambda\} \\
READ_{k*}(stack, X) & = & \bigcup\{CONCAT(\{a\}, READ_{(k-1)*}(stack + [q], a) \mid \\
 & & \quad a \in DR(TOP(stack), X) \\
 & & \quad q = GOTO_0(TOP(stack), X)\} \\
 & & \cup \bigcup\{READ_{k*}(stack + [q], Y) \mid \\
 & & \quad (TOP(stack), X)\ reads\ (q, Y)\} \\
 & & \cup \bigcup\{READ_{k*}(stack(1...SIZE(stack) - |\gamma|), C) \mid \\
 & & \quad C \rightarrow \gamma \bullet X \in TOP(stack), \\
 & & \quad |\gamma| + 1 < SIZE(stack)\}
\end{array}
\tag{5}
$$

$SHORT_k$ and $READ_k$ are then rewritten as:

$$
\begin{array}{lcl}
SHORT_k(p, A) & = & \{w \mid w \in READ_{k*}([p], A), 0 < |w| < k, B \rightarrow \alpha \bullet A\beta\ \in p\} \\
READ_k(p, A) & = & \{w \mid w \in READ_{k*}([p], A), |w| = k, B \rightarrow \alpha \bullet A\beta \in p\}
\end{array}
$$

Charles states that in the presence of cycles in the grammar, i.e., nonterminals that rightmost produce themselves and SCC's in the *reads* graph, his algorithm may not terminate. To guarantee termination, a verification of the non occurrence of these two conditions must always be performed. Later, the author discards Equation 4 as the bases of an algorithm to calculate $FOLLOW_k$. His main argument is that it does not match the format $F(x) = F'(x) \cup \bigcup\{F(y) \mid xRy\}$.

However, Equation 5 can still be used to calculate the strings that are read from a given transition even when the conditions pointed by Charles are not true. This is achieved if one keeps track of every reached stack and the string read so far while simulating the LRA_0 steps. Kristensen and Madsen [Kristensen and Madsen 1981] argue that a tree can be used to store such data. A node in this tree is a pair of the form $(M_0 \times \Sigma^*)$ and maps to a unique configuration, i.e., a stack of states and the corresponding string read at the moment. The correspondence between a node n and a configuration is guaranteed in the following way: the states in the path from the root node of the tree to n forms a stack. The string obtained by such stack is the string stored in n. During the simulated parsing, the tree will be expanded with a node each time a transition is carried out. If an attempt to add a node that is already in the tree is performed, then circularity is detected, and thus cycles are controlled.

A straightforward algorithm using these ideas and Equations 4 and 5 can be constructed. Such algorithm would clearly terminate, as it would depend solely on the values of $READ_{k*}(p, A)$ and $FOLLOW_k(q, B)$, where $(p, A)\ includes\ (q, B)$. This fact is attested by:

i) if (p, A) and (q, B) belong to an SCC in the *includes* graph, then their lookaheads are equal. This is assured because Equation 4 matches the format $F(x) = F'(x) \cup \bigcup\{F(y) \mid xRy\}$, where X is the set of nonterminal transitions and $F'(p, A)$ is

given by

$$
\begin{aligned}
F'(p,A) \;=\;& READ_k(p,A) \\
& \cup \bigcup \{ CONCAT(\{w\}, FOLLOW_{k-|w|}(q,B)) \mid \\
& \quad w \in SHORT_k(p,A), \\
& \quad B \to \alpha \bullet A\beta \in p, \\
& \quad q \in PRED(p,\alpha), \\
& \quad B \neq S \}
\end{aligned}
$$

This matching permits an easy control of cycles.

ii) $READ_{k*}$ is calculated using Equation 5. Storing each stack and the string read so far at each step in the calculation of $READ_{k*}$ permits the control of cycles, thus, preventing the algorithm to loop forever.

Furthermore, no restrictions are considered in the LRA_0 or any relation graph.

We are currently implementing an algorithm based on the discussed equations and cycle control scheme to generate the values in LA_k to determine the value of k necessary to solve a conflict. Its iteration starts with $k = 2$. If RED_k continues to report the conflict, a new iteration is performed, which increments the value of k. This continues until the conflict is "removed" or k becomes greater than k_{max}. The algorithm caches all calculated sets, since they might be used later for other conflicts.

If the classification fails to find a $k \leq k_{max}$ capable of removing the conflict and it is a non reduce/reduce conflict specific to LALR, the next attempt is ambiguity detection (situation (i)). It is known from the literature that this problem is undecidable. Therefore, a study to capture recurrent cases was performed and some patterns were noticed. A pattern consists of two sentential forms derivable from a nonterminal P. Expanding each sentential form will eventually lead to the ambiguity in study. These patterns were inferred from ambiguities found in the grammars of the programming languages Notus, Algol60 and Oberon2.

For each pattern, it must be asserted that $S' \stackrel{*}{\Rightarrow} \xi' P \xi''$, $\delta_i \stackrel{*}{\Rightarrow} \lambda$ and $P_i \stackrel{*}{\Rightarrow} P$. The symbols δ_i and P_i are used in the definition of the filters listed bellow:

Filter 1)
Pattern: $P \stackrel{*}{\Rightarrow} \delta_1 P_1 \delta_2 \alpha \delta_3 P_2 \delta_4$ and $P \stackrel{*}{\Rightarrow} \delta_6 \beta \delta_7 \delta_8$. This filter captures ambiguous constructions such as $E \to E + E \mid t$.

Filter 2) Pattern: $P \stackrel{*}{\Rightarrow} \delta_1 \alpha \delta_2 P_1 \delta_3$ and $P \stackrel{*}{\Rightarrow} \delta_5 \alpha P_2 \delta_6 \beta \delta_7 P_3 \delta_8$. This filter identifies dangling-else's instances.

Filter 3) $P \stackrel{*}{\Rightarrow} \delta_1 \alpha \delta_2 P_1 \delta_3$ and $P \stackrel{*}{\Rightarrow} \delta_5 P_2 \delta_6 \beta \delta_7$. This filter captures ambiguous constructions such as the rules *exp* → **let** *dcl* **in** *exp* **where** *exp* and *exp* → *exp* **where** *exp*.

Filter 4) $P \stackrel{*}{\Rightarrow} \delta_1 \alpha \delta_2 P_1 \delta_3 \beta \delta_4$ and $P \stackrel{*}{\Rightarrow} \delta_6 \alpha \delta_7 P_2 \delta_8 \beta \delta_9$. This filter captures alias between nonterminals.

We are managing to apply these filters on the derivation trees obtained by the first step of the methodology.

If a conflict is due to situation (ii) and (iv), SAIDE is unable to classify and assist the user.

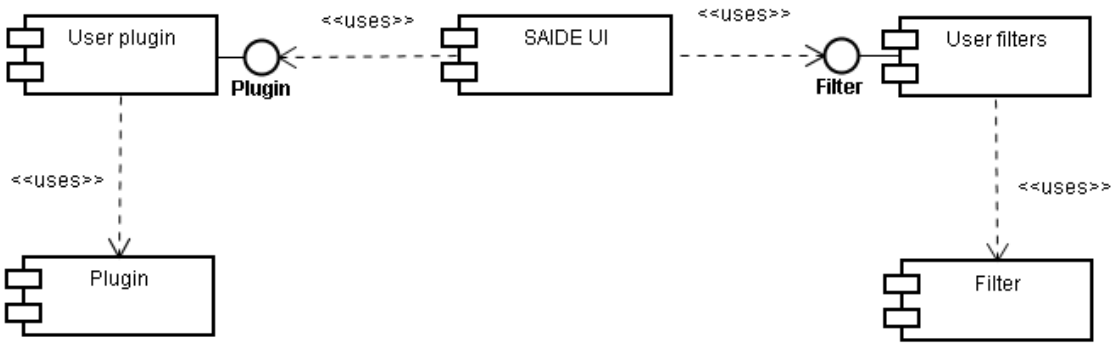


Figure 4. SAIDE's architecture illustrated as a component diagram in UML.

5.1.3. Conflict removal

Automatic conflict removal can only be accomplished if the parser is LALR(k) and $k \leq k_{max}$. There are two approaches for this problem.

The first one is to rewrite the grammar, starting from the productions involved in the conflict so that no reduction moves are performed until k tokens are read. [Mickunas et al. 1976] proposes a technique to transform LR(k) grammars into LR(1) correspondent ones, but it can deeply change the structure and the number of the rules in the grammar and is not generalized to LALR(k) grammars.

The other approach consists in the generation of an LALR(k) parser whose k varies. A conflict is removed if the parser generator, in this case SAIDE, can attest that nondeterminism is removed after examining k tokens ahead. The value of k in this case is local and is intended to solve only the given conflict.

5.1.4. Testing

To check if the conflict was been wiped out, SAIDE list all conflicts along with the total sum of conflicts found. The user browses this list and compares the current results with the ones previously presented.

5.2. Plugin facility

SAIDE's architecture, shown in Figure 4, permits its extensibility via plug-ins. A plugin instance must implement an interface with two methods responsible for returning `PluginParserFactory` and `HighlightLexerFactory` objects. `PluginParserFactory` is used by SAIDE to instantiate a parser capable of processing the specification file. The parsing result is an `Specification` instance used by the tool to generate data structures such as the LALR(1) automaton, *includes* and *reads* graphs, etc. and it appears throughout SAIDE's code. SAIDE's architecture permits the use of virtually any syntax specification language as long as there is a plugin implemented. In a similar way, the tool can be extended with filters in addition to the ones made available.

6. Conclusion

In this article, we presented the problem of conflict removal in non LALR(k) grammars. It was argued that this remains an arduous and time consuming task, considering that users continue to remove conflicts analysing extensive log files.

A methodology was proposed and the algorithms necessary to realize it were presented, showing the tool SAIDE.

The methodology is an important contribution to LALR parsing development and so is the outlined algorithm discussed in Section 5.1.2 for the calculation of lookahead strings of length not greater than k. Two important properties of such algorithm are: it is guaranteed to terminate, and no pre-conditions must be checked before running it.

At the present time, we are implementing the ambiguity detector, which will attempt to detect some previous cataloged ambiguity instances.

As future work, we intend to formulate an algorithm to automatically remove a subset of conflicts, either by redefinition of some rules of the grammar or generating an LALR parser whose k varies.

References

Aho, A. V. and Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling*. Prentice Hall Professional Technical Reference.

Charles, P. (1991). *A Pratical Method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery*. PhD thesis.

CUP (2007). Cup: Lalr parser generator in java. `http://www2.cs.tum.edu/projects/cup/`. Last access: 01/13/2007.

DeRemer, F. and Pennello, T. (1982). Efficient computation of lalr(1) look-ahead sets. *ACM Trans. Program. Lang. Syst.*, 4(4):615–649.

Frown (2007). Frown - an lalr(k) parser generator for haskell. `http://www.informatik.uni-bonn.de/~ralf/frown/index.html`. Last access: 01/13/2007.

Johnson, S. (1979). Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston.

Kristensen, B. B. and Madsen, O. L. (1981). Methods for computing lalr(k) lookahead. *ACM Trans. Program. Lang. Syst.*, 3(1):60–82.

Mickunas, M. D., Lancaster, R. L., and Schneider, V. B. (1976). Transforming lr(k) grammars to lr(1), slr(1), and (1,1) bounded right-context grammars. *J. ACM*, 23(3):511–533.

Tirelo, F. and Bigonha, R. (2006). Notus. Technical report, Federal University of Minas Gerais - Department of Computer Science, Programming Languages Laboratory.

Optimized Compilation of Around Advice for Aspect Oriented Programs

Eduardo S. Cordeiro[1], **Roberto S. Bigonha**[1], **Mariza A. S. Bigonha**[1], **Fabio Tirelo**[2]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais
Av. Presidente Antônio Carlos, 6627 – Campus Pampulha
31270-901 – Belo Horizonte – MG – Brazil

[2]Instituto de Informática – Pontifícia Universidade Católica de Minas Gerais
Av. Dom José Gaspar, 500 – Coração Eucarístico
30535-610 – Belo Horizonte – MG – Brazil

{cordeiro,bigonha,mariza}@dcc.ufmg.br, ftirelo@pucminas.br

Abstract. *The technology that supports Aspect-Oriented Programming (AOP) tools is inherently intrusive, since it changes the behavior of base application code. Advice weaving performed by AspectJ compilers must introduce crosscutting behavior defined in advice into Java programs without causing great performance overhead. This paper shows the techniques applied by the* ajc *and* abc *AspectJ compilers for around advice weaving, and identifies problems in code they produce. The problems analyzed are* advice and shadow implementation repetition *and* context variable repetition*. Performance gain provided by solving these problems is discussed, showing that bytecode size, running time and memory consumption can be reduced by these optimizations. It is assumed that the reader is familiar with AOP and AspectJ constructs.*

1. Introduction

Advice weaving is the process of combining crosscutting behavior, implemented in advice, into the classes and interfaces of a program. AspectJ defines three types of advice, which are activated upon reaching certain points in the execution of programs: before and after advice are executed in addition to join points; around advice may completely replace join points, though a special *proceed* command activates these points at some moment after the advice execution has begun.

The compilation of the *proceed* command in around advice at *bytecode* level requires join points to be extracted to their own methods. Furthermore, this command might appear inside nested types in the advice body, which requires passing context from the advice's scope to extracted join points. While discussing around advice weaving, join points are also called shadow points, or simply *shadows*. During weaving, a join point comprising a single Java command is often composed of several *bytecode* instructions.

There are two major AspectJ compilers: the official AspectJ Compiler (*ajc*) [AspectJ Team 2006], and the extensible, research-oriented AspectBench Compiler (*abc*) [Aspect Bench Compiler Team 2006]. *ajc* builds on the JDT Java compiler[1], and provides incremental compilation of AspectJ programs. It accepts Java and AspectJ code, as well

[1] http://www.eclipse.org/jdt

as binary classes, and produces modified classes as output. *abc* also accepts Java and AspectJ code, and its output is semantically equivalent to that of *ajc*, but instead of providing fast compilation by means of an incremental build process, *abc* produces optimized *bytecode*. This compiler is also a workbench for experimentation with new AspectJ constructs and optimizations, providing researchers with extensible front- and back-ends.

These compilers apply the same basic techniques for weaving before and after advice. Around advice, however, is woven differently. Performance analyses on a benchmark of AspectJ programs showed that around advice is one of the performance degradation agents in code produced by the *ajc* compiler [Dufour et al. 2004]. Based on this insight, the developers of *abc* created another approach for around advice weaving [Kuzins 2004, Avgustinov et al. 2005].

Both approaches for around advice weaving, however, still present problems related to repeated code generation. These problems are due to advice inlining and shadow extraction for advice applications, and can be fixed by small modifications in the advice weaving process. The remainder of this paper describes these problems, their proposed solutions and results obtained from their application to a small set of AspectJ programs.

1.1. AspectJ Compilers

The *ajc* compiler is built upon the extensible JDT Java compiler, which allows the introduction of hooks in the compilation process that modify its behavior [Hilsdale and Hugunin 2004]. These hooks are then used to adapt the front- and back-ends to compile both Java and AspectJ source-code. Java code for classes and interfaces is directly transformed into *bytecode*. Definitions of aspects, however, are handled in a different way: first, *bytecode* is produced to implement aspects as classes, so that code defined in advice and methods can be executed by standard JVMs. Finally, after *bytecode* has been generated for both Java and AspectJ source, the *weaver* introduces crosscutting behavior defined in aspects into the *bytecode* for the program's classes and interfaces, using crosscutting information gathered from the parsing phase.

Duringo compilation, in-memory representations of *bytecode* are used for code generation and weaving, and actual *bytecode* files are only generated at the end. The *ajc* compiler uses BCEL [Dahm et al. 2003] as a *bytecode* manipulation tool. BCEL interprets *bytecode* contained in class files, and builds in-memory representations of the classes and interfaces they define. It provides facilities for adding and removing methods and fields to existing classes, modifying method bodies by adding or removing instructions, and creating classes from scratch. Its representation of *bytecode* is very close to its definition [Lindholm and Yellin 1999], providing direct access to such low-level structures as a class' constant pool.

Extensibility in the *abc* compiler, as described in [Avgustinov et al. 2004], is achieved by combining two frameworks: Polyglot [Myers 2006] for an extensible front-end, and Soot [Vallée-Rai et al. 1999] for an optimizing, extensible back-end. Polyglot is a Java LALR(1) parser, and its grammar can be modified to add or remove productions. Soot implements several optimizations for Java *bytecode*, including peephole and flow-analysis optimizations such as copy and constant propagation. Extensions are linked to Soot at runtime, via a command line flag, thus requiring no modifications to its source code. This extension model, however, isn't flawless, and it can be difficult to implement

optimizations that modify the weaving algorithms for existing AspectJ constructs. Difficulties found during the development of this work are presented further in this paper.

Out of the four intermediate representations provided by Soot, *abc* uses only the Jimple representation. Jimple is a typed 3-address code that makes it easier to perform analyses like use-definition chains than the stack code of *bytecode*. Weaving is performed in *abc* with Jimple representations of classes and interfaces.

1.2. The Compilation Process

The compilation process for AspectJ programs differs from ordinary Java compilation in that crosscutting behavior defined in aspects must be combined to classes and interfaces. This process is called *weaving*, and is usually done at binary-code level. The *ajc* compiler performs weaving at *bytecode* using BCEL, and *abc* uses one of the Soot framework's intermediate representation for this, called Jimple.

In both compilers, Java and AspectJ source code is transformed into ASTs and then intermediary representations of the binary code. On *ajc*, *bytecode* is generated and manipulated directly via in-memory representations of its structure using the BCEL framework; on *abc*, Jimple code is used. The front-end is also responsible for generating crosscutting information for the weaver. This structure identifies locations on classes and aspects where advice must be woven into. The advice weaver then applies advice to join points, producing the final woven code for AspectJ programs.

An advice is transformed into regular a Java method, and the weaving phase applies calls to this method at its join points. For instance, the weaving of a before advice includes a call to the advice implementation before its join points, leaving the join points themselves unmodified. Weaving of around advice is more complex, however, as the original join points must be replaced by calls to advice implementations. This stage gives rise to problems with repeated code generation, and is discussed in detail in Section 2.

2. Around Advice Weaving

The most powerful type of advice defined in AspectJ is the around advice. It can be used to simulate the behavior of both before and after advice, as well as to modify or completely avoid join points. Context used in the shadow may be captured in the advice, but must also be passed on to shadow execution. The power of modifying the behavior of join points – also called shadows – inside around advice comes from the *proceed* command, which activates the shadow captured by the executing advice: context variables captured by the advice may be modified before the *proceed* call. Avoiding shadow execution altogether is achieved by omitting this command.

Listing 1 shows a small AspectJ program. Line 7 contains a shadow of the around advice defined in lines 21 - 23. This advice has no effect on the semantics of its join points, since it simply proceeds to shadow execution.

The remainder of this section presents the weaving techniques applied in the compilation of this program by the *ajc* and *abc* compilers.

2.1. The *ajc* Approach

Around advice weaving in the *ajc* compiler is briefly described in [Hilsdale and Hugunin 2004]. Since around advice shadows must be executed as a

```
1  public class Circle {
2      private int radius;
3      private int x,y;
4      public Circle(int x, int y, int radius) {
5          setX(x);
6          setY(y);
7          setRadius(radius);
8      }
9      public int  getRadius()           { return radius; }
10     public void setRadius(int radius) { this.radius = radius; }
11     public int  getX()                { return x; }
12     public int  getY()                { return y; }
13     public void setX(int x)           { this.x = x; }
14     public void setY(int y)           { this.y = y; }
15
16     public static void main(String[] args) {
17         Circle c = new Circle(0,0,10);
18     }
19 }
20 public aspect RadiusCheckAspect {
21     void around(): call(void Circle.setRadius(int)) {
22         proceed();
23     }
24 }
```

Listing 1. The running example for this paper.

result of *proceed* calls, these instructions are extracted from their original locations to separate methods, called *shadow methods*. The proceed call inside around advice bodies is then replaced by calls to these methods.

Figure 1 is a visual representation of this process. The darkened boxes in this figure represent the shadow in line 7 of Listing 1. Notice that only the parts affected by weaving are shown in this figure. The result of weaving the around advice defined in `RadiusCheckAspect` into class `Circle` is a modified version of this class. Each around advice shadow in a given class is extracted into its own method. For each shadow method, an inlined implementation of the advice is generated, whose proceed call is replaced with a call to the shadow method.

Method `shadow1` in the woven `Circle` class shown in Figure 1 contains a shadow, which is a call to method `setRadius`. The instance of `Circle` and the argument to this method, which are context variables required for executing this shadow, are passed from the join point to the advice implementation and then on to the shadow. Context passing can be seen in Figure 1 as the target object and the arguments from the shadow's `setRadius` call are passed as arguments to the advice and shadow methods.

The *proceed* call may appear inside nested types in the advice body. In this scenario, this call may attempt to access local variables in the advice environment after its scope has ended. A different approach is used to handle this special case, which involves

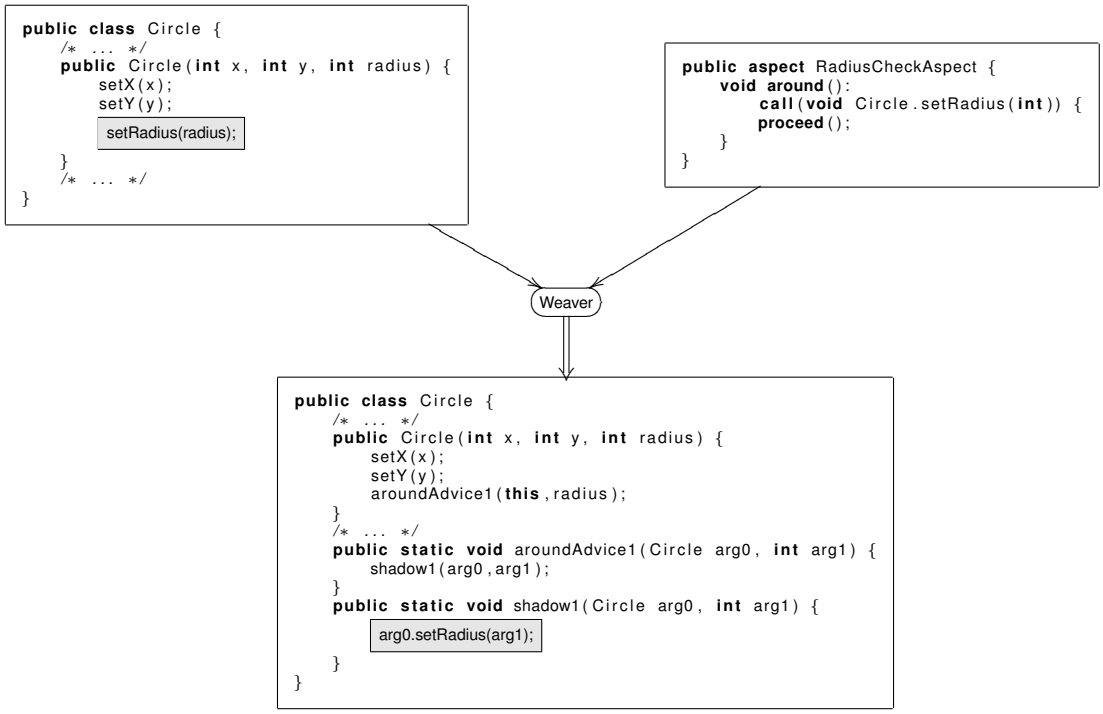


Figure 1. Around advice weaving in *ajc*.

creating an object to store both the advice environment variables and the shadow code to be executed at the *proceed* call.

Objects used to implement this type of around advice application are called *closure objects*. These objects are implementations of an interface called `AroundClosure`, which defines a method `run` to contain the shadow code. Environment variables are placed on the shadow environment as arguments to its closure's `run` method. Thus for each advice application at runtime an object must be created to cope with the *proceed* call.

2.2. The *abc* Approach

Kuzins details, in [Kuzins 2004], the structure used to implement around advice weaving in the *abc* compiler, and presents benchmarks suggesting that the code produced for around advice in *abc* is faster than the one produced by *ajc*. The performance gain is related to avoiding closure object creation, which is required in *ajc* for around advice that contains proceed calls inside nested types in the advice body.

In the *abc* approach, each shadow is labeled with an integer identifier, called *shadowID*, and the class that contains it is also labeled with an identifier called *classID*. All shadows for an around advice in a given class C are extracted to a single shadow method introduced into C. The shadow method for class C contains all its shadows, and execution is routed to each one via the *shadowID*. This identifier is a parameter to the shadow method and is set by every inlined advice implementation at the *proceed* call. Each advice implementation sets the *shadowID* according to its shadow.

The *classID-shadowID* pair appears on code generated by previous versions of

the *abc* compiler. On version 1.1.0, however, this approach has been taken a step further, avoiding shadow selection at runtime. This is achieved by inlining advice methods and shadows for each one of the advice's applications.

When closures are necessary to implement an around advice a, *abc* makes class C that contains shadows of a implement an interface called `AroundClosure`. This interface defines a method to which shadows are extracted. The advantage of this approach, when compared to the one adopted by *ajc*, is that the class containing a shadow is itself a closure object, and thus there is no need to create a new object at advice applications. This weaving strategy introduces fields in class C that are linked at runtime, during preparation for the advice call, to environment variables required for advice and shadow execution.

Code woven for around advice by *abc* is similar to that produced by the algorithm applied by *ajc*, except that advice methods are created in the *bytecode* class that represents the aspects in which they were declared, rather than the class where their shadows appear.

3. Repeated Advice Implementations

Repeated advice implementations are generated during around advice weaving when a class C contains several identical shadows of an around advice a. If a class contains n identical shadows of any given around advice, the around weaving strategy described in Section 2 create n identical pairs of advice and shadow implementations. This generation of repeated advice implementations appears on code compiled by both *ajc* and *abc*. It is due to the naive generation of inlined around advice implementations, with no regards as to whether or not other identical implementations have already been generated for identical shadows in the same class.

Consider modifying the code base presented in Listing 1 to add to the `main` method defined in class `Circle` a call to `setRadius`. Listing 2 shows the resulting `main` method. This creates another shadow of the around advice defined in Listing 1, thus producing repeated advice implementations.

```
16      public static void main(String[] args) {
17          Circle c = new Circle(0,0,10);
18          c.setRadius(-1);
19      }
```

Listing 2. The `main` method, modified to contain an advice shadow.

Class `Circle` has now two shadows of the existing around advice: one in its constructor, and another in the `main` method. The woven code for class `Circle`, originally shown in Figure 1, now contains another advice implementation, and can be seen in Listing 3. Notice, however, that the single difference between the advice implementations `aroundAdvice1` and `aroundAdvice2` is the shadow method called. Since these shadows are equivalent, it can be said that the advice implementations are also equivalent, and thus redundant.

Two methods are said to be equivalent if their signatures (parameters and return types) and instruction lists are the same. Eliminating any of these advice implementations and replacing the call to it with a call to the other one doesn't modify the semantics of this program. This reduces the size of generated code for AspectJ programs that use around advice. The optimized code for the example in Listing 3 would be free of methods `aroundAdvice2` and `shadow2`, and the call to `aroundAdvice2` in line 10 would be replaced with a call to `aroundAdvice1`. Notice that the resulting code is smaller, but still semantically equivalent to the original.

This optimization can be performed in two different approaches: a post-weaving *unification* phase, or advice implementation *caching* during the weaving process. At post-weaving, one must identify repeated advice implementations and eliminate all but one of them, and fix the calls to removed implementations. During weaving, one is required to detect that a given shadow has already been woven into in a class, and reuse the advice implementation created for that shadow instead of generating another inlined implementation.

```
1  public class Circle {
2      /* ... */
3      public Circle(int x, int y, int radius) {
4          setX(x);
5          setY(y);
6          aroundAdvice1(this,radius);
7      }
8      public static void main(String[] args) {
9          Circle c = new Circle(0,0,10);
10         aroundAdvice2(this,-1);
11     }
12     public static void aroundAdvice1(Circle arg0, int arg1) {
13         shadow1(arg0,arg1);
14     }
15     public static void shadow1(Circle arg0, int arg1) {
16         arg0.setRadius(arg1);
17     }
18     public static void aroundAdvice2(Circle arg0, int arg1) {
19         shadow2(arg0,arg1);
20     }
21     public static void shadow2(Circle arg0, int arg1) {
22         arg0.setRadius(arg1);
23     }
24 }
```

Listing 3. Class `Circle` after the weaving of two shadows.

The second approach, caching generated advice implementations during weaving, is more fitting for integration to the existing AspectJ compilers, since it avoids unnecessary work. Repeated advice implementations are never generated, and so they need not be removed. This approach has been suggested to *ajc* developers as a bug re-

port [Cordeiro 2006a].

The Soot optimization framework defines a phase model in which *bytecode* is modified gradually. In the *abc* compiler, only the peephole and flow analysis phases are activated. However, in these phases, one isn't able to modify the structure of the optimized program, and thus optimizations are restricting to handling method bodies. Since eliminating repeated advice implementations requires eliminating methods from classes as well as modifying method bodies, it is not possible to implement this optimziation as an *abc* back-end extension. During development of this study the developers of *abc* have implemented this solution by modifying the compiler's around weaving algorithm, as suggested to the *ajc* developers, integrating reuse of advice implementations in the weaving process.

3.1. Results

Removing advice and shadow implementation replicas from the code generated for an AspectJ program produces smaller code by eliminating from it several structures required to represent these methods in *bytecode* format. This decrease in code size is proportional to the number of around advice applications in each of the program's classes, as well as the size of advice and shadow bodies.

Table 1 shows the sizes of a set of AspectJ programs that use around advice. *Singleton* is the test program that accompanies Hannemann's Singleton pattern implementation [Hannemann and Kiczales 2002]. Its main method contains three identical shadows of an around advice.

SpaceWar is a sample AspectJ programs that features several language constructs and idioms. It is available along with the Eclipse AspectJ Development Tools[2] (AJDT). The around advice used in this program captures user and computer commands given to ships, ensuring that their respective ship is alive at the time the command is issued.

Laddad presents, in [Laddad 2003], a thread-safety aspect that can be applied to programs written using the Swing library. This aspect has been applied to the *Rin'G* program [Cordeiro et al. 2004], which is mostly based on user interaction and thus makes great use of Swing classes.

The reduction in *bytecode* size achieved by eliminating repeated advice and shadow implementations is shown in this table, as the optimized programs are smaller than the original ones. Decrease in *bytecode* size is proportional to the number of around advice applications in the program. The decrease percentage for a given program also depends on its total size: for instance, the decrease percentage for the *SpaceWar* program is smaller than for *Singleton*, since the latter is actually much smaller.

The greatest reduction presented in Table 1 is for the *Rin'G* program. Since this program is user-interface-oriented, there are roughly 500 around advice shadows spread over 83 classes, thus making the program size / reduction size ratio more noticeable.

4. Repeated Context Variables

Advice in AspectJ can capture context from join points, via the **`args`**, **`target`** or **`this`** clauses. Context information gathered by these clauses comprises arguments and targets

[2] `http://www.eclipse.org/ajdt`

Application	Original Code – A (*bytes*)	Optimized – B (*bytes*)	Decrease (%)
Singleton			
abc	8115	7539	**7.1**
ajc	17403	16667	**4.2**
SpaceWar			
abc	150869	145391	**3.9**
ajc	222446	215995	**2.9**
Rin'G			
abc	947179	805162	**15**
ajc	1212273	1001661	**17.4**

Table 1. Code generated for AspectJ programs by original and optimized compilers.

of method calls and member variable operations, as well as the executing object at the join points. Once captured, these variables are made available to the advice body. In around advice, captured context variables must be passed on to shadows in the *proceed* call.

However, even if the programmer doesn't capture context variables explicitly in pointcut expressions, the shadow's environment must be kept after it has been extracted to a shadow method during weaving. This is done by passing context as arguments from the original join point environment to the advice method, and then on to the shadow method, as can be seen in the woven code of Figure 1.

If the programmer uses the context capture clauses, there is always an intersection between this explicitly captured context and the set of variables required for shadow extraction. Therefore, whenever an around advice uses context capture clauses in its definition, redundant parameters are introduced in its implementations' signatures.

Context variable repetition leads to three problems in generated *bytecode* for around advice:

- redundant parameters add to the size of method definitions in *bytecode*, thus resulting in larger code;
- memory consumption is larger than necessary, since activations of advice methods in the execution stack allocate local variables for redundant parameters;
- execution time is wasted loading redundant arguments to advice method calls.

Consider replacing the around advice from Listing 1 with the one in Listing 4, which captures the argument of calls to `setRadius`. The woven `Circle` class after this modification is shown in Listing 5. Notice that the advice implementation contains an unused parameter, and the same local variable is used at the join point as an argument for both repeated parameters.

Capturing environment variables required for shadow execution is part of the shadow extraction process presented in Section 2. A corresponding parameter is added to the advice implementation's signature for each one of these variables in this step. While the advice is being inlined, variables explicitly captured by the programmer are also added as parameters to the advice implementation. Failure to detect the intersection between the sets of variables captured in these two separate steps leads to redundant parameters in advice implementations.

```
1  public aspect RadiusCheckAspect {
2      void around(int r): call(void Circle.setRadius(int)) && args(r) {
3          proceed(r < 0 ? 0 : r);
4      }
5  }
```

Listing 4. Around advice capturing and modifying context from its join points.

```
1  public class Circle {
2      /* ... */
3      public Circle(int x, int y, int radius) {
4          setX(x);
5          setY(y);
6          aroundAdvice1(this,radius,radius);
7      }
8      public static void aroundAdvice1(Circle arg0, int arg1, int arg2) {
9          if (arg1 < 0)
10             shadow1(arg0,0);
11         else
12             shadow1(arg0,arg1);
13     }
14     public static void shadow1(Circle arg0, int arg1) {
15         arg0.setRadius(arg1);
16     }
```

Listing 5. Class `Circle` after the weaving with context passing.

This problem can be fixed by keeping a record of captured local variables during shadow extraction, so that they won't be captured a second time while inlining the advice method. This solution has been suggested to both *ajc* and *abc* developers, and its implementation is currently being discussed [Cordeiro 2006b, Cordeiro 2006c].

Table 2 shows reduction in code size as a result of eliminating repeated context variables. *Production Line*, which serves as an example for this optimization, is a dynamic programming solution to the problem proposed in [Cormen et al. 2002, Chap. 15]; there are two production lines with equal sequences of machines that perform the same job, but at different latencies. The problem is to find the minimum time required to go through the production line, considering that artifacts produces by a machine in one line may be transfered to the other line at a time cost. In this case, however, AspectJ was used to implement transparent memoization in the recursive solution to the problem.

Eliminating parameters from advice implementations, at *bytecode* level, removes structures used to describe the type of these parameters and instructions to load them. When compared with total program size, this reduction is small, as can be seen in Table 2.

The entry in Table 2 for the code generated by *abc* for the *Production Line* program shows that the decrease percentage is ten times larger than for the *ajc* version. This is due to a collateral effect of eliminating repeated context variables, in which *abc* is able to eliminate repeated advice and shadow implementations. After weaving, the inliner attempts to identify replicas among the generated methods. Since this is done on Jimple code, there are local variables that bind context variables to advice parameters, which makes each inlined implementation different. Once context variables are eliminated, these local variables are also removed from the Jimple code, and the inliner manages to identify that the advice implementations are equivalent[3].

Application	Original Code (*bytes*)	Optimized (*bytes*)	Decrease (%)
Production Line			
ajc	14693	14568	**0.90**
abc	7481	6802	**9.00**
SpaceWar			
ajc	222446	222310	**0.06**
abc	150881	150746	**0.09**

Table 2. Code sizes for original and optimized AspectJ programs.

Table 3 shows average measures for the running time of calls to a few methods in the *SpaceWar* program. These methods are captured by an around advice, and thus measuring their execution time shows the effect in advice activation time of eliminating parameters from advice implementation signatures. The average times shown here were collected from a sample of 33 executions of each method for diminishing the impact of OS scheduling and other external runtime interferences.

Method	Original (ms)	Optimized (ms)	Decrease (%)
ajc			
`fire`	2.691	2.595	**3.57**
`rotate`	0.0117	0.0113	**3.42**
`thrust`	0.0125	0.0114	**8.80**
abc			
`fire`	2.358	2.291	**2.84**
`rotate`	0.0107	0.0104	**2.80**
`thrust`	0.0138	0.0120	**13.04**

Table 3. Average execution times for methods affected by an around advice in the *SpaceWar* program.

Though the impact in execution time of a single advice call is almost negligible, as shown in Table 3, the impact in the running time of programs with a great number of advice calls at runtime can add up to be quite large. This is especially the case when around advice applies to recursive methods, as in the *Production Line* algorithm; the

[3] Equivalence between methods is determined in *abc* by string representations of Jimple code, rather than their semantics.

running time of this program for random production lines of different sizes is shown in Table 4.

Another important consequence of eliminating repeated context variables from around advice is a decrease in memory consumption. Parameters are stored as local variables in *frames* by the JVM for each method activation. Thus removing some parameters from an advice implementation makes its activation frames smaller, which allows programs with around advice applied to recursive methods to run for larger inputs. This is shown in the last four lines of Table 4: the *bytecode* compiled by the original *ajc* runs for production lines of up to 967 machines, while the optimized version runs for inputs of up to 1017 machines – about 5% larger. The same happens with the code compiled by *abc*, with the optimized version running for inputs of about 8% more machines than the original.

Input size (machines)	Original (ms)	Optimized (ms)	Decrease (%)
100			
ajc	4.856	2.508	**48.35**
abc	4.562	2.388	**47.65**
500			
ajc	6.397	3.991	**37.61**
abc	5.724	3.411	**40.41**
900			
ajc	9.248	6.372	**31.10**
abc	6.045	3.622	**40.08**
967			
ajc	9.363	6.399	**31.66**
abc	7.529	4.844	**35.66**
1017			
ajc	–	6.472	**–**
abc	7.752	5.101	**34.20**
1230			
ajc	–	–	**–**
abc	7.083	4.860	**31.39**
1341			
ajc	–	–	**–**
abc	–	5.089	**–**

Table 4. Average execution times of the *Production Line* program for random inputs.

5. Conclusions

This paper presented a study of around advice weaving techniques applied by two AspectJ compilers: the AspectJ Compiler, *ajc*, and the AspectBench Compiler, *abc*. Code repetition problems have been identified in these techniques.

Repeated advice and shadow implementations appear in *bytecode* generated by *ajc* and *abc* when a single class contains several identical shadows of an around advice.

By eliminating advice and shadow implementation replicas, this optimization decreases the *bytecode* size for AspectJ programs. During development of this work the developers of *abc* have also implemented this optimization in their weaver.

While capturing context variables for around advice implementations, some local variables are captured more than once, producing repeated context variables in advice implementations. This problem appears when context variables are explicitly captured by the programmer in pointcut expressions, by means of the **`args`**, **`target`** and **`this`** clauses. Solutions to this problem have been experimentally integrated into the *abc* and *ajc* compilers, and operate in the weaving process. Once repeated variables have been eliminated, the resulting code is smaller, uses less memory and runs faster. Memory consumption and time reductions are more relevant in programs that have around advice applied to recursive methods, where several advice activation frames coexist in the execution stack.

Code generated by the *abc* compiler shows clearly that the AspectJ language constructs are not inherently expensive, but rather implemented in an expensive way in the *ajc* compiler. These constructs can, in fact, be implemented efficiently, as in the *abc* compiler, though this is not the priority for *ajc* developers. Efforts in *ajc* development have been concentrated on compilation and weaving speed, as well as the introduction of load time weaving for the AspectJ language.

The main contribution of this paper is the identification of two problems caused by around advice weaving in AspectJ compilers. Solutions to these problems have been proposed to the developers of these compilers and are currently under discussion [Cordeiro 2006a, Cordeiro 2006b, Cordeiro 2006c]. A formal evaluation of the proposed optimizations remains as a future work at the time of writing.

Though the AspectJ language is currently used in software development in production environments, this study shows that small optimizations may still improve the performance of programs written in this language, which indicates that the compilation techniques for aspect oriented programs are still in a stage of continuous evolution.

References

Aspect Bench Compiler Team (2006). Official *abc* project page. http://www.aspectbench.org. Last visited in December 2006.

AspectJ Team (2006). Official page for the AspectJ project and *ajc* compiler. http://www.eclipse.org/aspectj. Last visited in December 2006.

Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2004). Building the abc AspectJ compiler with Polyglot and Soot. Technical Report abc-2004-2, The abc Group.

Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2005). Optimising AspectJ. *PLDI'05*.

Cordeiro, E., Stefani, I., Soares, T., and Tirelo, F. (2004). Rin'g: Um ambiente não-intrusivo para animação de algoritmos em grafos. In *XII WEI, em Anais do SBC 2004 - XXIV Congresso da Sociedade Brasileira de Computação*, volume 1.

Cordeiro, E. S. (2006a). Around advice weaving generates repeated methods. *Bug report* available at `https://bugs.eclipse.org/bugs/show_bug.cgi?id=154253`.

Cordeiro, E. S. (2006b). Around weaving produces repeated context variables. *Bug report* available at `https://bugs.eclipse.org/bugs/show_bug.cgi?id=166064`.

Cordeiro, E. S. (2006c). Around weaving produces repeated context variables. *Bug report* available at `http://abc.comlab.ox.ac.uk/cgi-bin/bugzilla/show_bug.cgi?id=77`.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2002). *Algoritmos: Teoria e Prática*. Editora Campus.

Dahm, M., van Zyl, J., and Haase, E. (2003). Official BCEL Project Page. http://jakarta.apache.org/bcel. Last visited in December 2006.

Dufour, B., Goard, C., Hendren, L., et al. (2004). Measuring the Dynamic Behaviour of AspectJ Programs. *OOPSLA'04*.

Hannemann, J. and Kiczales, G. (2002). Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA. ACM Press.

Hilsdale, E. and Hugunin, J. (2004). Advice Weaving in AspectJ. *AOSD'04*.

Kuzins, S. (2004). Efficient Implementation of Around-advice for the AspectBench Compiler. Master's thesis, Oxford University.

Laddad, R. (2003). *AspectJ in Action*. Manning Publications Co.

Lindholm, T. and Yellin, F. (1999). *The Java Virtual Machine Specification*. Addison-Wesley Professional, second edition. Available at `http://java.sun.com/docs/books/vmspec/index.html`.

Myers, A. (2006). Official polyglot project page. `http://www.cs.cornell.edu/Projects/polyglot`. Last visited in December 2006.

Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., and Co, P. (1999). Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135.

A Visual Language for Animated Simulation *

Vladimir O. Di Iorio[1], **Débora P. Coura**[2],
Leonardo V. S. Reis[1], **Marcelo Oikawa**[1], **Carlos R. M. Junior**[1]

[1]Departamento de Informática – Universidade Federal de Viçosa (UFV)
Av. P.H. Rolfs, s/n
36.570-000 – Viçosa – MG – Brazil

[2]Unileste-MG – Centro Universitário do Leste de Minas Gerais
Av. Presidente Tancredo de Almeida Neves, 3500
35.170-056 – Coronel Fabriciano – MG – Brazil

vladimir@dpi.ufv.br,dcoura@terra.com.br,
{leo,moikawa,crmarques}@dpi.ufv.br

Abstract. *This paper presents a visual language for producing animated simulations. The language is implemented on a tool called* Tabajara Animator*, using principles of* Programming By Demonstration (PBD)*, which is a technique for teaching the computer new behaviour by demonstrating actions on concrete examples. The language is based on a formal model for concurrent update of agents, which represent the animated characters. The visual rules follow the "before-after" style, adopted by the most important similar tools. New features discussed by this work may produce a significant reduction on the number of required rules for producing animated simulations. This paper shows how these new features are implemented on a visual user-friendly interface, and how they are translated into structures of the formal model adopted.*

1. Introduction

Programming by Demonstration (PBD) is a technique for teaching the computer new behaviour by demonstrating actions on concrete examples [Lieberman 2001]. This technique has been used with success in several areas, for example, the construction of Web pages [Sugiura 2001], programming on Geographical Information Systems (GIS) [Traynor and Williams 2001], minimizing typing in small handheld devices [Masui 1998]. Tools for producing animated simulations represent an area with the first commercial systems that applied PBD successfully. Examples are *Stagecast Creator* [Smith et al. 2000] and *Agentsheets* [Repenning and Sumner 1995].

In [Coura et al. 2006], the authors propose three enhancements for PBD-based animated simulations systems. The most important enhancement is *first-person perspective* for visual rules. Stagecast Creator and Agentsheets use third-person perspective, leading to programs with several very similar visual rules, whose only difference is the orientation of the characters. The other new features are *negative conditions* and the use of *inheritance*. Using a significant example, it is shown that the enhancements may produce an important reduction on the number of visual rules required for a simulation. But important issues concerning the implementation of the enhancements are not solved.

*This work is partially supported by FAPEMIG, Brazil

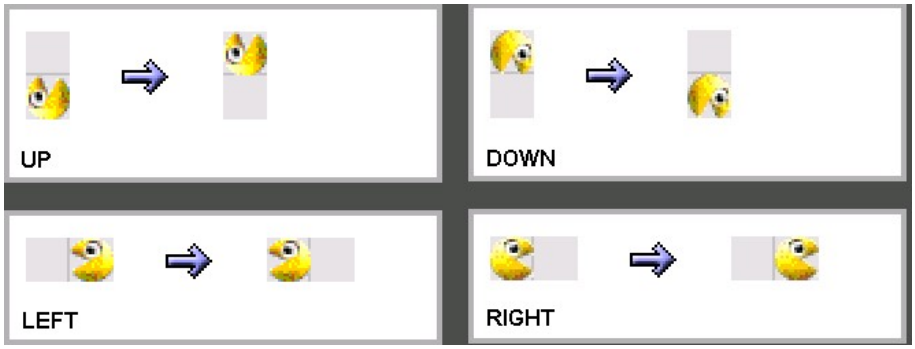


Figure 1. *Visual before-after rules* in Stagecast Creator.

This paper is an extension of the work published in [Coura et al. 2006], presenting the visual language of a system called *Tabajara Animator*. This system proposes visual solutions for the implementation of *negative conditions* and *inheritance*, and solves problems with the automatic generation of graphical representation of characters, associated with *first-person perspective*. The visual language is based on a formal model called ASM-OBJ, inspired by the Gurevich's *Abstract State Machines (ASM)* [Börger and Stärk 2003]. This paper also shows how visual programs containing the new features may be translated into programs of this model.

In Section 2, the most important tools for producing animation with PBD are presented. It is shown how the enhancements proposed in [Coura et al. 2006] may reduce the number of visual rules required, in situations commonly used. Section 3 introduces ASM-OBJ, a formal model for the definition of concurrent update of agents. In order to help explaining the semantics of the new visual language features, their translation into structures of the ASM-OBJ model is discussed, in the following sections. Section 4 presents an overview of the Tabajara Animator system. Sections 5, 6 and 7 analyzes the implementation of negative conditions, inheritance and first-person perspective for rules, respectively. In Section 8, the final conclusions are presented and future works are discussed.

2. Tools for Animated Simulation using PBD

The most successful tools for animated simulation using PBD are *Stagecast Creator* [Smith 2000, Smith et al. 2000] and *Agentsheets* [Repenning and Sumner 1995]. They have similar features, but the visual rules in Stagecast Creator are called *visual before-after rules*, while in Agentsheets, they are called *graphical rewriting rules*.

The *Kidsim* project [Smith et al. 1994], later called *Cocoa*, was finally designated *Stagecast Creator*. It has been used mostly in children education. As the original name implies, it was intended to allow kids to construct their own simulations, reducing the programming task to something that anyone could handle.

Figure 1 shows four Stagecast Creator rules, taken from a version of the classic *Pacman* game [DeMaria and Wilson 2003]. The main character, the Pacman, is controlled by the user, using the keyboard. It moves in a labyrinth, together with ghosts. The rules on Figure 1 define the movement for the Pacman, when there is an empty space in front of it, in four different directions. The visual representation of the rules are separated in two

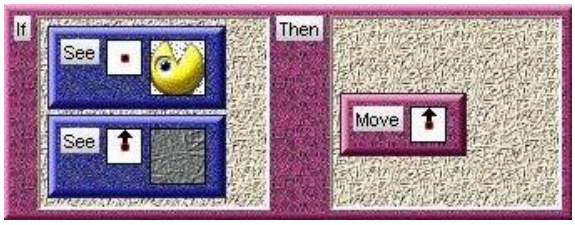


Figure 2. *Graphical rewriting rule* in Agentsheets.

parts, by a horizontal arrow. The left part, designated *before* clause, represents a possible situation occurred during a simulation. For example, the first rule represents a situation when the character is looking at a position on a cell over it, and this cell is empty. The right part of the rule, designated *after* clause, represents an action that must be executed when the condition of the *before* clause is satisfied. In order to define the *after* clause of the first rule, the user *demonstrates* his intention by moving the character up.

The Agentsheets environment aims to achieve a wide range of users, from children to professionals. The system includes a compiler which translates the visual rules to Java code, producing more efficient simulations than the ones built using Stagecast Creator.

Figure 2 shows a visual rule of Agentsheets with the same semantics as the first rule of Figure 1. In Agentsheets, to indicate that a character must move when an empty space is found, the user must define a position update on a graphical rewiriting rule. The condition to be satisfied defines that the character must have the given graphical representation, and the cell over it must be empty. The action executed when this condition is satisfied is a movement to a cell over the current position. Similarly to Stagecast Creator, additional rules must be built to define movements to other directions.

The Tabajara Animator project was created in order to implement the new features proposed in [Coura et al. 2006], for visual rules using PDB: *first-person perspective*, *negative conditions* and *inheritance*. The authors demonstrate that, with these innovations, the numbers of rules required for animated simulations can be significantly reduced.

In the examples of figures 1 and 2, as the character may be pointing to four different directions and the rules are written in third-person perspective, it is necessary to write four different rules to produce the desired behaviour. First-person perspective for rules is one of the new ideas implemented in Tabajara Animator. With first-person perspective, this behaviour can be defined by a single rule. Section 7 shows details of this important enhancement, and how it is implemented in Tabajara Animator.

In Stagecast Creator, the rules of Figure 3 may define the movement over empty spaces for a character representing a ghost, on the Pacman game. These rules are almost identical to the ones of Figure 1, the only difference is the character involved. This situation is a good opportunity to explore *inheritance*. The desired behaviour may be defined on a class named *Moveable*. Classes *Pacman* and *Ghost* may be subclasses of *Moveable*, inheriting all the rules of the superclass. Section 6 discusses the problems involved in the implementation of inheritance.

Figure 4 shows the rules defining the movement over cells containing a “vitamin”, for a character representing a Pacman. These rules are almost identical to the ones of

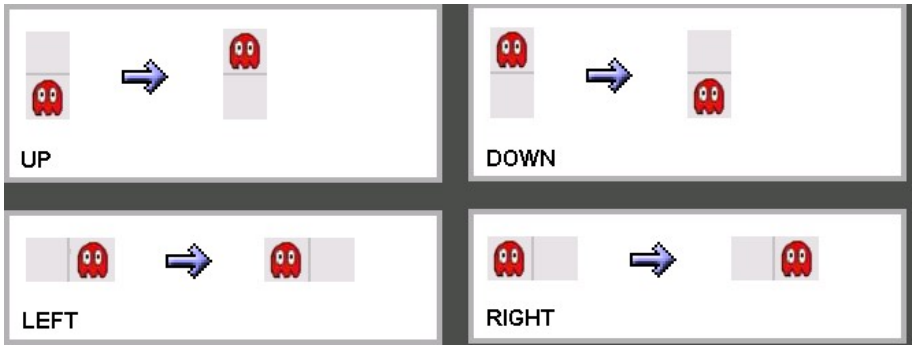


Figure 3. Rules for the movement of *ghost* characters in Stagecast Creator.

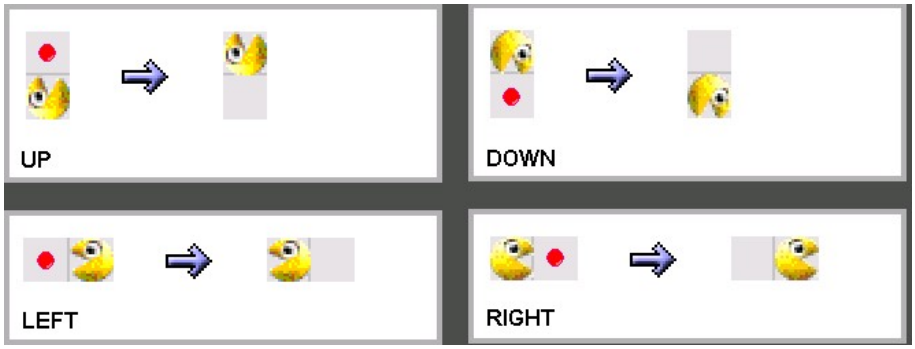


Figure 4. Movement of Pacman over vitamins, in Stagecast Creator.

Figure 1. The two sets of rules could be replaced by a rule with *negative condition*. The new rule could have the following semantics: the ghost must move to a cell in front of it, if this cell does **not** contain a wall of the labyrinth. Section 5 discusses the implementation of negative conditions.

3. The ASM-OBJ Formal Model

Abstract State Machines (ASM), formerly known as *Evolving Algebras*, are a formal model where the state of a system is represented by functions, and transitions are based on *function update*. A complete definition of this model can be found in [Börger and Stärk 2003]. ASM-OBJ is an extension of the ASM model which includes elements from object-oriented languages. Object-oriented extensions for ASM have been proposed in works like [Janneck and Kutter 1998] and [Zamulin 1998]. ASM-OBJ has been created especially for serving as basis for the Tabajara Animator visual language, and shares little similarities with these previous works. This section presents an informal definition of the main elements of the model. The semantics is given by associations with pure ASM. The complete definition of ASM-OBJ can be found in [Coura 2006].

The syntax of ASM-OBJ is defined using a XML schema [van der Vlist 2002], so ASM-OBJ programs are well-formed XML documents. A reason for choosing XML instead of conventional syntax is that the visual language is intended to be shared by different applications. The code is automatically generated by visual tools, so it is not necessary to actually write programs using XML syntax, what would be an important drawback. In this section, parts of the ASM-OBJ model definition are presented using a visual representation for XML schemas, and the semantics is explained using pure ASM.

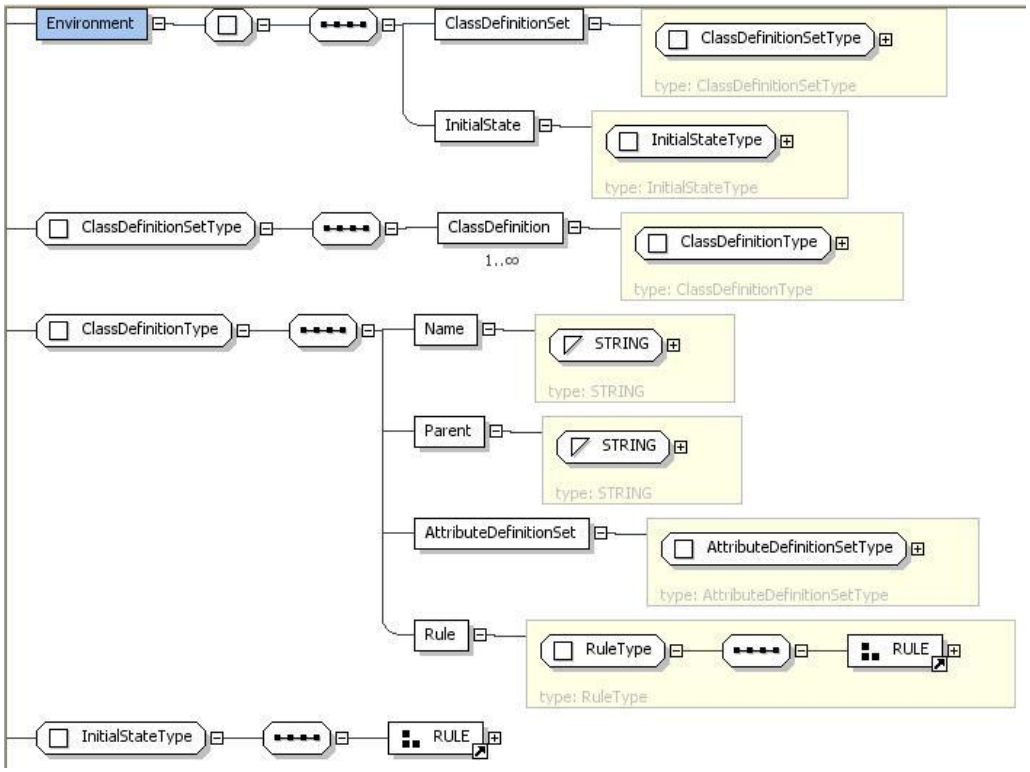


Figure 5. ASM-OBJ *environment* definition.

The visual representation used is the one proposed by the *Oxygen* system [Wheller 2002], with icons like ▣ to represent a complex type, ▬ to represent composition and ⁞ to represent a choice.

Figure 5 shows the definition of an *environment* in ASM-OBJ, composed by a set of *class definitions* and an *initial state*. The definition of a class includes its name, the name of its parent on the hierarchy, a set of *attributes* and a *rule*. The initial state is defined by a *rule*. The semantics is explained in terms of the ASM model, as follows. Each ASM-OBJ class with name C is equivalent to an ASM *universe* with the same name, i.e., an unary relation identified by the set of elements e such that $C(e) = \mathit{true}$. The definition of class attributes in ASM-OBJ, not shown in Figure 5, includes the name of the attribute and its type, which can be scalar types such as *Integer*, *Real* and *String*. Each type may be interpreted as another ASM universe. An attribute named A, inside a class named C, is interpreted as an ASM unary function $A : C \rightarrow T$, where T is the interpretation of the attribute type. An *agent* of a class named C acts like an *object* of OO languages, and is interpreted as an element of an ASM universe C.

ASM-OBJ defines several types of rules, very similar to ASM rules. Figure 6 shows part of the definition of some of these rules. The semantics of the *execution* of an ASM-OBJ rule is equivalent to *firing* an ASM rule. The execution of an ASM-OBJ *update* rule produces an update pair (*location*, *value*), just like in ASM, calculated using the *LeftSide* and *RightSide* components of the rule. A *location* defines an unique "address" which is associated to a value. In ASM, it is defined by a function and arguments. In ASM-OBJ, it is defined by an agent and an attribute of the agent´s class. The result of executing a *block* is the union of the execution of each of its subrules. The result of

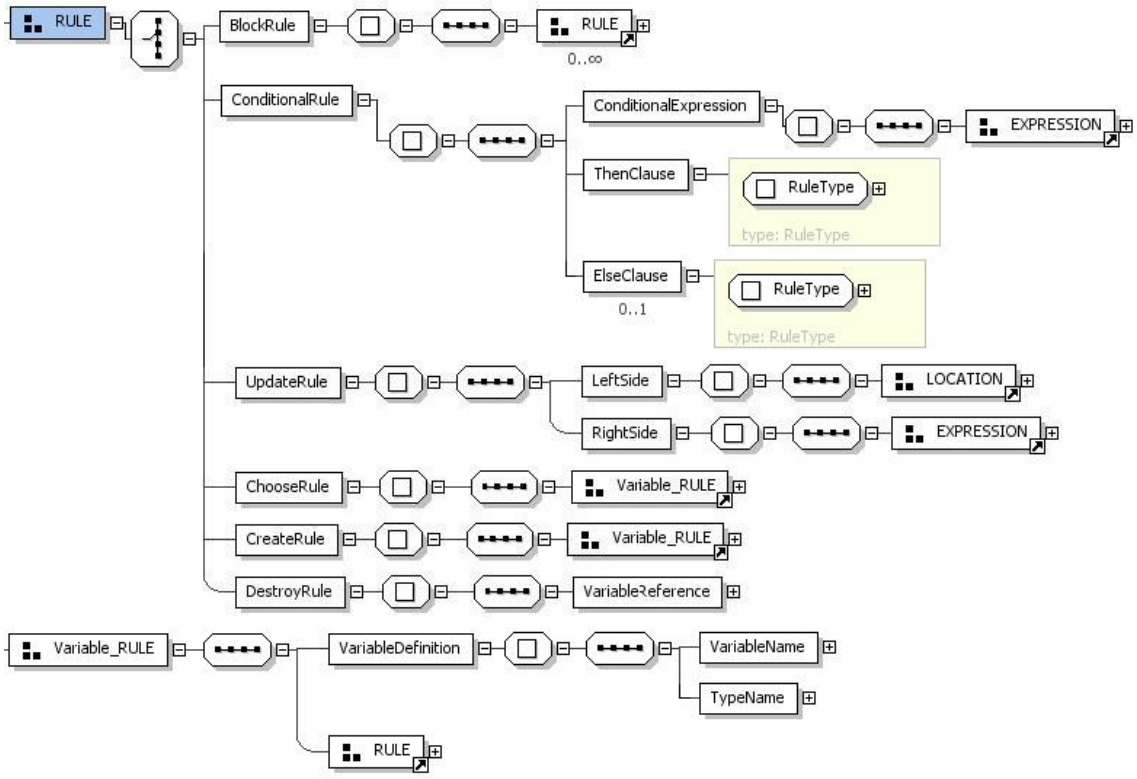
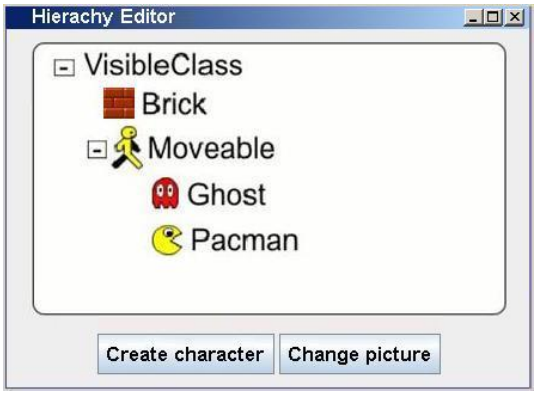


Figure 6. Some ASM-OBJ *rules*.

executing a *conditional* rule is the execution of the *then* clause, when the *conditional expression* is satisfied; otherwise, it is the execution of the optional *else* clause. In a *choose* rule with variable v of class C, an agent a of the class C is nondeterministically chosen and the result is the execution of the defined subrule, with the value of v set to a. In a *create* rule with variable v of class C, a new agent a of the class C is created and the result is the execution of its subrule, with the value of v set to a.

Inside the rule associated to a class, the reserved name *this* is an unbound variable. The *complete rule* of a class named C is the union of the rule associated to this class with the rules associated to all its superclasses, following the hierarchy defined by the environment. The execution of an agent a of a class C is the execution of the complete rule associated to the class C, with the value of the variable *this* set to a. The execution of an environment starts with the execution of its initial state rule. This rule is usually a block containing *create* rules, which will build an initial set of *live agents*. Then, each step of the execution consists of selecting a subset of agents from the live agents set, executing the selected agents and finally building a new state. A new state is built applying the produced update pairs to the current state. As in pure ASM, applying an update pair (l, k) to a state produces a new state where the value associated to location l is replaced by k.

ASM-OBJ defines a rich set of standard functions which can be used in expressions. Another set of functions, called *external functions*, may be extended by users. External functions are used, primarily, for the communication with the external environment. Two calls to an external function, in different steps of an execution, may return different values, even when given the same arguments.




```
...
<ClassDefinition>
  <Name>Moveable</Name>
  <Parent>VisibleClass</Parent>
  ...
</ClassDefinition>
<ClassDefinition>
  <Name>Pacman</Name>
  <Parent>Moveable</Parent>
  ...
</ClassDefinition>
...
```

Figure 7. Hierarchy editor of Tabajara Animator and representation in ASM-OBJ.

4. Overview of Tabajara Animator

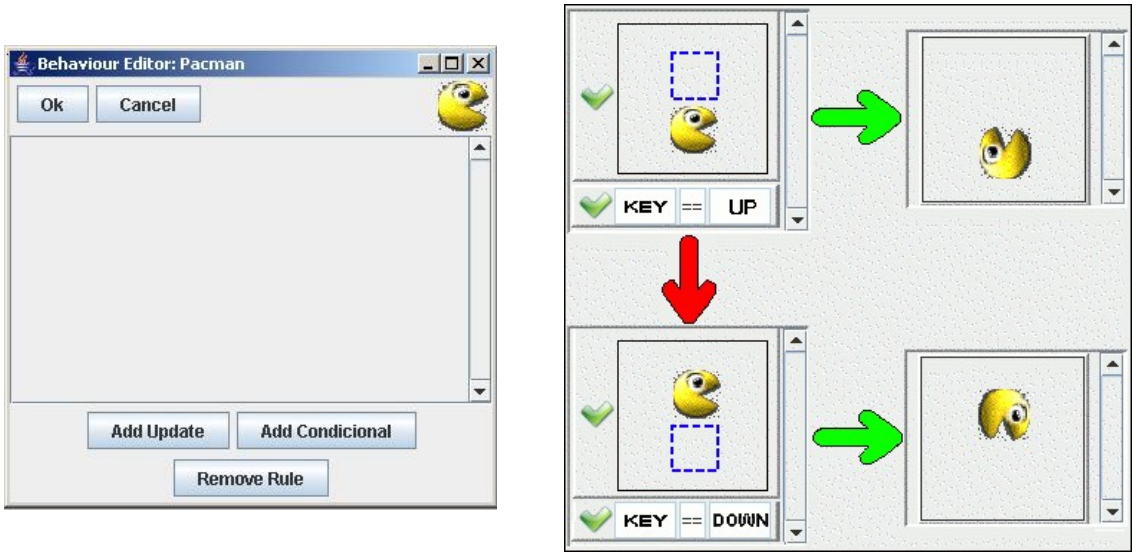
The Tabajara Animator is a system for the creation of animated simulations using programming by demonstration techniques. It is similar to Stagecast Creator and Agentsheets, but with the additional features proposed in [Coura et al. 2006].

To build visual programs, the system offers an *Hierarchy Editor*, which allows the definition of hierarchy relations, and a *Behaviour Editor*, which allows the definition of visual rules for each class. To present simulations, the system offers several different *animated simulation windows*. Simulations windows are discussed in Section 7.2.

The left side of Figure 7 shows a snapshot of the Hierarchy Editor window of Tabajara Animator. The user may create new classes anywhere in the hierarchy, defining a standard visual representation. In this example, *Brick*, *Moveable* and its two subclasses *Ghost* and *Pacman* are user-defined classes. The root of the hierarchy is the predefined class *VisibleClass*, representing any object with visual representation in an animated simulation. The right side of Figure 7 shows part of the *ClassDefinitionSet* element of an ASM-OBJ environment, representing these hierarchical relations.

Figure 8(a) shows the Behaviour Editor window, which can be activated from the Hierarchy Editor. After selecting a class and activating the Behaviour Editor, the user can create, delete or modify rules for this class. The visual rules have a strong correspondence with the structure of an ASM-OBJ program. Visual conditional rules are formed by a "before" clause, which represents the visual condition to be evaluated, an "after" clause, which represents the rule to be executed if the condition is satisfied, and an optional "else" clause, which represents the rule to be executed if the condition is not satisfied. An "after" clause is usually a visual update rule. An "else" clause may be a visual update or another visual conditional rule.

Figure 8(b) shows a visual conditional rule for the Pacman class. "Before" and "after" clauses are separated by horizontal (green) arrows. "Else" clauses appear behind vertical (red) arrows. Conditions, inside "before" clauses, are visual predicates connected by an implicit logic operator *and*. For example, the first condition shown in Figure 8(b) is satisfied if there is no character above the Pacman, **and** the *up arrow* keyboard is pressed. Visual update rules may define modifications on the visual attributes of characters: their position, rotation and visual representation. In Figure 8(b), the user demonstrated the intended behaviour by defining visual update rules with a rotation applied to the graphical



(a) Behaviour Editor Window. (b) A visual conditional rule.

Figure 8. Behaviour Editor Window and a complex visual conditional rule.

representation of the character. So, during a simulation, if the condition is satisfied, the character will be rotated as defined by the update rule.

Tabajara Animator offers a user-friendly interface to define visual conditional and update rules. The rectangular area with dashed sides, used in visual conditions, can be placed anywhere inside the window. If it is left empty, then the condition will be satisfied, during simulation time, only when there is no character inside the defined area. If characters are dragged into the rectangular area, then the condition is satisfied, during simulation time, when objects of the chosen classes are found inside the defined limits. Characters selected in "before" clauses appear automatically in "after" clauses, with the same position and rotation, unless the *negation operator* is applied to the visual condition (see Section 5). Moving and rotating these characters in "after" clauses, an user defines a visual update that will be executed on simulation time, when the visual condition is satisfied. Deleting characters or inserting new characters in "after" clauses, an user defines destruction and creation of characters, in simulation time.

An ASM-OBJ interpreter is integrated to the Tabajara Animator interface. Before carrying on simulations, the system translates the visual rules into ASM-OBJ code. The translation of visual elements discussed in this section is very obvious. Visual conditional rules are translated to ASM-OBJ conditional rules, with visual "before" clauses associated to conditional expressions, "after" clauses associated to *then* clauses and visual "else" clauses associated to ASM-OBJ *else* clauses. Complex visual conditions, with more than one predicate, are implemented using a call to an ASM-OBJ standard boolean function *and*. Visual update rules are translated to ASM-OBJ update rules, with class attributes representing position, rotation and other visual attributes.

5. Negative Conditions

The structure of a visual conditional rule is itself a way to define negative conditions. Any rule inside an "else" clause will be executed only if the condition is **not** satisfied.

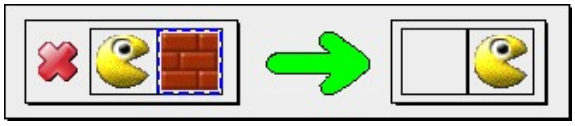


Figure 9. A rule with negative condition.

But the system offers also a visual *negation operator* which can be associated to any predicate. It indicates that a condition is true only if the predicate is **not** satisfied. The negation operation, together with the implicit *and* operator described in Section 4, allows the definition of complex conditions. In systems like Stagecast Creator and Agentsheets, users are restricted to a more limited set of possible conditions.

Figure 9 shows an example of a rule with negative condition, for the Pacman class. A symbol ✔ appears before every positive condition. If the user wants a negative condition, this symbol must be replaced by a symbol ✘. The interface allows the user to change from positive to negative conditions, and vice-versa, with a click of the mouse. Using also the concept of first-person perspective, the semantics of the rule shown is: if the character is looking at a cell which does **not** contain a brick, then the character is moved to this cell. The visual negation operator is straightforward translated to ASM-OBJ by using a call to a standard boolean function *not*.

6. Inheritance

In [Repenning and Perrone 2000], the authors present some reasons not to use inheritance in systems for the creation of animated simulations with visual languages. They argue that abstractions like inheritance are nontrivial for end users to understand, and are hard to represent visually. They present a different approach for generalization, called *programming by analogous examples*.

Examples presented in [Coura et al. 2006] show that inheritance may indeed reduce the number of required rules in some simulations. The authors agree that it may be an abstract concept not very easy to understand by end users, but they argue that a system may offer inheritance as an additional feature, reserved for more advanced users.

This work shows how Tabajara Animator solves the problem of representing inheritance relations and inherited rules on subclasses. As in many other systems, hierarchical relations are visually represented by a tree diagram, as shown in Figure 7. The class associated with a tree node is the superclass of the classes associated with the siblings of this node. For example, *Pacman* is a subclass of *Moveable*. Figure 10 shows two instances of the Behaviour Editor window. The window associated to the Moveable class shows a rule with negative condition. The window associated with the Pacman class shows the inherited rule, changing the graphical representation of the character. With this solution, it is easy to understand the behaviour of a class, even when it has inherited rules. But the system allows the edition of rules only on the classes where they are originally defined. On subclasses, inherited rules are “read-only”.

7. First-Person Perspective

Rules in Stagecast Creator and Agentsheets use third-person perspective. As the examples of Section 2 show, this strategy may lead to visual programs with several similar rules,

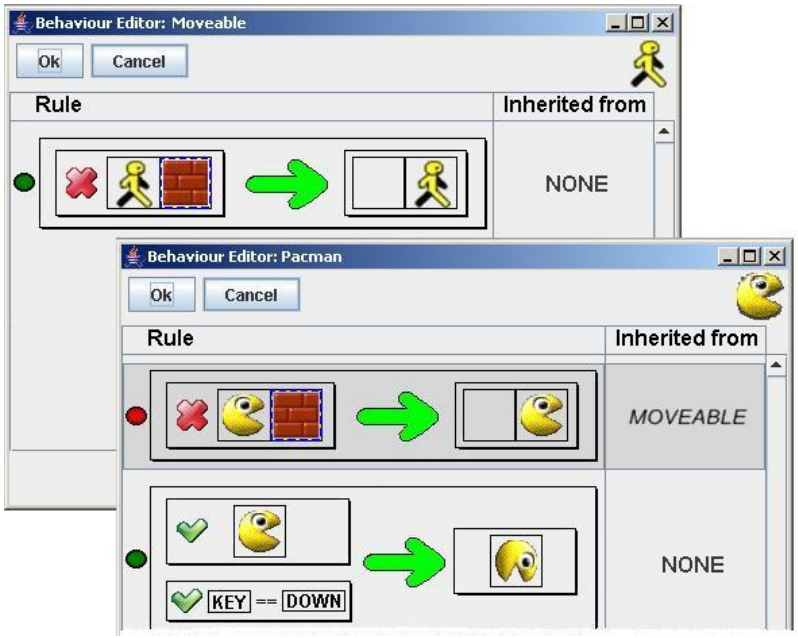
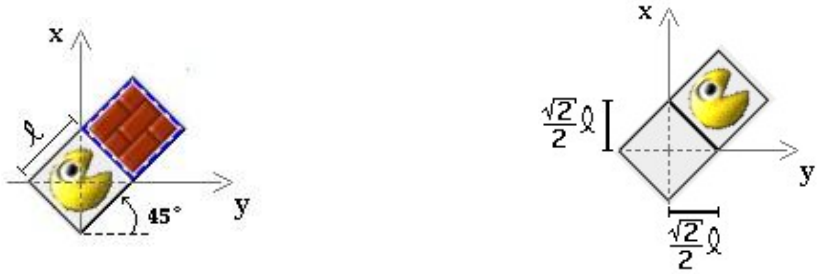


Figure 10. Representing inherited rules in subclasses.



(a) Verify Brick on a cell. (b) Move the character.

Figure 11. Calculations needed to implement first-person perspective.

where the only difference is the orientation of the characters. In order to implement first-person perspective for rules, it is not necessary to add new features to the interface. However, a lot of additional work must be done by the system, as discussed below.

The rule represented in Figure 9, when using first-person perspective, may be equivalent to several third-person perspective rules. The results produced in simulation time depend on the character current rotation. For example, suppose that the character is rotated 45 degrees counterclockwise. The real position of the rectangular area with dashed sides, where a Brick character may be positioned, must be calculated as shown in Figure 11(a). The update defined by the rule of Figure 9 represents only a horizontal movement. But in simulation time, this movement may have horizontal and vertical components, which must also be automatically calculated, as shown in Figure 11(b). All these calculations are carried out by ASM-OBJ code properly generated.

7.1. Code generated for managing first-person perspective features

Tabajara Animator defines *VisibleClass* as the superclass of any visible character. The attributes of VisibleClass define the current graphical representation, the current rotation

```
<ChooseRule>
 <VariableDefinition>
  <VariableName>x</VariableName>
  <TypeName>Brick</TypeName>
 </VariableDefinition>
 <ConditionalRule>
  <ConditionalExpression>
   <StandardFunctionCall>
    <FuncName>not</FuncName>
    <ArgumentList>
     <ExternalFunctionCall>
      <Name>RectangularArea</Name>
      <ParameterMap>...<ParameterMap>
     </ExternalFunctionCall>
    </ArgumentList>
   </StandardFunctionCall>
  </ConditionalExpression>
  <ThenClause> ...  </ThenClause>
 </ConditionalRule>
</ChooseRule>
```

(a)

```
<StandardFunctionCall>
 <FuncName>add</FuncName>
 <ArgumentList>
  <Location>
   <VariableRef>this</VariableRef>
   <AttribName>posX</AttribName>
  </Location>
  <StandardFunctionCall>
   <FuncName>Xproj</FuncName>
   <ArgumentList>
    <IntegerConst>30</IntegerConst>
    <IntegerConst>0</IntegerConst>
    <Location>
     <VariableRef>this</VariableRef>
     <AttribName>rotation</AttribName>
    </Location>
   </ArgumentList>
  </StandardFunctionCall>
 </ArgumentList>
</StandardFunctionCall>
```

(b)

Figure 12. Pieces of an ASM-OBJ program.

and the horizontal and vertical components of the current position.

A visual conditional rule including a rectangular area with dashed sides is translated into an ASM-OBJ *choose* rule. The translation of the visual condition defined in Figure 9 is the ASM-OBJ *choose* rule shown in Figure 12(a). This rule defines a variable named *x*, whose type is the class named *Brick*. Its execution forces the system to find an agent of the Brick class, satisfying conditions defined in the subrule. The subrule of the *choose* rule, in this case, is a *conditional* rule. The condition to be evaluated is the result of the application of the standard boolean function *not* to a call to an *external function* named *RectangularArea*.

The *RectangularArea* function has a boolean return type. Arguments passed to this function, not shown in Figure 12(a), represent the base object, the position of the rectangular area and a variable passed by reference, which may be instantiated by the function. The base object, in this case, is the Pacman character. The position of the rectangular area is defined by its top left and right down corners, relative to the position of the base object. These values are translated into real coordinates using the current position and the current rotation of the base character, as shown in Figure 11(a). Finally, the last argument is the variable *x*, bound by the *choose* rule. External functions receive an implicit argument that allows them to access the whole environment, during a simulation. Using this implicit argument and the arguments explained above, the *RectangularArea* function scans the environment and verify whether there is a Brick character inside the calculated area. If such a character is found, the value of variable *x* is set to it, and the function returns *true*. Otherwise, the function returns *false*.

As shown in Figure 11(b), a position update requires the calculation of horizontal and vertical displacements. To solve this, every position update is translated into two ASM-OBJ update rules, using predefined functions named *projX* and *projY*. Figure 12(b) shows one of the update rules resulting from the translation of the position update defined

in Figure 9. To the value of the *posX* attribute of the character, which represents the horizontal component of the current position, the rule adds the result of a call to the *projX* function. The *projX* function receives three parameters: a horizontal displacement, a vertical displacement and a value associated with the rotation of the character. The result is the *real* displacement on the horizontal axis. The code shown in Figure 12(b) supposes that the graphical representation of the Pacman character is a square with side length of 30 pixels. So the call to the *projX* function calculates the **horizontal** result of a horizontal movement of 30 pixels to the right, considering the current rotation of the character. The second required update rule is not shown, but it is similar to the code shown n Figure 12(b), replacing *posX* by *posY* and *projX* by *projY*. In this case, the call to the *projY* function calculates the **vertical** result of a horizontal movement of 30 pixels to the right, considering the current rotation of the character.

7.2. Problems in animated simulation windows

Rules in Tabajara Animator are defined using first-person perspective. But the system provides variations of first-person and third-person perspective animated simulation windows. On windows that present animated simulations with first-person perspective, a character is chosen to be positioned in the center of the window, and its graphical representation never changes. When this character suffers rotation, the window presents an inverse rotation of all other characters. In third-person perspective animated simulation windows, rotated graphical representations are automatically generated for each character, according to its rotation attribute.

A problem occurs in third-person simulation windows. Frequently, the graphical representations automatically generated for characters are not satisfactory. For example, suppose that the graphical representation of the Pacman, used on the rule of Figure 9, is rotated 180 degrees clockwise (or counterclockwise). The resulting picture would be an upside down character. In order to solve this problem, Tabajara Animator offers a visual tool called *Image Configurator* which allows users to define different graphical representations for characters, depending on their rotation attribute. An example is shown in Figure 13, with four different graphical representations defined for the Pacman character. Each different image is associated with a range of values for the rotation attribute of a Pacman character. During simulation time, the system verifies the rotation of each agent of the Pacman class and shows the corresponding image.

The system translates the operations defined in an Image Configurator to ASM-OBJ code. When using third-person perspective animated simulation windows, an additional ASM-OBJ conditional rule is generated. This rule updates the attribute defining the graphical representation of the character, according to its rotation attribute.

8. Conclusions and Future Works

In [Coura et al. 2006], inheritance, negative conditions and first-person perspective for rules are enhancements proposed for PBD-based animated simulation tools. An example of a simple application with common situations found in animated simulations is defined. The application is implemented in Stagecast Creator and Agentsheets, which are tools with third-person perspective rules, without inheritance and with little support for negative conditions. The number of rules required for these implementations is compared to a tool

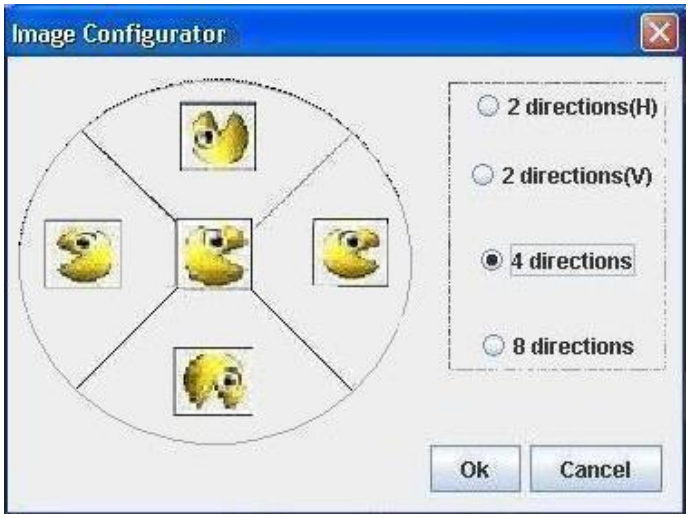


Figure 13. Defining graphical representation, depending on rotation.

with full support for the proposed enhancements. The results show that an important reduction on the number of rules may be achieved, using the enhancements.

The results presented in [Coura et al. 2006] are very significant, but they depend on an a successful implementation of all the proposed features. The main contribution of the work presented in this paper is the confirmation that inheritance, negative conditions and first-person perspective for rules may be successfully implemented. The paper shows details of the implementation, on a system called *Tabajara Animator*, using the help of a formal model to explain the semantics of the visual elements.

The implementation of the proposed features, in Tabajara Animator, may be summarized as follows. The interface of the system introduces new visual elements, not present in similar tools, in order to define negative conditions and inheritance. A visual negation operator may be applied individually to any predicate on a visual condition. It allows users to build more complex conditions than the ones provided by similar tools. Inheritance relations are represented on a tree diagram, and the problem of representing rules in subclasses is elegantly solved. First-person perspective for rules requires no additional visual elements on the interface of the system. The calculation of the real position of characters during simulation is carried out by predefined ASM-OBJ functions. An user-friendly visual tool allows the definition of different graphical representations for characters, solving the problems with automatically generated representations, in animated simulation windows.

The visual language of Tabajara Animator may be extended in several ways. An important extension may be the use of *rule abstractions*. The visual language of Agentsheets, called *Visual Agent Talk (VAT)*, offers such abstractions. A VAT *method* is a set of rules, which can be referenced by the name, inside other visual rules. An interesting work would be the implementation of a similar feature in Tabajara Animator. Because of inheritance, it would be necessary to define the behaviour in cases where a class has a method with the same name as a method defined in its superclass. Another possible extension to the Tabajara Animator visual language is the use of polymorphism. This feature has never been discussed together with languages for visual animation.

References

Börger, E. and Stärk, R. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag.

Coura, D., Di Iorio, V., Lima, A., Oliveira, A., and Andrade, M. V. (2006). Animações Através de Programação por Demonstração. In *Anais do Simpósio de Fatores Humanos em Sistemas Computacionais (IHC 2006)*, pages 81–90, Natal, Brazil.

Coura, D. P. (2006). Produzindo Animação Através da Programação por Demonstração. Master's thesis, Universidade Federal de Viçosa, Viçosa, Brasil.

DeMaria, R. and Wilson, J. L. (2003). *High Score!: The Illustrated History of Electronic Games*. McGraw-Hill Osborne Media.

Janneck, J. and Kutter, P. (1998). Object-based Abstract State Machines. TIK-Report 47, Swiss Federal Institute of Technology (ETH) Zurich.

Lieberman, H., editor (2001). *Your Wish is My Command: Programming by Example*. Morgan Kaufmann.

Masui, T. (1998). Integrating Pen Operations for Composition by Example. In *ACM Symposium on User Interface Software and Technology*, pages 211–212.

Repenning, A. and Perrone, C. (2000). Programming by example: programming by analogous examples. *Commun. ACM*, 43(3):90–97.

Repenning, A. and Sumner, T. (1995). Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. *IEEE Computer*, 28(3):17–25.

Smith, D. C. (2000). Building personal tools by programming. *Communications of the ACM*, 43(8):92–95.

Smith, D. C., Cypher, A., and Spohrer, J. (1994). KidSim: Programming Agents Without a Programming Language. *Communications of the ACM*, 37(7):54–67.

Smith, D. C., Cypher, A., and Tesler, L. (2000). Programming by example: novice programming comes of age. *Commun. ACM*, 43(3):75–81.

Sugiura, A. (2001). Web Browsing by Demonstration. In Lieberman, H., editor, *Your Wish is My Command: Programming by Example*, pages 61–86. Morgan Kaufmann.

Traynor, C. and Williams, M. G. (2001). End users and GIS: a demonstration is worth a thousand words. In *Your wish is my command: programming by example*, pages 115–134. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

van der Vlist, E. (2002). *XML Schema - The W3C's Object-Oriented Descriptions for XML*. O'Reilly.

Wheller, S. (2002). <oXygen/> User Manual. SyncRO Soft Ltd. (retrieved 25 Jan, 2007, from http://www.oxygenxml.com/).

Zamulin, A. (1998). Object-oriented Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University.

Improving Reusability in AspectLua

Mauricio B. C. Vieira[1] **, Thais V. Batista**[1]

[1]Computer Science Department
Federal University of Rio Grande do Norte – UFRN
Natal – RN – Brazil

vieira@ppgsc.ufrn.br, thais@dimap.ufrn.br

Abstract. *AspectLua is an extension of the Lua language to support dynamic aspect-oriented programming. It is based on AspectJ concepts and contains a meta-object protocol, LuaMOP, that handles a dynamic weaving process by exploiting the Lua reflective features. AspectLua specifications are limited in terms of aspect reuse, modularity and heterogeneous interaction. In order to address these limitations, this work proposes RE-AspectLua, a new version of AspectLua that uses the concepts of aspect interfaces with abstract joinpoints and connection language. Using these concepts, RE-AspectLua intends to break away from the syntactically manifest coding of aspects in which joinpoints are hard-coded into aspects, thereby promoting general reusability.*

Resumo. *AspectLua é uma extensão da linguagem Lua que dá suporte a programação orientada a aspectos dinâmica. AspectLua é baseada nos conceitos de AspectJ e faz uso de um protocolo de meta-objetos, LuaMOP, que efetua um processo de combinação dinâmica ao explorar as características reflexivas de Lua. As especificações em AspectLua são limitadas quanto ao reuso, modularidade e interação heterogênea do aspecto. Para resolver estas limitações, este trabalho propõe RE-AspectLua, uma nova versão de AspectLua que usa os conceitos de interfaces de aspecto com pontos de junção abstratos e linguagem de conexão. Ao usar estes conceitos, RE-AspectLua evita seguir a sintaxe de codificação de aspectos onde os pontos de junção são codificados nos aspectos e, portanto, promove reusabilidade.*

1. Introduction

Aspect-Oriented Programming (AOP) [Kiczales et al. 1997] has emerged to modularize elements that cut accross the basic decomposition modules of a system. In order to modularize such crosscutting concerns AOP introduces a new abstraction named *aspect*. AspectJ [Kiczales et al. 2001] was a pionneer aspect-oriented language that introduced a set of concepts: *joinpoints, advice, pointcuts* and *intertype declarations*. The main problem of traditional AOP approaches following AspectJ is that they promote a new modularity mechanism but the internal aspect code contains a direct reference to the element that the aspect crosscuts. The first drawback of this approach is that it limits the aspect reuse in different context. As the aspect internal code is tightly associated with the element that it crosscuts, it cannot be reused in other context, with other elements. The second drawback, also related to the first one, is that the way that the aspect composes with other elements

is also fixed within the aspect code. Ideally, an aspect should compose with different elements, in a different way. This heterogeneous composition ability is essential to promote aspect reusability. Reusability can be defined as the ability to reuse the aspect behaviour (advice) in different compositions. Heterogeneity is the ability using the aspect behaviour in a different way for each composition where the aspect is applied.

AspectLua [Cacho et al. 2005a] is an AOP language based on the Lua programming language [Ierusalimschy 2006]. Lua is an interpreted and dynamically typed language. These features introduce a different style for aspect-oriented programming where dynamism is a key issue, weaving is done at runtime and both the basic elements and aspects can be inserted into and removed from the application at runtime. In addition, the Lua philosophy is to be simple and small and AspectLua keeps this philosophy. AspectLua is built upon a meta-object protocol, LuaMOP [Fernandes and Batista 2004] that provides an abstraction over the reflective features of Lua and allows application methods and variables to be affected by the aspect definition. AspectLua follows AspectJ concepts and, as a consequence, presents the aforementioned reuse limitations.

In order to promote the reuse of aspects and the ability to support heterogeneous composition, in this work we present a new version of AspectLua, named RE-AspectLua. In RE-AspectLua aspects are defined at *aspect specification time* and their instantiation is defined later at *application composition time*. An aspect is defined by a set of *aspect interfaces*. Each aspect interface specifies an abstract join point and an advice. Abstract join points instantiation is defined at application composition time via a *connection language*. The connection language is the Lua language itself since it is a scripting language. The fact of using a scripting language as the connection language introduces a great flexibility to the aspect composition process as the glue code can contain conditional statements to decide which composition must be defined. In addition, as Lua is a dynamically typed and interpreted language, it allows the dynamic connection and even the adaptation of the aspect connection at runtime.

Some recent work also address some limitations of the AspectJ-based AOP [Aracic et al. 2005], [Gal et al. 2003], [McDirmid and Hsieh 2003], [Suvée et al. 2005], [Suvée et al. 2003]. However, none of them is based on an interpreted language and provides the set of AOP features supported by AspectLua (see [Cacho et al. 2005a]). The related work in terms of AOP languages built on top of scripting languages [Dechow 2004], [Bryant and Feldt 2002], [Hirschfeld 2003], are based on AspectJ concepts and none of them promotes reusability of aspects that is the main goal of RE-AspectLua.

Although some researchers do not associate the use of AOP with scripting languages because, in general, such languages are not intended to write large and complex software systems, we argue that the benefits of AOP target not only large and complex software systems but it also has an important role in embedded systems where the problem of composition is even harder. This type of system needs to maintain the application code small. Thus, separation of concerns is essential and AOP is a good technique to manage crosscutting concerns in embedded systems [Sztipanovits and Karsai 2002].

This work is organized as follows. Section 2 presents Lua, AspectLua and LuaMOP. Section 3 presents our proposal to support reuse and heterogeneous interaction in AOP and how they are included in RE-AspectLua. Section 4 presents a case study that

illustrates the concepts of RE-AspectLua. Section 5 discusses related work. Section 6 contains the final remarks.

2. AspectLua and LuaMOP

AspectLua and LuaMOP are presented in this section. Both are based on Lua programming language [Ierusalimschy et al. 1996].

Section 2.1 presents the main concepts of AspectLua and section 2.2 presents LuaMOP, the meta object infrastructure used by AspectLua.

2.1. AspectLua

AspectLua [Cacho et al. 2005a] is an aspect-oriented programming language based on the Lua [Ierusalimschy et al. 1996] language. AspectLua implementation uses LuaMOP [Fernandes 2004], a meta-object protocol that allows the dynamic weaving of aspects and components. Lua is dynamically typed, which means that variables are not bound to types but each value has an associated type. Lua syntax and control structures are similar to those of Pascal. Some non-conventional features of Lua: (i) functions are first-class values; (ii) Lua tables are the main data structuring facility in Lua. Tables implement associative arrays, are dynamically created objects, and can be indexed by any value in the language, except nil. Tables may grow dynamically. Lua offers reflective facilities and metatables are the main reflective abstraction in Lua. Metatables allow modification of the behavior of a table. More details about Lua can be found in [Ierusalimschy et al. 1996].

AspectLua combines a set of features to make AOP easier and powerful: (i) insertion and removal of aspects at runtime, (ii) the definition of precedence order among aspects, (iii) the possibility of using wildcards, (iv) the possibility of associating aspect with undeclared elements (*anticipated join points*), and (v) a dynamic weaving process via a meta-object protocol.

AspectLua follows AspectJ philosophy and, as a consequence, presents the same problems related to lack of reusing support. In particular, AspectLua supports set of join points: *call* for method invocations; *callone* for the specification of aspects that must be executed only once; *introduction* to add functions in objects (intertype declarations); and the *get* and *set* join points to capture operations on variables. AspectLua aspect also defines an advice to be executed when the set of join points specified is reached.

```
1  a = Aspect:new()
2  a:aspect( {name = 'aspect-name'},
3            {pointcutname = 'pointcut-name',
4             designator = 'designator-type',
5             list = {'some-class.method1()','another-class.*'},
6            {type ='advice-type', action = advice} )
7
8  function advice()
9    -- do something
10 end
```

Figure 1. – Generic code for aspect creation

Figure 1 illustrates the syntax of aspects definition in AspectLua:

- the first aspect parameter is its name.

- the second parameter is a Lua table that defines the elements of the join points: its name, its designator type (*call*,*callone*, *introduction*, *get* or *set*), and functions or variables that must be intercepted. The *designator* field indicates the join point type. The *list* field contains the functions or variables that will be intercepted. The '*' wildcard can be used. For instance, *another-class.** indicates that the aspect must be applied to all methods of a class *another-class*.
- the third parameter is a Lua table that defines the elements of the advice: the type (*after*, *before* or *around*), and the action that is executed when a join point is reached. In the example, the action is the function declared with the name *advice* (line 6).

As the aspect defines an explicit association with the affected components, the aspect reuse is not possible. Also, the heterogeneous composition is limited in AspectLua. To illustrate the lack of aspect reusing in AspectLua, let us consider a simple banking application. Suppose that a client wishes to register the access to the bank component (Figure 2). It has two operations: *deposit* and *cash*.

```
1  Bank = {balance = 0}
2  function Bank:deposit(amount)
3      self.balance = self.balance + amount
4  end
5  function Bank:cash(amount)
6      self.balance = self.balance - amount
7  end
```

Figure 2. – Bank component

Figure 3 shows two aspects, written with AspectLua, that affect the bank component. These two aspects are needed to log methods of the *Bank* component. *laspect_before* aspect (lines 1-6) determines that the *cash* method is preceded by the *logbalance* advice. The second aspect (lines 8-13) is used to advise *deposit* and *cash* just after they are called.

```
1  a = Aspect:new()
2  a:aspect( {name = 'laspect_before'},
3            {pointcutname = 'logged_methods',
4             designator = 'call',
5             list = {'Bank.cash'},
6            {type ='before', action = logbalance} )
7
8  b = Aspect:new()
9  b:aspect( {name = 'laspect_after'},
10           {pointcutname = 'logged_methods',
11            designator = 'call',
12            list = {'Bank.cash','Bank.deposit'},
13           {type ='after', action = logbalance} )
14
15
16 function logbalance(self)
17     print ('Balance is now: ', self.balance)
18 end
```

Figure 3. – Bank component logging

There is a clear lack of reuse illustrated by the example in figure 3. The aspect code is directly bound to its pointcuts, thus the aspect must be directly changed if one wishes to affect other joinpoints, and the heterogeneous interaction, i.e. aspect acting in a different way (before and after), can be achieved only by code duplication.

AspectLua provides the concept of *anticipated joinpoints*. Anticipated joinpoints are join points related to elements that do not exist at the *aspect specification time*. This joinpoint is useful to avoid the need of loading an application code before loading the aspect code that contains a joinpoint to the application. This facility is useful to allow lazy loading at the *aspect execution time*. More details can be found in [Cacho et al. 2005a].

2.2. LuaMOP

The Aspect weaving process used in AspectLua is provided by LuaMOP [Cacho et al. 2005b]. LuaMOP is a meta-object protocol that supports the creation of a meta-representation to each element that composes the Lua runtime environment: variables, functions, tables, userdata and so on. Each element is represented by a meta-class that provides a set of methods to query and to modify the behavior of each element of the base class. They are organized in a hierarchical way where *MetaObject* is the base meta-class (Figure 4). Derived from this meta-class are *MetaVariables*, *MetaFunctions*, *MetaCoroutine*, *MetaTable*, and *MetaUserData* meta-class. Furthermore, LuaMOP also provides a *Monitor* class to monitor the occurrence of events in the Lua runtime environment.

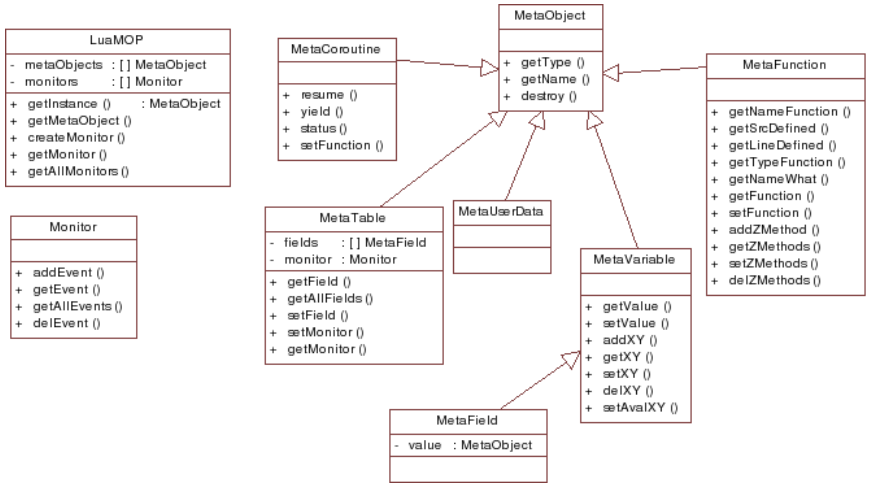


Figure 4. – LuaMOP class diagram.

The meta-representation provided by LuaMOP is created via the invocation of the *getInstance(instance)* method. This method returns the meta-object corresponding to the object with name or reference described by the instance parameter. This meta-object is an instance of a meta-class described above. For each meta-class there are methods that describe it and that support changing the behavior of a meta-object. Thus, *getType()* and *getName()* methods can be invoked by all meta-classes, since these methods are part of the MetaObject meta-class. These methods return, respectively, the meta-object type and name. The *destroy()* method is used to disconnect the meta-object from the base object and to destroy the meta-object. The *getInstance()* method can also be invoked, using as an input parameter a non-determined name. For instance: *getInstance("string.*")* returns a Lua table with meta-objects that represent the functions of the string package.

For the sake of brevity, only some *MetaFunction* methods are commented here, an extensive explanation of LuaMOP can be found in [Cacho et al. 2005b]. The *MetaFunction* meta-class offers the *addPreMethod*, *addPosMethod*, and *addWrapMethod* methods.

These methods define the place where the behavior is added: Pre(before), Pos(after), and wrap the execution of a function. An example of the use of these functions is illustrated in figure 5.

```
1  function reglog(self,value)
2      print("Deposited Value: ",value)
3  end
4  metafun = LuaMOP:getInstance("Account.deposit")
5  metafun:addPosMethod(reglog)
6  Account:deposit(10)
```

Figure 5. – LuaMOP example with addPosMethod [Cacho et al. 2005b]

The meta-object is obtained at line 4. At line 5, the *addPosMethod* method is invoked to add the *reglog* function defined from lines 1 to 3. When the deposit method is executed (line 6), the LuaMOP mechanisms automatically invoke the *reglog* method.

LuaMOP functionality goes beyond the provision of a meta-representation. It can also capture events from the runtime execution environment. A *Monitor* is created to handle events related to elements that have not yet been declared in the application. This facility is used to support the anticipated join points strategy (see in [Cacho et al. 2005b]).

The integration of these properties in an aspect environment is straightforward. For instance, the definition of the advice mechanism for a *call* join point is illustrated in figure 6.

```
1  function AspectDefinition:defineCall(id)
2      local aspect = AspectDefinition.aspectList[id]
3      local metaobject = LuaMOP:getInstance(aspect.pointcut.list)
4      if (aspect.advice.type == 'before') then
5          metaobject:addPreMethod (aspect.advice.action)
6      elseif (aspect.advice.type =='around') then
7          metaobject:addWrapMethod (aspect.advice.action)
8      else
9          metaobject:addPosMethod (aspect.advice.action)
10     end
11     aspect.mob = metaobject
12 end
```

Figure 6. – Aspect definition using LuaMOP features in AspectLua

For each aspect declared in AspectLua, it only suffices to define the advice method as a *pre*, *wrap*, or *pos* method to the metaobject representing the join point (figure 6, lines 4-10), depending on the type of aspect that can be *before*, *around* or *after* (figure 1, line 6).

3. Reusability and Heterogeneity Improvement in AOP

In this section we discuss the features of aspect-oriented programming languages needed to benefit aspect reuse, context independence and heterogeneous interaction and then we introduce RE-AspectLua, a solution for these problems written in the Lua language. Section 3.1 addresses the lack of reusability and heterogeneity in AOP languages, presents a solution, the concept of *aspect interfaces*, and discusses how a connection language may help in supporting reusability and heterogeneity in AOP languages. Section 3.2 presents how RE-AspectLua supports the features discussed in section 3.1 in order to promote aspect reuse and heterogeneous behavior.

3.1. Abstractions for Aspect Reuse

There are two essential features that aspect-oriented programming languages must include to benefit from aspect reuse, context independence and heterogeneous interaction:

Abstract Join Points. Abstract join points [Lieberherr et al. 1999] are declarations of join points that are not bound to the base element at specification time. The definition of the real instance is done later, at application composition time. In this way, abstract join points specify generic and context independent aspects. Different applications can reuse the aspect and instantiate the abstract join points with different elements. Thus, this strategy promotes aspect reuse and allows the heterogeneous interaction of the aspect with different components. There is a lot of work [Aracic et al. 2005], [Suvée et al. 2003], [Herrmann and Mezini 2001] that uses this concept.

Connection Language. Connectors are commonly used in component-based languages to connect components in order to compose the final system [Szyperski 2002]. In a similar way, connection languages can be used to support the connection between aspects and components. Some works use connectors to yield component-aspect composition [McDirmid and Hsieh 2003, Suvée et al. 2003, Suvée et al. 2005]. The use of such a mechanism, associated to abstract join points, is an interesting solution to reduce dependence relationships between aspects and components and to improve the flexibility of the interaction between them.

In order to address context independence, we borrow the concept of *aspect interface* defined in [Chavez et al. 2005] to model aspects at architectural level. A connection language is used to define the instantiation of the join points and the definition of the advice activation time. Thus, the connection language, also named configuration language, makes it possible the independence of the aspect in relation to the affected components. In this way, in the specification, the aspect does not determine the code that it affects.

3.1.1. Aspect Interfaces

At the programming language level, we propose the concept of *aspect interface*. Aspect interfaces are contract definitions of aspectual functionalities established at specification time. The aspect interface defines **refinements**. Refinements contain (i) the definition of the abstract join point (the elements that will be affected by the aspect); (ii) the definition of action to be taken (the advice) when the join points are reached. This set of join points and the advice type (*before*, *after*, or *around*) are defined by the connection language at application composition time.

The aspect interface abstract syntax in BNF is illustrated in figure 7:

```
<AspectInterfaceDeclaration> ::=  <AspectInterfaceName> "=" <AspectInterfaceInstantiation>
<AspectInterfaceName> ::=  <identifier>
<AspectInterfaceInstantiation> ::= "AspectInterface:new()"
<RefinementDeclaration> ::=  <AspectInterfaceName> ":" <RefinementInstantiation>
<RefinementInstantiation> ::= "refinement( { name = '" <identifier> "'},"
                              "            { refine = '" <identifier> "',"
                              "              action = " <identifier> "})"
```

Figure 7. – BNF Syntax of aspect interfaces

The BNF syntax defines the declaration of aspect interfaces, and its refinements. An aspect interface has a identifier and is instantiated by the use of *AspectInterface:new()*. It may have one or more refinements. The refinements are declared for each aspect interface, and must define its name (*name*), the abstract pointcut name (*refine*) and the method to act as advice for the abstract join point (*action*)

3.1.2. Connection Language

In order to express the relationships between an aspect and base elements, the use of connectors, as we discussed in section 3, can give a proper support to promote the aspect independence in relation to the usage context.

Scripting languages have been used to support the interconnection of elements in component-based systems [Batista 2000]. It acts as a configuration language that defines the relationship between the components. Scripts can also be used to adapt the component interconnection when interfaces are not compatible and, in this case, are called *glue code*.

In a similar way, scripting languages also act as connection language between aspects and components. In this case, the scripting language must instantiate the abstract join points. It cannot break the component interface contract and it must define the precedence between aspects that act on the same join point. In addition, the inherent flexibility of most scripting languages allows the definition of complex relationships between aspects, such as (i) the conditional removal of an aspect behavior in the presence of other aspect, (ii) higher semantics by enabling complex aspect protocols.

3.2. RE-AspectLua

RE-AspectLua includes the abstractions aforementioned. An aspect in RE-AspectLua is a component that contains a set of aspect interfaces (*AspectInterface*).An aspect interface can have one or more definitions of refinements. Refinements are declarations of abstract join points and advice (figure 8).

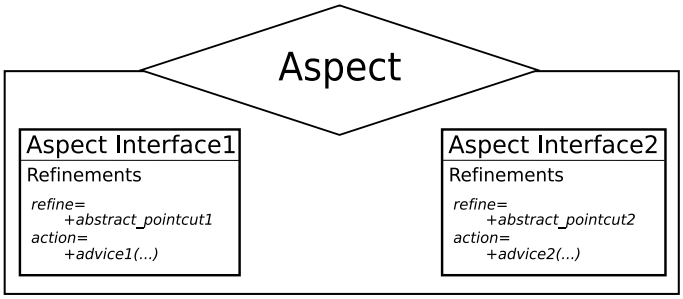


Figure 8. Generic Aspect model

In RE-AspectLua, common aspects are also called **aspectual components** by acting like a component with aspectual behaviour.

3.2.1. Aspect Definition

Figure 9 shows an example with two aspect interfaces declared: *ai1* and *ai2*. The *ai1* interface (line 4-7) defines a refinement that declares a behavior (*advice1* method) to be executed when a set of abstract join points *abstractpointA* is reached. The moment when the method will be executed (*before*, *after* or *around*), and the exact join points that the *refine* declaration represents are defined at application composition time, using a connection language. The *ai2* interface (lines 9-12) is similar to the *ai1* interface. However, it defines another behavior to be executed when the set of abstract join points *abstractpointB* is reached.

```
1  aspectA = Aspect:new( {name = "Aspect A"} )
2  aspectB = Aspect:new( {name = "Aspect B"} )
3
4  ai1 = AspectInterface:new()
5  ai1:refinement( {name = 'interface1'},
6                  {refine = 'abstractpointA',
7                   action = advice1} )
8
9  ai2 = AspectInterface:new()
10 ai2:refinement( {name = 'interface2'},
11                 {refine = 'abstractpointB',
12                  action = advice2} )
13
14 aspectA:interface(ai1)
15 aspectA:interface(ai2)
16
17 aspectB:interface(ai1)
```

Figure 9. – Aspect definition in RE-AspectLua

In RE-AspectLua, the aspectual components can share definitions of aspect interfaces. In the example of figure 9, the *aspectA* aspect has two interfaces, while *aspectB* has just the first interface declared. The two aspects have the same aspect interface but interact in a different way with the application. This illustrates the heterogeneity interaction ability of RE-AspectLua.

RE-AspectLua offers "syntax sugar" to reduce the code needed to define an aspect interface and to associate it to an aspect. Figure 10 presents the syntax sugar used to define aspect interfaces and an aspect.

```
1  ai1 = AspectInterface:new_refinement( {name = 'interface1'},
2                                        {refine = 'abstractPointA',
3                                         action = advice1} )
4
5  ai2 = AspectInterface:new_refinement( {name = 'interface2'},
6                                        {refine = 'abstractPointB',
7                                         action = advice2} )
8
9  aspectA = aspect ({ai1, ai2})
```

Figure 10. – Syntax sugar for the definition of aspects in RE-AspectLua

Figure 10 (lines 1-3) illustrates the creation of the same *interface1* of figure 9. The *aspectA* aspect is created containing the two interfaces *ai1* and *ai2* (line 9). In this case, the aspect name is not explicitly defined.

3.2.2. Aspect Instantiation

The interaction and precedence relationships of the aspects are defined also via scripts of the connection language. In RE-AspectLua, the language used to connect the elements is the Lua language itself with a library that allows the instantiation of join points and the definition of the relationships between aspects. This library offers the *get_interface* method, to dynamically get an aspect interface by its name, and the *bind_refinement* method to bind a refinement by instantiating concrete join points for it.

Figures 11 and 12 illustrate the definition of the aspectual connection to the aspects defined in figure 9.

```
1  int1 = aspectA:get_interface{'interface1'}
2  int1.abstractpointA =
3    {designator = 'call',
4     pointcut-list = {'some-class:method1()'},
5     type = 'before'}
6
7  int2 = aspectA:get_interface{'interface2'}
8  int2.abstractpointB =
9    {designator = 'callone',
10    pointcut-list = {'another-class.*'},
11    type = 'after'}
```

Figure 11. – Instantiating the interfaces of *aspectA*

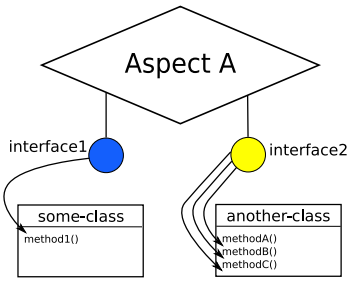


Figure 12. AspectA model

Figure 11 shows how the connection language is used to declare the instantiation of *aspectA*'s aspect interfaces. First, the *ai1* aspect interface is retrieved by a *get_interface()* call on *aspectA* (line 1). Then the *abstractpointA* refinement has its instantiation defined: the *advice1* method (figure 9, line 7) will be executed just before *some-class:method1()* call. In this language it is possible to quantify the join points using the wildcard '*' as can be seen in the declaration of the join points of the *abstractpointB* refinement (figure 11, line 8).

```
1  int3 = aspectB:
2     get_interface{'interface1'}
3  int3.abstractpointA =
4    {designator = 'execution',
5     pointcut-list = 'third_class.*',
6     type = 'before'}
```

Figure 13. – Instantiating the interface of *aspectB*

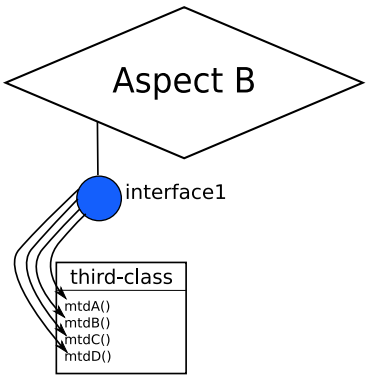


Figure 14. AspectB model

Another crosscutting relationship is illustrated in figures 13 and 14. The *interface1* aspect interface in *aspectB* has a set of join points that intercepts the executions of all methods of the *third-class* component and executes the *advice1* advice (defined in the aspect interface *ai1* at figure 9, line 7).

4. Case Study

A simple banking application is shown to illustrate advices in RE-AspectLua. AspectLua does not offer a suitable solution for this case unless by some code duplication. This section shows how RE-AspectLua handles it.

Figure 15 contains a RE-AspectLua generic aspect code for logging bank transactions. The *logaspect* aspect is declared at line 3. The *logged_methods* refinement is associated to the *logging* aspect interface at lines 5-7; and the interface is linked to the *logaspect*. The advice code is at lines 10-12.

```
1  require 'REAspectLua'
2
3  logaspect = Aspect:new( {name = "logaspect"} )
4  aint = AspectInterface:new ()
5  aint:refinement ( {name = 'logging'},
6                    {refine = 'logged_methods',
7                     action = logbalance} )
8  logaspect:interface(aint)
9
10 function logbalance(self)
11     print ('Balance is now: ', self.balance)
12 end
```

Figure 15. – RE-AspectLua aspect declaration

In order to instantiate the *logged_methods* refinement, a connection script is needed. Figure 17 shows multiple refinement instantiation. First, the aspect and the aspect interfaces are retrieved by their names (lines 1-2). Then, the refinement is instantiated (or bound) 3 times. The *logbalance* advice is executed after the *deposit* method of the *Bank* component (lines 4-6); and before and after the invocation of the *cash* method.

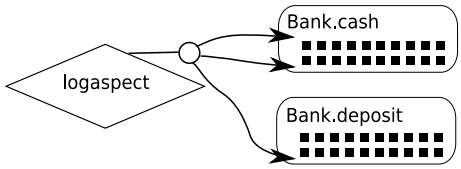


Figure 16. – Instantiating the interface of logaspect

```
1  laspect = Aspect:get("logaspect")
2  log_int = laspect:get_interface('logging')
3
4  log_int:bind_refinement("logged_methods", { designator = 'call',
5                                              pointcut_list = {'Bank.deposit'},
6                                              type = 'after'})
7  log_int:bind_refinement("logged_methods", { designator = 'call',
8                                              pointcut_list = {'Bank.cash'},
9                                              type = 'after'})
10 log_int:bind_refinement("logged_methods", { designator = 'call',
11                                              pointcut_list = {'Bank.cash'},
12                                              type = 'before'})
```

Figure 17. – Instantiation script

The pure AspectLua implementation of this functionality would need more than one aspect due to lack of reuse and heterogeneity abilities of AspectLua (discussion in section 2.1, figure 3).

5. Related Work

AspectLagoona [Gal et al. 2003] defines aspects as component-like modules containing advices. The pointcut language is the method definition in component interface. The aspect is defined for component interface, not component implementation, crosscutting all implemetations of interface. This way, there's low independence of aspect from the component, leading to limited reuse of aspect. RE-AspectLua allows context independence, providing more reusable aspects than AspectLagoona.

Jiazzi [McDirmid and Hsieh 2003] supports component developments for Java. Components are linked by a connection language. The connection language doesn't allow advice-like behaviour, although permits that some crosscutting concerns are better modularized by the use of open classes and open signatures. New code statements and a post-compilation phase are needed in Jiazzi. RE-AspectLua also defines aspect interaction by using a connection language, but does not require any further step to bind aspect behaviour into component code.

JAsCo [Suvée et al. 2003] provides aspects in JavaBeans component model. Aspect beans are defined containing advice associated to abstract pointcuts. Special Connector entities instantiate the abstract pointcuts, allowing high levels of reuse. RE-AspectLua is inspired on JAsCo, by providing abstract pointcuts to be instantiated in later phase. As RE-AspectLua uses Lua as connection language, dynamic advice implementation is simpler than in JAsCo, where all JavaBeans must be traced in the execution environment.

FuseJ [Suvée et al. 2005] proposes a new component model where all components can be aspects and all the crosscutting is done by connection language. This approach restricts application domains of language, since all components must follow the component FuseJ component model.

CaesarJ [Aracic et al. 2005] integrates components and aspects with mixin composition of family classes. This way, CaesarJ doesn't use connection language to link aspect-components. A wrapper mechanism is used to adhere crosscutting structure and behavior to components of different family classes. RE-AspectLua does not use wrapper mechanism, but refinements in aspect interface, to link crosscutting behavior to base components.

LAC – Lua Aspectual Components [Herrmann and Mezini 2001] – is a Lua extension whose main goal is to support the idea of Aspectual Components (AC) [Lieberherr et al. 1999]. It defines an explicit connector module to bind the participants. In contrast, RE-AspectLua is more flexible as it uses the Lua scripting capabilities in the definition of the connection between aspects and base elements. This flexibility allows the definition of complex aspect composition.

6. Final Remarks

The abstractions to the modularization of the crosscutting concerns at traditional aspect-oriented languages are limited in terms of reusability and heterogeneous interaction capability. In this paper we use the concepts of aspect interface and connection language, commonly used in component-based systems, in order to promote aspects reuse. We propose the use of a scripting language, the Lua language, to the definition of the interaction between aspects and base elements. Although the concept of *abstract join points* is com-

monly found in some works, we use this concept in conjunction with a dynamically typed and interpreted connection language, that adds a great deal of flexibility to the composition of the reusable aspect and the base code. AspectJ supports the concept of abstract pointcuts. However, the flexibility provided by usign abstract pointcuts with a connection language is not offered by AspectJ.

The concepts of abstract join points, aspect interface and connection language are instantiated in the definition of a new version of AspectLua, RE-AspectLua, that promotes reusability, context independence and a better organization of the heterogeneous interaction between aspects and base elements. Future work include the use of RE-AspectLua in a more complex application to explore the aspect reuse in many heterogeneous ways.

We also presented a case study that show that RE-AspectLua enriches the modularization approach of AOP to allow a more effective reuse and heterogeneous interaction.

7. Acknowledgements

This work has been partially supported by Brazilian National Agency of Petroleum under grant No.2001.7999-5 for Mauricio B. C. Vieira.

References

Aracic, I., Gasiunas, V., Mezini, M., and Ostermann, K. (2005). An Overview of CaesarJ. Technical Report TUD-ST-2005-01, Darmstadt University of Technology, Darmstadt, Germany.

Batista, T. V. (2000). *LuaSpace: um Ambiente para Reconfiguração Dinâmica de Aplicaçõe Baseadas em Componentes*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro.

Bryant, A. and Feldt, R. (2002). Aspectr - simple aspect-oriented programming in ruby. `http://aspectr.sf.net/`. Last visualization in March 2007.

Cacho, N., Batista, T., and Fernandes, F. (2005a). Aspectlua - A dynamic AOP approach. *Journal of Universal Computer Science (J.UCS)*, 11(7):1177–1197.

Cacho, N., Fernandes, F., and Batista, T. (2005b). Handling Dynamic Aspects in Lua. In *SBLP2005: Proceedings of XIX Brazilian Symposium on Programming Languages*, pages 76–89, Recife, Brasil.

Chavez, C. V. F. G., Garcia, A., Kuleska, U., Sant'Anna, C., and Lucena, C. (2005). Taming Heterogeneous Aspects with Crosscutting Interfaces. In *SBES2005: Proceedings of XIX Brazilian Symposium on Software Engineering*, pages 216–231, Uberlândia, Brasil.

Dechow, D. R. (2004). Advanced separation of concerns for dynamic, lightweight languages. In *5th Generative Programming and Component Engineering*.

Fernandes, F. (2004). Combinando Aspectos e Componentes: uma abordagem interpretada. Master's thesis, Universidade Federal do Rio Grande do Norte, Natal.

Fernandes, F. and Batista, T. (2004). A Dynamic Approach to Combine Components and Aspects. In *SBES2004: Proceedings of XVIII Brazilian Symposium on Software Engineering*, pages 102–112, Brasília, Brazil.

Gal, A., Franz, M., and Beuche, D. (2003). Learning from components: Fitting AOP for system software. In *Proceedings of the AOSD 2003 Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Boston, MA, USA.

Herrmann, S. and Mezini, M. (2001). Combining Composition Styles in the Evolvable Language LAC. In *Workshop on Advanced Separation of Concerns in Software Engineering at 23rd ICSE*.

Hirschfeld, R. (2003). AspectS - aspect-oriented programming with squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK. Springer-Verlag.

Ierusalimschy, R. (2006). *Programming in Lua*. Lua.org, Rio de Janeiro, BR, second edition.

Ierusalimschy, R., de Figueiredo, L. H., and Filho, W. C. (1996). Lua an extensible extension language. *Software: Practice and Experience*, 26(6):635–652.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK. Springer-Verlag.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York.

Lieberherr, K., Lorenz, D., and Mezini, M. (1999). Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA.

McDirmid, S. and Hsieh, W. C. (2003). Aspect-oriented programming with Jiazzi. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 70–79, New York, NY, USA. ACM Press.

Suvée, D., Vanderperren, W., and Jonckers, V. (2003). Towards a symbiosis between aspect-oriented and component-based software development. In *Proceedings of the SCI 2003 international conference*, pages 442–447, Orlando, USA.

Suvée, D., Vanderperren, W., Wagelaar, D., and Joncker, V. (2005). There are no Aspects. In *Electronic Notes in Theoretical Computer Science, Proceedings of the Software Composition Workshop (SC 2004)*, volume 114, pages 153–174. Elsevier Science.

Sztipanovits, J. and Karsai, G. (2002). Generative programming for embedded systems. In *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 180–180, New York, NY, USA. ACM Press.

Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Programming through Spreadsheets and Tabular Abstractions

Carlos Henrique Q. Forster

Instituto Tecnológico de Aeronáutica
Divisão de Ciência de Computação
Pça. Mal. Eduardo Gomes, 50 Vila das Acácias
12228-900 – São José dos Campos – SP – Brazil
forster@ita.br

Abstract. *The spreadsheet metaphor has, over the years, proved itself valuable for the definition and use of computations by non-programmers. However, the computation model adopted in commercial spreadsheets is still limited to non-recursive computations and lacks abstraction mechanisms that would provide modularization and better reuse (beyond copy and paste). We investigate these problems by identifying a minimal set of requirements for recursive computations, designing a spreadsheet-based language with an abstraction definition mechanism, prototyping an interpreter and evaluating it with examples.*

Keywords. *Spreadsheet languages, end-user programming, call-by-need, lazy structures, recursion, abstraction.*

1. Introduction

Spreadsheets are a popular document model for structuring information and interacting with data. With spreadsheets a user can also define small computations with data that allow for instance statistical summarization, simulation and information highlighting provided that this user is able to express these computations in a formula language. Spreadsheet systems have also been used as a data entry interface for other software.

While the textual paradigm of programming daunts the general user, we can see non-programmers using spreadsheets and defining simple computations expressed through spreadsheet formulae. This can be a good starting point for learning to program. A spreadsheet user will perceive desired computations that he is unable to describe in a spreadsheet and feel the urge to move to a textual programming language. It is expected that enhancements on the expressive power of spreadsheets to allow advanced computations would fill the gap between spreadsheet and textual programming.

One criticism about current commercial spreadsheets is their lack of abstraction mechanisms. Users are limited to a set of predefined functions. Modularization is simulated by copy-and-paste of template spreadsheets. This makes spreadsheets hard to maintain. Code replication forces the users to keep track of all copies in order to update them properly when new information arrives or a bug is found. Proper reuse mechanism would allow tested and trustful components to be safely adopted and easily replaced. Additionally, some features can be implemented through spreadsheets but in a very cumbersome way. Long expressions referring to many cells are truly hard to decipher and maintain. Having means to define abstract operations and creating instances of

these operations as needed would make complex spreadsheets more readable just like subroutine calls make programs more readable.

Another criticism is the lack of scalability. Although the supposition of infinite-sized problems can strike us as superfluous, recursive computations are important when we deal with abstractions. Most of the times when the problem is bigger than the spreadsheet were prepared to handle, we need just to insert a number of columns or rows and get a new spreadsheet ready to do the calculation. In the case of abstract spreadsheet definition, nothing should be assumed about the size of the problem, we just need one abstraction to be able to solve the problem and we cannot manually create another spreadsheet instance for each "subroutine call" because the number of instances can grow rapidly.

This paper investigates the problem of defining recursive and abstract computations in the tabular format, preserving important properties such as the spatial structuring of information and referential transparency. Usually an imperative scripting language is included in a spreadsheet system to add support to advanced computations. We intend to follow the reverse path, allowing complex computations to be defined in spreadsheet format and turning the spreadsheet into an environment for scripting application behavior or connecting software components.

We point some application areas for spreadsheet programming. Spreadsheet programming can be used for the reconfiguration of software applications through defining small computations or data flow among software components. Particularly, Information Visualization is an area that would benefit from the reconfiguration capability and the representation of data and data transformation in the tabular format. Also, in Education, teaching of Math-related subjects could benefit from end-user definition of computations [Misner and Cooney, 1991].

The paper is organized as follows. Section 2 of the paper revisits many concepts on which spreadsheets are based and points to a few references from the literature. The respective design decisions for the proposed system are presented in section 3. The development of our contribution starts with the identification of a set of primitive constructs that allow a spreadsheet language to represent recursive computations (section 4). A model of abstraction definition in the tabular format for the spreadsheet language is presented (section 5). The prototype implementation of an interpreter for the defined language and a spreadsheet editor are described (section 6). The prototype system is evaluated by implementing some examples (section 7). In section 8, we provide a general conclusion of the present work and point future directions.

2. Design Issues Peculiar to Spreadsheet Languages

The design of a language for spreadsheets is quite different from the design of a general programming language being the main differences, the presence of a visual editor and the requirement for fast updates for interactive computing. Another important difference is the organization of information in a tabular structure. We describe in this section some design issues that are considered in our design.

Coordinate-based references are the foundation of the spatial organization of spreadsheets, but imply some design difficulties. On one hand, coordinates can be easily entered through point-and-click in the editor interface. Additionally, coordinate-based formulae can be copied, moved and extended (copied to a larger area) maintaining

appropriate relative references to, for instance, “the same line”, “the line above” or “two columns to the right”. On the other hand, formulae based on coordinate mnemonics are very difficult to read. Consequently, programs are difficult to write or maintain. Verifying correctness of choice between absolute, relative or mixed-coordinate references is pretty difficult. In Visicalc-based spreadsheets [Bricklin and Frankston, 1999], absolute coordinates represented such as $A$1, when copied or extended are kept as $A$1. Relative coordinates like A1, when extended to the right becomes B1, when copied to the cell immediately below becomes A2. Mixed coordinates are represented like A$1 or $A1.

The semantics of cell copy, movement and extension operations is a source of confusion. The behavior of those operations can vary concerning how cell references are maintained. Some systems may update cells that used to point to a previous location of another cell. However, copying would be handled differently. In some editors, movement keeps track of which cells refer to the cell being moved. It is confusing because it can modify something very far away from the area where the user is changing. Lisper and Malmström [2002] point at the problem of concatenation of tables on the same page. If we concatenate two tables side-by-side, the insertion of a new row to the first table would spoil the alignment of the second table. In order to cope with row or column insertion concatenation of separate tables must be diagonal, what is logically correct but functionally and visually inappropriate. This is a good clue that data and programs must be organized beyond two-coordinate axes. A data structure holding multiple tables should be necessary.

Another way to reference cells considers cell names instead of coordinates. The use of names is recommended because references become more readable and provide a form of access similar to tuple structures. Systems based only on names, like dataflow systems, unfortunately rarely make use of the tabular spatial structure and can easily become cluttered both in name space and canvas space. Notable exceptions include Forms/3 [Burnett *et al.*, 2001] and Haxcel [Lisper and Malmström, 2002].

While, for the textual programming languages, there is the read-eval-print loop of an interpreter or the compile-eval-print loop of compiler systems, spreadsheets would have a definition-eval-display loop. Traditional spreadsheet systems would evaluate an expression and all of its dependencies on any change. A **call-by-need spreadsheet system** would evaluate only formulae that are directly or indirectly needed for the current visualization. The formula language of a spreadsheet doesn’t have the attribution operation. The absence of side-effects due to attribution provides the property of referential transparency. This is the property that two identical expressions yield identical values. The preservation of this property along with call-by-need will allow lazy-evaluation, mechanism of evaluation that goes beyond call-by-need by reusing computations so that identical expressions should not be evaluated twice. (See for example [Harrison and Field, 1988]).

Recursive formulae through the means of cycles are not generally allowed. A spreadsheet system may permit the definition of cyclic dependent cell formulae. This will constitute an equation system or a constraint system, which may be solved by a constraint solver or through relaxation, where each cell is computed multiple times until some convergence criterion is satisfied or recognized as unreachable.

Some spreadsheet systems allow data structures to be included inside cells or to include spreadsheets as members of data structures such as a folders (list with hierarchy) or trees. Complex objects such as graphics or images are allowed as cell values in some spreadsheets such as image spreadsheets [Levoy, 1994] or visualization spreadsheets [Chi *et al*., 1998] [Nuñez, 2002].

3. Design Decisions

We describe now our design decisions. In order to keep semantics as clear and simple as possible, we recommend using **coordinate-awareness** providing functions namely row() and col(). Such functions allow the formula of cell to know the coordinates of that cell. We propose that relative cell references be built upon these functions. Copying a formula from one cell to another is the correct cell copy operation. Extension of a cell to an area is implemented simply as copying the formula of the cell to each cell in the area. Alternatively, names are also allowed as absolute references.

Our model of spreadsheet editing and computation assumes that cells only accept formulae as input. We assume also that spreadsheet formulae cannot modify cells, only the editor is able to modify the formula of a cell. We define an **editor** as the only type of entity able to modify cells so that interactive computation should be obtained through the mediation of an editor. Complex objects may be represented or abstracted as spreadsheet data and it is the role of an editor to provide the correct visualization for this data.

We consider that evaluation of complex computations may not be deliverable in time, so our proposed editor has two modes: a formula-viewing mode and a value-viewing mode. Formulae are evaluated only when their values must be displayed (or are referenced by some other cell or evaluation chain whose value will be displayed). In formula-viewing mode, values are not displayed and then evaluation does not happen. This behavior of performing only the presently needed computations corresponds to the desired call-by-need behavior. Sharing the evaluation of expressions is implemented through caching of computed values of cells while intermediate values passed as function arguments are not cached and their resources are freed by a garbage collector. This behavior approaches lazy evaluation that could be properly obtained if graph rewriting was implemented, which was used for example by de Hoon [1995].

Spreadsheets are first-class objects and can be contained in spreadsheet cells. The cell reference operation is extended to provide access to cells of **inner spreadsheets** and **outer spreadsheets**. The coordinate-awareness functions can provide not only the coordinates of the cell being evaluated, but the location where its spreadsheet is stored and so on until a **top-level spreadsheet** is reached.

At the moment, cyclical references are not allowed and, when detected, are reported with an error symbol being delivered as the value of the cell. We provide other forms to describe recursive computations that we find more appropriate and are described in the next sections.

4. A Sufficient Model of Computation through Spreadsheets

We describe a computational model for spreadsheets. This model was created based on an experience of implementing a Turing Machine in a spreadsheet language. Therefore, the constructed model is expressive enough to represent any computable function.

However, this sufficiency does not help to make program development easy. Readability of code, clear intuitive semantics and other human factors should be considered in the design of a spreadsheet language.

Coordinate cell references are important to exploit the spatial structure of information. Lets make all cell references through the use of a function ref(i,j) where i and j are respectively row and column numbers. So an absolute reference such as $B$1 is written as ref(1,2). An editor can adopt a syntax sugar of seeing or accepting ref(1,2) as $B$1. Now we consider coordinate-awareness constants i and j for which row and column numbers are automatically assigned. A cell self-reference would be ref(i,j), while a reference to the immediately above cell would be ref(i-1,j) and the cell immediately to the right is ref(i,j+1). If current cell is C5, ref(i,j+1) can be displayed by the editor as D5. If we enter D5, the editor converts it to ref(i,j+1). If we enter C$4, it is converted to ref(4,j). When considering copying a cell with this kind of representation, there is nothing to change in the formula. The same happens when extending and moving. The semantics is then very clear and closer to traditional textual programming languages.

Our view of this kind of representation is to understand all cells as functions. While a spreadsheet S is a function mapping a (row,column) pair into a cell, a cell C is a function mapping (row,column) into a value. So, a cell with formula ref(i,j+1) holds indeed a function λij.ref(i,j+1) in lambda-calculus notation. The value of this cell is obtained applying this function to the cell coordinates such as λij.ref(i,j+1) row column. Approaching cell definition as a function of its location allows the preservation of referential transparency. Complaints that relative references would damage referential transparency are, therefore, not effective for this design.

Indirection is also accomplishable and important for the implementation of conditionals. Conditionals need not to be a primitive if we have indirection such as ref(ref(1,1),ref(1,2)). Implementation of a conditional can be seen in table 1.

Table 1. Implementation of conditional.

ANS=IF(COND; THEN; ELSE) for example ANS=IF(A>B; A; B)		
	J=1	J=2
I=1	COND: A>B	ELSE: B
I=2	ANS: ref(ref(1,1)+1, 2)	THEN: A

In the considered case, the comparison operator > returns 0 for FALSE or 1 for TRUE. This value can be added to a row or column index and used to select the proper cell. Implementation of conditional with indirection is similar to parameter selection in lambda-calculus. In lambda calculus, TRUE and FALSE are combinators that select the first or the second parameter.

We opt to use two types of recursive definitions. The first one is the “ellipsis” **tabular recursion**; the second one is function-call recursion which is defined in section 5 under the name of tabular abstraction for which some examples are discussed in section 7. Tabular recursion allows construction of infinite tables and provides a means to specify induction. A three dots symbol (…) in a cell implies that this cell and all cells to the right of it have the same formula which is the formula of the cell immediately to the left of the current cell. The vertical counterpart is a three columns symbol (:::). A

diagonal ellipsis would be represented by ***. See, for example, the implementation of factorial and Fibonacci functions in table 2.

To illustrate that this model is sufficient to describe any computable function we implemented a Turing Machine, first in Excel, then in our prototype. In tables 3 and 4 there is an implementation of a TM that accepts the language $\{1^n2^n \mid n \geq 0\}$. The symbol 5 is the blank symbol for the TM tape. Each row of spreadsheet (a) contains an instantaneous description of the TM, the first element is the machine state, the second is the head position (column number) and the remaining elements are tape symbols.

A Turing Machine requires an infinite-length tape and may enter an endless loop, requiring that the spreadsheet that simulates it be infinite in both axes. Infinite-length spreadsheets can be defined by the use of the ellipsis operator. Due to the call-by-need evaluation scheme, this does not lead to infinite computations because only the values that are needed for calculation or presentation are evaluated. Our spreadsheets can be seen as lazy structures.

Although, in the Turing Machine example, all computations are represented by the infinite-length spreadsheet, this doesn't mean that the computations are carried out. We will only see a computed value if displayed by an editor, so we have to "scroll down" the spreadsheet until we see the TM stopped. If the TM enters an endless loop we would theoretically need to scroll down forever. From this observation, we conclude that a function to force the evaluation of cells is needed. We propose a "scan" function that will traverse the spreadsheet forcing evaluations until a criterion is met. In the case of the TM, this function will traverse the spreadsheet until the TM stops. If TM never stops, "scan" should also theoretically never stop.

5. A Model for Abstraction Definition

We propose an abstraction model very similar to the one in Forms/3 [Burnett *et al.*, 2001]. The abstraction scheme resembles the delegation mechanism used as inheritance in prototype-based languages such as JavaScript. Every spreadsheet is a candidate for abstraction. It happens by overriding the formulae of some cells of a template spreadsheet and collecting the results of computed cells of the new formed spreadsheet.

We call **tabular abstraction** the use of a spreadsheet or table as a function. Our liberal proposal permits that any cells of a template spreadsheet be overridden. The **override** of a spreadsheet consists of a spreadsheet with the substitution cells and a reference to the template spreadsheet whose cells are being overridden. The new spreadsheet formed this way will have cells whose formula will evaluate based on the other overridden cells. We use the following syntax: extend(template_spreadsheet, override_spreadsheet) which evaluates to a spreadsheet that can be kept inside a cell. A spreadsheet can extend itself using the ref() function without arguments. In terms of lambda-calculus, we create an abstraction defining which cells become parameters and already apply the abstraction to values and formulae that override the cells.

Table 2. Implementation of factorial and Fibonacci functions.

ANS=FACTORIAL(N)				
	J=1	J=2	J=3	J=4
I=1	1	mul(ref(row(),sub(col(),1)), col())	…	
I=2	N:	ANS: ref(1,ref(N))		

	J=1
I=1	1
I=2	1
I=3	add(ref(sub(row(),1),1),ref(sub(row(),2),1))
I=4	:::

Table 3. This is an implementation of a Turing Machine in Excel. We suppose lines and columns are infinite. Three basic formulae are given in (e), additional formulae is produced by the extension mechanism.

a)Instantaneous Descriptions

A (state)	B (head)	C (tape)...					
1	3	1	1	2	2	5	5
2	4	3	1	2	2	5	5
2	5	3	1	2	2	5	5
3	4	3	1	4	2	5	5
3	3	3	1	4	2	5	5
1	4	3	1	4	2	5	5
2	5	3	3	4	2	5	5
2	6	3	3	4	2	5	5
3	5	3	3	4	4	5	5
3	4	3	3	4	4	5	5
1	5	3	3	4	4	5	5
4	6	3	3	4	4	5	5
4	7	3	3	4	4	5	5
5	8	3	3	4	4	5	5
#TRUE	8	3	3	4	4	5	0

5 is the blank symbol

This machine accepts $\{1^n 2^n \mid n \geq 0\}$

b) Symbol to Write on Tape

symbol	*=1*	*=2*	*=3*	*=4*	*=5*
state=1	3			4	
state=2	1	4		4	
state=3	1		3	4	

c) State Transition Table

symbol	*=1*	*=2*	*=3*	*=4*	*=5*
state=1	2	#FALSE	#FALSE	4	#FALSE
state=2	2	3	#FALSE	2	#FALSE
state=3	3	#FALSE	1	3	#FALSE
state=4	#FALSE	#FALSE	#FALSE	4	5

d) Tape Head Movement

symbol	*=1*	*=2*	*=3*	*=4*	*=5*
state=1	1			1	
state=2	1	-1		1	
state=3	-1		1	-1	

e) Excel Formulae

A3=ÍNDICE(State!$A$1:$E$5;Tape!$A2;ÍNDICE($A2:$Z2;1;$B2))

B3=ÍNDICE(Move!$A$1:$E$5;Tape!$A2;ÍNDICE($A2:$Z2;1;$B2))+$B2

C3=SE(COL()=$B2;ÍNDICE(Write!$A$1:$E$5;Tape!$A2;
ÍNDICE($A2:$Z2;1;$B2));C2)

Table 4. Implementation of a Turing Machine in our prototype.

	J=1	J=2	J=3	4	5	6	7	8
I=1	state:grid(…)	write:grid(…)	move:grid(…)					
I=2	1	3	1	1	2	2	5	…
I=3	ref**(**state, ref(sub(row(),1),1)**,** ref(sub(row(),1), ref(sub(row(),1),2))**)**	add**(** ref(sub(row(),1),2)**,** ref(move, ref(sub(row(),1),1), ref(sub(row(),1), ref(sub(row(),1),2)))**)**	if**(** eq(col(), ref(sub(row(),1),2))**,** ref(write, ref(sub(row(),1),1), ref(sub(row(),1), ref(sub(row(),1),2)))**,** ref(sub(row(),1),col())**)**	…				
I=4	:::	:::	:::	***				

The difference of our proposal to [Burnett *et al.*, 2001] is that tabular organization is enforced and depended upon. Coordinate-awareness functions can provide access to the cells of the outer (or the "calling") spreadsheet as absolute references or references relative to the location of the extend() function. As a result, spreadsheet cells can be also overridden with ellipsis, allowing a sort of infinite argument list. Return of multiple values is done by accessing computed values of the spreadsheet instances. Infinite list of values can then also be returned.

6. Implementation

We implemented the prototype of the interpreter and the editor in Java. Java was chosen as development platform for several reasons: we already have a garbage collector; it is easier to write a graphical editor; it is easy to write a FFI to extend the system using class files and the reflection mechanism; it is multiplatform; it is free.

Spreadsheets were implemented as Java Hashtables. The objects used for keys are coordinate pairs and, for elements, a class containing a formula string, the expression tree of the parsed formula and a cache for the cell value. Each spreadsheet also has a hash table of named cells (mapping symbolic name to coordinates) and a linked list of ellipsis cells. When the value of a cell is requested, the spreadsheet checks if the cell value was cached. If it is not the case, then it checks if the cell is influenced by some ellipsis cell, returning the corresponding ellipsis reference or returning the cell expression tree otherwise. When any formula is changed, all cached values are discarded.

Spreadsheet persistence is made by a linear description of spreadsheets in the formula language. This allows also the inclusion of a spreadsheet as a cell element inside another spreadsheet. The syntactical construction is grid(row,column,formula,…). This construction is not evaluated as a function; its parameters are considered "quoted".

The editor has two modes: a formula-view mode and a value-view mode that can be switched through a button widget. Formulae for the cells can be entered in either mode. A cell is selected by a single click. A ctrl+click combination produces the expression of relative reference to the cell, while a ctrl+shift+click produces the absolute reference. Double-clicks on cells that contain a spreadsheet will open an editor window with that inner spreadsheet. The textual representation of a spreadsheet can be obtained by an option on the menu and can be pasted as the content of any spreadsheet cell.

The lexical analyzer was written with the aid of Java regular expressions. The parser is a top-down depth-first backtracking parser based on recursive function calls. A symbol table (based in a hash table) is kept to make quick comparisons of symbolic strings. The evaluation of some integer expressions was implemented: addition, multiplication, arithmetic negation, subtraction, division, comparison (equal, greater than *etc*.). In the case of a spreadsheet inside of another, an expression ref (0, 2, 7, 0) refers to the cell at row 7 column 0 of the spreadsheet located in the cell at row 0 column 2. This function can receive as arguments integer coordinates, names of cells and spreadsheets, so that the expression ref (ref (0, 2), 7, 0) has the same meaning of the one above. Functions col() and row() return the coordinates of the cell being evaluated and admit a numeric parameter *n* to refer to the *n*-th outer level coordinates.

The function up() returns the spreadsheet that contains the spreadsheet with the cell being evaluated.

7. Examples and Evaluation

We implemented several experiments including Pascal triangle with tabular recursion (Fig. 1), functional-recursive Fibonacci and factorial functions (Fig. 2) and the already mentioned TM. The computation of Fibonacci is well-known as costly to compute when we not reuse previously computed values. We implemented efficient computation of Fibonacci series through value cache. The cache is limited to cell values and is accessed during the evaluation of the ref() function. These basic examples show the expressiveness of the language. The option to create windows to show inner spreadsheets for intermediary computations is illustrated in figure 2 and was very important while constructing the spreadsheets.

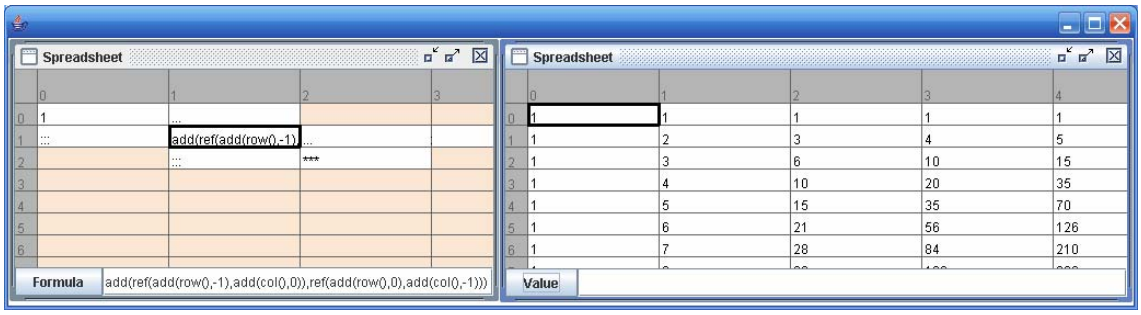


Figure 1. Pascal triangle spreadsheet.

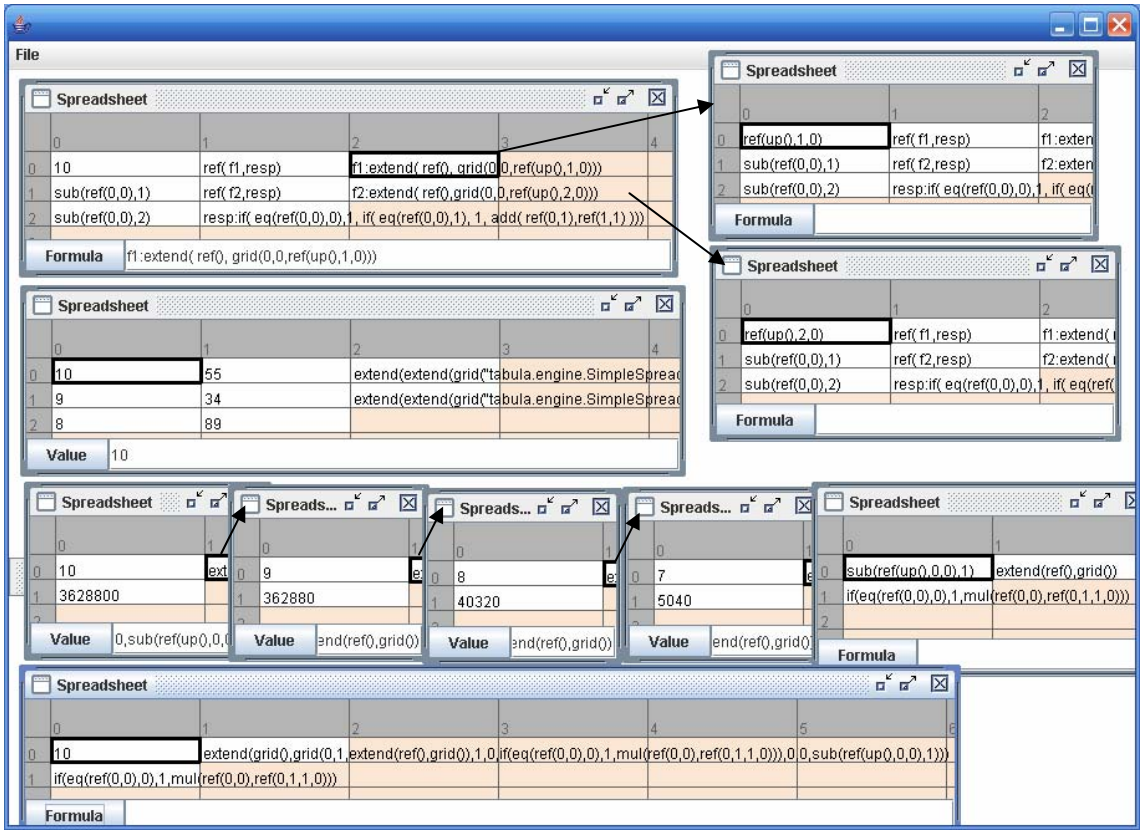


Figure 2. Recursive factorial and Fibonacci spreadsheets. Arrows show how spreadsheet windows open to allow visualization of intermediate results.

8. Conclusion

We presented a programming model based on spreadsheets instead of traditional text file programs. Although spreadsheets are already computationally expressive for many tasks, we intended to improve them to allow the definition of recursive computations and better modularity and reusability of components. We justify our design through informal theoretical discussion and examples of its expressiveness.

We expect the developed environment should meet several application needs, improving the expressive power of users when dealing with computations and allowing non-programmers to enter code in the intuitive form of spreadsheets as long as they are able to understand and use the simple formula language.

The intent of this work, in the present stage, is the exploration of concepts. We are not yet concerned with providing an optimal implementation and the language and the editor are still not appealing for end-users. The designed language is targeted to a prototype system valuing completeness, simplicity and ease of implementation. A lot more of syntax sugar, basic functions and constructs should be added in order to deliver the systems to users. In the future, we plan to improve the interpreter by implementing graph rewriting, better data structure construction functions and a design for the “scan” operation. We intend to apply the prototype in education and information visualization scenarios.

References

Bricklin, D. and Frankston, B. (1999) VisiCalc: Information from its creators. http://www.bricklin.com/visicalc.htm

Burnett, M., Atwood, J., Djang, R., Gottfried, H., Reichwein, J., and Yang, S. (2001) Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm, Journal of Functional Programming 11(2), March, 155-206.

Chi, E. H., Riedl, J., Barry, P. and Konstan, J. (1998) Principles for information visualization spreadsheets. In IEEE Computer Graphics and Applications (Special Issue on Visualization) July/August. IEEE CS, p. 30-38.

Field, A. J. and Harrison, P. G. (1988) Functional Programming, Addison-Wesley.

de Hoon, W., Rutten, L., van Eekelen, M. (1995) Implementing a Functional Spreadsheet in CLEAN, Journal of Functional Programming 5(3), July, pp 383-414.

Levoy, M. (1994) Spreadsheet for images. In Computer Graphics (SIGGRAPH '94 Proceedings), volume 28, pp 139-146. SIGGRAPH, ACM Press.

Lisper, B. and Malmström, J. (2002) Haxcel: A Spreadsheet Interface to Haskell, Proc. 14th International Workshop on the Implementation of Functional Languages, p 206-222, Madrid, September.

Misner, C. W. and Cooney, P. J. (1991). Spreadsheet Physics. Addison-Wesley, 1991.

Nuñez, F., (2000) An Extended Spreadsheet Paradigm for Data Visualisation Systems, and Its Implementation, M. Sc. thesis Department of Computer Science, Faculty of Science, The University Of Cape Town. URL citeseer.ist.psu.edu/543469.html

A New Architecture for Concurrent Lazy Cyclic Reference Counting on Multi-Processor Systems

Andrei de A. Formiga, Rafael D. Lins

Departamento de Eletrônica e Sistemas
Universidade Federal de Pernambuco (UFPE) – Recife, PE – Brazil
andrei.formiga@gmail.com, rdl@ufpe.br

Abstract. *Multi-processor systems have become the standard in current computer architectures. Software developers have the possibility to take advantage of the additional computing power available to concurrent programs. This paper presents a way to automatically use additional processors, by performing memory management concurrently. A new architecture with little explicit synchronization for concurrent lazy cyclic reference counting is described. This architecture was implemented and preliminary performance tests point at significant efficiency improvements over the sequential counterpart.*

1. Introduction

Automatic memory management (often called *garbage collection*) has become widespread in current programming systems as witnessed by the wide adoption of garbage-collected execution environments such as Sun Java and Microsoft .NET platform. Automatic memory management programming systems improve programmer productivity and the reliability of resulting programs by relieving programmers from the burdensome and error prone task of having to manage memory manually [Jones and Lins, 1996].

Most current systems, implemented on sequential architectures, use a sequential garbage collector, where memory management tasks alternate with tasks related to the user process; these two tasks are never executed concurrently. However, current trends in processor design and marketing indicate that single-chip multiprocessors are becoming of widespread use in recent computer systems. Such change in computer architecture opens the possibility of taking advantage of the extra computing power available. Executing memory management tasks concurrently with the user processes, programs can automatically take advantage of the available processors on a computer, with no changes required to the user programs.

Reference counting, first developed by G.E. Collins [Collins, 1960], is one of the main methods for automatic memory management. This paper proposes a new architecture for a concurrent reference counting memory management system, based on a similar proposal by Lins [Lins, 2005]. The motivation for a new architecture was the need to minimize explicit synchronization between the collector and the rest of the program, particularly to avoid the use of locks; Lins' original proposal was the basis of the garbage collector of several commercial machines, but their code was never freely available. The architecture proposed herein was implemented in a real programming

system, and preliminary performance tests show promising gains in overall performance of programs.

This paper is organized as follows: Section 2 briefly presents the reference counting algorithm in its simplest form and some milestones in its development, leading to previous architectures for parallel and concurrent reference counting, some of which were used as the basis for the one presented here; Section 3 details the new architecture for concurrent reference counting on multiprocessors that minimizes explicit synchronization, and emphasizes its differences in relation to its predecessors; the architecture in Section 3 is limited to one collector and one user process, thus Section 4 suggests ways in which the proposed architecture may be extended to work with multiple user processes, and multiple collector processes; finally, Section 5 presents the current implementation of the proposed architecture and the results of some initial performance measurements.

2. Reference Counting

The reference counting method consists of associating with each object in the heap a counter that stores the number of references to it. An object is active when it is currently in use and somehow accessible by the user program, which is indicated by it being transitively connected by references to one of the roots of the computation. The user process alters the connectivity of the objects both increasing and decreasing their reference count. Whenever the counter of an object reaches zero, it means that it is no longer active thus it is *garbage* and may be reclaimed to be later reused. It is usual to consider the heap as organized in a directed graph, where objects are the nodes and references are the edges. After Dijkstra [Dijkstra *et al.*, 1976] it is common to call *mutator* the part of the program that is concerned with the user process and *collector* the garbage collection system.

A distinctive advantage of reference counting over other methods of garbage collection is that its operations are interleaved with mutator activity, resulting in a technique that is naturally incremental. Other techniques, like mark-scan [McCarthy, 1960] and copying collection [Fenichel and Yochelson, 1969], require that the mutator stops for some time while the collector performs its work; this leads to pauses in interactive systems, which may be a nuisance. However, standard reference counting has a serious drawback: it can not reclaim cyclic data structures, as noticed by J.H.McBeth in 1963 [McBeth, 1963]. Thereafter, many solutions to this problem were proposed, but either they were not general-use, or were actually a proposal to use two memory management systems, one of them solely to reclaim cycles. The first general solution to the problem of reference counting cyclic structures was published in 1990 by Martinez, Wachenchauzer and Lins [Martinez *et al.*, 1990]; the idea is to identify points in the graph where cycles can be formed, and perform from there a local mark-scan to detect potential *space-leaks*, garbage cycles. In subsequent papers, Lins extended this solution in many ways, for instance by making the local cycle analysis lazy [Lins, 1992a] and by identifying critical points in the graph that could speed up the analysis [Lins, 2002].

To keep track of the state of local analysis in a local mark-scan, objects have to maintain not only a reference count, but also a color. The current color of a cell indicates its state regarding the local mark-scan.

Lins also developed a series of parallel reference counting algorithms targeted at multi-threaded or multi-processor systems [Lins, 1991; Lins, 1992b; Lins, 2005], but none of them were implemented in real multi-processor architectures. However, they were the basis for the two IBM Java machines implemented at IBM-T.J.Watson [Bacon et al., 2001] and IBM-Haifa [Levanoni and Petrank, 2006]. The architecture presented in this paper is based on the latest version of Lins' algorithm for parallel multi-processed lazy cyclic reference counting [Lins, 2005], detailed in the next section. This new architecture was implemented, with promising performance results, as detailed in Section 5 of this paper.

3. The Proposed Architecture

The new architecture proposed in this paper, in its basic version, is designed to use two processors executing concurrently: the mutator and the collector. Figure 1 shows a schematic view of this architecture. The two processes share access to three queues: a list of free objects, an increment queue and a decrement queue. The mutator alters graph connectivity getting memory from the *free-list* whenever a new object is created. When new references are made, the mutator inserts increment requests onto the *increment queue*; likewise, when references are destroyed, the mutator inserts decrement requests onto the *decrement queue*. The collector manages memory, removing and processing requests for increment and decrement from the appropriate queues and detecting when objects become garbage; when this happens, memory reclaimed from garbage objects is added to the free list. A detailed presentation of how the algorithm works is presented in the next subsections. For a matter of simplicity of the management of the free-list, it is considered that objects are fixed-size cells.

3.1. Mutator Operations

From the point of view of garbage collection, the mutator can only create new objects (getting cells from the free-list), create new references to objects, or destroy existing references to objects. Besides standalone operations for creation and destruction of references, it's often useful to have a single update operation that combines the deletion of a reference to an object and the creation of a reference to another object – e.g., when a pointer that references an object is changed to point to another one – a reference to the former object is destroyed, while the latter has a new reference to it. So, in this model the mutator performs three operations:

- `New` – to allocate new cells, getting them from the free list
- `Del(S)` – used when a reference to cell `S` is destroyed; generates a decrement request
- `Update(R, S)` – used to update a reference from object `R` to object `S`

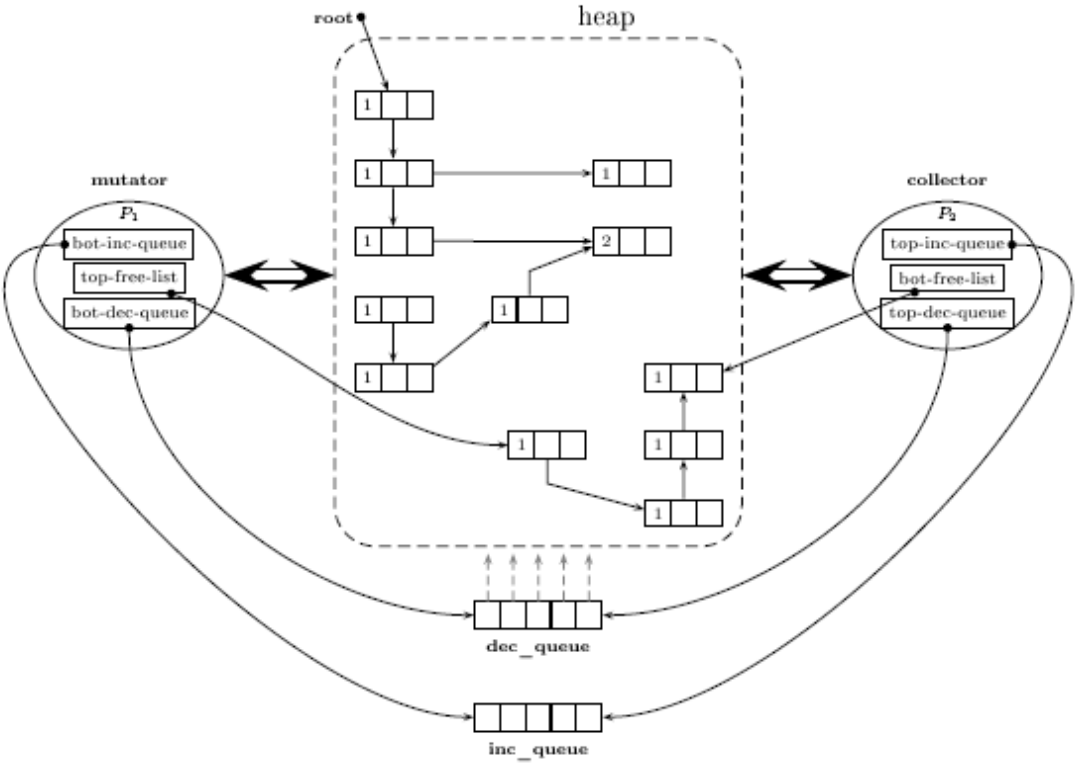


Figure 1. Basic architecture for concurrent reference counting with one mutator and one collector

The mutator never changes directly the reference count of a cell, nor its color. All memory management is done by the collector. This has implications for the necessary synchronization of mutator and collector, as will be seen later.

The algorithm for operation `New` is presented below. The mutator simply checks the existence of cells in the free list and gets one if available; otherwise it remains busy waiting for new cells. This is reasonable, because in this case the mutator can not continue its computation, and must wait for the collector to make cells available.

```
New() =
   if not empty(free-list) then
      newcell := get_from_free_list()
   else
      newcell := New()
   return newcell
```

The operation `Del` is even simpler than in standard reference counting: the mutator only inserts a new decrement request into the decrement queue. The collector takes care of the remaining graph adjustment.

```
Del(S) =
   add_decrement(S)
```

Finally, `Update` must replace a reference by another, thus it must call `Del` to notify the destruction of a reference, and insert a reference onto the increment queue.

```
Update(R, S) =
   Del(*R)
   add_increment(S)
   *R := S
```

It is easy to see in the operations above that the mutator only removes cells from the free list (actually organized as a queue) and inserts elements onto the increment and decrement queues. This results in simpler management of the three queues, shared by two concurrent processes.

3.2. Collector Operations

The memory management responsibilities, for a reference counting system, are to keep track of reference counts and detect when objects become garbage. It is also necessary to manage the cycle analysis, performed by the local mark-scan procedures. In the proposed architecture, the collector must process increment and decrement requests placed on the corresponding queues; adjust reference counts accordingly; detect any cells that have become garbage and insert them onto the free list; and besides that to keep track of possible cyclic structures that may have become garbage. This outline of how the collector works is reflected in its main operation, `Process_queues`, detailed below.

The analysis of cyclic structures requires that cells must have one of three possible colors: green, red and black. Green cells are active cells that are not scheduled for a local mark-scan. Red cells are currently under a local mark-scan. Finally, black cells are marked and scheduled for a local mark-scan at some point in the future. Only shared cells may be part of a cycle, and such a cycle becomes garbage when a shared cell has its reference count decremented. This means that when a cell has its reference counted decremented to a value greater than zero, it must be marked for future analysis. Cells that were marked for a local mark-scan are kept in a data structure called the *status analyzer*. It is a collection of cells that were marked black and must be analyzed later on. The original algorithm by Martinez, Wachenchauzer and Lins [Martinez et al., 1990] was eager: whenever a shared cell had its counter decremented, it was immediately analyzed with a local mark-scan. Experience has proved that doing the analysis lazily, as proposed by Lins [Lins, 1992a], yields far better performance.

When a cell in the status analyzer is selected, a local mark-scan takes place: the cell and its sub-graph is painted red and their reference counters are decremented, eliminating references that are internal to this sub-graph. If there remain cells with reference counter greater than zero at the end of this red-marking phase, it means that there are external references to the sub-graph, thus it is necessary to paint cells green and restore their counters. If no red cell in the sub-graph has a counter greater than zero, the whole sub-graph is a garbage cycle and must be collected so that the cells can be returned to the free list. The status analyzer is also used to keep objects that are critical points in the object graph, points that when analyzed will allow the algorithm to decide in less steps whether a sub-graph is cyclic garbage or not. This is a very brief

description of the cyclic reference counting algorithms developed by Lins. For further details one may refer to [Lins, 2002].

The main operation in execution by the collector is `Process_queues`, whose basic function is to process requests for increment and decrement of reference counts, getting them from the appropriate queues. Adjustment of reference counter in objects will then trigger most other actions of the collector, like freeing the memory occupied by garbage objects and marking objects for cycle analysis. The only operation that will not be triggered by adjusts in reference counters is scanning the status analyzer to detect cycles, so this is included in `Process_queues`: it first processes the increment queue, taking increment requests from it and performing them (cells are assumed to have a reference counter field `rc` and a color field `color`); then it processes the decrement queue, calling operation `Rec_del` to destroy a reference; finally, after processing both queues, the status analyzer is processed by a call to `scan_status_analyzer`. Operation `Rec_del` must be used to delete a reference because if the cell has its counter hit zero, all its outgoing references must be deleted recursively.

```
Process_queues() =
   while not empty(inc-queue)
      S := get_increment()
      S.rc := S.rc + 1
      S.color := green
   if not empty(dec-queue) then
      S := get_decrement()
      Rec_del(S)
   else
      scan_status_analyzer()
   Process_queues()
```

The recursive reference deletion operation, `Rec_del`, is thus defined:

```
Rec_del(S) =
   if S.rc == 1 then
      S.color := green
      for T in S.children do
         Rec_del(T)
      add_to_free_list(S)
   else
       S.rc := S.rc - 1
      if S.color != black then
         S.color := black
         add_to_status_analyzer(S)
```

It first checks to see if the removed reference is the last one to the object; in this case, it must be deleted and its outgoing references removed; the field `children` in a cell is supposed to be a collection of all its outgoing references. If the cell has reference count greater than one, it is either a potential candidate to have an external reference

transitively connecting it to root or being a knot-tying point to a cycle, thus it must be painted black and added to the status analyzer.

The remaining operations of the collector are all related to the analysis and detection of cyclic garbage. `scan_status_analyzer` will be called whenever there are no increment or decrement requests to be processed (in the worst case, it will be called because of memory exhaustion – the mutator will be blocked if it can not allocate a new object). Its definition is presented below. When a cell S is in the status analyzer, it may be either black or red. If it is black, a local mark-scan is scheduled to be performed in the sub-graph below S. Operation `mark_red` is called to start the local analysis. If the cell is red, it means it was identified in `mark_red` as a critical point that may be connected to external references; its counter is checked, and if it is greater than zero the sub-graph below it is active, so it is necessary to undo the effects of operation `mark_red` by calling `scan_green`. The function calls itself recursively, looking for further items to process; if there are none left, the remaining cells still colored red are garbage in cyclic structures, so `collect` is called to recycle them.

```
scan_status_analyser () =
   S := select_from(status_analyser)
   if S == nil then return
   if S.color == black then
      mark_red(S)
   if S.color == red && S.rc > 0 then
      scan_green(S)
   scan_status_analyser ()
   if S.color == red then
      collect(S)
```

`mark_red` performs the red-marking phase of the local mark-scan. It tests if the cell is red, coloring it red if not; then the reference counter of all its children (objects referenced by it) is decremented, canceling references that may be internal to a cycle. `mark_red` is called recursively through the whole sub-graph of S. If a cell is already painted red – which means it already had its reference counter decremented – and still has a counter with value greater than zero, it is a critical point in the graph that may be connected to external references. Adding it to the status analyzer will speed up the process of deciding if a sub-graph is a garbage cycle or not [Lins, 2002]. This accounts for red-colored cells found in the status analyzer, as seen previously while describing `scan_status_analyzer`.

```
mark_red(S) =
   if S.color != red then
      S.color := red
      for T in S.children do
         T.rc := T.rc - 1
      for T in S.children do
         if T.color != red then
            mark_red(T)
         if T.rc > 0 && T not in status-analyzer then
            add_to_status_analyser(T)
```

If a sub-graph is found to have external references, it is necessary to restore counters decremented by mark_red; This is the task of `scan_green`. It paints a cell green and increments the counter of its children.

```
scan_green(S) =
   S.color := green
   for T in S.children do
      T.rc := T.rc + 1
      if T.color != green then
         scan_green(T)
```

The only remaining operation to be described is `collect`, whose responsibility is to free the memory associated with each garbage object detected in a sub-graph. Its definition is shown below. Cells returned to the free list are painted green and assigned a reference counter with value 1, to simplify cell initialization when they need to be later reallocated.

```
collect(S) =
   for T in S.children do
      Delete(T)
      if T.color == red then
         collect(T)
   S.rc := 1
   S.color := green
   add_to_free_list(S)
```

3.3. Synchronization

Now that the operations of both processes were described, it is now considered how they can work concurrently in a safe and efficient manner. These operations were adapted from the sequential version of the cyclic reference counting algorithm [Lins, 2002], and the whole architecture presented here was based on that of Lins [Lins, 2005]. Lins' architecture used only two shared queues between collector and mutator, the free list and a decrement queue (called a *delete queue* in the original paper). The result is that both collector and mutator read and update reference counts in cells, so this configures a potential interference between them [Andrews, 1999]. Lins dealt with this by assigning priorities to the processors, so that one would always be allowed to access reference counter fields before the other in a situation of contention. This would be a good solution, if current hardware provided a way to ensure the priorities. However, this is not the case in current shared-memory multi-processor architectures, so implementing Lins' architecture in current hardware would create the necessity of explicit synchronization between collector and mutator when reading or updating reference counts in cells, and as these are frequent operations, the overhead involved with synchronization would compromise performance. The architecture presented here solves this by using a technique of disjoint variables [Andrews, 1999], where only the collector reads and updates the values of reference counts in cells, and only the mutator changes the value of references in the memory graph. This works by having two queues, one for requests for increment and another for requests of decrements, shared between the two processes. The resulting architecture has little need for explicit synchronization.

None is needed for reads and updates of reference counts or reads or updates of pointers.

A remaining concern regarding synchronization is about the three shared queues. Although it is not described here, there are known ways to implement concurrent queues with little need for explicit synchronization, especially for the case of a single reader and single writer [Valois, 1994]. The chosen implementation of concurrent queues will dictate how operations like `add_increment`, `add_decrement`, `get_from_free_list` and their counterparts will be ultimately implemented.

4. Multiple Mutators and Collectors

In Section 3 the basic architecture for concurrent reference counting was presented, considering a designation of one process as the mutator and one as the collector. In many programs, it may be desirable to have more than one mutator or thread executing concurrently, thus the architecture is now generalized. Figure 2 shows a schematic version of this multi-mutator architecture.

The main difference in relation to the single mutator version is that now all mutators must share access to one end of the three queues. To accomplish this, mutator processes no longer access the queues directly, but through shared registers, called `top-free-list`, `bot-dec-queue` and `bot-inc-queue`. Access to these registers must be synchronized to avoid interference; this can be done by using *compare-and-swap* type operations, present in hardware in most current processors. Implementation of lock-free queues with many readers (for the free list) or many writers (for the request queues) is still possible [Valois, 1994]. Otherwise, the algorithms work just like the single mutator case. In particular, the collector does not need any changes to accommodate this architecture.

Another direction of extension to the architecture presented in Section 3 is to allow for many collector processes to work cooperatively in a parallel garbage collection architecture. This could be combined with the suggestions for multiple mutators to obtain a multi-mutator, multi-collector architecture. The issues involved in such architecture, however, are more complicated and include greater problems of interference and load-balancing between collectors, and between them and the mutators. This possibility is not considered further in this paper. Previous work by Lins [Lins, 2005] presents such architecture in greater detail.

5. Implementation and Results

The architecture of Section 3 was implemented as the garbage collection sub-system for a programming language. A compiler for a lazy, purely functional programming language (similar to a subset of Haskell) was written to generate benchmarks for the implemented garbage collection system. The compiler uses well-known techniques for implementation of functional programming languages [Peyton-Jones, 1987] and generates code for an execution environment similar to the G-machine; this runtime environment includes an implementation of the garbage collector described in Section 3.

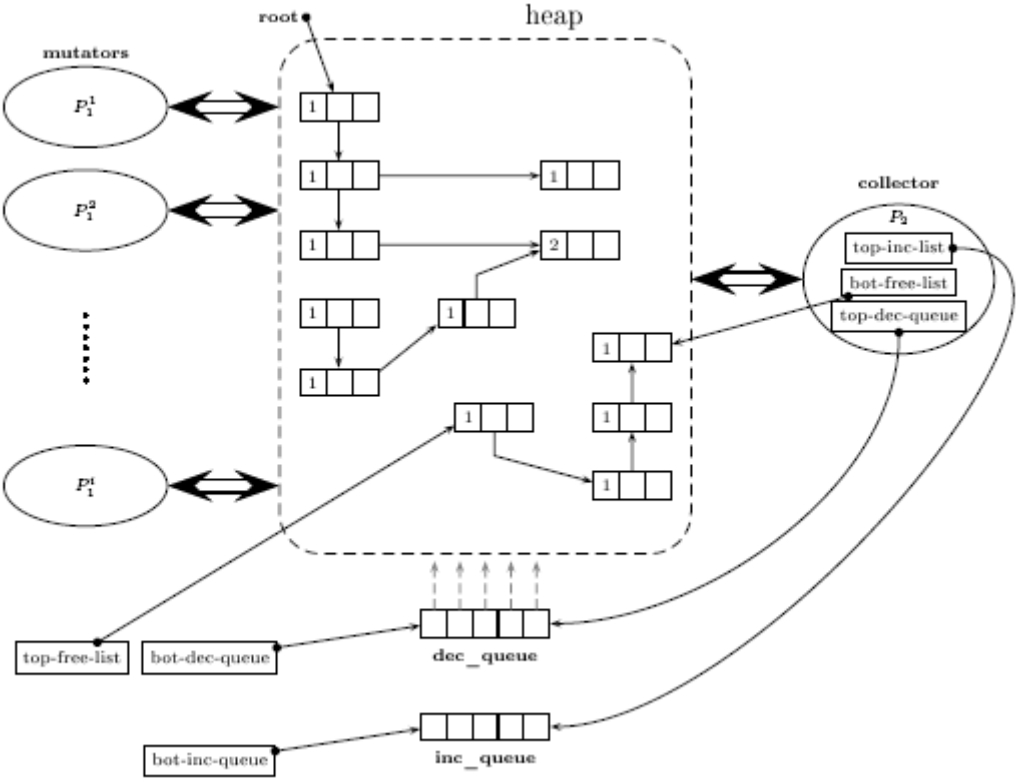


Figure 2. Architecture for concurrent reference counting with more than one mutator process.

This implementation was used in experiments to assess the performance of the new architecture with relation to the sequential version of the same algorithm. Table 1 shows the results for execution of six test programs compiled to work with a sequential reference counting algorithm. All tests were executed in a single 1.6GHz Athlon MP computer with 2 processors and 512Mb of RAM, with a free list of 5000 cells and a status analyzer structure that could hold a hundred items.

Table 1. Results for the execution of six test programs with sequential reference couting.

Benchmark	alloc	scan_sa	mark_red	scan_green	collect	time(s)
acker	253175	4013	40574	40127	447	0,0351
conctwice	42834	521	5468	5406	62	0,005
fiblista	14837640	27892	857421	836317	21104	3,0812
recfat	335959	4985	49872	49338	534	0,0872
somamap	94309	1356	10521	10409	112	0,0473
somatorio	35298	499	4987	4933	54	0,0274

The tests where selected for being highly recursive programs, and in some cases for using many list objects. Recursion was implemented by knot tying the Y-combinator, yielding cycles in the graph-reduction machine [Turner 79] which is part of the execution environment. Programs with many recursive calls generate many cycles, exercising the cycle analysis capabilities of the algorithm. The last column on Table 1

reports overall execution (CPU) times, in seconds. The other columns list the number of calls to operations related to memory management: the **alloc** column contains the number of allocated cells during the execution of the program; **scan_sa** is the number of calls to `scan_status_analyzer`; **mark_red**, **scan_green** and collect each record the number of times the respective operations were called.

The results on Table 1 make sense only when compared with those of Table 2, which shows results for the same tests, but executed in a system with the concurrent reference counter algorithm of Section 3. The meaning of the columns is the same.

Table 2. Results for the execution of six test programs with concurrent reference couting.

Benchmark	alloc	scan_sa	mark_red	scan_green	collect	time(s)
acker	253175	9304	57966	57434	532	0,025
conctwice	42834	897	7210	7127	83	0,0038
fiblista	14837640	35112	956874	935527	21347	2,732
recfat	335959	11421	112663	112075	588	0,0702
somamap	94309	1647	11896	11773	123	0,0298
somatorio	35298	532	6052	5980	72	0,0208

The two tables show that the number of calls for memory management functions is greater in the concurrent architecture. This happens because the collector calls `scan_status_anlyzer` every time the increment and decrement queues are both empty, while the sequential version only examines the status analyzer whenever there are no free cells [Lins, 2002]. However, overall execution times where, on average, about 20% smaller for the concurrent reference counting system, in relation to the sequential version. Albeit these performance results are preliminary – for example, the total execution time of the benchmark programs above is still too small – they indicate that the concurrent architecture of Section 3 represents a performance gain to programs that use it, without any need to alter the source code of user programs; besides, such a garbage collector uses better the available hardware in multi-processor systems, even when the executing programs are not concurrent themselves.

6. Conclusions and lines for further works

This paper presented a new architecture for concurrent lazy cyclic reference counting on multi-processor systems, targeted for implementation on current commercial hardware and with less need for explicit synchronization than its predecessors. This architecture was implemented in a 1.6GHz Athlon MP computer with 2 processors and 512Mb of RAM. A lazy functional compiler was developed, together with the garbage collection subsystem, and its performance was tested against the sequential version of the same algorithm. Six benchmarks were used to test the performance of the architecture proposed. A rise in throughput of about 20% was observed between the sequential and the concurrent version for the programs tested.

The implementation of the language and the garbage collector serves as a testbed for further and more complex benchmarks as well as allowing for the possibility of testing different garbage collection strategies. Considering that the architecture proposed in this

paper was based on a previous proposal by Lins [Lins, 2005], a comparison between both versions would be interesting to verify the real gains achieved after minimizing the amount of required synchronization. However, Lins' architecture was never implemented; this is planned for future works. Another direction for further investigation is the use of more computer intensive benchmark programs – preferably real-world programs – that could provide more realistic figures on how the proposed algorithms would behave in production situations. In a recent paper Lins and Carvalho Jr. [Lins and Carvalho, 2007] introduce the notion of "permanent" or "tenured" objects to cyclic reference counting, making the local mark-scan more efficient. The extension of the architecture presented herein to encompass such objects is on progress.

The source code for the benchmarks, compiler and garbage collector may be found at: http://postele.homelinux.net/~andrei/faul.tar.gz

References

Andrews, G. R. (1999), Foundations of Multithreaded, Parallel, and Distributed Programming, Addison-Wesley Professional.

Bacon, D. F., Attanasio, C. R., Lee, H. B., Rajan, R. T. and Smith, S. (2001) "Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector", In: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, June, 2001 (SIGPLAN Not. 36,5).

Collins, G. E. (1960) "A method for overlapping and erasure of lists", In: Communications of the ACM, vol. 3, 12, pp. 655-657.

Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S. and Steffens, E. F. M. (1976) "On-the-fly garbage collection: An exercise in cooperation", In: Lecture Notes in Computer Science, No. 46. Springer-Verlag.

Fenichel, R. and Yochelson, J. (1969) "A Lisp garbage collector for virtual memory computer systems" In: Communications of the ACM, vol. 12, 11, pp. 611-612.

Jones, R. and Lins, R. D. (1996), Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons ltd.

Levanoni, Y. and Petrank, E. (2006) "An on-the-fly reference counting garbage collector for Java", In: ACM Transactions on Programming Languages and Systems, vol. 28, 1, pp. 1-69.

Lins, R. D. (1991) "A shared memory architecture for parallel cyclic reference counting", In: Microprocessing and Microprogramming, 34, pp. 31-35.

Lins, R. D. (1992) "Cyclic reference counting with lazy mark-scan", In: Information Processing Letters, vol. 44, 4, pp. 215-220.

Lins, R. D. (1992) "A multi-processor shared memory architecture for parallel cyclic reference counting", In: Microprocessing and Microprogramming, 35, pp. 563-568.

Lins, R. D. (2002) "An efficient algorithm for cyclic reference counting", In: Information Processing Letters, vol. 83, 3, pp. 145-150.

Lins, R. D. (2005) “A new multi-processor architecture for parallel cyclic reference counting”, In: SBAC-PAD ’05: Proceedings of the 17th International on Computer Architecture on High Performance Computing, pp. 35-43, IEEE Press.

Lins, R. D. and Carvalho Jr, F. H. de (2007) “Cyclic reference counting with permanent objects”, in this volume.

Martinez, A. D., Wachenchauzer, R. and Lins, R. D. (1990) “Cyclic reference counting with local mark-scan”, In: Information Processing Letters, 34, pp. 31-35.

McBeth, J. H. (1963) “On the reference counter method”, In: Communications of the ACM, vol. 6, 9, pp. 575.

McCarthy, J. (1960) “Recursive functions of symbolic expressions and their computation by machine”, In: Communications of the ACM, vol. 3, 3, pp. 184-195.

Peyton-Jones, S. L. (1987) The Implementation of Functional Programming Languages, Prentice-Hall International.

Valois, J. D. (1994) “Implementing lock-free queues”, In: Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, pp. 64-69.

Turner, D.A. (1979) “A new implementation technique for applicative languages”, Software Practice and Experience, 9, pp. 31-49.

Cyclic Reference Counting with Permanent Objects

Rafael Dueire Lins[1], Francisco Heron de Carvalho Junior[2]

[1]Departamento de Eletrônica e Sistemas, CTG, Universidade Federal de Pernambuco
50.740-530 – Recife – PE – Brazil

[2]Departamento de Computação, Universidade Federal do Ceará
50.740-530 – Fortaleza - CE – Brazil

rdl@ufpe.br, heron@lia.ufc.br

Abstract. *Reference Counting is the memory management technique of most widespread use today. Very often applications handle objects that are either permanent or get tenured. This paper uses this information to make cyclic reference counting more efficient.*

Keywords: *Memory management, garbage collection, reference counting, cyclic graphs, permanent objects, tenured objects.*

1. Introduction

Reference counting [Collins 1960] [Jones and Lins, 1996] is a simple memory management technique in which each data structure keeps the number of external references (or pointers) to it. It was developed by Collins [Collins 1960] to avoid user process suspension provoked by the mark-scan algorithm in LISP. Reference counting performs memory management in small steps interleaved with computation. In 1963, J.H.McBeth [McBeth 1963] noticed that reference counting was unable to reclaim cyclic structures, because the counter of cells on a cycle never drops to zero, causing a space-leak, as may be observed in Figure 1.

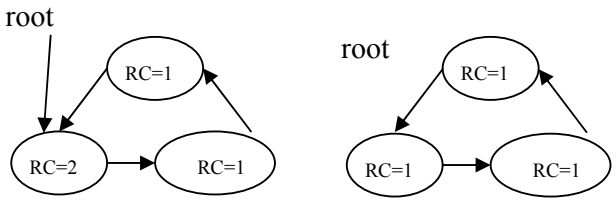


Figure 1. Isolating a cycle from root causes a space-leak

In real applications cyclic structures appear very often. For instance, recursion is frequently represented by a cyclic graph and web pages have hyperlinks to other web pages that frequently point back to themselves [Lins 2006]. These are two examples that may give an account of the importance of being able to handle cycles in reference counting. Several researchers looked for solutions for this problem. The first general solution for cyclic reference counting was presented in reference [Martinez, Wachenchauzer and Lins 1990], where a local mark-scan is performed whenever a

pointer to a shared data structure is deleted. Lins largely improved the performance of the algorithm in two different ways. The first optimization [Lins 1992] widely acknowledged as the first efficient solution to cyclic reference counting, postpones the mark-scan, as much as possible. This algorithm is implemented is both IBM Java machines developed at IBM T.J.Watson and IBM-Israel in Cooperation with the Technion, both of them reporting excellent performance [Bacon and Rajan 2001][Bacon et al 2001]. The second optimization [Lins 1993] relies on a creation-time stamp to help in cycle detection.

A decade later than the general solution to cyclic reference counting was presented, Lins introduced the *Jump_stack*, a data structure which largely increases the efficiency of the previous algorithms [Lins 2002]. This data structure stores a reference to the "critical points" in the graph while performing the local marking (after the deletion of a pointer to a shared cell). These nodes are revisited directly, saving a whole scanning phase in.

One of the strategies used in optimizing compilers and applications is to recognize whenever data is permanent or is "so old" that may be tenured. Handling such objects in a distinctive fashion avoids making copies of them and all the computational effort involved in its management. This paper introduces permanent objects to cyclic reference counting, increasing the efficiency of the previous algorithms.

2. Efficient Cyclic Reference Counting

The algorithm with permanent objects is designed on top of the efficient cyclic reference counting algorithm [Lins 2002]. Thus, it is explained in this section. The general idea of the algorithm is to perform a local mark-scan whenever a pointer to a shared structure is deleted. The algorithm works in two steps. In the first step, the sub-graph below the deleted pointer is scanned, rearranging counts due to internal references, marking nodes as possible garbage and also storing potential links to root in a data structure called the "*Jump-stack*". In step two, the cells pointed at by the links stored in the Jump-stack are visited directly. If the cell has reference count greater than one, the whole sub-graph below that point is in use and its cells should have their counts updated. In the final part of the second step, the algorithm collects garbage cells.

Now, implementation details of the algorithm are presented. As usual, free cells are linked together in a structure called *free-list*. A cell $\boldsymbol{B}$ is *connected* to a cell $\boldsymbol{A}$ ($\boldsymbol{A \rightarrow B}$), if and only if there is a pointer $\boldsymbol{<A, B>}$. A cell $\boldsymbol{B}$ is *transitively connected* to a cell $\boldsymbol{A}$ ($\boldsymbol{A \overset{*}{\rightarrow} B}$), if and only if there is a chain of pointers from $\boldsymbol{A}$ to $\boldsymbol{B}$. The initial point of the graph to which all cells is use are transitively connected is called *root*. In addition to the information of number of references to a cell, an extra field is used to store the color of cells. Two colors are used: green and red. Green is the stable color of cells. All cells are in the free-list and are green to start with.

There are three operations on the graph:

New(R) gets a cell U from the free-list and links it to the graph:

```
New (R) = select U from free-list
          make_pointer <R, U>
```

Copy(R, <S,T>) gets a cell R and a pointer <S, T> to create a pointer <R, T>, incrementing the counter of the target cell:

```
Copy(R, <S,T>) = make_pointer <R, T>
                 Increment RC(T)
```

Pointer removal is performed by Delete:

```
Delete (R,S) = Remove <R,S>
               If (RC(S) == 1) then
                  for T in Sons(S) do
                      Delete(S, T);
                  Link_to_free_list(S);
               else Decrement_RC(S);
                    Mark_red(S);
                    Scan(S);
```

A cell T belongs to the bag Sons(S) iff there is a pointer <S,T>. One can observe that the only difference to standard reference counting in the algorithm above rests in the last two lines of Delete, which will be explained below. Mark_red is a routine that "analyzes" the effect of the deleted pointer in the sub-graph below it. The sub-graph visited has the counts of cells decremented. Whenever a cell visited remains with count greater than one two possibilities may hold:

1. The cell is an entry point of root into the sub-graph below it.
2. The value is a transient one and may become zero at a later stage of Mark_red, indicating that it is not an entry point from root.

To perform this analysis whenever a cell is met by Mark_red with count greater than one after decrementing, it is placed in the Jump_stack. The code for Mark_red follows:

```
Mark_red(S) =  If (Color(S) == green) then
                  Color(S) = red;
                  for T in Sons(S) do
                      Decrement_RC(T);
                      if (RC(T)>0 &&
                          T not in Jump_stack)
                       then Jump_stack = T;
                      if (Color(T) == green)
                       then  Mark_red(T);
```

Scan(S) verifies whether the Jump_stack is empty. If so, the algorithm sends cells hanging from S to the free-list. If the jump-stack is not empty there are nodes in the graph to be analysed. If their reference count is greater than one, there are external pointers linking the cell under observation to root and counts should be restored from that point on, by calling the ancillary function Scan_green(T).

```
Scan(S) =  If RC(S)>0 then Scan_green
           else
             While (Jump_stack ≠ empty) do
               T = top_of_Jump_stack;
               Pop_Jump_stack;
               If (Color(T) == red && RC(T)>0)
               then
                    Scan_green(T);
            Collect(S);
```

Procedure Scan_green restores counts and paints green cells in a sub-graph in use, as follows,

```
Scan_green(S) = Color(S) = green
                for T in Sons(S) do
                    increment_RC(T);
                    if color(T) is not green
                    then
                        Scan_green(T);
```

Collect(S) is the procedure in charge of returning garbage cells to the free-list, painting them green and setting their reference count to one, as follows:

```
Collect(S) =  If (Color(S) == red) then
              for T in Sons(S) do
                 Remove(<S, T>);
                 RC(S) = 1;
                 Color(S) = green;
                 free_list = S;
                 if (Color(T) == red) then
                    Collect(T);
```

3. Cyclic Reference Counting with Permanent Objects

As already mentioned in the introduction of this paper, permanent objects appear very often in real implementations of systems and languages. Treating such objects differently from temporary ones is a way to increase the efficiency of the cyclic reference counting algorithm presented above. Below, the algorithm presented in the last section is modified to handle permanent objects efficiently. Operations are explained in terms of the same atomic actions presented above.

New(R) gets a cell U from the free-list and links it to the graph. The color of the new object depends on its nature. Temporary objects are set as "green" while permanent objects are set as "white". Permanent objects have their reference count set to "overflow".

```
New (R) = select U from free-list
          make_pointer <R, U>
          if  R is permanent then (color(R):= white); RC(R):=∞;
                                  else (color(R):= green)
```

Although it may at first sight that permanent objects increase the complexity of the allocation routine New, in reality this is not the case. Permanent objects appear during graph creation, instead of during graph manipulation, without any need to perform the testing during run-time. Another low-cost alternative is to allow two different combinators for cell creation one for permanent objects and another for temporary ones:

```
New Perm (R) = select U from free-list
               make_pointer <R, U>
               color(R):= white; RC(R):=∞;

New_Temp (R) = select U from free-list
               make_pointer <R, U>
               color(R):= green
```

Copy(R, <S,T>) gets a cell R and a pointer <S, T> to create a pointer <R, T>. If the target T cell is temporary its count gets incremented:

```
Copy(R, <S,T>) = make_pointer <R, T>
                 if  color(T) is not  white
                 then Increment RC(T)
```

Again, the increase in the complexity of this operation is apparent. The color test needs not to be performed in the case of having counts with overflow. Thus it is possible to make use of the definition of Copy as before, provided that one has in account that the

value of counts in permanent object is not consistent, i.e. does not stand for the number of pointers to it.

```
Copy(R, <S,T>) = make_pointer <R, T>
                 Increment RC(T)
```

Pointer removal is performed by Delete, which may remain unaltered it one assumes that the decrement of overflow remains the same.

```
Delete (R,S) = Remove <R,S>
               If (RC(S) == 1) then
                     for T in Sons(S) do
                          Delete(S, T);
                     Link_to_free_list(S);
               else Decrement_RC(S);
                    Mark_red(S);
                    Scan(S);
```

The real changes in the algorithm with permanent objects appear during the mark-scan. Permanent objects never have their counts altered. Whenever a permanent object is part of a cycle under the local mark-scan it stops further analysis. Although Mark_red remains with the same definition as before, one must observe that the analysis does not propagate through *white* (permanent) cells.

```
Mark_red(S) =  If (Color(S) == green) then
                  Color(S) = red;
                  for T in Sons(S) do
                       Decrement_RC(T);
                       if (RC(T)>0 &&
                           T not in Jump_stack)
                        then Jump_stack = T;
                       if (Color(T) == green)
                        then  Mark_red(T);
```

As before, Scan(S) verifies whether the Jump_stack is empty. If so, the algorithm sends cells hanging from S to the free-list. If the jump-stack is not empty there are nodes in the graph to be analyzed. If their reference count is greater than one, there are external pointers linking the cell under observation to root and counts should be restored from that point on, by calling the ancillary function Scan_green(T).

```
Scan(S) =  If RC(S)>0 then Scan_green
           else
             While (Jump_stack ≠ empty) do
               T = top_of_Jump_stack;
               Pop_Jump_stack;
               If (Color(T) == red && RC(T)>0)
               then
                    Scan_green(T);
            Collect(S);
```

Procedure Scan_green visits only red cells, restores their counts and paints green cells in a sub-graph in use. Thus it is slightly modified to:

```
Scan_green(S) = If Color(S) = red then
                   Color(S): = green
                   for T in Sons(S) do
                        increment_RC(T);
                        if color(T) is  red
                        then
                             Scan_green(T);
```

Collect(S) is the procedure in charge of returning garbage cells to the free-list, setting their reference count to one, as follows:

```
Collect(S) =  If (Color(S) == red) then
                  for T in Sons(S) do
                   if (Color(T) == red) then
                        Collect(T);
                  RC(S):= 1;
                  free_list := S;
```

4. Tenuring objects

The generational hypothesis states that "young objects die young and old objects tend to remain in use until the end of computation". Taking this into account, very often systems and languages tend to give a permanent status to objects that "live" over a certain time or operational barrier. This change of status is called "tenure".

Several different tenuring policies may be adopted. One of them is change into white the color of a given object whenever it were green and have a reference count greater than a certain threshold value "*t*". This strategy changes the code for Copy into:

```
Copy(R, <S,T>) = make_pointer <R, T>
                 If RC(T) ≥ t and  color(T)== green
                      then color(T);= white
                      Increment RC(T)
```

Notice that tenuring polices may yield to space-leaks, as tenured objects may become garbage. The code for Delete remains unchanged. One should observe that it may claim white cells provided one is removing the last reference to it. Tenured cells only avoid the propagation of the local mark-scan through them. That means that the space leak only involves cyclic structures of tenured cells.

4.1. Avoiding Space-leaks

A possibility of having tenured objects and avoiding permanent cell loss that cause space leaks is to introduce a new color, "*grey*" to tenured objects and place them in the Jump-stack for later analysis. This will cause the redefinition of Copy as:

```
Copy(R, <S,T>) = make_pointer <R, T>
                 If RC(T) ≥ t and  color(T)== green
                      then color(T);= grey
                           Jump_stack:= T
                 Increment RC(T)
```

One should notice that grey objects hold their actual reference count value. Tenured objects are analyzed only at last, i.e. whenever the free-list is empty. At this moment, the Jump_stack is empty. Thus, the code for New is now written as:

```
New (R) = if free-list not empty then
                select U from free-list
                make_pointer <R, U>
                 if  R is permanent then (color(R):= white); RC(R):=∞;
                                 else (color(R):= green)
          else
              If Jump_stack not empty then
                 For T in Jump_stack  do
                          NMark_red(T)
                 Scan(T)
                 New(R)
            else
                  write_out "No cells available; execution aborted";
```

NMark_red is a new procedure that takes into account the possibility of cells being grey.

```
NMark_red(S) =  If (Color(S) == green or grey) then
                    Color(S) = red;
                    for T in Sons(S) do
                        Decrement_RC(T);
                        if (RC(T)>0 &&
                            T not in Jump_stack)
                         then Jump_stack = T;
                        if (Color(T) == green or grey)
                         then  Mark_red(T);
```

The code for Delete remains unchanged. One should observe that it may claim grey cells directly, provided one is removing the last reference to it. Grey cells only avoid the propagation of the local mark-scan.

The tenuring policy must be carefully adopted as it either may cause space-leaks or a high operational overhead.

5. Proof of the Correctness

Providing formal proofs of the correctness of algorithms is not a simple task. This section elucidates the on the correction of the algorithms presented in this paper. The starting point is assuming the correctness of the algorithm for efficient cyclic reference counting [Lins 2002].

5.1. Cyclic RC with Permanent Objects

Permanent objects may be seen as objects that are permanently linked to root. Thus, the only role they play is to stop the propagation of the local mark-scan. In doing so the algorithm saves the need of placing the object onto the Jump_stack during Mark_red and later, during Scan removing it from the Jump_stack and, as its reference count will never drop to allow the object to be collected, having to call Scan_green on it. The changes introduced into the code of the routines implement the operations described above making explicit the possibility of handling *white* objects.

5.2. Tenured Objects

The policy presented above to decide when an object may be considered permanent was focused on the number of references to it. There is a hidden assumption that the higher the number of references the longer lived will be the object. Under such hypothesis, all the algorithm does is to tenure a temporary object making it a permanent one. As one has already argued for the correctness of the algorithm with permanent objects, this change in status does not alter the overall behavior of the algorithm, provided one may be able to accept the possibility of a space-leak. This is the case when the deletion of the last pointer isolates an island from root with no elements to any later analysis.

5.3. Tenured Objects without Space-leaks

The whole idea of the algorithm for tenured objects without space-leaks is to keep references (in the Jump_stack) for deciding later about the validity of a tenured

object. The analysis of possible candidates for recycling is performed in extreme circumstances only, i.e. whenever the free-list is empty.

6. Conclusions

Permanent objects appear very often in real applications, thus addressing them in an efficient way is a matter of concern to whoever implement systems, languages, etc. This paper shows how to introduce permanent objects to cyclic reference counting in a simple way, with almost no overhead to atomic operations, but avoiding the unnecessary propagation of the local mark-scan. Besides that, this paper shows two different ways of working with tenured objects. The first alternative may cause space-leaks of cyclic data structures encompassing tenured objects. The second alternative makes possible to reclaim all garbage at a high operational cost. Both alternatives point in the direction of having a conservative tenuring policy to avoid overheads.

The presented algorithm will certainly have a large impact in the decreasing amount of communication exchanged between processors either in shared-memory architectures [Lins 1991][Lins 1992a] or in distributed environments [Lins and Jones 1993][Lins 2006], thus bringing more efficiency in tightly and loosely coupled systems.

7. Acknowledgements

This work was sponsored by CNPq – Brazilian Government.

8. References

D.F.Bacon and V.T.Rajan (2001) “Concurrent Cycle Collection in Reference Counted Systems”, In: Proceedings of European Conference on Object-Oriented Programming, Springer Verlag, LNCS vol. 2072.

D.F.Bacon, C.R.Attanasio, H.B.Lee, R.T.Rajan and S.Smith (2001) “Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector”, In: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (SIGPLAN Not. 36,5).

G.E. Collins (1960) “A method for overlapping and erasure of lists”, In: Comm. of the ACM, vol. 3, n. 12, pages 655—657.

R.E. Jones and R.D. Lins (1996) “Garbage Collection Algorithms for Dynamic Memory Management”, John Wiley & Sons.

R.D. Lins (1991) “A shared memory architecture for parallel cyclic reference counting”, In: Microprocessing and microprogramming, vol. 34, pages 31—35.

R.D. Lins (1992a) “A multi-processor shared memory architecture for parallel cyclic reference counting”, In: Microprocessing and microprogramming, vol. 35, pages 563—568.

R.D.Lins (1992b) “Cyclic Reference counting with lazy mark-scan”, In: Information Processing Letters, 44, pages 215—220.

R.D.Lins (1993), “Generational cyclic reference counting”, In: Information Processing Letters 46, pages 19—20.

R.D.Lins (2002) “An Efficient Algorithm for Cyclic Reference Counting”, In: Information Processing Letters, vol. 83, n. 3, pages 145-150, North Holland, August.

R.D.Lins (2006) “New algorithms and applications of cyclic reference counting”. In: Proceedings of ICGT 2006 – International Conference on Graph Transformation and Applications, Invited Keynote Paper, LNCS, Springer Verlag.

R.D. Lins and R.E.Jones (1993), “Cyclic weighted reference counting”, In: K. Boyanov (ed.), Proc. of Intern. Workshop on Parallel and Distributed Processing, NH.

J.H. McBeth (1963), “On the reference counter method”, In: Comm. of the ACM, 6(9):575.

A.D. Martinez, R. Wachenchauzer and R.D. Lins, (1990) “Cyclic reference counting with local mark-scan”, Information Processing Letters 34, pages 31—35, North Holland.

C APIs in extension and extensible languages

Hisham Muhammad
Roberto Ierusalimschy

[1]Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-RIO)
Rio de Janeiro, RJ – Brazil

Abstract. *Scripting languages are used in conjuction with C code in two ways: as extension languages, where the interpreter is embedded as a library into an application; or as extensible languages, where the interpreter loads C code as add-on modules. These two scenarios share many similarities, as in both of them two-way communication of code and data needs to take place. However, the differences between them impose design tradeoffs that affect the C API that bridges the two languages, often making a scripting language more suitable for extending than embedding, or vice-versa. This paper discusses how these tradeoffs are handled in the APIs of popular scripting languages, and the impact on their use as embedded or extensible languages.*

1. Introduction

There are many situations in which it is necessary or interesting to have interaction between programs written in different languages. A typical case is the use of external libraries, such as graphic toolkits, APIs for database access, or even operating system calls. Another scenario involves applications developed using more than one programming language, in order to optimize parts where performance is critical or to allow extensibility through scripts written by end-users.

Regardless of purpose, communication between programs written in different languages brings up a number of design issues, not only in the development of the applications, but of the languages themselves. There are many ways to obtain this kind of interoperability, but ideally, a language should provide a foreign language interface that allows programmers to send and receive both calls and data to another language [Finne et al. 1998].

A model for interaction between languages that has shown to be especially relevant nowadays is that between statically typed compiled languages, such as C and C++, and dynamically typed interpreted languages, such as Perl and Python. In [Ousterhout 1998], Ousterhout categorizes these two groups as *systems programming languages* and *scripting languages*.

These two categories of languages have fundamentally different goals. Systems programming languages emerged as an alternative to assembly in the development of applications, having as main features static typing, which eases the understanding of data structures in large systems, and being implemented as compilers, due to concerns with performance. In contrast, scripting languages are dynamically typed and are implemented as interpreters or virtual machines. Dynamic typing and the extensive use of higher-level constructs as basic types, such as lists and hashes, bring greater flexibility in the

interaction between components; in static languages, the type system imposes restrictions to those interactions, often requiring the programmer to write adaptation interfaces, which makes the reuse of components harder.

Scripting languages have the distinction that, by design, they are developed having interaction with code written in other languages in mind. Because of the popularity of the C language and the support it enjoys in most popular operating systems, a considerable number of implementations of foreign language interfaces are, in practice, C APIs.

Scripting languages are used in conjuction with C code in two ways: extending a C application, where the interpreter is embedded as a library; or by having C code extend the language, through add-on modules written as C libraries. These two scenarios share many similarities, as in both of them two-way communication of code and data needs to take place. However, the differences between them impose tradeoffs that affect the design of the resulting C API.

This paper discusses how the design of a language's C API affects its suitability for different application scenarios. In Section 2, we discuss the different roles of scripting languages. In Section 3, the main issues involving interaction of C code with scripting language runtime environments are presented, followed by a discussion in Section 4 on how popular scripting languages address those issues and the effect of their designs in their applicability as extension and extensible languages. Finally, Section 5 concludes the paper.

2. Extension and extensible languages

Scripting languages are designed to be used in two-language scenarios. Originally, they had an auxiliary role, in which user scripts allow for customization of applications. With the increased popularity of scripting languages, a different usage model has also risen to prominence, in which the scripting language performs a more central role. Typical examples are graphical applications where the interface is described by scripts controlling components implemented in C and games where the logic is described in scripts and the runtime engine is implemented in lower-level languages.

In these scenarios, there is a clear distinction between a lower-level layer where performance is a critical factor and another, higher-level layer that coordinates operations on elements of the lower layer. Scripting languages cease to be just an extension mechanism: the application itself is written using the scripting language and libraries written in lower-level languages are loaded as extension modules.

It makes sense, then, when discussing language interaction, to make a distinction between *extensible languages* and *extension languages*. Extensible languages are those that can be extended through external modules implemented in other languages. Extension languages are those which runtime environment can be embedded in an application, allowing to use them to extend the application. Typically, scripting languages can be used, with variable degrees of convenience, as either extensible or extension languages.

Another interesting observation is that, while in one model the scripting language serves as an extension language for the lower-level language in which the application is written, in the other model the opposite happens: we can look at add-on modules written using the language's C API as a way to extend the scripting language using C; in this

perspective, C becomes the extension language.

This way, the set of features provided by an API between C and a scripting language tends to be symmetric in case it is desired to provide language extensibility as well as promote its use as an extension language. In both situations, code and data manipulation features need to be provided in both directions. A few common issues arise when implementing interaction between C and scripting languages; they are discussed in the following section.

3. Interaction between C and scripting languages

Interfaces provided by scripting languages are usually understood as "extension APIs": they extend the virtual machine with features not originally offered by it, or alternatively, they extend an external application with the features offered by the runtime environment of the language, embedding it to the application. The first scenario is the one used in the programming model where the high-level coordination is made by an interpreted language and modules written in languages such as C and C++ are used to access external libraries or to implement performance-critical parts. The second scenario, in general, will also encompass the first one, when exposing to the embedded virtual machine extensions that will allow it to talk to the host application.

Both scenarios involve the same general problems: data transfer between the two languages, including how to allow the scripting language to manipulate structures declared in C and vice versa; handling the difference between the memory management models, more specifically the interaction between garbage collection in the virtual machine and explicit deallocation in C; calling functions declared by the scripting language from C; and the registration of C functions so that they can be invoked by scripts.

3.1. Data transfer

The main complexity in the interaction between programming languages is not the difference in syntax or semantics from their control flow structures, but in their data representations. In the communication between code written in two different languages, data flow in various forms: as parameters, object attributes, elements in data structures, etc.

Since the format how these data are represented often differs, the alternatives to perform data transfer between languages involve either converting the data or manipulating it opaquely through some kind of handle. The duplication that takes place when converting data limits the applicability of this method, restricting its use typically to numeric types and, in minor scale, strings. When exposing handles, the source language may explicitly offer facilities in the target language to manipulate these data, that is, the data remains opaque, but the language can access its contents through an API.

Because of its focus on the manipulation of pointers and structures, C provides a small set of basic types. Besides, C is very liberal with regard to the internal representation of its structured types, with each different platform having to define its own application binary interface (ABI). There may also be the need to handle conversion of endianness and format of floating point numbers. So, even in cases where it is possible to link C code directly, bindings libraries are still usually needed to make the manipulation of complex types more convenient.

For types such as strings, the size of values also brings performance concerns. In many cases the internal representation used for strings is the same as used in C, so an option is to simply pass to the C code a pointer to the address where the string is stored, which avoids copying of data, under risk of allowing the C code to modify the contents of the string. Exposing to C code pointers to memory areas within the runtime environment of the other language may also bring concurrency problems, in case the environment uses multiple threads.

When exposing data of structured types to C, the conversion to a native C type, in many cases, is not an option. Structured types in C are defined statically, therefore not serving to represent conveniently data of dynamic structures, such as objects that may gain or lose attributes or even change class during runtime. Even in languages with static typing, like Java, copying objects is not usually an interesting option due to the volume of data. Copying of structured objects tends to be restricted to specific operations such as manipulation of arrays of primitive types.

The alternative to allowing C code to operate over structured data, thus, is to provide an API that exposes the operations defined over those types as C functions. This also avoids the need to control the consistency between two copies of a given structure. Consistency problems, however, may occur if the API allows the C code to store pointers to objects from the language – this makes it necessary for the programmer to manage explicitly the synchronicity between pointers and the life cycles of objects that may be subject to garbage collection.

3.2. Garbage collection

From the moment when C code gains access to handles to data from the storage space of another language, the programmer must take into consideration the differences between the memory management models involved. For example, the C program may deallocate an object referenced by data in the scripting language, or the scripting language may remove an element from a structure causing it to be collected.

It is necessary, then, to indicate in C that the data remain accessible from it and must not be collected. In a complementary way, when transferring the control of C objects to the domain of the other language – for example, when storing them in a data structure of the other language – it is necessary to indicate to the language how to deallocate the memory of the structure when the garbage collector detects that it is no longer in use. The way how the API will provide this functionality depends not only on the design of the C API, but also on the garbage collection mode employed by the implementation of the language.

3.3. Function calls and registration

When bridging C and a scripting language, it is necessary to provide a form of invoking, from C, functions to be executed by the scripting language, and vice-versa. This combines the issues of data transfer, for passing arguments and receiving results between these two “spaces”, and the implications that this brings about the objects’ lifetime, affecting garbage collection. The tasks involved are always the same – perform conversion of input data, pass parameters to the other language, specify which function to call, obtain return values, convert them back to the other language – but approaches employed in scripting

language APIs vary widely. In the next section we will discuss how some APIs implement these tasks and the impact of their design on their usability as extension and extensible languages.

Because of the static typing of C, it is not possible to use a transparent syntax for calling functions registered at runtime. It is therefore necessary to define an API of functions for performing calls to the scripting language. Conversely, to allow the invocation of C functions from code written in a scripting language, its API must provide a way to register these functions in the execution environment. In statically typed languages, such as Java, to make it possible to call external functions using the same syntax as native calls, the set of external functions must be declared *a priori* in some way. On the other hand, in dynamically typed languages, functions can be used directly; defining them at some point in time before their call is sufficient. This way, one can declare external functions at runtime through C code using the scripting language API.

4. Scripting language API designs

A pioneering example of an embedded, extension language is Tcl [Ousterhout 1994]. Four main goals were set in its original design [Ousterhout 1990]: focus as a command language (designed to write short programs); extensibility; simplicity in its implementation; simple interface with C applications. We observe in those goals principles that are now understood as fundamental features of extensible and extension languages: extensibility was listed as a goal explicitly; the last two goals point out its focus as an extension language.

Aiming to simplify the interaction with C code, Tcl uses strings as its single data type. This minimalism, which has shown to be an advantage for Tcl as an extension language, makes it seem limited compared to languages like Python, which provide a more complete feature set as an extensible language. Scripting languages have grown beyond Tcl's focus as a command language, and thus, Tcl gradually lost space in the scripting world. Its historical importance, however, is undeniable: it was the concept introduced by Tcl of implementing scripting languages as C libraries that pushed strongly the development of extensible applications.

In this section, we discuss the design of the C APIs of four popular scripting languages, Python [van Rossum 2006b], Perl [Wall et al. 2000], Ruby [Thomas and Hunt 2004] and Lua [Ierusalimschy 2006], in terms of the interaction issues outlined in the previous section, while also contrasting them with the C API of Java [Gosling et al. 2000]. Unlike the others, Java uses static typing – which allows us to observe how typing affects the design of an API – but like them it is based on a virtual machine model, features automatic memory management and allows dynamic loading of code, and most importantly, it can both be embedded as an extension language and be extended with native C code.

4.1. Data transfer

The basic set of functions for manipulating data in scripting language APIs is usually the same: they provide functions for converting values from the language to basic C types and vice-versa. A central design issue lies in how to represent a value between languages. All values in the Python virtual machine are represented as objects, mapped to

the C API as the `PyObject` structure [van Rossum 2006a]. More specific types such as `PyStringObject`, `PyBooleanObject` and `PyListObject` are `PyObject`s by structural equivalence, that is, they can be converted through a C cast. Similarly, in Ruby, the API defines a C data type called `VALUE`, which represents a Ruby object. `VALUE` may represent both a reference to an object (that is, a pointer to the Ruby heap) as well as an immediate value. In particular, the constants `Qtrue`, `Qfalse` and `Qnil` are defined as immediate values, allowing them to be compared in C using the == operator. Perl also provides handles to its data in C, but these C values are better understood as containers to Perl values: types of Perl variables are mapped to C structs `SV` for scalars, `AV` for arrays, `HV` for hashes. A scalar variable in Perl has an `SV` associated to itself; however, one can create in C an `SV` that is not associated to any Perl variable name.

Lua, in contrast, employs a different approach for manipulating data in C: no pointers or handles to Lua objects are ever exposed to C code, and instead, operations are defined in terms of indices of a virtual stack. So, data transfer from C to Lua takes place through functions that receive C types, convert them to Lua values and stack them. While this results in the simplest and most orthogonal data manipulation API among the ones mentioned, code in which values are associated to stack indices tends to be less natural-looking than code using C variables – the manipulation of, say, a Ruby `VALUE` is syntactically similar to that of other C types: an assignment to a `VALUE` is done in C with an assignment.

All of these languages also offer API functions for manipulating their fundamental structured types (tables in Lua, arrays and hashes in Ruby and Perl, lists and dictionaries in Python). Python, in particular, defines an extensive function API for operations on its built-in classes; most of these functions could be performed using the generic API for method invocation, but they are offered directly in C as a convenience. In Java, static typing reduces greatly the need for explicit data conversion in C code. The Java Native Interface (JNI) [Sun 2003] defines C types equivalent to each of Java's primitive types (`jint` for `int`, `jfloat` for `float`, and so on). While to return an integer to Python from C one would have to use a command such as `return PyInteger_New(42)`, when interfacing Java they could simply write `return 42`. Reference types, such as classes and objects, are exposed to C as opaque references, instances of `jobject`. On the other hand, treatment of multi-threading complicates the access of types such as strings and arrays.

An important task when bridging C code to a scripting language is the creation of data in the scripting language environment containing C structures. Perl, Ruby and Lua provide simple mechanisms for this task. Ruby offers the `Data_Wrap_Struct` macro which receives a C structure and returns a Ruby `VALUE`. Lua defines a basic type in the language especially for this end, called *userdata*, which contains a memory block managed by the Lua VM that is accessible to C code but is an opaque object when accessed from Lua. In Perl, one can create `SV`s containing arbitrary memory blocks for use in C. In Python, the process is not as straightforward. Creating a Python class from C involves declaring parts of it statically and other parts dynamically, being usually necessary to define three different C structures, which are closely tied to the implementation of the Python VM. The complexity of code that interacts with C data types using the Python API tends to be less problematic in an isolated piece of code such an extension module

(which is typically centered around the declaration of these types and their methods) than when inserted in a larger body of code, as it happens with an embedded interpreter. Using the JNI, it is not possible to create new Java types from C; one can only load precompiled classes.

Another common need when interacting with C is to store pointers in the data space of the scripting language. Python, Lua and Perl offer features to do this directly. In Python, a `PyCObject` is a predefined type that holds a void pointer accessible from C. Lua offers a built-in type for this end, *light userdata*, which differs from *userdata* in that the memory block it points to is not managed by the virtual machine. In Perl, the same can be achieved storing a pointer in the data area of an `SV`. In Ruby and Java, there is no direct way to store pointers. The alternative is to convert pointers and store them as numbers. In fact, this happens internally in the implementation of Ruby, and the portability limitations of this approach are made evident by the fact that the compilation of Ruby fails if `sizeof(void*) != sizeof(long)`.

4.2. Garbage collection

Garbage collection aims to isolate, as much as possible, the programmer from memory management. This way, ideally an API should also be as independent as possible from the garbage collection algorithm used in the implementation of the virtual machine. Perl and Python perform garbage collection based on reference counting, and this shows through in the reference increment and decrement operations frequently needed during the use of their APIs.

Ruby uses a mark-and-sweep garbage collector. Its API manages to abstract this fact well for manipulation of native Ruby objects, but the implementation of the collector is evident in the creation of Ruby types in C, where we need to declare a mark function when there are C structures that store reference to Ruby objects. The Lua API goes further when isolating itself from the implementation of the garbage collector: the only point of the API where the use of an incremental garbage collection is apparent is in the routine for direct interaction with the collector, `lua_gc`, where its parameters can be configured.

Of the five languages discussed in this work, the only one whose API abstracts entirely the implementation of the garbage collector is Java. The only interfacing operation provided by the language, `System.gc()`, does not receive any arguments and does not specify how or when the collection should be done[1]. Indeed, the various available implementations of the JVM use different algorithms for garbage collection. We observe, then, that while most languages abstract the specifics of the garbage collector, details of the garbage collection algorithm tend to show up in the APIs. Since in pragmatic terms the API compatibility of an implementation is as important as language compatibility, this means that, due to the API, language implementations end up tied to specific garbage collection algorithms because of their API even if they are transparent to the language itself.

Another issue that arises in the communication between C and scripting languages is the management of references. For manipulating data through the API, Lua and Ruby demand the least concerns from the programmer about managing references. Lua avoids

[1]The documentation is purposely vague, stating only that this method "suggests that the Java Virtual Machine expend effort toward recycling unused objects".

the problem altogether, by keeping its objects in the virtual stack and not returning references to C code; accessing data from Lua, thus, always involves function calls for getting the data into the stack.

In Ruby, only objects stored in C globals and not referenced from Ruby need to be notified, using the `rb_global_variable` function; objects in the local scope of a C function do not need to be notified. The way how Ruby ensures the validity of local `VALUE`s is remarkably peculiar: when performing the mark phase, the garbage collector scans the C stack looking for values that look like `VALUE` addresses, that is, numeric sequences that correspond to valid `VALUE` addresses. These addresses can be identified because objects are always allocated within heaps maintained by the Ruby interpreter. Each `VALUE` found in the stack is then marked. This ensures that any `VALUE` locally accessible by C code becomes invalidated, but may generate "false positives" stopping data that could be collected from being so.

In spite of programmer convenience, such approach is extremely non-portable. The implementation of the garbage collector in Ruby 1.8.2 has `#ifdefs` for IA-64, DJGPP, FreeBSD, Win32, Cygwin, GCC, Atari ST, AIX, MS-DOS, Human68k, Windows CE, SPARC and Motorola 68000. Besides, the collector forces the discharge of registers to the stack using `setjmp`, to prevent variables of the `VALUE` type that may have been optimized into registers by the compiler from being missed.

Both Perl and Java handle the issue of references stored in local variables in a similar way, by distinguishing references as either global or local (local references are called "mortal variables" in Perl). Local references allow for mostly implicit management. API functions in Java return local references by default, which can be converted to global ones with the API call `NewGlobalRef`. In Perl, the opposite happens, and global references can be converted to local ones with the `sv_2mortal` function. Java's approach is more interesting, as normally more locally-scoped than globally-scoped variables are used.

4.3. Function calls

In Python, Lua and Perl, functions can be accessed as language objects and invoked. Python allows any `PyObject` to be called as a function, as long as they implement the `__call__` method, which can be written in either Python or C (as a function registered in the object's `PyTypeObject` struct). Like in data manipulation, Python offers an extensive API, with several convenience functions allowing parameters to be passed as Python tuples, as Python objects given as varargs, as C values to be converted by the invocation function, etc. In Lua, there is a built-in primitive type, *function*, which represents both Lua functions and C functions registered in the Lua VM. Perl also allows functions to be manipulated as first-class objects using its C API, returning SV structs representing them.

In Ruby as well as Java, methods are not first-class objects, and therefore their APIs define specific C types used to reference them – `jmethodID` in Java and `ID` in Ruby[2]. Java also offers a large number of method invocation functions and, due to static typing, input parameters can be passed as varargs in a direct way, without having to spec-

[2] An ID is merely a reference to the symbol table entry corresponding to the method's name, and not a unique identifier for the method itself.

ify how their conversion should be made. Ruby also offers some variants of invocation functions.

Lua separates the function call routine from argument passing, which is done in a previous step by setting up the contents of the stack. This is a very simple solution, but the resulting code is less clear than the equivalent calls in languages such as Ruby and Python, in which arguments to the function call are written in C as arguments to the C API call. Perl also features function calls using a stack model, but its use is exceedingly complex, demanding a macro protocol to be followed which exposes the internal workings of the interpreter [Marquess 2006]. Another complicating factor is the handling of return values, for these vary according to the Perl context in which the function is called.

In Lua and Python, the occurrence of errors can be checked through the function's return value. In a similar way, Perl allows detecting errors in the most recent call checking a special variable, `$@`; in Java, this is done calling an API function. In Ruby, error handling is more convoluted: the API offers a function for invoking C functions in protected mode, but lacks an equivalent for calling Ruby functions. It is necessary to write a wrapper function in those cases.

4.4. Registration of C functions

Python and Ruby offer to the programmer various options for C function signatures that are recognized by the API, which is practical, given that this way one can choose different C representations for the input parameters (collected in an array, obtained one by one, etc.) according to their use in the function. Lua offers only one possible signature for C functions to be registered in its virtual machine, as arguments are passed through the stack and not as arguments to the C function.

In Java, function signatures are created through the `javah` tool [Liang 1999] – due to its static type system, types of input parameters passed by Java are converted automatically by the JNI, which is very convenient as it avoids explicit operations for conversion and type checking in the function. Because of their dynamic type systems, the other languages offer specific API functions for performing these checks.

The interface between Perl and C was designed having in mind that the connection between C functions and the Perl interpreter is made through generated code from a description given in a higher-level language, XS [Roehrich 2006]. Instead of isolating the access to Perl's internals through a public API, the proposed approach is to encapsulate the process of generating wrapper code using interfaces written in `.xs` files. These files contain C code along with annotation that simplifies the handling of input and output parameters. In fact, Perl does not expose a documented API for registering functions [Okamoto and Roehrich 2006]. Because of that, it is not practical for an application to embed a Perl interpreter and expose it to a set of C functions using C code only. The alternative is to write a Perl extension using XS and import the resulting package in the embedded Perl interpreter.

Registration of functions in Ruby and Lua is simple. In Lua, in particular, it is an assignment (made through API calls), not different from any other object. In Python, there are features for batch registering, using arrays of the `PyMethodDef` struct (Lua offers a similar feature with `luaL_register` function), but there is no simple way to register a single function – again, this shows a focus on extending rather than embedding:

extension modules tend to register many functions at once, while embedded interpreters often register global functions. Both in Java and Perl, function registration is done implicitly by the generation tools, and there are no public API functions for registering new C functions at runtime in either of them.

5. Conclusion

Choosing a scripting language depends on a series of factors, many of them relative to the language itself, others relative to its implementation. When we deal with multi-language development scenarios, an aspect that should not be neglected is the design of interfaces between languages. Be it extending the scripting language through C code, or making a C application extensible through a scripting language, the API offered by the language has a fundamental role, often influencing the design of the application.

Although the same general problems, such as data transfer, function registration and calling, are common to different usage scenarios of a scripting language API, applications embedding a virtual machine tend to demand more from the API than libraries implementing extension modules. This point is illustrated by the difficulties imposed by the Python API both in the access to global variables and registration of global functions; and, more evidently, by the complexity of Perl's API for function calls.

The fact that the Python API makes the use of global variables and functions difficult, favoring the use of modules, can be justified as a way to promote a more structured programming discipline. This is interesting when using the API for developing extension modules, given that using global variables and functions is extremely harmful in those cases, as it would pollute the namespace of Python applications. For the case where the language is embedded to provide scripting support for a C application, the absence of a convenient way to define global functions in the scripts' namespace is questionable.

The approach adopted by Perl, using a pre-processor which generates automatically code for converting data when passing parameters and return values, has shown to be inadequate for scenarios involving embedded interpreters. Although the use of a pre-processor simplifies the simpler cases of declaration of C functions, the lack of a well-defined API for handling data transfer between the Perl interpreter and C code becomes apparent in more elaborate cases.

Interesting observations resulted from the comparison of the Java API with that from the other four scripting languages, given that, although it shares several traits with those languages, Java is not considered a scripting language. While static typing does reduce considerably the need for explicit data conversion in C code for primitive types of the language, in practice type checking for objects and the linking of fields and methods happens in a dynamic way, as these have to be performed at runtime by the JNI. Thus, regarding interaction of the virtual machine with C, advantages brought by static typing are reduced. Besides, dynamic resolution of fields and methods through C has subtle differences in behavior when compared to what occurs in native Java code, which can be a source of programmer errors.

Throughout the development of this work, we implemented as a case study a C library called LibScript[3], which provides an extensibility architecture for applications in

[3] `http://libscript.sourceforge.net`

a language-independent manner: scripting language VMs are loaded dynamically as LibScript plugins. In the implementation of these plugins, we had a chance to exercise the APIs of the different scripting languages performing similar tasks.

The disparity between languages with regard to the availability of documentation deserves mention. Java, Python and Lua feature extensive documentation, both for the languages themselves and to their C APIs. For those languages, we were able to largely base our study and the implementation of the case study on the provided documentation. The documentation of Ruby relative to its C API is sparser; in [Thomas and Hunt 2004] only part of its public API is covered. One has to make use of undocumented functions for tasks as fundamental as freeing global references registered through C.

The balance between simplicity and convenience is another recurring theme when comparing APIs. Python's extensive API, containing 656 public functions, contrasts with the 113 functions exposed by the Lua API (79 from the core API, 34 in its auxiliary API). In many situations, Python API functions abbreviate two, three ore even more calls, as in the case of powerful functions such as `Py_BuildValue` and `PyObject_CallFunction`, resulting in short and readable C code. The approach defended by Lua is that of a minimalistic API, offering mechanisms with which more elaborate functionality can be built. In fact, in [Ierusalimschy 2006] a C function equivalent to `PyObject_CallFunction` is presented, using the Lua API.

Ruby exports 530 functions in its header and Perl 1209, but as only a small fraction of those is documented, it is hard to evaluate the size of their "public API" and how many of these are just functions for internal use exposed in their headers[4]. This also shows that the documentation is not only relevant as support material for development, but it also indicates how well-defined an API is.

The Java API is well-documented, like that from Python and Lua, but the number of exported functions is not a good parameter for comparison with the other APIs as, because of its statically defined types, many functions have a variant for each primitive type. Java exports its API as a structure containing function pointers; 228 functions in total are exported in this structure.

An aspect that is equally important when extending or embedding is the concern on not polluting the C namespace. Python, Java and Lua define all its functions and C types with prefixes that aim to avoid conflicts with other names, which in the case of embedding are defined by the application, and in the case of extending are defined by the library being exposed to the language. Perl and Ruby define names in a disorganized fashion, which occasionally causes problems. Perl has options to disable a series of macros and force a common prefix in its functions, but this feature is incomplete and using it hampers the functionality of its headers.

Another point that could be observed in this work is that the consistency of an API depends greatly on the consistency of the language it exposes. Constructions where a language lacks orthogonality, such as code blocks in Ruby or the differences when manipulating scalar and array values in Perl, end up increasing the complexity of the API and demand from the programmer specific handling in C code.

[4]Some functions are marked as being for internal use only, but most of them have no indication whatsoever.

The focus in extending or embedding adopted by a language's C API has as much impact in its suitability for one or other scenario as the design of the language itself. The interaction between the design of the language, its implementation and its API all affect each other in often subtle ways – APIs like those from Lua and Java, which allow multiple interpreters to run concurrently, show a design concern on embedding, while those from Perl, Python and Ruby focus on providing facilities to make it easier to write extension modules. Given that an API designed towards embedding also encompasses the needs of APIs for extension modules, and that module generation tools such as SWIG [Beazley 1996] (as well as language-specific tools such as [Ewing 2006, Niemeyer 2006, Manzur and Celes 2006]) are becoming increasingly powerful and popular, we observe that a C API aiming to support both extension and embedding should focus on the latter, as that tends to demand more from both the API and the language implementation.

References

Beazley, D. M. (1996). SWIG: an easy to use tool for integrating scripting languages with C and C++. In Association, U., editor, *4th Annual Tcl/Tk Workshop '96*, pages 129–139, Berkeley, CA, USA. USENIX.

Ewing, G. (2006). Pyrex - a language for writing Python extension modules. `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`.

Finne, S., Leijen, D., Meijer, E., and Jones, S. P. (1998). H/Direct: a binary foreign language interface for Haskell. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 153–162, New York, NY, USA. ACM Press.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2000). *The Java Language Specification*. Addison-Wesley, Boston, MA, USA, 2nd edition.

Ierusalimschy, R. (2006). *Programming in Lua*. Lua.org, 2nd edition.

Liang, S. (1999). *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Manzur, A. and Celes, W. (2006). toLua++ reference manual. `http://www.codenix.com/~tolua/tolua++.html`.

Marquess, P. (2006). *perlcall(1)*. Perl 5 Porters, 5.8.8 edition. `http://perldoc.perl.org/perlcall.html`.

Niemeyer, G. (2006). Lunatic Python. `http://labix.org/lunatic-python`.

Okamoto, J. and Roehrich, D. (2006). *perlapi(1)*. Perl 5 Porters, 5.8.8 edition. `http://perldoc.perl.org/perlapi.html`.

Ousterhout, J. K. (1990). Tcl: An embeddable command language. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 133–146, Berkeley, CA. USENIX Association.

Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit*. Addison Wesley.

Ousterhout, J. K. (1998). Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30.

Roehrich, D. (2006). *perlxs(1)*. Perl 5 Porters, 5.8.8 edition. `http://perldoc.perl.org/perlxs.html`.

Sun (2003). *Java Native Interface 5.0 Specification*. Sun Microsystems, 5.0 edition. `http://java.sun.com/j2se/1.5.0/docs/guide/jni/`.

Thomas, D. and Hunt, A. (2004). *Programming Ruby: The Pragmatic Programmer's Guide*. Addison Wesley Longman, Inc., Boston, MA, USA, 2nd edition.

van Rossum, G. (2006a). *Extending and Embedding the Python Interpreter*, 2.4.3 edition. `http://docs.python.org/ext/ext.html`.

van Rossum, G. (2006b). *Python Reference Manual*. Python Software Foundation, 2.4.3 edition. `http://docs.python.org/ref/`.

Wall, L., Christiansen, T., and Orwant, J. (2000). *Programming Perl*. O'Reilly, 3rd edition.

Higher-Order Lazy Functional Slicing*

Nuno F. Rodrigues and Luís S. Barbosa
{nfr, lsb}@di.uminho.pt

DI-CCTC, Universidade do Minho
4710-057 Braga, Portugal

Abstract. Program slicing is a well known family of techniques intended to identify and isolate code fragments which depend on, or are depended upon, specific program entities. This is particularly useful in the areas of reverse engineering, program understanding, testing and software maintenance. Most slicing methods, and corresponding tools, target either the imperative or the object oriented paradigms, where program slices are computed with respect to a variable or a program statement.
Taking a complementary point of view, this paper focuses on the slicing of higher-order functional programs under a lazy evaluation strategy. A prototype of a Haskell slicer, built as proof-of-concept for these ideas, is also introduced.

1 Introduction

Introduced by Weiser [13, 11, 12] in the late Seventies, program slicing is a family of techniques for isolating parts of a program which depend on or are depended upon a specific computational entity referred to as the *slicing criterion*. In Weiser's view, program slicing is an abstraction exercise that every programmer has gone through, aware of it or not, every time he undertakes source code analysis.

Weiser's original definition has been since then re-worked and expanded several times, leading to the emergence of different methods for defining and computing program slices. Despite this diversity, most of the methods and corresponding tools target either the imperative or the object oriented paradigms, where program slices are computed with respect to a variable or a program statement.

Weiser approach corresponds to what would now be classified as a *backward*, *static* slicing method. A dual concept is that of *forward slicing* introduced by Horwitz et al [3]. In forward slicing one is interested on what depends on or is affected by the entity selected as the slicing criterion. Note that combining the two methods also gives interesting results. In particular the union of a backward to a forward slice for the same criterion n provides a sort of a selective window over the code highlighting the *region* relevant for entity n.

Another duality pops up between *static* and *dynamic* slicing. In the first case only static program information is used, while the second one also considers input values [4, 5] leading frequently, due to the extra information used, to smaller and easier to analyse slices, although with a restricted validity.

Taking a different perspective, this paper focuses on the problem of slicing *functional* programs [1]. Since slicing is a technique intended to be used by programmers while developing, analyzing or transforming source code in a production context, it should target real languages and in a most complete way. Otherwise, such techniques would be useless by failing to stand up to the expectations of their natural clients *i.e.*, programmers and software analysts. Thus, our approach to functional slicing targets an emerging programming paradigm: *functional programs with higher-order constructs sharing a lazy strategy evaluation.* Additionally it will be shown in section 7, that the strict version of the proposed technique can be easily derived from the lazy one, and that the removal of higher-order constructs represents a trivial simplification of the method introduced here.

* The research reported in this paper is supported by FCT, under contract POSI/CHS/44304/2002, in the context of the PURe project.

By the beginning of this work we thought that the development of higher-order lazy functional slicer represented a more or less straightforward engineering problem that could be easily solved by making use of some combination of parsing and syntax tree traversal operations. However, all attempts to build such a tool resorting to a direct implementation of these operations invariantly ended by the discovery of some particular case where the resulting slices did not correspond to the expected correct ones. Even more, by performing minor changes in the implementations in order to correctly cover some special cases, one often ended up introducing new problems or preventing the treatment of other special cases.

Soon, however, we realized the problem complexity had been underestimated from the outset. This lead to the development of a semantic-based approach, providing a suitable level of abstraction, in which the lazy slicing problem could be specified and solved. Furthermore, the formed framework makes it possible to state and verify relevant properties of the slicing process.

The research background of this paper amounts to the development of HaSlicer [1], a functional slicer targeting the functional language Haskell [1]. Before developing HaSlicer it was decided to pay special attention to high order entities, somehow related to an architectural view over functional systems. Thus, HaSlicer deals with code entities such as modules, data-types and functions, ignoring completely more fine grained entities like functional combinator expressions. Although the success of this decision is attested by the system high level views given by the Functional Dependency Graph visualizer [10], there was still a lack of proper foundations and techniques for what may be called low level slicing of functional programs. This paper is a step in that direction.

The context for this research is a broader project on *program understanding and re-engineering* of legacy code supported by formal methods. Actually, if forward software engineering can be regarded as an almost lost opportunity for formal methods (with notable exceptions in areas such as safety-critical and dependable computing), reverse engineering looks more and more a promising area for their application, due to the engineering complexity and exponential costs involved. In a situation in which the only quality certificate of the running software artifact still is life-cycle endurance, customers and software producers are little prepared to modify or improve running code. However, faced with so risky a dependence on legacy software, managers are more and more prepared to spend resources to increase confidence on — *i.e.*, the level of understanding of — their code.

The paper is organised as follows. Section introduces the functional language FL, a "sugared" λ-calculus used to express our programs. Section 3 discusses the relationship between slicing and evaluation and justifies the use of a semantic approach to reason about slicing of functional programs. Sections 4 and 5 present two algorithms for performing lazy functional slicing. In section 6 a strict version of the slicing algorithm presented in section 5 is discussed. Section 7 resorts to the semantics presented in sections 5 and 6 to prove that lazy slices are smaller than or equal to their strict counterparts. Finally section 8 concludes and discusses topics for future work.

Contributions. We formally introduce a dynamic slicing algorithm for higher-order lazy functional languages which, to the best of our knowledge, is a first attempt to address slicing for this kind of programs. It is also shown how the same formal setting can be used to state and prove slicing properties. The whole approach is prototyped in a library developed for Haskell a pure higher-order lazy functional language, as a proof-of-concept[2].

2 Related Work

While we regard this work as a first incursion on higher-order lazy functional slicing, there are a number of related works that should be mentioned.

In [8] Reps and Turnidge provide a static functional slicing algorithm but, in contrast to our approach, theirs target first-order strict functional programs. Besides considering a different language class (first-order) and a different evaluation strategy (strict), the authors define slicing

[1] The tool is available online at http://labdotnet.di.uminho.pt/HaSlicer/HaSlicer.aspx
[2] The library is available online at http://alfa.di.uminho.pt/∼nfr/Tools/Tools.html

criteria by means of projection functions, a strategy that we regard as more rigid when compared to our own approach which resorts to a subexpression labeling mechanism.

In [7] the authors present a strategy to dynamically slice lazy functional languages. Nevertheless, they leave higher-order constructs as a topic for future work, and base their approach on redex trails. This leads to a slicing criterion definition (which consists of a tuple containing a function call with full evaluated arguments, its value in a particular computation, the occurrence of the function call and a pattern indicating the interesting part of the computed value) which is much more complex to use in practice than our own. The latter, by pointing out a specific (sub)expression in the code, represents a more natural way for the analyst to encode the relevant aspects of the code that he/she wants isolated.

Perhaps the work most related to ours is [2], where the author presents an algorithm for dynamic slicing of strict higher-order functional languages followed by a brief adaptation of the algorithm to lazy evaluation. A major difference with the approach proposed in their paper is that, recursive calls must be explicitly declared in the language and there is no treatment of mutual recursive functions which, as pointed out by the author, results in a considerable simplification of the slicing process. Again, we believe that our definition of the slicing criterion is more precise than the one used in [2], which consists of the value computed by the program in question (even though more flexible slicing criteria are briefly discussed).

Finally, it should be emphasized that a slicing criterion, like the one we propose, that permits to choose any (sub)expression of the program under analysis, deeply influences and augments the complexity of the slicing process, specially under a lazy evaluation framework like the one we address. In fact, this aspect is the responsible for the evolution of the slicing algorithm from a one phase process, like the one presented in section 5, to a two phase process where one must first keep track of internal (sub)expression lazy dependencies before calculating the final slicing with respect to the relevant (sub)expressions.

3 The Functional Language

Given that one is not interested on focusing on a single functional language, but rather to come up with a technique that is potentially applicable to all higher-order lazy functional languages, one has decided to introduce a common level functional language which can easily serve several functional programming language implementations.

The process of choosing such a syntax had to fulfill two main requisites. The language could not be excessively broad since this would introduce an unnecessary notational burden in the representation. On the other hand it could not be excessively small because this would make translations from/to real functional languages too complex to achieve.

Values	z	$::= (\lambda x.e)$	
		$\mid (C\ x_1 \cdots x_a)$	$a \geq 0$
Expressions	e	$::= z$	
		$\mid e\ x$	
		$\mid x$	
		$\mid$ `let` $x_n = e_n$ `in` e	$n > 0$
		$\mid$ `case` e `of` $\{(C_j\ x_{1j} \cdots x_{aj}$ `->` $e_j\}_{j=1}^{n}$	$n > 0,\ a \geq 0$
Programs	*prog*	$::= x_1 = e_1, \ldots, x_n = e_n$	

Fig. 1. The FL syntax

Thus, one had to find a tradeoff between this conditions to make the entire process feasible. Such a tradeoff is captured in language FL where the syntax is presented in figure 1. FL notation is basically a λ-Calculus enriched with `let` and `case` statements. It introduces the domain U of

values, the domain E of expressions, the domain P of programs and the domain of V of variables. Note that values are also expressions by the first rule in the definition of expressions.

A very important detail about the language in figure 1 is that functional application cannot occur between two arbitrary functional expressions, but only between an expression and a variable previously defined. In practice this implies that at evaluation time the applied expression must have been previously added to the heap so that it can be used on a functional application. This requisite may seem strange for now, but it is necessary to deal correctly with the semantics upon which we define the slicing process.

It requires, however, some care when converting concrete functional programs to FL. In practice, the translation is achieved by the introduction of a new free variable with a let expression and the subsequent substitution of the expression by the newly introduced variable.

Of course, to treat real functional languages, some other straightforward syntactic translations are in demand. These includes the substitution of `if then else` by `case` expressions with the respective `True` and `False` values or the substitution of `where` constructions by `let`.

Such syntactic transformations have been implemented, as a prove of concept, in a front end functional language (Haskell). Even more, they were implemented isomorphically because by the end of the slicing process, one wants to be able to reconstruct the slice exactly like the original program except by the removal of some sliced expressions.

Finally, we have to uniquely identify the functional expressions and sub-expressions of a program, such that the slicing process refer to these identifiers in order to specify what parts of the program belong to a specific slice. Such identifiers, collected in a set L, are introduced by expression labeling as shown in figure 2.

Values	$z ::= (\lambda x : l_1 . e) : l$	
	$\mid\ (C\ x_1 : l_1 \cdots x_a : l_a) : l$	$a \geq 0$
Expressions	$e ::= z$	
	$\mid\ e\ (x : l') : l$	
	$\mid\ x : l$	
	$\mid\ \texttt{let}\ x_n = e_n : l_n\ \texttt{in}\ e : l$	$n > 0$
	$\mid\ \texttt{case}\ e\ \texttt{of}\ \{(C_j\ x_{1j} : l_{1j} \cdots x_{aj} : l_{aj}) : l'\ \texttt{->}\ e_j\}_{j=1}^{n} : l$	$n > 0,\ a \geq 0$
Programs	$prog ::= x_1 = e_1, \ldots, x_n = e_n$	

Fig. 2. Labeled FL syntax

For the moment, one may look at labels from L as simple unique identifiers of functional expressions. Latter, these labels will be used to capture information about the source language representation of the expression they denote, so that, by the end of the slicing process, one can be able to construct a sliced source code version of the program under analysis.

4 Slicing and Evaluation

Slicing of functional programs is an operation that largely depends on the underlying evaluation strategy for expressions. This can be exemplified in programs where strict evaluation introduces non termination whereas a lazy strategy produces a result. As an example, consider the following functional program.

```
fact :: Int -> Int
fact 0 = 1
fact k = k * fact (k-1)

ssuc :: Int -> Int -> Int
```

```
ssuc r y = y + 1

g :: Int -> [Int] -> [Int]
g x z = map (ssuc (fact x)) z
```

If we calculate the slice of the above program w.r.t. expression `g (-3) [1,2]`, taking into account that the program is being evaluated under a strict strategy, the evaluation will never terminate and the slice fails to compute.

On the other hand, under a lazy evaluation strategy, the evaluation is possible because `succ` is not strict over its arguments, and therefore `(fact x)` which introduces non terminating behaviour is not computed. Thus, slicing is now feasible and one would expect to obtain the following slice:

```
ssuc :: Int -> Int -> Int
ssuc r y = y + 1

g :: [Int] -> [Int]
g z = map (ssuc (fact x)) z
```

Note that strictly speaking the computed slice is not executable. Actually this would require definition of function `fact` in order to be interpreted or compiled. This was a deliberate choice because, in a functional framework, if one calculates executable slices (without using any further program transformation), it often happens that such slices take enormous proportions when compared to the original code. Nevertheless, and because the expressions that are sliced away do not interfere with the selected slicing criterion, a program transformation to be used for this case is to substitute the expression in question by some special value of the same type. In HASKELL, for instance, and because types have a cpo structure, one could use the bottom value (usually denoted by $\perp$) of the type in question to signal the superfluous expression. These and other possible code transformations that target the execution of slices are, however, beyond the scope of this paper.

The approach to low level slicing of functional programs proposed in this paper is mainly oriented (but see section 6) to lazy languages. Our motivation was that slicing has never been treated under such an evaluation strategy (combined with higher-order constructs). Moreover, intuition suggests, as in the example above, that lazy slices tend to be smaller than their strict counterparts.

Therefore, our starting point was a lazy semantics for FL introduced by Launchbury in [6], which is presented in figure 3. In this semantics, expression $\Gamma \vdash e \Downarrow \Delta \vdash z$ states that expression e under heap Γ evaluates to value z producing heap Δ as result.

In figure 3 and throughout the paper the following syntactic abbreviations are used: $\hat{z}$ standing for $\alpha-$conversion, $[x_i \mapsto e_i]$ for $[x_1 \mapsto e_1, \ldots, x_i \mapsto e_i]$, $\Gamma[x_i \mapsto e_i]$ to express the update of mapping $[x_i \mapsto e_i]$ in heap Γ and $e[x_i/y_i]$ for the substitution $e[x_1/y_1, \ldots, x_i/y_i]$.

5 Lazy Forward Slicing

We start by analyzing a simplified version of the more general problem of higher-order lazy functional slicing, which we have called *lazy print*. The calculation of this particular kind of slice is completely based on the lazy evaluation coverage of a program, without taking any extra explicit slicing criterion. This means that a *lazy print* calculation amounts to extracting the program fragment that has some influence on the lazy evaluation of an expression within that program. For an example, consider the following trivial functional program.

```
fst :: (a, b) -> a
fst (x, y) = x

sum :: [Int] -> Int
sum []    = 0
```

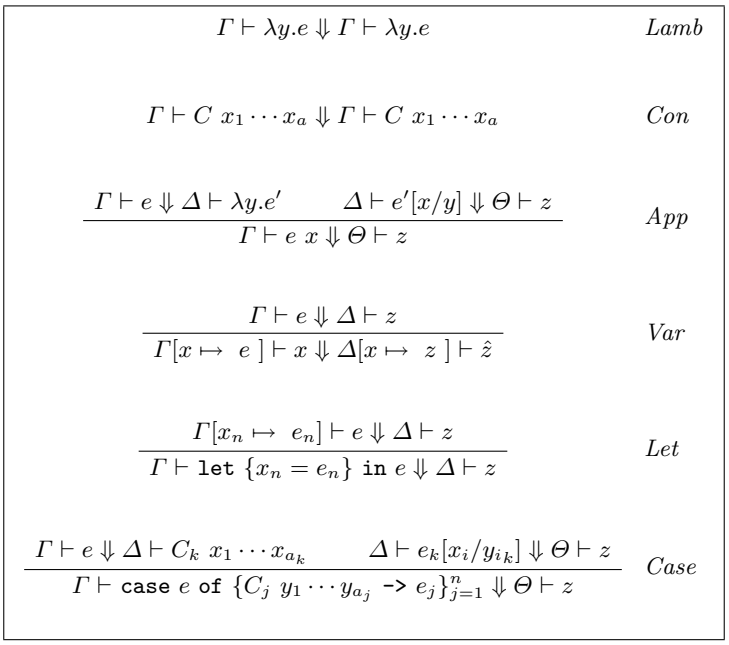


Fig. 3. Lazy Semantics

```
sum (h:t) = h + (sum t)

g :: ([Int], Int) -> Int
g z = sum (fst z)
```

The *lazy print* of this program w.r.t. the evaluation of `g ([], 3)` is

```
fst :: (a, b) -> a
fst (x, ) = x

sum :: [Int] -> Int
sum []    = 0


g :: ([Int], Int) -> Int
g z = sum (fst z)
```

Automating this calculation entails the need to derive an augmented semantics from the lazy semantics presented in figure 3. This extends Launchbury semantics with an extra output value of type set of labels (S), for the evaluation function $\Downarrow$. The purpose of this set S is to collect all the labels from the expressions that constitute the *lazy print* of a given evaluation. Motivated by implementation reasons, instead of using an alpha conversion in the original rule *Var*, we introduce a fresh variable in rule *Let* to avoid variable clashing.

The *lazy print* semantics uses two auxiliary functions, namely $\varphi : \mathrm{E} \times \mathrm{V} \rightarrow \mathbb{P}\,\mathrm{L}$ and $\mathcal{L} : \mathrm{E} \rightarrow \mathbb{P}\,\mathrm{L}$. Function φ collects the labels from all the occurrences of a variable in an expression and function $\mathcal{L}$ returns all the labels in an expression.

The intuition behind this augmented semantics is that it works by collecting all the labels from the expressions as they are being evaluated by the semantic rules. The only exception is rule *Let*, which does not collect all the expression labels immediately. This is explained by the fact that there is not sufficient information available when rule *Let* is applied to decide which variable

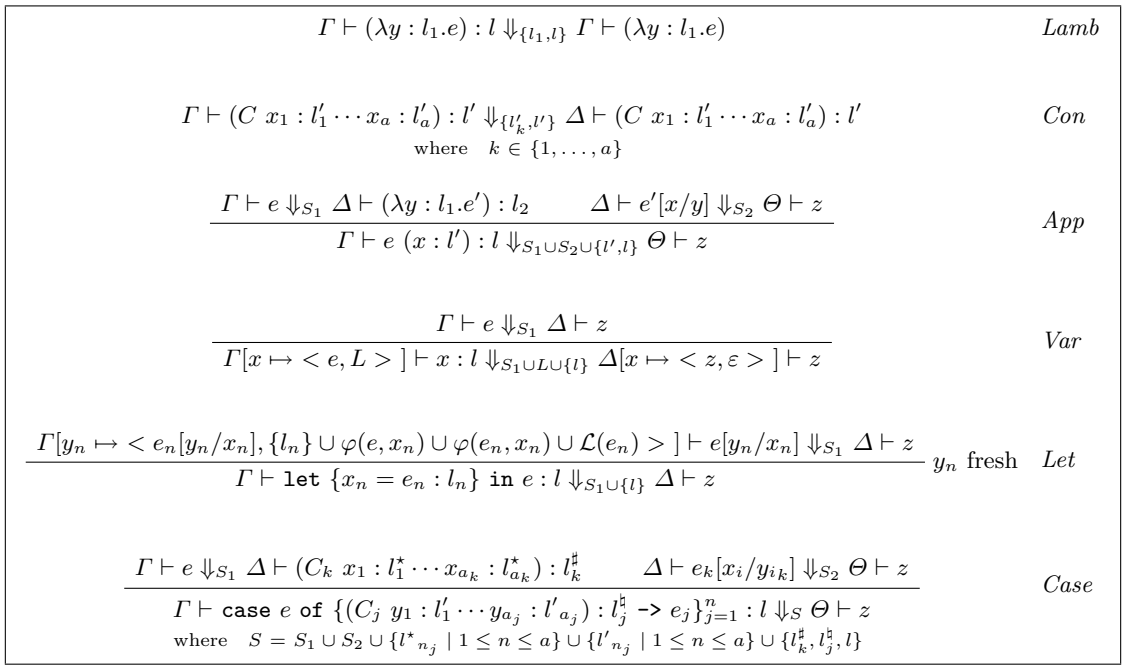


Fig. 4. Lazy Print Semantics

bindings will be needed in the remainder of the evaluation towards the computation of the final result. A possible solution for this problem is to have a kind of memory associating pending labels and expressions such that, if an expression gets to be used then not only their labels are included in the evaluation labels set, but also the pending labels that were previously registered in the memory.

A straightforward implementation of such a memory mechanism is the heap itself. Thus, by extending the heap from a mapping between variables and expressions to a mapping from variables to pairs of expressions and sets of labels, the semantics becomes able to capture the "pending labels" introduced by the *Let* rule.

A problem is spotted however in slices computed on top of the *lazy print* semantics given in figure 4. As an example, consider the following fragment that calls some complex and very cohesive functions `funcG` and `funcH` which do indeed contribute to the computation of the values in `x` and `y`:

```
f z w = let x = funcG z w
            y = funcH x z
        in (x, y)
```

When computing the *lazy print* of such a program, no matter what values are chosen for `z w`, the returned slice is always

```
f z w =

        (x, y)
```

as if the variables introduced by the `let` expression would have no effect on the result of the overall function, which completely contradicts what one already knew about the behaviour of functions `funcG` and `funcH`.

The reason for such a deviating behaviour induced by the lazy print semantics is explained by the *Con* rule. Because $C\ x_1 : l_1 \cdots x_a : l_a$ expressions are considered primitive in the language, the

Con rule simply collects the outer labels of such expressions and returns the expression exactly as it was received.

Indeed this explains the odd behaviour of the above example, where function f returns a pair which falls into the $C\ x_1 : l_1 \cdots x_a : l_a$ representation in FL. Therefore, the only semantic rule being applied during the lazy print calculation was the *Con* rule which does not evaluate the constructor (Pair) arguments and their associated expressions. Thus, one may now understand why the only labels that the semantics yields during the evaluation are the ones visible at the time of application of the *Con* rule.

A possible approach to solve this problem of extra laziness induced by the semantics would be to evaluate every data constructor parameter in a strict way. This, however, would throw away most of the lazy motto of the semantics since the evaluation would become strict on every data type.

A much more effective solution is to divide the slicing calculation into two phases. The first phase uses the semantics from figure 4. The second one takes the value and the heap returned by the first phase and passes them to a processor which restates to a semantics similar to the one used in the first phase except for rule *Con* which is substituted by the one from figure 5.

$$\frac{\Gamma[x_k \mapsto < e_k, L_k >] \vdash x_k \Downarrow_{S_1} \Delta \vdash z_k}{\Gamma[x_k \mapsto < e_k, L_k >] \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l' \Downarrow_S \Delta \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l'} \quad Con$$
$$\text{where} \quad k \in \{1, \ldots, a\}$$
$$S = L_k \cup \{l'_k, l'\} \cup S_1$$

Fig. 5. Con Rule for Strict Evaluation of the Result Value

This way, constructor strict evaluation is introduced only over the resulting value, leaving all intermediate values being evaluated as lazy as possible.

6 Lazy Forward Slicing with Slicing Criterion

However, despite the relevance of the *lazy print* in, e.g., program understanding, a further step towards effective slicing techniques for functional programs requires the explicit consideration of slicing criteria. In this section, we present an approach where slicing criteria is specified by sets of program labels.

The slicing process proceeds as in the previous case, except that now one is interested in collecting the program labels affected not only by a given expression, as before, but also by the expressions associated to the labels introduced by the user as a slicing criterion.

A first and straightforward approach to implement a slicer with such a slicing criterion involves taking into account the set of collected labels on both the output and the input of the evaluation function $\Downarrow$. Therefore, the semantic rule for λ-expressions changes to the one presented by the rule of figure 6.

$$S_i, \Gamma \vdash (\lambda y : l_1.e) : l \Downarrow \Gamma \vdash (\lambda y : l_1.e) : l, S_f \quad Lamb$$
$$\text{where } S_f = S_i \cup \bigcup \{\varphi(e, y) \mid l_1 \in S_i\} \cup \{l \mid l_1 \in S_i\}$$

Fig. 6. Improved Semantics

This extra rule enables the semantics to evaluate expressions taking into account a set of labels S_i supplied as a slicing criterion and its impact on the resulting slice S_f. Putting it in another

$$\frac{}{\Gamma \vdash (\lambda y : l_1.e) : l \Downarrow_F \Gamma \vdash (\lambda y : l_1.e) : l} \quad \textit{Lamb}$$
where $F = [l_1 \mapsto \varphi(e, y) \cup \{l\}]$

$$\frac{}{\Gamma \vdash (C\ x_1 : l_1 \cdots x_a : l_a) : l \Downarrow_F \Gamma \vdash (C\ x_1 : l_1 \cdots x_a : l_a) : l} \quad \textit{Con}$$
where $k \in \{1, \ldots, a\}$, $F = [l_k \mapsto l]$

$$\frac{\Gamma \vdash e \Downarrow_F \Delta \vdash (\lambda y : l_1.e') : l_2 \qquad \Delta \vdash e'[x/y] \Downarrow_G \Theta \vdash z}{\Gamma \vdash e\ (x : l') : l \Downarrow_H \Theta \vdash z} \quad \textit{App}$$
where $H = F \oplus G \oplus [l' \mapsto \{l, l_1\}]$

$$\frac{\Gamma \vdash e \Downarrow_F \Delta \vdash z}{\Gamma[x \mapsto < e, L >\ ] \vdash x : l \Downarrow_G \Delta[x \mapsto < z, \varepsilon >\ ] \vdash z} \quad \textit{Var}$$
where $G = F \oplus [l \mapsto L]$

$$\frac{\Gamma[y_n \mapsto < e_n[y_n/x_n], \{l_n, l\} \cup \varphi(e, x_n) \cup \varphi(e_n, x_n) >\ ] \vdash e[y_n/x_n] \Downarrow_F \Delta \vdash z}{\Gamma \vdash \texttt{let}\ \{x_n = e_n : l_n\}\ \texttt{in}\ e : l \Downarrow_G \Delta \vdash z}\ y_n \text{ fresh} \quad \textit{Let}$$
where $G = F \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)]$

$$\frac{\Gamma \vdash e \Downarrow_F \Delta \vdash (C_k\ x_1 : l_1^\star \cdots x_{a_k} : l_{a_k}^\star) : l_k^\sharp \qquad \Delta \vdash e_k[x_i/y_{i_k}] \Downarrow_G \Theta \vdash z}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{(C_j\ y_1 : l_1' \cdots y_{a_j} : l'_{a_j}) : l_j^\natural\ \texttt{->}\ e_j\}_{j=1}^n : l \Downarrow_H \Theta \vdash z} \quad \textit{Case}$$
where $G = F \oplus G \oplus [l_m^\star \mapsto \varphi(e_k, y_m) \cup \{l'_m, l_k^\natural\} | 1 \le m \le a_k] \oplus$
$[l_k^\natural \mapsto \{l\}] \oplus [l'_m \mapsto \varphi(e_k, y_m) \cup \{l_k^\natural\} | 1 \le m \le a_k]$

Fig. 7. Higher-Order Slicing Semantics

way, each rule has to compute the resulting set of labels S_f considering the effect that the input labels in S_i may have in the slice being computed.

Soon, however, it became difficult to specify the semantic rules taking into account the impact of the receiving set of labels. The problem of specifying the rules is related to the fact that in many cases there is not enough information to enable the decision of including a certain label or not.

For instance, in the *App* rule one may not immediately decide whether to include or not label l_1 in the resulting label set. The reason for this is that one has no means of knowing in advance whether a particular expression in the heap will ever become part of the slice. If such an expression is to be included into the slice, sometime along the remainder of the slicing process, then label l_1 will also belong to the slice as well as all the labels that l_1 affects by the evaluation of the first premiss of rule *App*.

$$\frac{\Gamma[x_k \mapsto < e_k, L_k >\ ] \vdash x_k \Downarrow_{F_k} \Delta \vdash z_k}{\Gamma[x_k \mapsto < e_k, L_k >\ ] \vdash (C\ x_1 : l_1' \cdots x_a : l_a') : l' \Downarrow_G \Delta \vdash (C\ x_1 : l_1' \cdots x_a : l_a') : l'} \quad \textit{Con}$$
where $k \in \{1, \ldots, a\}$, $G = F_k \oplus [l_k' \mapsto l']$

Fig. 8. Con Rule for Strict Evaluation of the Result Value

In order to overcome this problem, one should look for some independence in the slicing process over the partial slices that are being calculated by each semantic rule. Thus, instead of calculating partial slices on the application of every rule, one computes partial dependencies between labels. This entails the need for a further modification in the rules which are now intended to compute maps of type $\mathrm{L} \to \mathbb{P}\ \mathrm{L}$, called *lmap*'s, rather than sets, such that all labels in the codomain depend on the labels in the domain. The resulting semantics is presented in figure 7.

In the sequel the following three operations over *lmap*'s are required: an application operation, resorting to standard finite function application, defined by

$$F(x) = \begin{cases} F\ x & \text{if } x \in dom\ F, \\ \{\} & \text{otherwise.} \end{cases}$$

a *lmap* multiplication $\oplus$, defined as

$$(F \oplus G)(x) = F(x) \cup G(x)$$

and, finally, a range union operation *urng*, defined as

$$urng\ F = \bigcup_{x \in dom\ F} F(x)$$

Again, this semantics suffers from the problem identified in the *lazy print* specification i.e., the semantics is "too lazy". Once more, to overcome such undesired effect, one introduces a new rule (Fig. 8) to replace the original *Con* rule, and the slicing process is similarly divided into two phases.

By changing the output of the evaluation function from a set to a *lmap* of labels, we no longer have a slice of the program by the end of the evaluation. What is returned, instead, is a *lmap* specifying the different dependencies between the different expressions that form the program under analysis. The desired slice is computed as the transitive closure of such dependencies *lmap*.

Furthermore, splitting the slicing process into a dependencies calculation and the computation of a slice for the set of pertinent labels makes easier the calculation of slices that only differ on the set of pertinent labels. For such cases, one can rely on a common dependencies *lmap* and the whole process amounts to the calculation of the transitive closure for redefined set of labels.

7 Strict Evaluation

Slicing under strict evaluation is certainly easier. A possible semantics, as the one considered in figure 9, can be obtained by a systematic simplification of the one used in the lazy case. Of course, this is not the only possibility. To make comparisons possible between the lazy and strict case, however, we chose to keep specification frameworks as similar as possible, although we are aware that many details in the strict side could have been simplified. For example, strict semantics can always return slices in the form of sets of labels instead of calculating maps capturing dependencies between code entities.

Moreover, in the strict case there is no need to capture pending labels in the heap, since `let` expressions are evaluated as soon as they are found. This leads to a simplification of the heap from a mapping between variables and pairs of expressions and set of labels to a mapping between variables and values.

As for the rules, the *App* and *Let* rules need to be changed, along with some minor adaptation of the rules that deal with the (newly modified) heap.

Another decision taken in the strict slicing semantics specification was to keep value sharing *i.e.*, sharing of values that are stored in the heap. Nevertheless, one can easily derive a slicing semantics without any sharing mechanism, for which case one could probably remove the heap from the semantics.

Finally note that now there is no need to introduce a new *Con* rule to force the evaluation of unevaluated expressions inside result value. Thus, unlike the two previous versions of lazy slicing, strict slicing is accomplished in a single evaluation phase.

8 Some Considerations About the Slicing Processes

All slicing algorithms presented in this paper were introduced as (evaluators of) a specific semantics. Such an approach provides an expressive setting on top of which one may reason formally

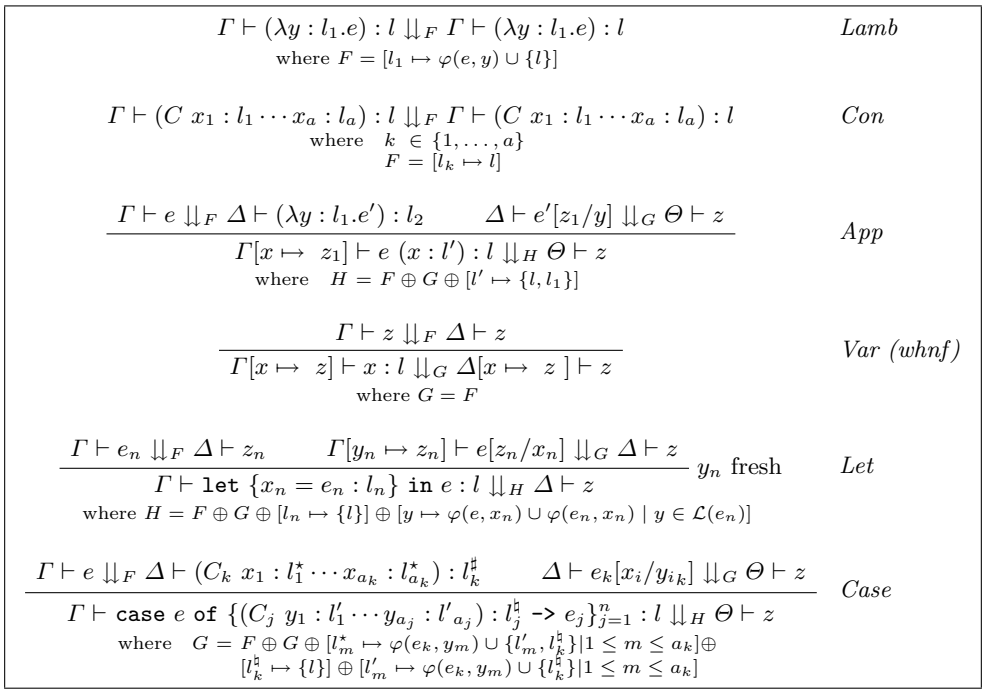


Fig. 9. Strict Slicing Semantics

about slices and slicers. This is illustrated in this section to confirm the intuitive fact that, in general, lazy slices are smaller than strict slices.

In the case of the *lazy print* semantics, such a proof amounts to showing that the set of labels returned by the lazy print is a subset of the set of labels yielded by an hypothetical strict print semantics.

But, since both the higher-order lazy slicing semantics and the strict one do not return sets of labels but maps of dependencies, one has to restate the proof accordingly. This can be achieved in two ways: either including the final transitive closure calculation in the slicing process, or introducing a partial order over the dependency *lmap*'s that respects subset inclusion.

We chose the latter alternative, and introduce the following partial order over *lmap*'s, which is the standard inclusion order on partial functions.

$$F \preceq G \Leftrightarrow dom(F) \subseteq dom(G) \wedge (\forall x \in dom(F).F(x) \subseteq G(x))$$

Now, the property that "lazy slices are smaller than strict slices" is formulated as follows.

If $\Gamma \vdash e \Downarrow_F \Delta \vdash z$ and $\Gamma \vdash e \Downarrow_G \Theta \vdash z$ then $F \preceq G$

The proof proceeds by induction over the rule-based semantics. First notice that the property is trivially true for all identical rules in both semantics. Such as the cases of rules *Lamb*, *Con* and *Case* for which the resulting *lmap*'s are equal. The remaining cases follows.

Case *App*: Evaluation of expressions under these rules take the following form, according to the evaluation strategy used.

$$\frac{\Gamma \vdash e \Downarrow_F \Delta \vdash (\lambda y : l_1.e') : l_2 \qquad \Delta \vdash e'[x/y] \Downarrow_G \Psi \vdash z}{\Gamma \vdash e\ (x : l') : l \Downarrow_H \Theta \vdash z} \quad App$$
$$\text{where} \quad H = F \oplus G \oplus [l' \mapsto \{l, l_1\}]$$

$$\frac{\Gamma \vdash e \downarrow\downarrow_I \Theta \vdash (\lambda y : l_1.e') : l_2 \qquad \Theta \vdash e'[z_1/y] \downarrow\downarrow_J \Phi \vdash z}{\Gamma[x \mapsto\ z_1] \vdash e\ (x : l') : l \downarrow\downarrow_K \Phi \vdash z} \quad App$$
$$\text{where} \quad K = I \oplus J \oplus [l' \mapsto \{l, l_1\}]$$

By induction hypothesis one has that $F \preceq I$. By definition of *Let* rule, which is the only rule that changes the heap, one has that $\mathcal{L}(\Delta) \cup urng\ F = \mathcal{L}(\Theta) \cup urng\ I$, where function $\mathcal{L}$ is overloaded to collect all the labels of the expressions in a heap. It follows that

$\mathcal{L}(\Delta) \cup urng\ F = \mathcal{L}(\Theta) \cup urng\ I$
$\Rightarrow$ {Induction Hypothesis}
$\mathcal{L}(\Delta) \setminus \mathcal{L}(\Theta) \subseteq urng\ I$
$\Rightarrow$ {Defintion of $\oplus$, noting that every possible label that G may collect from heap Δ is already in I}
$G \preceq I \oplus J$
$\Rightarrow$ {Induction Hypothesis}
$F \preceq I \wedge G \preceq I \oplus J$
$\Rightarrow$ {Definition of $\oplus$}
$F \oplus G \preceq I \oplus J$
$\Rightarrow$ {Definition of $\oplus$}
$F \oplus G \oplus [l' \mapsto \{l, l_1\}] \preceq I \oplus J \oplus [l' \mapsto \{l, l_1\}]$
$\Rightarrow$ {Defintion of G and H}
$G \preceq H$

Case *Let*: Evaluation of expressions under these rules takes the following format, according to the evaluation strategy used.

$$\frac{\Gamma[y_n \mapsto < e_n[y_n/x_n], \{l_n, l\} \cup \varphi(e, x_n) \cup \varphi(e_n, x_n) >] \vdash e[y_n/x_n] \Downarrow_F \Delta \vdash z}{\Gamma \vdash \texttt{let}\ \{x_n = e_n : l_n\}\ \texttt{in}\ e : l \Downarrow_G \Delta \vdash z}\ y_n \text{ fresh} \quad Let$$
$$\text{where } G = F \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)]$$

$$\frac{\Gamma \vdash e_n \downarrow\downarrow_H \Theta \vdash z_n \qquad \Gamma[y_n \mapsto z_n] \vdash e[z_n/x_n] \downarrow\downarrow_I \Phi \vdash z}{\Gamma \vdash \texttt{let}\ \{x_n = e_n : l_n\}\ \texttt{in}\ e : l \downarrow\downarrow_J \Phi \vdash z}\ y_n \text{ fresh} \quad Let$$
$$\text{where } J = H \oplus I \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)]$$

By induction hypothesis and because $\mathcal{L}(e_n) \subseteq urng\ H$ one has that $F \preceq H \oplus I$. It follows that

$G = F \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)]$
$\Rightarrow$ $\{F \preceq H \oplus I\}$
$G \preceq H \oplus I \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)]$
$\Rightarrow$ {Definition of K}
$G \preceq K$

Case *Var*: Evaluation of expressions under these rules take the following form, according to the evaluation strategy used.

$$\frac{\Gamma \vdash e \Downarrow_F \Delta \vdash z}{\Gamma[x \mapsto < e, L >] \vdash x : l \Downarrow_G \Delta[x \mapsto < z, \varepsilon >] \vdash z} \quad \textit{Var}$$
$$\text{where } G = F \oplus [l \mapsto L]$$

$$\frac{\Gamma \vdash z \Downarrow\!\Downarrow_H \Delta \vdash z}{\Gamma[x \mapsto\ z] \vdash x : l \Downarrow\!\Downarrow_I \Delta[x \mapsto\ z\ ] \vdash z} \quad \textit{Var}$$
$$\text{where } I = H$$

By induction hypothesis one has that $F \preceq H$. Since the only way to add entries to the heap is via the *Let* rule, and because, in strict semantics, such rule increments the dependencies *lmap* with every label from the newly introduced expressions, it follows that increments to the strict evaluation *lmap* will contain every mapping that is pending on the modified higher-order slicing heap. Thus, even though it may happen that at the time of evaluation of the *Var* rule, one may have $I \preceq G$, in the overall evaluation tree the dependency *lmap* for the lazy evaluation is always smaller or equal to the strict evaluation *lmap*.

9 Conclusions and Future Work

This paper introduced a semantic-based approach to low level slicing of functional programs, highlighting a strong relationship between the slicing problem and the underlying evaluation strategy.

Due to space restrictions, we have not been able to expose here a real code example that would highlight the strengths of the presented method. Actually, a realistic application example needs to have at least one large (more than 20 lines) function with several calls to other functions which would also had to be presented in order to achieve an understandable practical example. Nevertheless, we have tested the method against the Haskell implementation of the semantics presented in section 6, and the interested reader may consult the results at `http://www.di.uminho.pt/~nfr/Results/HoSlicingResults.html`. The need for examples with large function definitions to demonstrate our method capabilities is because in such cases one can point out as a slicing criterion a tag indicating the particular (sub)expression inside the large function definition, and by doing so one can more precisely identify the relevant part of the code that he/she is interested in. Other approaches that rely on slicing criterion defined by a return value cannot achieve slices as precise as ours.

Although the techniques introduced here are oriented to forward slicing, we strongly believe that a correct inversion of the dependencies *lmap*'s, followed by the same transitive closure calculation, will capture the backward cases.

This research adds to our previous work on high level functional slicing *i.e.*, slicing defined over "high level" program entities such as functions, modules, or data-types, as documented in [10]. Reference [9] reports on a completely alternative approach to the slicing problem based on the so called Bird-Meertens calculus. The common context of this research effort is a project on program understanding and re-engineering[3], currently running at Minho University, Portugal.

References

1. R. Bird. *Functional Programming Using Haskell*. Series in Computer Science. Prentice-Hall International, 1998.
2. S. K. Biswas. *Dynamic slicing in higher-order programming languages*. PhD thesis, 1997. Supervisor-Carl A. Gunter.
3. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conf. on Programming Usage, Design and Implementation*, pages 35–46. ACM Press, 1988.
4. B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.
5. B. Korel and J. Laski. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, 1990.

[3] http://wiki.di.uminho.pt/twiki/bin/view/PURe

6. J. Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993.
7. C. Ochoa, J. Silva, and G. Vidal. Dynamic slicing based on redex trails. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 123–134, New York, NY, USA, 2004. ACM Press.
8. T. W. Reps and T. Turnidge. Program specialization via program slicing. In *Selected Papers from the International Seminar on Partial Evaluation*, pages 409–429, London, UK, 1996. Springer-Verlag.
9. N. Rodrigues and L. Barbosa. Program slicing by calculation. *Journal of Universal Computer Science*, 12(7):828–848, 2006.
10. N. Rodrigues and L. S. Barbosa. Component identification through program slicing. In L. S. Barbosa and Z. Liu, editors, *Proc. of FACS'05 (2nd Int. Workshop on Formal Approaches to Component Software)*, volume 160, pages 291–304, UNU-IIST, Macau, 2006. Elect. Notes in Theor. Comp. Sci., Elsevier.
11. M. Weiser. *Program Slices: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Methods*. PhD thesis, University of Michigan, An Arbor, 1979.
12. M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.
13. M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

Open and Closed Worlds for Overloading: a definition and support for coexistence

Carlos Camarão[1], **Cristiano Vasconcellos**[2], **Lucília Figueiredo**[3], **João Nicola**[1]

[1] Universidade Federal de Minas Gerais (UFMG), DCC-ICEX,
Caixa Postal 702, Belo Horizonte 30123-970, MG, Brasil
{camarao,joaoraf}@dcc.ufmg.br

[2]Universidade Federal de Pelotas,
Campus Universitário s/n, Caixa Postal 354, Pelotas 96100-900, RS, Brasil
cristiano.damiani@ufpel.edu.br

[3]Universidade Federal de Ouro Preto, DECOM-ICEB,
Caixa Postal 140, Ouro Preto 35400-000, MG, Brasil
lucilia@dcc.ufmg.br

Abstract. *The type system of Haskell and some related systems are based on an open world approach for overloading. In an open world, the principal type of each overloaded symbol must be explicitly annotated (in Haskell, annotations occur in type class declarations) and a definition of an overloaded symbol is required to exist only when overloading is resolved. In a closed world, on the other hand, each principal type is determined according to the types of definitions that exist in the relevant context and, furthermore, overloading resolution for an expression considers only the context of the definition of its constituent symbols. In this paper we formally characterize open and closed worlds, and discuss their relative advantages. We present a type system that supports both approaches together, and compare the defined system with Haskell type classes extended with multi-parameter type classes and functional dependencies. We show in particular that functional dependencies are not necessary in order to support multi-parameter type classes, and present an alternative route.*

1 Introduction

The type system of Haskell [23, 12, 10, 20] and related type systems[31, 2, 26, 13, 4, 8] are based on an *open world* approach for overloading. In an open world, the principal type of each overloaded symbol must be explicitly annotated, and a definition of an overloaded symbol is required to exist only when overloading is resolved.

In Haskell, type annotations occur in *type class* declarations, and definitions of overloaded symbols are given in *instance* declarations.

For example, the principal types of `(==)` ("equal") and `(/=)` ("not equal") are annotated in type class *Eq* (defined in the Haskell prelude) as follows:

```
class Eq a where
  (==), (/=) :: a → a → Bool
  x /= y     =  not (x == y)
  x == y     =  not (x /= y)
```

A class may contain, apart from type annotations of overloaded symbols, also *default definitions*, as shown in class *Eq* above. The following is an instance of class *Eq* for values of type *Int*, assuming that *primEqInt* is a function for comparing values of type *Int* for equality. The definition of `(/=)` in this instance is the default definition given in class *Eq*, since it is not explicitly given in the instance declaration.

```
instance Eq Int where
  (==) = primEqInt
```

In an open world, a definition of an overloaded symbol is required to exist only when overloading is resolved. For instance, no definitions of equality are required to be in the context where a definition of (polymorphic) equality of lists is given.

In a closed world, on the other hand, for any given expression the types of definitions available in the context where this expression occurs determine if the occurrence of this expression is well-typed or not and, in the first case, its principal type. A closed world is "closed" only in the level of modules, which introduce separate typing contexts. If, say, x is imported from a module M into another module M', then the uses of x in M' consider only the definitions of x that occur in M. If new definitions of x need to be given or used in M', an open world *must* be used. On the other hand, inside a module, a closed world is, in fact, "more open" than an open world, in the sense that a new definition of an overloaded symbol is not required to be an instance of any given annotated type. Each new definition of an overloaded symbol x implies a redefinition of x's principal type, as the least common generalization of the types of definitions of x in the typing context.

In this paper we construct a framework that allows us to give precise definitions of open and closed worlds, and discuss their relative advantages. A useful result of this is the presentation of an alternative to the use of functional dependencies in an extension of Haskell with multi-parameter type classes. The paper starts by giving some preliminary definitions, in Section 2. Section 3 presents constraint-set satisfiability and simplification. In Section 4 we give formal definitions of open and closed worlds; relative advantages are compiled in subsection 4.1. Formal definitions of type systems to support both closed and open worlds are presented in Section 5. Inference of principal typings is discussed in Section 6, together with some relevant implementation issues. A brief discussion of ambiguity is given in Section 7. Section 8 concludes.

2 Preliminaries

We use types and terms of a language that is basically core-ML [21, 3, 22] extended with the possibility of introducing overloaded definitions in the outermost program scope, by means of a `leto` construct which does not introduce nested scopes. In this way, typing contexts are allowed to be stepwisely extended and may have more than one assumption for the same variable. The context-free syntax of expressions, their types and kinds of types is presented in Figure 2.

Each simple type expression has a *kind*, identified by an upper index in a simple type expression. A simple type is a type expression of kind $\star$, which is usually omitted. Meta-variables α, β, a, b, c dente type variables and C type constructors, of any kind.

Types of expressions are constrained polymorphic types. A set of constraints κ is a possibly empty set of pairs $x : \tau$, where x denotes an overloaded symbol and τ a simple

Programs	p	$::= e \mid \texttt{leto}\ x = e\ \texttt{in}\ p$	
Expressions	e	$::= x \mid \lambda x.\, e \mid e_1\, e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$	
Kinds	ι	$::= \star \mid \iota_1 \rightarrow \iota_2$	
Simple type expressions	τ^ι	$::= \alpha^\iota \mid C^\iota \mid \tau_1^{\iota \rightarrow \iota'}\ \tau_2^\iota$	
Constraints	κ	$::= \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$	$(n \geq 0)$
Types	σ	$::= \tau \mid \kappa.\, \tau \mid \forall \alpha.\, \sigma$	

Figure 1: Context-free syntax of expressions and types

type. A constrained polymorphic type is written as $\forall \alpha_1. \ldots . \forall \alpha_n.\, \kappa.\, \tau$, where $n \geq 0$. If $\kappa = \emptyset$, we have an unconstrained polymorphic type.

The set of free type variables of type σ is defined as usual and denoted by $tv(\sigma)$.

We use $\forall \bar{\alpha}.\, \kappa.\, \tau$ as an abbreviation for $\forall \alpha_1. \cdots \forall \alpha_n.\, \kappa.\, \tau$, for some $n \geq 0$, and similarly for $\bar{\tau}$, $\bar{\sigma}$. Naturally, $\forall \bar{\alpha}.\, \sigma = \sigma$ if $n = 0$, and $\forall \bar{\alpha}.\emptyset.\tau = \forall \bar{\alpha}.\tau$. We also use $\bar{\alpha}$ as a set of type variables, as in $\bar{\alpha} = tv(\tau)$.

A substitution S is a kind-preserving function from type variables to simple type expressions. The identity substitution is denoted by id. $S\sigma$ represents the capture-free[1] operation of substituting $S\alpha$ for each free occurrence of type variable α in σ. This operation is extended to constraints in the usual manner. We define $dom(S) = \{\alpha \mid S\alpha \neq \alpha\}$. It is sometimes convenient to use a finite mapping notation for substitutions, where $S = \{(\alpha_j \mapsto \tau_j)\}^{j=1..m}$ is used to denote the substitution such that $dom(S) = \{\alpha_j\}^{j=1..m}$ and $S\alpha_j = \tau_j$, for $j = 1, \ldots, m$. We also write $S \dagger \{\alpha_i \mapsto \tau_i\}^{i=1..n}$ to denote the substitution S' such that $S'\beta = S\beta$, if $\beta \notin \{\alpha\}^{i=1..n}$, and $S'\alpha_i = \tau_i$, for $i = 1, \ldots, n$. We define $\sigma[\bar{\tau}/\bar{\alpha}] = (id \dagger (\bar{\alpha} \mapsto \bar{\tau}))\sigma$.

In type systems with support for (universal) polymorphism, the type ordering is such that $\forall \alpha.\, \sigma \geq \sigma[\tau/\alpha]$, for *all* τ. The principal type of an expression in a typing context is the least upperbound, in this ordering, of all types that can be derived for this expression in this typing context. If $\sigma \geq \sigma'$, then σ' is called a *generic instance* of σ.

3 Constrained Polymorphism, Satisfiability and Simplification

In type systems with support for overloading, a typing context may include multiple assumptions for an overloaded symbol. The set of valid type assumptions which constitute a typing context is usually restricted by an overloading policy.

The principal type of an overloaded symbol x is obtained from the least common generalization (lcg) of the set of types in assumptions for x in the relevant typing context (unless the principal type of x is explicitly annotated). We say "the" least, instead of "a" least, by considering polymorphic types as equivalent up to renaming of bound type variables. Let $\tau \geq_S \tau'$ if $S\tau' = \tau$, and also $\tau \geq \tau'$ if there exists S such that $\tau \geq_S \tau'$. The lcg of a set of types $\{\sigma_i = \forall \bar{\alpha}_i.\, \kappa_i.\, \tau_i\}^{i=1..n}$ is $\forall \bar{\alpha}.\, \tau$, where $\bar{\alpha} = tv(\tau)$ and: i) $\tau_i \geq \tau$ (i.e. $\forall \bar{\alpha}.\, \tau \geq \forall \alpha_i.\, \tau_i$, $\bar{\alpha}_i = tv(\tau_i)$, for $i = 1, \ldots, n$; and ii) $\tau_i \geq \tau'$ implies $\tau \geq \tau'$ (i.e. $\forall \bar{\beta}.\tau' \geq \forall \bar{\alpha}.\, \tau$, where $\bar{\beta} = tv(\tau')$).

[1]The operation of applying substitution S to σ is capture-free if $tv(S\sigma) = tv(S(tv(\sigma)))$, where application of substitution S to a set of type variables $\{\alpha_i\}^{i=1..n}$ is given by $\{S\alpha_i\}^{i=1..n}$.

The *constrained least common generalization* of the types of x in a typing context Γ is the type $\sigma = \forall\bar{\alpha}.\,\{x : \tau\}.\,\tau$, where $\forall\bar{\alpha}.\,\tau$ is the lcg of $\Gamma(x)$ — written as $clcg(x, \Gamma, \sigma)$.

EXAMPLE 1. Consider that the assumptions for `(==)` in a typing context $\Gamma_{(==)}$ are `(==)` : *Int* $\rightarrow$ *Int* $\rightarrow$ *Bool* and `(==)` : *Float* $\rightarrow$ *Float* $\rightarrow$ *Bool*. The following types can be derived for `(==)` in this typing context:

$$\textit{Int} \rightarrow \textit{Int} \rightarrow \textit{Bool}, \textit{Float} \rightarrow \textit{Float} \rightarrow \textit{Bool}, \forall a.\,\{\texttt{(==)} : a \rightarrow a \rightarrow \textit{Bool}\}.\,a \rightarrow a \rightarrow \textit{Bool}$$

The last one is the principal type of `(==)` in $\Gamma_{(==)}$. It can be instantiated to types of the form $\{\texttt{(==)} : \tau \rightarrow \tau \rightarrow \textit{Bool}\}.\,\tau \rightarrow \tau \rightarrow \textit{Bool}$, for which the constraint is *satisfiable* in Γ — in this particular case, τ can be either *Int* or *Float* or α, for some type variable α. If τ is *Int* or *Float*, the set of constraints can be simplified to an empty of constraints. □

EXAMPLE 2. Consider the following definition of function *ins*, that uses `(==)`:

```
ins a []     = [a]
ins a (b:x) = if a==b then b:x else b:ins a x
```

The principal type of this definition, obtained as the $clcg$ of types of `(==)` in $\Gamma_{(==)}$, is: $\forall a.\,\{\texttt{(==)} : a \rightarrow a \rightarrow \textit{Bool}\}.\,a \rightarrow [a] \rightarrow [a]$. Thus *ins* can be used in any context with a type that is an instance of $\forall\bar{\alpha}.\,\tau \rightarrow \tau \rightarrow \textit{Bool}$, where $\bar{\alpha} = tv(\tau)$, if constraint-set $\{\texttt{(==)} : \tau \rightarrow \tau \rightarrow \textit{Bool}\}$ is satisfiable in this context.

EXAMPLE 3 (Functions overloaded over distinct type constructors). Assume that there exist distinct definitions of function *ins*, in a typing context Γ_{ins}, for inserting elements in lists and trees, with types $\forall a.\,\{\texttt{(==)} : a \rightarrow a \rightarrow \textit{Bool}\}.\,a \rightarrow [a] \rightarrow [a]$, and $\forall a.\,\{\texttt{(==)} : a \rightarrow a \rightarrow \textit{Bool}\}.\,a \rightarrow \textit{Tree}\ a \rightarrow \textit{Tree}\ a$. In this context, the principal type of *ins* is $\forall a.\,\forall c.\,\{\textit{ins} : a \rightarrow c\ a \rightarrow c\ a\}.\,a \rightarrow c\ a \rightarrow c\ a$, where c is a type variable of kind $\star \rightarrow \star$. Note that this type does not contain a constraint on `(==)`. Such constraint is automatically recovered from constraints on types of assumptions for *ins* in Γ_{ins}, if and when overloading is resolved, thus creating automatically a hierarchy of dependencies between overloaded symbols. □

EXAMPLE 4. Consider that we also want to overload *ins* in typing context Γ'_{ins} by including definitions with the following types, in addition to those of Γ_{ins} of Example 3, that include a comparison operator as argument for working on ordered lists and trees (this could not be in Haskell extended with MPTCs if the principal type of *ins* was fixed a priori in a type class as $\forall a.\,\forall c.\,a \rightarrow c\ a \rightarrow c\ a$):

$$\begin{array}{l}\textit{ins} : \forall a.\,(a \rightarrow a \rightarrow \textit{Bool}) \rightarrow a \rightarrow [a] \rightarrow [a]\\ \textit{ins} : \forall a.\,(a \rightarrow a \rightarrow \textit{Bool}) \rightarrow a \rightarrow \textit{Tree}\ a \rightarrow \textit{Tree}\ a\end{array}$$

In Γ'_{ins}, the type of *ins* is $\forall a, b, c.\,\{\textit{ins} : a \rightarrow b \rightarrow c\}.\,a \rightarrow b \rightarrow c$, where a, b, c are respectively the lcgs of:

$$\begin{array}{llll}\{\ a, & a, & a \rightarrow a \rightarrow \textit{Bool}, & a \rightarrow a \rightarrow \textit{Bool}\ \}\\ \{\ [a], & \textit{Tree}\ a, & a, & a\ \}\\ \{\ [a], & \textit{Tree}\ a, & [a] \rightarrow [a], & \textit{Tree}\ a \rightarrow \textit{Tree}\ a\ \}\end{array}$$

3.1 Constraint-set satisfiability

A constraint-set κ on a type $\sigma = \forall\bar{\alpha}.\,\kappa.\,\tau$ restricts the set of types to which σ can be instantiated, on a given typing context Γ, according to the type assumptions in Γ for the overloaded symbols that occur in κ.

A definition of constraint-set satisfiability independent on provability in a type system was given in [1]. We present below a simpler definition ($\circ$ denotes composition):

DEFINITION 1 (Constraint and constraint-set satisfiability).

$$\frac{}{\Gamma \models_{id} \emptyset} \qquad (\mathtt{SAT_0})$$

$$\frac{S\tau = S\tau' \quad \Gamma \models_{S'} S\kappa}{\Gamma \cup \{x : \forall\bar{\alpha}.\,\kappa.\,\tau'\} \models_{S' \circ S} \{x : \tau\}} \qquad (\mathtt{SAT_1})$$

$$\frac{\Gamma \models_S \{x : \tau\} \quad \Gamma \models_S \kappa}{\Gamma \models_S \kappa \cup \{x : \tau\}} \qquad (\mathtt{SAT_n})$$

A *constraint-set satisfiability problem* is a problem of determining, for a given pair (Γ, κ), where Γ is a typing context and κ is a constraint-set, whether $\Gamma \models_S \kappa$ is provable, for some substitution S, which is called a *solution* to the satisfiability problem.

A solution S is *principal* if for any other solution S' there exists a substitution R such that $S' = R \circ S$. The application of the principal solution followed by constraint-set simplification is called *improvement*[12, 14].

If $\Gamma \models_S \kappa$ is provable, for some S, then we write that $\Gamma \models \kappa$, otherwise $\Gamma \not\models \kappa$.

The constraint-set satisfiability problem has been proved to be undecidable [29]. However, practical implementations have been used, that either restrict the set of types of overloading symbols that may be introduced in typing contexts, in order to guarantee decidability, or use an iteration limit in order to prevent nontermination [17, 19, 11, 1].

Let a solution S to a satisfiabilty problem of κ in Γ be *minimal* if, for any other solution S', $dom(S) \cap tv(\kappa) \subseteq dom(S') \cap tv(\kappa)$ (informally, a solution is minimal if it "modifies κ less" than any other solution).

DEFINITION 2 (Overloading resolution). Let Γ be any typing context and κ be any constraint-set involving a constraint on x — i.e. let $\kappa = \{x : \tau\} \cup \kappa'$, for some constraint-set κ'. *Overloading of x is resolved*, in an expression that has a principal type with constraint-set κ, if there exists a minimal solution S to the satisfiability of κ in Γ such that $S\tau = S'\tau$, for all other minimal solutions S'. □

3.2 Constraint-set Simplification

Constraints can be simplified either by removal of resolved constraints or substitution of constraints. For instance, $\{\,(==) : \mathit{Int} \to \mathit{Int} \to \mathit{Bool}\}$ can be simplified to an empty set of constraints, in typing context $\Gamma_{(==)}$ of Example 1, and $\{\mathit{ins}\colon \alpha \to [\alpha] \to [\alpha]\}$ can be simplified to $\{\,(==) : \alpha \to \alpha \to \mathit{Bool}\,\}$, in typing context Γ_{ins} of Example 3.

Simplification yields equivalent constraint-sets. Equivalence between constraint-sets in a given typing context Γ is defined as the reflexive, symmetric and transitive closure of the simplification relation $\Gamma \models \kappa \gg \kappa'$ defined below.

DEFINITION 3 (Constraint-set Simplification).

$$\frac{\Gamma \cup \{x : \forall \bar{\alpha}.\, \kappa'.\, \tau'\} \models_S \{x : \tau\} \cup \kappa \qquad \Gamma \not\models \{x : \tau\} \cup \kappa}{\Gamma \cup \{x : \forall \bar{\alpha}.\, \kappa'.\, \tau'\} \models \{x : \tau\} \cup \kappa \gg S(\kappa \cup \kappa')}$$

The premise of the rule above implies that overloading of x is resolved.

3.3 Open world satisfiability

Open world constraints — i.e. constraints introduced due to the specification, by programmers, of the principal type of overloaded symbols — are tested for satisfiability only when overloading of some symbol must be resolved. This can be formalized by means of the constraint projection operation $\kappa|^*_{tv(\tau) \cup tv(\Gamma)}$, which returns all constraints with type variables "dependent" on type variables of τ and Γ, transitively.

DEFINITION 4 (Constraint projection).

$$\kappa|_V = \{x : \tau \in \kappa \mid tv(\tau) \cap V \neq \emptyset\} \qquad \kappa|^*_V = \begin{cases} \kappa|_V & \text{if } tv(\kappa|_V) \subseteq V \\ \kappa|^*_{tv(\kappa|_V)} & \text{otherwise} \end{cases}$$

DEFINITION 5 (Open world satisfiability). $\Gamma \models^V_S \kappa$ holds if $\Gamma \models_S \kappa' \gg \emptyset$, where $\kappa' = \kappa - \kappa|^*_{V \cup tv(\Gamma)}$. $\Gamma \models^V \kappa$ holds if $\Gamma \models^V_S \kappa$ holds, for some S. □

EXAMPLE 5. Consider the definition *h* = *g True*, in typing context $\Gamma_g = \{g :$ *Bool* $\rightarrow$ *Char*, $g :$ *Char* $\rightarrow$ *Bool*, *True* : *Bool*$\}$. In a closed world approach, *h* has type *Char*. In an open world, consider for example that the definitions of g with these types are instances of the multi-parameter type class `class G a b where g:: a → b`. In this case, *h* has principal type *G Bool b* $\Rightarrow b$, where b is a fresh type variable (written $\forall\beta.\, \{g : Bool \rightarrow \beta\}.\, \beta$ in `CT`). The reason for this is that it is possible for *h* to be exported and used in a program context where other definitions of g exist, and one of these could be used in the evaluation of *g True* (for example, a definition with type $Bool \rightarrow Int$). According to Definition 5, we have: $\Gamma \models^{\{\beta\}} \{g :$ *Bool* $\rightarrow \beta\}$, for any Γ, that is, no definition of g is required to exist in order to type *g True*.

An extension of Haskell with functional dependencies allows programmers to "close the world". In this example, type variable b can be explicitly defined to "depend on a" (or a can be specified to determine b), by annotation of a functional dependency: `class G a b | a -> b where g:: a → b`. With such a functional dependency, the world is closed, i.e. satisfiability is checked in typing context Γ_g, returning a substitution that, applied to type *G Bool b* $\Rightarrow b$, improves (or simplifies) it to *Char*. □

4 Open and Closed Worlds: a Formal Definition

In this section we give formal definitions of open and closed worlds, based on the definitions given before. Our characterization is somewhat different from the one based on open and closed refinement kinds of Duggan and Ophel[7]. Refinement kinds are defined so as to a priori allow constrained polymorphic types to be "incrementally extended" (in the case of open kinds) or not (for closed kinds). Closed refinement kinds completely characterize ("close up"), by themselves, the possibly infinite set of instance types. For this, refinement kinds use intersection types and fixed point operators. In contrast, our definitions are always respective to a given typing context.

DEFINITION 6 (Open World). An *open world* is characterized by:

1. The principal type of each overloaded symbol is specified by a single type annotation and $\Gamma \models^{tv(\tau_x)} \kappa_x$ holds, where Γ is the typing context and $x : \forall\bar{\alpha}.\, \kappa_x.\, \tau_x$ is the type assumption corresponding to this annotation.
2. In such Γ, each definition of x must have a type $\sigma = \forall\bar{\beta}.\, \kappa.\, \tau$ such that $\bar{\beta} = tv(\kappa.\, \tau)$, $\tau = S\tau_x$ and $\kappa \supseteq S\kappa_x$, for some S, and $\Gamma \models^{tv(\tau)} \kappa$.
3. $\Gamma \models^{tv(\tau)} \kappa$ holds for all Γ, κ, τ such that $\Gamma \vdash \kappa.\, \tau$ is derivable.

In $\Gamma \models^{tv(\tau)} \kappa$, the set $tv(\tau)$ can be restricted, e.g. by functional dependencies in Haskell with MPTcs, so as to close the world (as illustrated in Example 5). In this case, $tv(\tau)$ should be subtracted by a set of variables in κ that are specified as targets (i.e. that occur at the right-hand side) of some functional dependency and for which the corresponding source type variables have been instantiated (i.e. do not occur in κ).

DEFINITION 7 (Closed World). A *closed world* is characterized by:

1. The principal type of x in Γ is the $clcg$ of the types of x in Γ.
2. A type annotation for e is an instance of e's principal type.
3. $\Gamma \models \kappa$ holds for all Γ, κ, τ such that $\Gamma \vdash \kappa.\tau$ is derivable.
4. If definitions of x are given in a module M — in which type assumptions are given by Γ — and x is imported into a module N, then $\Gamma(x)$ gives the types in type assumptions for x used to type uses of x in N.

4.1 Open versus closed

This section compiles relative advantages of open and closed worlds. For space reasons, we only include major issues. We start with the advantages of an open world:

- Overloaded symbols may be used without requiring definitions of overloaded symbols to be visible (they must be visible only when overloading is resolved).
- The type inference algorithm can behave more efficiently in an open world, due to satisfiability being tested only when overloading must be resolved. This is rather significant in the case of frequently used overloaded symbols (e.g. `(==)`).

However, in a closed world:

- Types of overloaded symbols need not be annotated. The annotation of constraints of types of overloaded symbols require that programmers decide, in advance, which particular definitions an overloaded symbol might possibly have (in

Haskell, programmers must decide, in particular, how to organize type class hierarchies). In some situations, this requirement can be rather inconvenient (as pointed out by e.g. Odersky, Wadler and Wehr [24, page 137]). Since constraints include information that is dependent on the implementation of a function (i.e. on the fact that an implementation uses some particular overloaded symbols), if the implementation of a function is changed, so that the new implementation uses a different set of overloaded symbols, program parts that use this function need to be modified, if they include type annotations, even if the types of argument and result of the function remain the same. In a closed world, the hierarchy of dependencies between overloaded symbols need not be given by programmers, but are recovered automatically by the types of overloaded symbols (see e.g. Example 3).

- A closed world allows the inference of more informative types, (possibly causing overloading to be resolved), due to earlier constraint-set satisfiability checking. Practical examples for which an analysis of the types of overloaded symbols available in a typing context can be used to instantiate the types of expressions that use these overloaded symbols can be found in e.g. [14, 8].
- A closed world approach opens possibilities for optimizations in code generation which do not depend (unlike the case of an open world approach [6, 18]) on a global analysis of the entire program for efficient dynamic dispatching to an appropriate definition of an overloaded symbol.

5 Type system

Figure 2 presents a version of system CT to support a closed world. This is extended in Section 5.1 in order to support both open and closed worlds together. Typing formulas $\Gamma \vdash e : \sigma$ express that expression e has type σ in typing context Γ. following are used in Figure

For any typing context Γ, $(\Gamma\,;\, x : \sigma)$ denotes $\Gamma \cup \{x : \sigma\}$ if $\Gamma \cup \{x : \sigma\}$ is a valid typing context (according to the adopted overloading policy), and $(\Gamma,\ x : \sigma) = (\Gamma \ominus x)\,;\, x : \sigma$, where $\Gamma \ominus x = \Gamma - \{x : \sigma \mid x : \sigma \in \Gamma\}$. Predicate $gen(\kappa.\,\tau, \sigma)$ is defined to hold if $\sigma = \forall\bar{\alpha}.\,\kappa.\,\tau$, where $\bar{\alpha} = tv(\kappa.\,\tau)$. Instantiation of constrained polymorphic types is restricted by constraint-set satisfiability:

DEFINITION 8 (Instance relation of constrained types).

$$\frac{\sigma \geq \forall\bar{\alpha}.\,\kappa.\,\tau \qquad \Gamma \models \kappa}{\sigma \geq_\Gamma \forall\bar{\alpha}.\,\kappa.\,\tau}$$

Note that if $\Gamma(x)$ is a singleton $\{x{:}\tau\}$ then $clcg(x, \Gamma, \forall\bar{\alpha}.\,\{x{:}\tau\}.\,\tau)$ and $clcg(x, \Gamma, \tau)$ also hold, as $\forall\bar{\alpha}.\,\{x{:}\tau\}.\,\tau$ and τ are equivalent modulo constraint-set simplification in Γ.

5.1 Selectively opening a closed world

An extension of system CT to support also an open world is presented in Figure 5.1. It is based on the use of special type annotations, that specify the types of overloaded symbols, as in the following example: `assume (==)` :: $a \to a \to$ *Bool*.

A clause `assume` $x :: \tau$ introduces in the typing context an *open-world assumption* $x : \forall\bar{\alpha}.\,\{x : \tau\}.\,\tau$, where $\bar{\alpha} = tv(\tau)$. Note that the form of an `assume` clause contributes to restoring data abstraction, since it does not include constraints.

$$\frac{clcg(x,\Gamma,\sigma) \quad \sigma \geq_\Gamma \kappa.\,\tau}{\Gamma \vdash x : \kappa.\,\tau} \qquad \text{(VAR)}$$

$$\frac{\Gamma, x : \tau' \vdash e : \kappa.\,\tau}{\Gamma \vdash \lambda x.\, e :\ \kappa.\,\tau' \to \tau} \qquad \text{(ABS)}$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \kappa_2.\,\tau_2}{\Gamma \vdash e_1\, e_2 : \kappa_1 \cup \kappa_2.\,\tau_1} \quad \Gamma \models \kappa_1 \cup \kappa_2 \qquad \text{(APPL)}$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_1 \quad \Gamma, x : \sigma_1 \vdash e_2 : \kappa_2.\,\tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \kappa_2.\,\tau_2} \quad gen(\kappa_1.\,\tau_1, \sigma_1) \qquad \text{(LET)}$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_1 \quad \Gamma; x : \sigma_1 \vdash p : \kappa_2.\,\tau_2}{\Gamma \vdash \texttt{leto}\ x = e_1\ \texttt{in}\ p : \kappa_2.\,\tau_2} \quad gen(\kappa_1.\,\tau_1, \sigma_1) \qquad \text{(LETO)}$$

Figure 2: Type system CT supporting a closed world

The support for open world in system CT is based on characterizing type assumptions and constraints as open or closed. Open world type assumptions and constraints are introduced in a typing context only by means of assume clauses, and the second by lambda, let or leto bindings (type derivation for lambda and let bound variables are done in the same manner as for leto bound variables). We define that ow and cw filters out open and closed world assumptions, respectively, from a typing context or a constraint-set ($\Gamma = ow(\Gamma) \cup cw(\Gamma)$ and similarly for κ). A typing context Γ must be such that $ow(\Gamma)$ satisfies the requirements in Definition 6. $\Gamma(x)$ is redefined to mean $ow(\Gamma)(x)$ if $ow(\Gamma)(x) \neq \emptyset$, otherwise $cw(\Gamma)(x)$. The satisfiability relation $\Gamma \models^{*V} \kappa$ holds if both $\Gamma \models^V ow(\kappa)$ and $\Gamma \models cw(\kappa)$ hold. The instantiation relation is modified in order to use the combined satisfiability relation: $\dfrac{\sigma \geq \forall\bar{\alpha}.\,\kappa.\,\tau \qquad \Gamma \models^{*tv(\tau)} \kappa}{\sigma \geq_\Gamma \forall\bar{\alpha}.\,\kappa.\,\tau}$.

$$\frac{clcg(x,\Gamma,\sigma) \quad \sigma \geq_\Gamma \kappa.\,\tau}{\Gamma \vdash x : \kappa.\,\tau} \qquad \text{(VAR}_\text{o}\text{)}$$

$$\frac{\Gamma, \{x : \tau'\} \vdash e : \kappa.\,\tau}{\Gamma \vdash \lambda x.\, e :\ \kappa.\,\tau' \to \tau} \qquad \text{(ABS}_\text{o}\text{)}$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \kappa_2.\,\tau_2}{\Gamma \vdash e_1\, e_2 : \kappa_1 \cup \kappa_2.\,\tau_1} \quad \Gamma \models^{*tv(\tau_1)} \kappa_1 \cup \kappa_2 \qquad \text{(APPL}_\text{o}\text{)}$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_1 \quad \Gamma, \{x : \sigma_1\} \vdash e_2 : \kappa_2.\,\tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \kappa_2.\,\tau_2} \quad gen(\kappa_1.\,\tau_1, \sigma_1) \qquad \text{(LET}_\text{o}\text{)}$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_1 \quad \Gamma; \{x : \sigma_1\} \vdash p : \kappa_2.\,\tau_2}{\Gamma \vdash \texttt{leto}\ x = e_1\ \texttt{in}\ p : \kappa_2.\,\tau_2} \quad \begin{array}{l} gen(\kappa_1.\,\tau_1, \sigma_1) \\ ow(\Gamma)(x) = \{\sigma\} \text{ implies } \sigma \geq_\Gamma \sigma_1 \end{array} \qquad \text{(LETO}_\text{o}\text{)}$$

Figure 3: Type system CT supporting open and closed worlds

To summarize the modifications to support also an open world: i) a modified constraint-set satisfiability relation, on the side condition of rule APPL$_\text{o}$, considers the possibility of occurrence of open and closed constraints together and, for open world

constraints, defers satisfiability to when overloading is resolved, ii) a correspondingly modified instantiation relation is used in rule $\texttt{VAR}_\text{o}$, and iii) a side condition in rule $\texttt{LETO}_\text{o}$ restricts the type of definitions of overloaded symbols to be an instance of its principal type, if this type is explicitly specified.

6 Type Inference

A prototype implementation of the front-end of a compiler for a language that is essentially "Haskell without type classes" but with support for constrained polymorphism as defined in system `CT`, is available at `http://www.dcc.ufmg.br/~camarao/CT/`. The type inference algorithm in this prototype supports both open and closed worlds and is an extension — to support (possibly mutually recursive) binding groups — of the algorithm defined in Figure 6 . This algorithm is defined as a syntax-directed proof system of judgements $\Gamma \vdash e : (\kappa.\, \tau, \Gamma')$, where $(\forall \bar{\alpha}.\, \kappa.\, \tau, \Gamma')$ is the principal typing of e in Γ [9, 28], where $\bar{\alpha} = tv(\kappa.\, \tau) - tv(\Gamma)$. The algorithm uses the following functions.

Function $clcg_a$, for computing the constrained least common generalization of the set of types of some symbol x in a typing context Γ, is given by $clcg_a(x, \Gamma) = \forall \bar{\alpha}.\, \{x : \tau\}.\, \tau$, where $\forall \bar{\alpha}.\, \tau = lcg_a(\Gamma(x))$. Function lcg_a computes the lcg of a given set of types, based on a function for computing the lcg of any set of simple type expressions. Finite mappings $\mathcal{S}$ from type variables to pairs of simple types are used in lcg' to "remember" generalizations already performed. For example, lcg applied to the set of types $\{\alpha_1 \to \beta_1 \to \alpha_1,\ \alpha_2 \to \beta_2 \to \alpha_2\}$ gives $\alpha \to \beta \to \alpha$, where α, β are fresh (and not, say, $\alpha \to \beta \to \alpha'$). lcg' needs to remember generalizations only inside a pair of type expressions. χ is used to denote a type variable or constructor.

$$lcg_a(\{\sigma_i\}^{i=1..n}) = \forall \bar{\alpha}.\, \tau, \text{ where } \sigma_i = \forall \bar{\alpha}.\kappa_i.\, \tau_i, \text{ for } i = 1, \ldots, n,\quad (\tau, \mathcal{S}) = lcg'(\{\tau_i\}^{i=1..n}, \emptyset), \text{ for some } S, \text{ and } \bar{\alpha} = tv(\tau)$$

$$lcg'(\{\tau\}, \mathcal{S}) = (\tau, \mathcal{S})$$

$$lcg'(\{\chi_1^{\iota_1}, \chi_2^{\iota_2}\}, \mathcal{S}) = \begin{cases} (\chi_1^{\iota_1}, \mathcal{S}) & \text{if } \chi_1 = \chi_2 \text{ (which implies } \iota_1 = \iota_2) \\ (\alpha, \mathcal{S}) & \text{if } \mathcal{S}(\alpha) = (\chi_1^{\iota_1}, \chi_2^{\iota_2}), \text{ for some } \alpha \\ (\alpha', \mathcal{S} \dagger \{\alpha' \mapsto (\chi_1^{\iota_1}, \chi_2^{\iota_2})\} & \text{otherwise, where } \alpha' \text{ is fresh} \end{cases}$$

$$lcg'(\{\tau_1^{\iota_1}\ \tau_2^{\iota_2},\ \tau_3^{\iota_3}\ \tau_4^{\iota_4}\}, \mathcal{S}) =$$

$$\textbf{if } S(\alpha) = (\tau_1^{\iota_1}\ \tau_2^{\iota_2},\ \tau_3^{\iota_3}\ \tau_4^{\iota_4}), \text{ for some } \alpha, \textbf{ then } (\alpha, \mathcal{S})$$
$$\textbf{else if } i_1 \neq i_3 \text{ or } i_2 \neq i_4 \textbf{ then } (\alpha', \mathcal{S} \dagger \{\alpha' \mapsto (\tau_1^{\iota_1}\ \tau_2^{\iota_2},\ \tau_3^{\iota_3}\ \tau_4^{\iota_4})\}), \text{ where } \alpha' \text{ is fresh}$$
$$\textbf{else } (\tau_a^{\iota_1}\ \tau_b^{\iota_2}, \mathcal{S}_2), \text{ where: } (\tau_a^{\iota_1}, \mathcal{S}_1) = lcg'(\{\tau_1^{\iota_1}, \tau_3^{\iota_3}\}, \mathcal{S}),\ (\tau_b^{\iota_2}, \mathcal{S}_2) = lcg'(\{\tau_2^{\iota_2}, \tau_4^{\iota_4}\}, \mathcal{S}_1)$$

$$lcg'(\{\tau_1, \tau_2\} \cup \mathcal{T}, \mathcal{S}) = lcg'(\{\tau\} \cup \mathcal{T}, id) \text{ where } (\tau, \mathcal{S}') = lcg'(\{\tau_1, \tau_2\}, id)$$

Function *simplify* simplifies constraint-sets, as defined in Section 3.2.

$$simplify(\kappa, \Gamma) = simplify(\kappa, \Gamma, \emptyset)$$

$$simplify(\emptyset, \Gamma, \kappa_0) = \emptyset$$

$$simplify(\{o : \tau\}, \Gamma, \kappa_0) = \textbf{if } tv(\tau) = \emptyset \textbf{ then } \emptyset \textbf{ else}$$
$$\textbf{if } o : \tau \in \kappa_0 \textbf{ then } \{o : \tau\} \textbf{ else}$$
$$\textbf{if } \exists \text{ a single } \forall \bar{\alpha}.\kappa'.\tau' \in \Gamma(o) \text{ s.t. } S\tau' = S\tau, \text{ for some } S$$
$$\textbf{then } simplify(S\kappa', \Gamma, \kappa_0 \cup \{o : \tau\}) \textbf{ else } \{o : \tau\}$$

$$simplify(\{o : \tau\} \cup \kappa, \Gamma, \kappa_0) = simplify(\{o : \tau\}, \Gamma, \kappa_0) \cup simplify(\kappa, \Gamma, \kappa_0)$$

Function sat implements a practical solution to the satisfiability problem by returning a principal solution whenever one exists and by using an iteration limit to stop, when it cannot find a solution. The definition of sat is given in [1].

Function $unify$ implements unification[2] (see e.g. [22]). We assume that all types in outermost typing contexts are closed (otherwise unification would be needed in the case of leto-bindings, as done for let-bindings), and also use $st(x, \Gamma, \Gamma') = \{\tau = \tau' \mid x : \tau \in \Gamma, x : \tau' \in \Gamma'\}$ and $\Gamma^*_x = \{x : \sigma_i\}^{i=1..n} \cup \bigcup_{y:\tau\in\kappa_i, \text{ for some } \tau, i\in\{1..n\}} \Gamma^*_y$, where $\Gamma(x) = \{\sigma_i = \forall\bar{\alpha}_i.\, \kappa_i.\, \tau_i\}^{i=1..n}$.

$$\frac{\forall\bar{\alpha}.\, \{x : \tau\}.\, \tau = clcg_a(x, \Gamma) \quad \kappa = simplify(\{x : \tau\}, \Gamma)}{\Gamma \vdash x : (\kappa.\, \tau, \Gamma^*_x)}$$

$$\frac{\Gamma \vdash (e : \kappa.\, \tau, \Gamma')}{\Gamma \vdash \lambda x.\, e : (\kappa.\, \tau' \to \tau, \Gamma' \ominus x)} \qquad \tau' = \begin{cases} \tau_x & \text{if } x : \tau_x \in \Gamma' \\ \alpha & \text{otherwise, } \alpha \text{ fresh} \end{cases}$$

$$\frac{\Gamma \vdash e_1 : (\kappa_1.\, \tau_1, \Gamma_1) \quad \Gamma \vdash e_2 : (\kappa_2.\, \tau_2, \Gamma_2)}{\Gamma \vdash e_1\, e_2 : (S'S(\kappa_1 \cup \kappa_2.\, \alpha), S\Gamma_1 \cup S\Gamma_2))} \quad \begin{array}{l} S = unify(\{\tau_1 = \tau_2 \to \alpha\} \cup st(x, \Gamma_1, \Gamma_2)) \\ S' = sat(\kappa_1 \cup \kappa_2, \Gamma),\ \alpha \text{ fresh} \end{array}$$

$$\frac{\Gamma \vdash e_1 : (\kappa_1.\, \tau_1, \Gamma_1) \quad \Gamma, \{x : \sigma_1\} \vdash e_2 : (\kappa_2.\, \tau_2, \Gamma_2)}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : (S'S(\kappa_2.\, \tau_2), S\Gamma_2 \ominus x)} \quad \begin{array}{l} S = unify(st(x, \Gamma_1, \Gamma_2)\}) \\ S' = sat(\kappa_1 \cup \kappa_2, \Gamma) \end{array}$$

$$\frac{\Gamma \vdash e_1 : (\kappa_1.\, \tau_1, \Gamma_1) \quad \Gamma; \{x : \sigma_1\} \vdash p : (\kappa_2.\, \tau_2, \Gamma_2)}{\Gamma \vdash \texttt{leto}\ x = e_1\ \texttt{in}\ p : (S(\kappa_2.\, \tau_2), \Gamma_2 \ominus x)} \quad \begin{array}{l} S = sat(\kappa_1 \cup \kappa_2, \Gamma) \\ ow(\Gamma)(x) = \{\sigma\} \text{ implies } \sigma \geq_\Gamma \sigma_1 \end{array}$$

Figure 4: Algorithm for Inference of Principal Typings

Proofs of theorems on soundness and principal typing of the type inference algorithm (which are not even stated in this paper for lack of space) can be obtained by induction on the type inference rules, using proofs of correctness of sat, $simplify$ and lcg_a. We are unfortunately still working on small details of the proofs.

The implementation of the type inference algorithm uses the core of an algorithm by Mark Jones [15] and is based also on other works by the authors, covering constraint-set satisfiability and polymorphic recursion [1, 28, 9]. We have very recently "glued" our front-end to GHC's[19] back-end, and a compiler will thus be publicly available soon.

Our experience with our prototype implementation, with respect to whether the iteration limit will only be exercised in rare cases in practice, is unfortunately rather limited up to now, because we have been mostly exercising the prototype with "desugared" versions of *valid* Haskell programs (with type classes replaced by assume clauses and instances becoming normal declarations).

Even if constrained types are restricted as e.g. in Haskell 98, where type classes can only have a single parameter, time complexity of satisfiability of a constraint on x still grows exponentially with the number of assumptions for x in the typing context [25, 30]. In cases of heavily overloaded symbols, the performance of the type inference algorithm thus degrades *if* an open world approach is not used (so as to restrict satisfiability to when

[2]Two types σ, σ' are unifiable if there exists a substitution S such that $S\sigma = S\sigma'$.

overloading is resolved). A typical situation occurs with the overloading of `(==)`, which is defined for basic values (integers, floating point numbers etc.), lists, pairs, triples etc.

Recent work has been developed in order to define suitable conditions on types of overloaded symbols that guarantee decidability of type inference and are not too restrictive in practice [27, 5, 16]. However, we are reluctant at the moment to incorporate such restrictions, because of the added complexity to the language, the abscence of a semantic characterization for such restrictions, and because restrictions prevent nontermination but not termination after a very long time (shouldn't the bottom line be lower/stricter?).

7 Ambiguity

A full discussion of type ambiguity and semantics coherence in the context of type systems with support for constrained polymorphism is, in our view, a subject that has not yet been investigated in sufficient depth. Usually, an expression e is considered *ambiguous* if two distinct denotations can be obtained for it, using a semantics defined inductively on the derivation of a type for e [22]. A so-called *coherent* semantics does not specify a meaning to such ambiguous expressions. With respect to derivations in the type system, we can translate this requirement so as to avoid the existence of two or more derivations of the same type for e that assign distinct non-unifiable types for some subterm of e. There is also a third, more syntactic characterization, which is the one we shall briefly consider below (again, we are not aware of any in-depth work relating any of these).

Consider a type with constraint-set κ and simple type τ, occurring in a typing context Γ. We can consider, in type systems for constrained polymorphism:

DEFINITION 9. $\kappa.\tau$ is ambiguous if there exist two distinct minimal solutions S_1 and S_2 to the satisfiability of κ in Γ such that $S_1\tau = S_2\tau$. □

This definition can be relaxed, allowing a greater set of expressions to be well-typed, if we consider instead:

DEFINITION 9'. $\kappa.\tau$ is ambiguous if there exists a minimal solution S to the satisfiability of κ in Γ such that all other minimal solutions S' are such that $S\tau = S'\tau$. □

With Definition 9, an expression such as, for example, $f\ x$ is not considered ambiguous in a typing context containing $\{f : \mathit{Int} \to \mathit{Int}, f : \mathit{Int} \to \mathit{Float}, f : \mathit{Float} \to \mathit{Float}, x : \mathit{Int}, x : \mathit{Float}\}$. The motivation for Definition 9 is clear: even though there exist two distinct type derivations for $f\ x : \mathit{Float}$, it can be effectively used in a context requiring it to have type *Int*. It is worthwhile to note though that Definition 9 would contradict (in the example above) a usual definition of semantics coherence.

Though we haven't unfortunately proved it yet, we expect that ambiguity as given in Def. 9 is sufficient to reject all derivations that would lead to semantic incoherence.

In an open world, ambiguity must be tested, as satisfiability itself, only when the satisfiability condition indicates so. Also, the use of constraint-projection, on which the satisfiability condition is based upon, to avoid erroneous ambiguity detection — in Haskell, in the case of MPTCs — eliminates the need for the programmer to specify functional dependencies (FDs), although it has the drawback of removing from the programmer the possibility of using FDs in order to close the world. With the help of constraint projection, the world is closed automatically when (but only when) defined by the

satisfiability-trigger condition. In our view, the use of constraint projection is an adequate tool for closing the world, releaving the programmer from the burden of specifying FDs. For example, the use of constraint projection would avoid the ambiguity that would be reported if FDs are removed from the class declaration below, since the type of g would be $SM\ m\ r \Rightarrow m\ a$.

```
class SM m r | m -> r, r -> m where
  { new:: a → m(r a); read:: r a → m a; write:: ...}

g x = do { r ← new x; read r }
```

8 Conclusion

We provide formal characterizations, in the context of constrained polymorphism, of open and closed worlds, constraint-set satisfiability and simplification, overloading resolution and type ambiguity. We define a type system for supporting both open and closed world approaches together and an algorithm for computing principal typings. Relative advantages between open and closed worlds are presented and briefly discussed, which suggest that both should be supported in a programming language. The theoretical framework and discussions presented provide a contribution to a better understanding of concepts and to further work and evolution of programming languages in this area.

References

[1] Carlos Camarão, Lucília Figueiredo, and Cristiano Vasconcellos. Constraint-set Satisfiability for Overloading. In *Proc. of ACM PPDP'04*, pages 67–77, 2004.

[2] K. Chen, Paul Hudak, and Martin Odersky. Parametric Type Classes. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 170–181, 1992.

[3] Luís Damas and Robin Milner. Principal type schemes for functional programs. In *Proc. of POPL'82*, pages 207–212, 1982.

[4] Fergus Henderson David Jeffery and Zoltan Zomogyi. Type Classes in Mercury. Technical Report 98/13, University of Melbourne, 1998.

[5] Gregory J. Duck, Simon Peyton Jones, Peter J. Stuckey, and Martin Sulzmann. Sound and Decidable Type Inference for Functional Dependencies. In *Proceedings of ESOP'2004 (European Symposium on Programming)*, 2004.

[6] Dominic Duggan, Gordon Cormack, and John Ophel. Kinded type inference for parametric overloading. *Acta Informatica*, 33(1):21–68, 1996.

[7] Dominic Duggan and John Ophel. Open and closed scopes for constrained genericiy. *Theoretical Computer Science*, 275(1–2):215–258, 2002.

[8] Dominic Duggan and John Ophel. Type checking multi-parameter type classes. *Journal of Functional Programming*, 12(2):135–158, 2002.

[9] Lucília Figueiredo and Carlos Camarão. Principal Typing and Mutual Recursion. In *Proc. WFLP'2001 (Int'l Workshop on Funct. & Logic Prog.)*, pages 157–170, 2001.

[10] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type Classes in Haskell. *ACM TOPLAS*, 18(2):109–138, 1996.

[11] Fergus Henderson et al. The Mercury Project, 2003. http://www.cs.mu.oz.au/research/mercury.

[12] Mark Jones. *Qualified Types*. Cambridge University Press, 1994.

[13] Mark Jones. A system of constructor classes: overloading and higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–36, 1995.

[14] Mark Jones. Simplifying and Improving Qualified Types. In *Proc. FPCA'95: ACM Conf. on Functional Prog. and Comp. Architecture*, pages 160–169, 1995.

[15] Mark Jones. Typing Haskell in Haskell. In *Proc. of Haskell Workshop'99, TR UU-CS-1999-28, Comp. Science Dept., Utrecht Univ.*, 1999. http://www.cse.ogi.edu/~mpj/thih.

[16] Mark Jones. Type Classes with Functional Dependencies. In *Proc. of ESOP'2000*, 2000. Springer-Verlag LNCS 1782.

[17] Mark Jones et al. Hugs98. http://www.haskell.org/hugs/, 1998.

[18] Mark P. Jones. Dictionary-free Overloading by Partial Evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Prog. Manipulation*, 1994.

[19] Simon P. Jones et al. GHC: The Glasgow Haskell Compiler. http://www.haskell.org/ghc.

[20] Simon P. Jones et al. *Haskell 98 Language and Libraries*. Cambridge Univ. Press, 2003.

[21] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[22] John Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[23] Tobias Nipkow and Christian Prehofer. Type Reconstruction for Type Classes. *Journal of Functional Programming*, 1(1):1–100, 1993.

[24] Martin Odersky, Philip Wadler, and Martin Wehr. A Second Look at Overloading. In *Proc. of ACM Conf. on Functional Prog. and Comp. Archit.*, pages 135–146, 1995.

[25] Helmut Seild. Haskell Overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.

[26] Geoffrey Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2-3):197–226, 1994.

[27] Martin Sulzmann et al. Understanding functional dependencies via Constraint Handling Rules. *Journal of Functional Programming*, 17(1), 2007.

[28] Cristiano Vasconcelos, Lucília Figueiredo, and Carlos Camarão. Practical Type Inference for Polymorphic Recursion: an Implementation in Haskell. *Journal of Universal Computer Science*, 9(8):873–890, 2003.

[29] Dennis Volpano and Geoffrey Smith. On the Complexity of ML Typability with Overloading. In *Proc. ACM Symp. Func. Prog. Comp. Archit.*, pages 15–28, 1991. LNCS 523.

[30] Dennis M. Volpano. Haskell-style Overloading is NP-hard. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 88–94, 1994.

[31] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proc. of ACM POPL'89*, pages 60–76. ACM Press, 1989.

Using Visitor Patterns in Object-Oriented Action Semantics

André Murbach Maidl[1], **Cláudio Carvilhe**[2], **Martin A. Musicante**[1,3]

[1]PPGInf - Programa de Pós-Graduação em Informática
Universidade Federal do Paraná (UFPR) – Curitiba – PR – Brazil

[2]Departamento de Informática
Pontifícia Universidade Católica do Paraná (PUCPR) – Curitiba – PR – Brazil

[3]DIMAp - Programa de Pós-Graduação em Sistemas e Computação
Universidade Federal do Rio Grande do Norte (UFRN) – Natal – RN – Brazil

murbach@inf.ufpr.br, carvilhe@ppgia.pucpr.br, mam@dimap.ufrn.br

Abstract. *Object-Oriented Action Semantics incorporates some object-oriented concepts to the Action Semantics formalism. Its main goal is to obtain more readable and reusable semantics specifications. Furthermore, it supports syntax-independent specifications due to the way classes are written. In a previous work, a library of classes (called LFL) was developed to improve specification reuse and to provide a way to describe semantics concepts, independent from the syntax of the programming language. This paper aims to address some problematic aspects of LFL, as well as a case study, where a specification is built by using the Visitor Pattern technique. The use of this pattern allows a clear separation between the syntax of a programming language and its different semantic aspects.*

1. Introduction

Action Semantics [Mosses 1992, Watt 1991] is a formal framework for defining the semantics of programming languages. A main goal of Action Semantics is to provide a notation that is intuitive for programmers. This notation has been used to describe the semantics of real programming languages and presents good reusability and extensibility properties [Mosses and Musicante 1994]. However, the standard Action Semantics lacks some syntactic support for the definition of libraries, and reusable components [Gayo 2002].

A modular approach to Action Semantics is presented in [Doh and Mosses 2001]. Their goal is to extend the Action semantics notation to define modules. The interpretation of modules introduces some new, potential problems to the semantics of the whole specification [Doh and Mosses 2001]. These problems are related to the way the modules and their hierarchy are interpreted.

Based on the modular approach, an object-oriented view of Action Semantics is proposed in [Carvilhe and Musicante 2003]. This approach is called Object-Oriented Action Semantics (OOAS). Using OOAS it is possible to organize Action Semantics descriptions into classes. OOAS does not have the problems of Modular Action Semantics since the class hierarchy is taken into account for the interpretation of a semantic description.

The **LFL** (Language Features Library) has been proposed in [Araújo and Musicante 2004] to create new OOAS descriptions by composing and

reusing classes. The main goal of LFL is to define generic classes to describe general semantic concepts. These classes can be instantiated to be used in conjunction to the syntax of a specific programming language. LFL introduced some syntactic problems to Object-Oriented Action Semantics descriptions. In particular, the way in which generic classes were defined is cumbersome, resulting in less readable specifications.

In this work, we propose LFLv2 (read LFL version 2), a new version of LFL to solve the problematic issues in the original library. Moreover, we incorporate in LFLv2 some of the ideas presented in [Mosses 2005, Iversen and Mosses 2005], in order to separate the syntax of the programming language from the specification of its semantics. In addition to it, we give examples of Object-Oriented Action Semantics descriptions using the Language Features Library version 2 and also using the Visitor Pattern technique [Gamma et al. 1998, Appel and Palsberg 2003] to improve the modularity aspects observed in the object-oriented approach.

This work is organized as follows: the next section briefly introduces Action Semantics and OOAS. Section 3 sumarizes the LFL. In section 4 we introduce the use of the Visitor Patterns technique, as well as LFLv2. A simple imperative language is specified as a case study in section 5. Section 6 gives the conclusions of this work.

2. Action Semantics and OOAS

Action Semantics [Mosses 1992, Watt 1991] is a formal framework developed to improve the readability and the way of describing programming languages semantics. The framework is based on denotational semantics and operational semantics. In Action Semantics, semantic functions specify the meaning of the phrases of a language using *actions*. These actions represent the denotation of the language phrases. The action notation is defined operationally and contains the actions and action combinators needed in Action Semantics descriptions.

The three main mathematical entities that compose Action Semantics descriptions are: *actions*, *data* and *yielders*. Actions can be performed to represent the concepts of many programing languages, such as: control, data flow, scopes of bindings, effects on storage and interactive process. When an action is performed it can produce one of the following outcomes: normal termination (*complete*), exceptional termination (*escape*), unsuccessful termination (*fail*) or non-termination (*diverge*). Action notation provides some basic actions which are written using English words in order to improve readability in semantics specifications. These actions can be combined to obtain more complex actions.

A data notation is used in Action Semantics to provide the needed sorts of data and operations related to them. These sorts of data can be: truth-values, numbers, characters, strings, tuples, maps, tokes, messages, etc.

Yielders are entities that can produce data during an action performance. Such produced data are called *yielded data*. The produced data depend on the current information processed, for instance: the *transient data* passed, the *bindings* received and the current *storage* state. Actions have *facets*, according to their data flow:

- **basic** - deals with pure control flow;
- **functional** - deals with actions that process transient data;

- **declarative** - deals with actions that produce or receive bindings;
- **imperative** - deals with actions that manipulate the storage;
- **reflective** - deals with abstractions;
- **communicative** - deals with actions that operate under distributed systems.

Object Oriented Action Semantics (OOAS) [Carvilhe and Musicante 2003] is a method to organize Action Semantics specifications. Its main goal is to provide extensibility and reusability in Action Semantics by splitting semantic descriptions into classes and treating semantic functions as methods.

In order to overcome the lack of modularity in Action Semantics, as reported by [Gayo 2002], Object-Oriented Action Semantics introduces an extension of the standard Action Notation. Class constructors and several other object-based operators are available, to provide an object-oriented way of composing specifications. An Object-Oriented Action Semantics description is a ***hierarchy*** of classes. Each class encapsulates some particular Action Semantics features. Those features can be easily reused or specialized by other objects, using well-known object-orientation concepts. As an example, we will take a look at a simple command language, whose BNF is defined as follows:

```
Command ::= Identifier ":=" Expression | Command ";" Command |
            "if" Expression "then" Command 'else" Command "end-if" |
            "while" Expression "do" Command
```

This equation defines syntactic trees for commands, containing assignments, sequences, conditionals and iterations. A simple strategy for defining an Object-Oriented Action Semantics description of this language is to define Command as an abstract class, and to define sub-classes of commands for each class of phrase. A base-class can be written as:

```
Class Command
    syntax:
        Cmd
    semantics:
        execute _ : Cmd → Action
End Class
```

Notice that Command is the abstract class. Initially it introduces the syntactic sort Cmd, detailed in the syntax section. Semantics part defines the semantics of a Command, using standard Action Notation. In this case, it establishes that a semantic function execute maps a Command to an Action. A plain semantic function like execute is seen as a method.

The Command definition provides a foundation to specify the other classes. We can now define every particular class as a sub-class of Command. Lets take a look at the Selection class as follows:

```
Class Selection
    extending Command
    using E:Expression, C1:Command, C2:Command
    syntax:
        Cmd ::= "if" E "then" C1 "else" C2 "end-if"
    semantics:
        execute ⟦ "if" E "then" C1 "else" C2 "end-if" ⟧ =
            | evaluate E then execute C1 else execute C2
End Class
```

Selection is the specialized class. The extending directive states a particular Command behavior (in this case, a conditional command). The using directive makes other objects available.

A complete Object Oriented Action Semantics specification can be found in [Carvilhe and Musicante 2003].

3. Language Features Library (LFL)

In programming languages specifications it is common to find pieces of the specification that have similar concepts. When it happens is interesting to reuse these pieces instead of having duplicated code. The classes concept introduced by Object-Oriented Action Semantics [Carvilhe and Musicante 2003] helps with the code reuse due to the common structure provided by organizing the specification into classes.

In [Araújo and Musicante 2004], was proposed a library of classes which was called **LFL** (Language Features Library). The function of LFL is to congregate and organize classes, that have common specifications, into a structure and work as a repository of classes for Object-Oriented Action Semantics descriptions.

A tree structure was adopted to represent the classes' organization in LFL. LFL was branched off into three main classes: *Syntax*, *Semantics* and *Entity*.

The node *Semantics* was the only one implemented in the initial version of LFL and it also forked into another three nodes: *Declaration*, *Command* and *Expression*. Pieces of code that manipulate bindings are treated in *Declaration*; *Command* represents semantics definitions that are concerned about data flow and the storage; while those classes for the the manipulation os values belong to *Expression*.

Each one of the above-defined nodes was split in two: *Paradigm* and *Shared*. The first one is responsible to represent classes that contain features from a specific programming language paradigm, such as: *Object-Oriented*, *Functional*, *Logical* and *Imperative*. The node *Shared* contains classes which represent features that are common to more than one programming paradigm.

The following example illustrates the definition of a LFL class. This class defines the semantics of an if-then-else command:

```
Class Selection
        ≪ Command implementing < execute _ : Command → Action >
        Expression implementing < evaluate _ : Expression → Action > ≫
    locating LFL.Semantics.Command.Shared
    using E:Expression C1:Command, C2:Command
    semantics:
        execute-if-then-else(E, C1, C2) =
            evaluate ⟦ E ⟧ then
            | check(the given TruthValue is true) and then execute ⟦ C1 ⟧
            or
            | check(the given TruthValue is false) and then execute ⟦ C2 ⟧
End Class
```

The Selection class provides the semantics of a selection command in a syntax-independent style. The classes Command and Expression are passed as (higher order)

parameters. Notice that the generic arguments must comply with the restrictions defined by the implementing directive: They must provide some methods with a specific type as sub-parameters.

The directive locating identifies the place where the class is located in the LFL structure.

The programming language specifications which use the LFL are similar to the plain Object-Oriented Action Semantics descriptions. Let us now exemplify the use of the generic Selection class to define the semantics of a selection command:

```
Class MyCommand
    syntax:
        Com
    semantics:
        myexecute _ : Com → Action
End Class

Class MySelection
    extending MyCommand
    using C1:MyCommand, C2:MyCommand, E:MyExpression,
        objSel:LFL.Semantics.Command.Shared.Selection ≪
            MyCommand<myexecute>, MyExpression<myevaluate> ≫
    syntax:
        Com ::= "if" E "then" C1 "else" C2 "end-if"
    semantics:
        myexecute ⟦ "if" E "then" C1 "else" C2 "end-if" ⟧ =
            objSel.execute-if-then-else(E, C1, C2)
End Class
```

The classes MyCommand and MySelection, as defined above, specify the behaviour of a selection command using the LFL. Notice that the LFL generic class Selection is instantiated. The argument for this LFL class are the MyCommad and MyExpression classes. Notice that the sub-paramenters are also provided.

LFL brings an interesting concept to Object-Oriented Action Semantics: the definition of semantics descriptions which are independent from the syntax of the programming language. This definition uses the inclusion of classes provided by a library of classes. A main issue with the way in which LFL is defined is that its parameter-passing mechanism is obscure and let us back to the readability problems found in other semantics descriptions of programming languages.

4. Visitor Patterns

In the previous sections we have presented Object-Oriented Action Semantics, an approach for language definition using Action Semantics, and Object-Oriented concepts. We also have presented LFL, a library of classes for Object-Oriented Action Semantics descriptions. The former is based on the modularity in Action Semantics by splitting descriptions into classes, the latter is a collection of Object-Oriented Action Semantics classes to be used and instanced in different programming languages projects.

As it is proposed in [Mosses 2005, Iversen and Mosses 2005], we are looking for approaches that allow us to describe programming languages semantics syntax indepen-

dently. LFL clearly does it, however it has a main problematic issue, related to the way instantiation of classes is done.

In this work, we propose some changes in LFL for its improvement, namely, using LFL with no parameters passing. Visitor Patterns [Gamma et al. 1998] are added to OOAS descriptions that use LFL in order to try to increase the modularity aspects observed in the object-oriented framework. Visitor Patterns are used to define operations that can be performed in the elements of an object structure. They allow the definition of a new operation without changing the classes of the elements in which these operate.

4.1. Object-Oriented Action Semantics and Visitor Patterns

Visitor Pattern is a common object-oriented technique used in compiler construction. Visitor Pattern helps the compiler writer to provide several semantics to the same syntactic tree.

We can use the Visitor Pattern technique to give an interpretation to each syntax tree in OOAS. This interpretation is an (OOAS) object, which has a *visit* method for each syntax tree defined. Each Object-Oriented Action Semantics class should implement an *accept* method to serve as a hook for all interpretations. When an *accept* method is called by a visitor, the correct *visit* method is invoked. Such control can go back and forth between visitors and classes [Appel and Palsberg 2003].

Informally, when the visitor calls the *accept* method it is in fact asking: "who are you?". The question is answered by the *accept* method as a result of the calling of the correspondent *visit* method of the visitor. The following examples illustrate how to use Object-Oriented Action Semantics with the Visitor Pattern. Notice that each *accept* method takes a visitor as a parameter and that each *visit* method takes a syntax tree object as a parameter.

```
Class Command
    syntax:
        Cmd
    semantics:
        accept _ : Visitor → Action
End Class

Class Selection
    extending Command
    using E:Expression, C1:Command, C2:Command
    syntax:
        Cmd ::= "if" E "then" C1 "else" C2 "end-if"
    semantics:
        accept V:Visitor = visit V
End Class

Class Visitor
    syntax:
        Vis ::= Command
    semantics:
        visit _ : Command → Action
End Class
```

```
Class Interpreter
    extending Visitor
    semantics:
        visit ⟦ "if" E "then" C1 "else" C2 "end-if" ⟧ =
            evaluate E then accept C1 else accept C2
End Class
```

Notice that a *visitor* is in fact a Command syntax tree object encapsulated by the Visitor class. When the accept method is performed, the correct visit method is called to interpret the syntax tree argument that actually is a *visitor*. In section 2 the interpreter was implemented in the execute methods while now it is in the Interpreter class. The evaluate method is still used in the traditional way of Object-Oriented Action Semantics.

With the Visitor Patterns we can add new interpretations without editing existing classes since that each class has its own accept method. In section 5 we will see that accept and visit methods can be used with different names when we need more than just one visitor, as was used in this section.

4.2. LFLv2 and Visitor Patterns

LFL is a good approach to be used as a programming language semantics description tool for specifying programming languages semantics syntax independently since it has support for generic classes definitions [Araújo and Musicante 2004]. Nonetheless, the obscure LFL syntax took us back to the readable problems found in semantic frameworks. An alternative way to inhibit parameters passing in LFL is the use of abstractions of actions to specify the semantics of some common concepts of programming languages.

Furthermore, we also propose the LFL usage just for semantic concepts. It implies in the exclusion of the LFL nodes: *Syntax*, *Semantics* and *Entity*. Thereby, we can bind the nodes *Declaration*, *Command* and *Expression* directly to the main LFL class since we will use LFL just to represent semantic concepts. The nodes *Shared* and *Paradigm* are maintained, as well as their respective subdivisions. The former is concerned with constructs that commonly appear in programming languages and the latter represents some specialized constructs that appear in specific paradimgs.

As in the first version of the library, the set of classes is very reduced and can be extended to support new classes. According to our proposal, the Selection class example shown in section 3 would be written in the following way:

```
Class Selection
    locating LFL.Command.Shared.Selection
    using: Y1:Yielder, Y2:Yielder, Y3:Yielder
    semantics:
        if-then-else(Y1,Y2,Y3) =
            | enact Y1 then enact Y2 else enact Y3
End Class
```

Now, LFL classes are more compact since they just define their own location in the LFL hierarchy, the methods that describe the semantics of some concepts of programming languages and the objects used in the specified methods.

Instead of using methods that have to be passed as parameters, each method implemented by the new library of classes receives a number of abstractions that are performed

according to the language behavior that is being represented. The enact action receives *yielders* that result in abstractions (and encapsulate actions). These actions are performed independently from the methods described in the programming language specification. Notice that the use of the Visitor Pattern technique is something that is not tied to the new library. Althoug it has been used since it might improve modularity in OOAS descriptions that use LFL. Let us now see how to use the new LFL together with the Visitor Pattern:

```
Class MyCommand
    syntax:
        Com
    semantics:
        myexecute _ : Visitor → Action
End Class

Class MySelection
    extending MyCommand
    using E:MyExpresssion, C1:MyCommand, C2:MyCommand
    syntax:
        Com ::= "if" E "then" C1 "else" C2 "end-if"
    semantics:
        execute V:Visitor = visit V
End Class

Class Visitor
    using O:LFL
    syntax:
        Vis ::= MyCommand
    semantics:
        visit _ : MyCommand → Action
End Class

Class Interpreter
    extending Visitor
    semantics:
        visit ⟦ "if" E "then" C1 "else" C2 "end-if" ⟧ =
            O.Command.Shared.Selection.execute-if-then-else(
                    closuse abstraction of (myevaluate E),
                    closuse abstraction of (myexecute C1),
                    closuse abstraction of (myexecute C2) )
End Class
```

The classes are again defined as in Object-Oriented Action Semantics using Visitor Patterns. The language syntax and the semantic function myexecute are also defined in function of the visitors approach. In this way, the semantics descriptions by the library usage are given in the visitor implementation.

The object *O*, defined in the Visitor class, represents all LFL classes and they can be used due to the hierarchy of classes created by LFL. The visit method describes MyCommand semantics used in the language passing the abstractions to the execute-if-then-else method specified in LFL.

The specification above defines a language with static bindings. If the language being described has dynamic bindings, the directives *closure* should be taken from the specification.

The notation encapsulate X will be used in the rest of this paper as an abbreviation of closure abstraction of (X).

5. A case study

In this section we present the specification of a toy language called μ-Pascal. This language is fairly similar to the Pascal language. μ-Pascal is an imperative programming language containing basic commands and expressions. The μ-Pascal language will be specified using Object-Oriented Action Semantics, the new LFL proposed in section 4.2 and using the Visitor Pattern technique.

5.1. Abstract syntax

Now we present the abstract syntax for the μ-Pascal language:

(1) Program ::= "begin" Declaration ";" Command "end"

(2) Declaration ::= "var" Identifier ":" Type ⟨ = Expression ⟩$^?$ |
"const" Identifier ":" Type "=" Expression |
Declaration ";" Declaration

(3) Type ::= "boolean" | "integer"

(4) Command ::= Identifier ":=" Expression | Command ";" Command |
"if" Expression "then" Command ⟨ "else" Command ⟩$^?$ "end-if" |
"while" Expression "do" Command

(5) Expression ::= "true" | "false" | Numeral | Identifier |
Expression ⟨ "+" | "-" | "*" | "<" | "=" ⟩ Expression

(6) Identifier ::= Letter ⟨ Letter | Digit ⟩*

(7) Numeral ::= Digit ⟨ Digit ⟩*

In the above abstract syntax we have defined that a Program is a sequence of Declaration and Command. Declarations can be integer or boolean constants and variables. Assignments, sequences, selections and iterations are defined as Commands. Expressions might be arithmetical or logical.

5.2. μ-Pascal semantics using LFLv2 and Visitor Patterns

In this section we will demonstrate the use of the Object-Oriented Action Semantics approach with the advantages of the new LFL and the Visitor Pattern technique.

First of all, we will define the basic classes that implement some particular concepts of the example language, like identifiers, numerals and types. After that we will exemplify how declarations, commands and expressions are treated. Then we will show how the *visitors* can join all together, giving the semantics of the language.

```
Class Identifier
    syntax:
        Id ::= letter [ letter | digit ]*
End Class
```

The class Identifier has just the syntactic part since it is used only for representing the name of variables and constants used in the language.

```
Class Numeral
    syntax:
        N ::= digit+
    semantics:
        valuation _ : N → integer
End Class
```

In the class Numeral we define what kind of numbers are used by the example language. In this case the numbers are integers and the method valuation is defined to map a numeral received as a parameter to its respective integer. Notice that the valuation parameter is a syntactic entity.

```
Class Type
    syntax:
        T ::= "boolean" | "integer"
    semantics:
        sort-of _ : T → Sort
        sort-of ⟦ "boolean" ⟧ = truth-value
        sort-of ⟦ "integer" ⟧ = integer
End Class
```

The Type class is used to define that the language data types can be boolean or integer. The method sort-of maps the language data types to their correct Object-Oriented Action Semantics *sorts*.

Hence, we will see the methods parameters as *visitors* that carries syntactic entities. In fact, the syntactic entities are encapsulated by the *visitor* objects and they will be interpreted by the *visitor* which implements the semantics of the encapsulated syntax.

```
Class Declaration
    syntax:
        Dec
    semantics:
        elaborate _ : Visitor → Action
End Class
```

The class Declaration works as an abstract class. It introduces the sort Dec which will be redefined in the sub-classes to represent each Declaration. Both methods accept and visit, from section 4, are represented by elaborate and visitDec. Notice that Declaration is just an abstract class, reason why visitDec appears just in its sub-classes.

```
Class Variable
    extending Declaration
    using I:Identifier, E:Expression, T:Type
    syntax:
        Dec ::= "var" I ":" T [ "=" E ]
    semantics:
        elaborate V:Visitor = visitDec V
End Class
```

Constructing the declarations classes hierarchy, now we have defined the Variable class. The Dec token is redefined giving the variables declarations syntax used in the

language. The elaborate method is overloaded to express the semantics of a variable declaration using the visitDec method.

The method elaborate takes a *visitor* as an argument and through the method visitDec gives the declaration performance in the Visitor class. The declaration is possible to be performed in the Visitor class since V represents the encapsulated syntax. In this manner, the syntax carried by the *visitor* object can be checked by the correct visit method.

Class Command
    syntax:
        Cmd
    semantics:
        execute _ : Visitor → Action
End Class

We now have the Command class. In this class we introduce the syntactic sort Cmd which will be redefined in Command sub-classes. In commands we will define the accept method as execute and the visit method as visitCmd. Notice that, like Declaration, Command also is an abstract class.

Class While
    extending Command
    using C:Command, E:Expression
    syntax:
        Cmd ::= "while" E "do" C
    semantics:
        execute V:Visitor = visitCmd V
End Class

In the While class we express how a while-loop works in the language. To achieve the before mentioned result, we have created the super-class Command and the sub-class While. Using Visitor Patterns, in the abstract class we specify just the abstract method which defines that a *visitor* results in an *action*. A command is correctly executed by calling the execute method with the command syntax. The correct semantics will be given by the visitCmd method since it takes the *visitor* object and interprets the syntax, that is in the object, in the Visitor class.

Class Expression
    syntax:
        Exp
    semantics:
        evaluate _ : Visitor → Action
End Class

Again we have an abstract class, like Declaration and Command. The Expression class is the super-class for the expressions definitions. The accept and visit methods will be represented by evaluate and visitExp, respectively. The sub-classes of Expression can be defined similarly to Declaration and Command sub-classes definitions. Now we will see how the Visitor class works.

```
Class Visitor
    using: O:LFL
    syntax:
        Vis ::= Declaration | Command | Expression
    semantics:
        visitCmd _ : Command → Action
        visitDec _ : Declaration → Action
        visitExp _ : Expression → Action
End Class
```

In the Visitor class we specify all the signatures of the visit methods which represent the maps of a class, containing a syntactic tree that will be visited, to an action. We also specify that a Visitor may carry a Declaration, a Command or an Expression syntax tree object. The Object-Oriented Action Semantics descriptions are given in the Interpreter class using the methods provided by **LFL**. The LFL methods are accessed through the object *O* which is a LFL instance.

```
Class Interpreter
    extending Visitor
    semantics:
        visitDec ⟦ "var" I ":" T ⟧ =
            O.Declaration.Paradigm.Imper.VarDec.elaborate-variable(I, sort-of T)
        visitDec ⟦ "var" I ":" T "=" E ⟧ =
            O.Declaration.Paradigm.Imper.Variable.elaborate-variable(I, sort-of T,
                encapsulate evaluate E)
        ...
        visitCmd ⟦ C1 ";" C2 ⟧ =
            O.Command.Shared.Sequence.execute-sequence(
                encapsulate execute C1,
                encapsulate execute C2)
        ...
        visitCmd ⟦ "while" E "do" C ⟧ =
            O.Command.Shared.While.execute-while(encapsulate evaluate E,
                encapsulate execute C)
        ...
        visitExp ⟦ N ⟧ = give valuation N
        visitExp ⟦ I ⟧ = O.Expression.Shared.Identifier.evaluate-identifier(I)
        visitExp ⟦ E1 "+" E2 ⟧ =
            O.Expression.Shared.Sum.evaluate-sum(encapsulate evaluate E1,
                encapsulate evaluate E2)
        ...
End Class
```

Notice that the *visitors* visit a syntax tree, specified in the Object-Oriented Action Semantics classes, and give the correct semantics to the syntax tree visited. It is possible by calling the accept method, this method takes a *visitor* represented by a syntax tree object and then the correct visit method will be performed to give the semantics of the syntax tree carried by the *visitor*.

The semantics expressed by the visit methods might be merely an *action*, like in the numeral valuation, just a LFL method that results in an *action*, like in the identifiers evaluation, or we can use accept calls to visit the needed syntactic entities to be used with LFL methods; in the example the *actions* are encapsulated for generating *abstractions* and

these *abstractions* are passed to some LFL method that implements the desired semantics. We can also pass *tokens* and *sorts* to the new LFL [Maidl 2007].

```
Class Micro-Pascal
    using D:Declaration, C:Command
    syntax:
        Prog ::= "begin" D ";" C "end"
    semantics:
        run _ : Prog → Action
        run ⟦ "begin" D ";" C "end"⟧ = elaborate D hence execute C
End Class
```

Since Object-Oriented Action Semantics allow us to organize the specification into classes, we can define a main class. The main class in this example is the Micro-Pascal class. In this class we specify the syntax of a μ-Pascal program and that it is mapped to an action. The declaration semantics and command semantics are given by calling their accept methods and passing their respective *visitors*.

6. Conclusions

Object-Oriented Action Semantics was designed to improve modularity in Action Semantics descriptions. LFL has improved the reusability in the object-oriented approach and also has provided a way to describe the semantics of programming languages in a syntax-independent style. The problems found in the first version of LFL were solved by the new version of it, through the use of abstractions of actions, instead of passing classes and methods as arguments to LFL classes. The library of classes was also restructured to implement only those pieces that are concerned with representing the semantics of programming languages.

LFL became more concise and simplified due to the use of the reflective facet. The use of the Visitor Pattern improves modularization in OOAS. Notice that this Pattern can be used either with just plain OOAS or with OOAS plus LFL. Since the definition of the Visitor Pattern is inherently related to the Object-Oriented paradigm, the use of this pattern in conjunction with an Object-Oriented specification language like OOAS leads to clear and modular specifications.

In [Maidl 2007] the author proposes an implementation of Object-Oriented Action Semantics in Maude and LFLv2 as a case study of it.

As future work, we would investigate the use of the Visitor Pattern technique for compiler generation from OOAS descriptions and we also would trace a careful comparsion between LFLv2 and the construtive approach [Mosses 2005, Iversen and Mosses 2005].

The main contributions of this work can be summarized as:

- Modularity is improved. The use of Object-Oriented Action Semantics and Visitor Patterns provide a new view of modularity in semantics descriptions of programming languages.
- LFL is patched. We propose a cleaner and usable library of classes by its own restructuration and for the fact of changing arguments by abstractions.
- LFL can describe semantics syntax-independently. The hierarchy of classes made this possible.

Acknowledgments

We would like to thank the anonymous referees for their constructive comments and also Flávia Erika Shibata for her careful English review.

References

Appel, A. W. and Palsberg, J. (2003). *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA.

Araújo, M. and Musicante, M. A. (2004). Lfl: A library of generic classes for object-oriented action semantics. In *XXIV International Conference of the Chilean Computer Science Society (SCCC 2004), 11-12 November 2004, Arica, Chile*, pages 39–47. IEEE Computer Society.

Carvilhe, C. and Musicante, M. A. (2003). Object-oriented action semantics specifications. *Journal of Universal Computer Science*, 9(8):910–934.

Doh, K. and Mosses, P. (2001). Composing programming languages by combining action semantics modules. In *First Workshop on Language Descriptions, Tools and Applications*.

Gamma, E., Vlissides, J., Johnson, R., and Helm, R. (1998). *Design Patterns CD: Elements of Reusable Object-Oriented Software, (CD-ROM)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gayo, J. E. L. (2002). Reusable semantic specifications of programming languages. In *SBLP 2002 - VI Simpósio Brasileiro de Linguagens de Programação*.

Iversen, J. and Mosses, P. D. (2005). Constructive action semantics for core ML. *IEE Proceedings Software*. Special issue on Language Definitions and Tool Generation. To appear in Brics RS-04-37.

Maidl, A. M. (2007). Uma implementação da semântica de ações orientada a objetos em maude. Master's thesis, Universidade Federal do Paraná. (In preparation).

Mosses, P. (1992). Action semantics. In *Action Semantics*. Cambridge University Press.

Mosses, P. D. (2005). A constructive approach to language definition. *Journal of Universal Computer Science*, 11(7):1117–1134.

Mosses, P. D. and Musicante, M. A. (1994). An action semantics for ML concurrency primitives. Number 873 in Lecture Notes in Computer Science, Barcelona, Spain. FME, Springer-Verlag.

Watt, D. (1991). In *Programming Language Syntax and Semantics*. Prentice Hall International (UK).

Uma Linguagem para Especificação e Combinação Dinâmica de Aspectos em Aplicações Orientadas a Serviços

Nabor C. Mendonça, Clayton F. Silva, Ian G. Maia, Tiago Cordeiro

Mestrado em Informática Aplicada, Universidade de Fortaleza

Av. Washington Soares, 1321, CEP 60811-905 Fortaleza – CE

nabor@unifor.br, clayton_fsilva@yahoo.com.br, iangmaia@gmail.com, tiagomail2@yahoo.com.br

Abstract. *This article presents the Web Service Aspect Language (WSAL), which integrates fundamental aspect-oriented programming concepts into the context of service-oriented development. Differently from existing solutions that aim at integrating these two emerging software development paradigms, WSAL supports a novel aspect model, in which aspects are also implemented and executed as services. This characteristic allows WSAL aspects to be dynamically woven into the message flow exchanged between service consumer and service provider applications, in a way that is completely decoupled from their implementation technologies. The article also reports on an initial implementation of an aspect weaver for WSAL, which is based on an existing HTTP intermediary technology.*

Resumo. *Este artigo apresenta a linguagem WSAL (Web Service Aspect Language), que integra conceitos fundamentais da programação orientada a aspectos ao contexto do desenvolvimento orientado a serviços. Diferentemente de outras soluções propostas na literatura que buscam integrar esses dois emergentes paradigmas de desenvolvimento de software, WSAL suporta um modelo de aspectos inovador, onde aspectos também são implementados e executados na forma de serviços. Essa característica permite a combinação dinâmica de aspectos especificados em WSAL ao fluxo de mensagens trocadas entre as aplicações consumidoras e provedoras de serviços, de uma maneira totalmente independente das tecnologias utilizadas na sua implementação. O artigo descreve ainda a implementação inicial de um combinador de aspectos para WSAL, baseado em uma tecnologia de intermediários HTTP existente.*

1. Introdução

A *Computação Orientada a Serviços* (COS) é um emergente paradigma da computação que utiliza serviços como elementos fundamentais de projeto [Papazoglou e Georgakopoulos 2003]. Para operar em um ambiente da COS, as aplicações provedoras de serviços devem, de forma declarativa, definir as características de suas operações (estrutura das mensagens de requisição e de resposta, protocolo de transporte, modo de invocação, etc.) em um formato previamente acordado e neutro [Curbera *et al*. 2003]. A partir dessas declarações, aplicações consumidoras de serviços podem, de forma automática ou semi-automática, descobrir, selecionar e invocar operações de serviços de seu interesse.

A manifestação mais comum da COS se dá no contexto da Web, com os chamados *serviços web* [Cauldwell *et al*. 2001]. Serviços web implementam funções básicas da COS utilizando um conjunto padronizado de tecnologias baseadas na linguagem de marcação XML, tais como: WSDL [Christensen *et al*. 2001], uma linguagem para descrição de serviços; UDDI [Bryan *et al*. 2002], um serviço de registro que permite a descoberta e localização de outros serviços; e SOAP [Gudin *et al*. 2003], um protocolo para a troca de mensagens entre serviços. Outras funções mais avançadas, como a composição de serviços e o estabelecimento de acordos para a provisão de serviços com garantias de qualidade de serviço, também já estão sendo implementadas por uma nova geração de tecnologias [Curbera *et al*. 2003].

No entanto, a natureza distribuída, heterogênea e fracamente acoplada das aplicações baseadas em serviços web faz com que muitos de seus interesses (ou preocupações) de projeto, em geral de cunho não-funcional, sejam difíceis de modularizar usando apenas as tecnologias tradicionais de implementação. Por dizer respeito tanto às aplicações provedoras quanto às aplicações consumidoras dos serviços envolvidos, o tratamento desses interesses tende a se espalhar pelo código fonte de diversas aplicações, misturando-se à implementação de seus interesses funcionais. Claramente, tal propriedade pode afetar negativamente o desenvolvimento de aplicações orientadas a serviços, dificultando não apenas a sua implementação, mas também a sua subseqüente manutenção e evolução.

A constatação de que existem propriedades de um sistema de software que não são facilmente modularizáveis utilizando modelos de desenvolvimento tradicionais (ou seja, modelos que têm a decomposição funcional como princípio fundamental de projeto), bem como a necessidade de oferecer novos mecanismos de composição para implementar tais propriedades de forma localizada, em separado dos demais módulos, constituem a principal motivação por trás do paradigma da *Programação Orientada a Aspectos* (POA) [*Kiczales et al*. 1997]. A POA propõe um novo tipo de abstração – denominado *aspecto* – o qual permite separar explicitamente interesses não-funcionais de um sistema que do contrário estariam misturados ao seu código funcional. A POA fornece ainda novos mecanismos de composição de software que permitem combinar interesses não-funcionais, implementados como aspectos, com o código funcional do sistema em locais ou pontos de execução (conhecidos como *pontos de junção*) definidos pelo programador.

Em vista da dificuldade de se separar interesses não-funcionais fortemente presente na COS, e dos recursos de separação explícita de interesses oferecidos pela POA, há uma demanda natural por soluções que busquem integrar esses dois paradigmas. Embora alguns trabalhos já tenham sido propostos recentemente nesta direção [Verheecke e Cibrán 2003, Verspecht *et al*. 2003, Charfi e Mezini 2004, Courbis e Finkelstein 2004, Henkel *et al*. 2005], todos, em menor ou maior grau, estão acoplados a uma determinada linguagem de programação ou plataforma de execução. Tal acoplamento está presente tanto na implementação das aplicações consumidoras e provedoras de serviços, quanto na implementação de seus interesses não-funcionais na forma de aspectos. O alto nível de acoplamento presente nessas soluções é indesejável por duas razões importantes: (i) ele limita o leque de tecnologias de implementação à disposição dos programadores, e (ii) ele vai de encontro à própria natureza heterogênea e fracamente acoplada que caracteriza a COS. Portanto, uma solução mais eficaz para

integrar aspectos e serviços deveria ter a característica de independência de tecnologia de implementação como um princípio básico de projeto.

Neste artigo, apresentamos a linguagem WSAL (*Web Service Aspect Language*), que permite a integração natural de alguns conceitos fundamentais da POA, como aspectos (*aspects*), pontos de junção (*join points*), adendos (*advices*) e combinação (*weaving*), ao contexto da COS. Diferentemente de outras soluções existentes que também buscam integrar esses dois paradigmas, WSAL suporta um modelo de aspectos inovador, onde aspectos também são implementados e executados na forma de serviços. Essa característica permite a combinação dinâmica (ou seja, no momento da execução das aplicações envolvidas, quando estas trocam mensagens) de aspectos especificados em WSAL ao fluxo de mensagens trocadas entre as aplicações consumidoras e provedoras de serviços, de uma maneira totalmente independente das tecnologias utilizadas na sua implementação.

A linguagem WSAL é descrita em detalhes na próxima seção. A seção 3 reporta a implementação inicial de um conjunto de ferramentas para dar suporte à linguagem, incluindo um combinador de aspectos que está sendo construído com base em uma tecnologia de intermediários HTTP existente. A seção 4 apresenta os resultados de uma avaliação preliminar do desempenho do combinador em diferentes cenários de execução. A seção 5 compara WSAL com diversos trabalhos relacionados. Por fim, a seção 6 conclui o artigo com as principais contribuições do trabalho e sugestões para trabalhos futuros.

2. A Linguagem WSAL

WSAL é uma nova linguagem da POA que tem como base um novo modelo de combinação de aspectos, proposto especificamente para o contexto do desenvolvimento orientado a serviços [Mendonça e Silva 2006]. Nesse modelo, aspectos (e não apenas as aplicações) também são implementados e executados na forma de serviços web fracamente acoplados, chamados *serviços aspectuais* [Mendonça e Silva 2005]. Dessa forma, o processo de combinação dos aspectos passa a ser realizada em nível de rede (portanto, externamente ao ambiente de execução das aplicações), por meio de um mecanismo de interceptação dinâmica de mensagens que faz o papel de combinador de aspectos da linguagem. O combinador tem como principal função interceptar as mensagens SOAP trocadas entre as aplicações através da rede e, uma vez que tenham sido identificados eventos de interesse nas mensagens interceptadas, invocar as operações (adendos) apropriadas do serviço aspectual especificado.

Essa estratégia de implementação de aspectos como serviços, utilizada originalmente em WSAL, fundamenta-se na visão de que, para que os paradigmas da POA e COS possam ser integrados de forma natural, é necessário que o próprio processo de combinação de aspectos seja realizado de forma desacoplada de tecnologia.

2.1. Processo de Combinação

A Figura 1 ilustra o processo de combinação de aspectos em WSAL. O processo inicia com a especificação (passo 1), interpretação (passo 2) e implantação (passo 3) dos aspectos especificados em WSAL pelo combinador de aspectos da linguagem. A combinação em si acontece em tempo de execução, através da interceptação de mensagens SOAP de interesse (passo 4), invocação dos serviços aspectuais apropriados

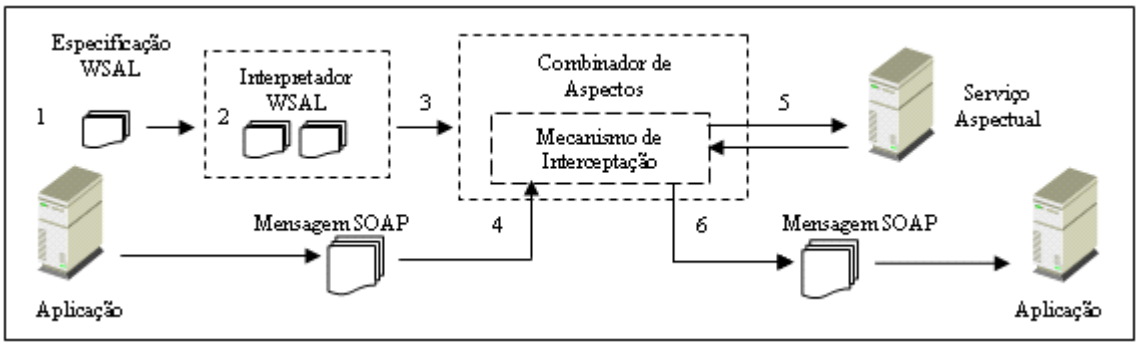


Figura 1. Processo de combinação de aspectos em WSAL.

(passo 5) e, se pertinente, retransmissão das mensagens interceptadas (possivelmente modificadas) às aplicações de destino (passo 6). Note que o processo não impõe nenhuma restrição ao papel desempenhado pelas aplicações cujas mensagens são interceptadas, de forma que ele pode ser aplicado tanto às mensagens de requisição de serviço quanto às mensagens de resposta de um ou mais serviços.

O principal benefício desse modelo de combinação em nível de rede é que ele torna o mecanismo de combinação completamente independente de quaisquer tecnologias utilizadas (como linguagem de programação e infra-estrutura de middleware), tanto na implementação dos componentes das aplicações quanto dos aspectos. Esta é uma característica chave para o modelo de combinação de WSAL, pois dá aos desenvolvedores das aplicações uma maior flexibilidade na escolha das tecnologias que melhor satisfazem suas necessidades e preferências.

2.2. Elementos da Linguagem

Como a maioria das linguagens de especificação da COS, WSAL tem sua sintaxe definida e validada utilizando um esquema XML próprio. Os elementos sintáticos da linguagem representam recursos típicos da POA, como (conjuntos de) pontos de junção, adendos e aspectos [Kiczales *et al*. 1997]. As subseções abaixo detalham alguns desses elementos e descrevem como eles podem ser agrupados na especificação de aspectos em WSAL.

2.2.1. Pontos de Junção

Em WSAL, pontos de junção correspondem a eventos definidos a partir das características das mensagens SOAP trocadas entre as aplicações consumidoras e provedoras de serviços web. WSAL define seis tipos de pontos de junção, todos associados à estrutura das mensagens SOAP ou a propriedades do protocolo de transporte subjacente. São eles:

- *namespace*: identifica um ou mais espaços de nomes dentre aqueles definidos na descrição WSDL de um serviço web, por meio da identificação dos URIs referentes aos espaços de nomes;
- *messagePart*: identifica um ou mais elementos que compõem a mensagem SOAP de requisição ou de resposta usada pelas operações definidas em uma descrição WSDL de um serviço web;
- *serviceOperation*: identifica uma ou mais operações dentre aquelas definidas na

```
<!-- ponto de junção composto -->
<pointcut name="pj1" type="composite">
   <and>
      <pointcut type="serviceOperation" pattern="doGoogleSearch"/>
      <pointcut type="clientLocation" pattern="188.188.*"/>
   </and>
</pointcut>

<!-- ponto de junção simples -->
<pointcut name="pj2" type=" messagePart" pattern="&lt;query*&gt;web 2.0&lt;/query&gt;">
```

Figura 2. Exemplos de pontos de junção em WSAL.

descrição WSDL de um serviço web;

- *serviceLocation*: identifica um ou mais URLs onde um serviço web pode ser provido;
- *clientLocation*: identifica um ou mais endereços de rede (expresso na forma de um endereço IP ou nome de domínio) em que pode residir a aplicação consumidora de um serviço web;
- *composite*: indica um ponto de junção composto por um conjunto de pontos de junção unidos pelos operadores lógicos *and* (conjunção), *not* (negação) e *or* (disjunção).

Este conjunto de tipos de pontos de junção é flexível o suficiente para permitir que pontos de junção sejam especificados para qualquer mensagem SOAP, tanto em termos de seu conteúdo (usando os tipos *namespace, messagePart* e *serviceOperation*) quanto em termos das propriedades relacionadas ao seu protocolo de transporte (usando os tipos *serviceLocation* e *clientLocation*). Além disso, usando o tipo *composite*, WSAL permite a composição arbitrária dos outros cinco tipos de pontos de junção, oferecendo um mecanismo bastante flexível para a definição de eventos de interação de diferente natureza e complexidade.

Sintaticamente, um ponto de junção é representado pelo elemento *pointcut* de WSAL. Além do atributo *type*, que indica o tipo do ponto de junção, esse elemento inclui também o atributo *pattern*, cujo valor especifica o contexto do ponto de junção. Para obter mais flexibilidade, valores para o atributo *pattern* podem ser definidos usando expressões regulares. A Figura 2 ilustra um ponto de junção simples e um ponto de junção composto definidos segundo a sintaxe de WSAL.

2.2.2. Adendos

Na POA, os adendos (*advices*) especificam o comportamento adicional a ser incluído no ponto de junção por ele referenciado. Em WSAL, cada adendo é associado a um ponto de junção (ou a um conjunto de pontos de junção, usando um ponto de junção composto) e a uma operação de um serviço aspectual, a ser invocado em tempo de execução pelo combinador de aspectos da linguagem.

Adendos são especificados com o elemento *advice* de WSAL. Esse elemento inclui o atributo *type*, para indicar o tipo do adendo. Os tipos de adendos definidos em WSAL são derivados dos diferentes tipos de eventos de interação que podem ocorrer em tempo de execução entre as aplicações provedoras e consumidoras de serviços web, por meio do protocolo SOAP. Três tipos de eventos são considerados: requisição de um

serviço, resposta de um serviço, e falha (ou exceção) na execução de um serviço. A partir desses três tipos de eventos foram definidos sete tipos de adendos em WSAL. São eles:

- *beforeRequest*: o comportamento a ser adicionado pelo serviço aspectual é invocado antes do combinador enviar a mensagem de requisição SOAP interceptada ao seu destino original. A mensagem interceptada não é alterada;
- *uponRequest*: similar ao *beforeRequest*, sendo que agora a mensagem de requisição interceptada pode ser alterada pelo serviço aspectual;
- *afterResponse*: o comportamento do serviço aspectual é invocado após o combinador interceptar uma mensagem de resposta SOAP, recebida do serviço web originalmente invocado por alguma aplicação cliente. A mensagem interceptada não é alterada;
- *uponResponse*: similar ao *afterResponse*, sendo que agora a mensagem de resposta interceptada pode ser alterada pelo serviço aspectual;
- *afterException*: o comportamento do serviço aspectual é invocado após o combinador interceptar uma mensagem SOAP indicando falha na execução do serviço originalmente solicitado pela aplicação cliente. A mensagem interceptada não é alterada;
- *uponException*: similar ao *afterException*, sendo que agora a mensagem de falha inteceptada pode ser modificada pelo serviço aspectual;
- *around*: o serviço aspectual substitui o serviço que originalmente seria invocado pela aplicação cliente. Para isso, o combinador repassa a mensagem de requisição SOAP interceptada diretamente ao serviço aspectual, cuja resposta é então devolvida para a aplicação cliente.

Cada adendo também pode ser definido com o tipo de informação que será repassada ao serviço aspectual, com o uso do atributo *context*. Usando este atributo, o desenvolvedor do aspecto pode controlar a quantidade de informações repassadas ao serviço aspectual, evitando, assim, uma sobrecarga na sua invocação pelo combinador. Entre as opções de informações de contexto a serem repassadas estão dados referentes ao protocolo de transporte das mensagens (no caso, HTTP), e ao próprio conteúdo das mensagens SOAP interceptadas.

Além do tipo e do contexto, um adendo em WSAL ainda pode ser definido como síncrono ou assíncrono, usando o atributo *mode*. Um adendo assíncrono será invocado de forma assíncrona pelo combinador de aspectos, ou seja, o combinador não ficará bloqueado aguardando o fim da execução do serviço aspectual, e repassará a mensagem SOAP interceptada ao serviço ou aplicação de destino imediatamente após a sua invocação.

2.2.3. Aspectos

A especificação de um aspecto em WSAL é feita utilizando o elemento *aspect*. Esse elemento associa um conjunto de pontos de junção (elementos *pointcut*) a um elemento *aservice*, que em WSAL representa um serviço aspectual. O elemento *aservice*, por sua vez, define o nome e a localização do serviço aspectual (atributos *name* e *location*, respectivamente), e encapsula a especificação de um conjunto de adendos (elementos

```
<aspect id="ClientAuthenticationAspect">
  <pointcut name="googleService"
    type="serviceLocation"
    pattern="http://api.google.com/search/beta2"/>
  <aservice name="AuthenticationService"
    location="http://server1/axis/AuthService">
    <advice type="uponRequest"
      pointcut-ref="googleService"
      operation="AddCredential"
      context="infoRequest" />
  </aservice>
</aspect>
```

(a)

```
<aspect id="ServiceBillingAspect">
  <pointcut name="UseService"
    type="serviceLocation"
    pattern="http://server1/axis/upperService"/>
  <aservice name="BillingService"
    location="http://server2/axis/BillingService">
    <advice type="afterResponse"
      pointcut-ref="UseService"
      operation="BillingPerUse"
      context="infoHTTP"
      mode="asynchronous"/>
  </aservice>
</aspect>
```

(b)

Figura 3. Dois exemplos de aspectos especificados em WSAL: um aspecto de autenticação de clientes (a) e um aspecto de cobrança (b).

advice). Dessa forma, um elemento *aspect* contém toda a informação necessária para que o combinador de aspectos de WSAL possa interceptar mensagens SOAP de acordo com os critérios definidos nos pontos de junção especificados no aspecto, e então invocar as operações do serviço aspectual especificadas nos adendos, quando pontos de junção de interesse forem identificados nas interações entre as aplicações.

A Figura 3 mostra dois exemplos completos de aspectos especificados em WSAL. O primeiro exemplo (Figura 3(a)) corresponde a um aspecto de autenticação de clientes. Esse aspecto tem como objetivo encapsular o procedimento de autenticação necessário para que aplicações clientes possam acessar o serviço de busca do Google. O procedimento de autenticação consiste, basicamente, em inserir as devidas credenciais, previamente fornecidas pelo Google, em todas as mensagens de requisição SOAP enviadas ao seu serviço de busca. Para isso, o aspecto define um ponto de junção do tipo *serviceLocation*, cujo atributo *pattern* contém a URL referente à localização do serviço de busca, e um elemento *aservice* com um único adendo do tipo *uponRequest*, associado à operação *AddCredential* do serviço aspectual *AuthenticationService*. Essa operação ficará responsável por inserir as credenciais requeridas pelo serviço de busca do Google em uma mensagem de requisição interceptada e recebida como parâmetro do combinador, e então retornar a mensagem modificada ao combinador, para que ele possa encaminhá-la de volta ao seu destino original.

O segundo exemplo (Figura 3(b)) corresponde a um aspecto de cobrança. Esse aspecto é utilizado para registrar cada acesso realizado com sucesso a um ou mais serviços web, de modo a permitir a subseqüente cobrança pelo uso do(s) serviço(s) junto a seus usuários (aplicações clientes). Note que o aspecto está associado ao serviço aspectual de nome *BillingService*, e contém um único adendo, do tipo *afterResponse*, associado à operação *BillingPerUse* do serviço aspectual. Note ainda que essa operação foi definida para ser invocada de modo assíncrono pelo combinador, uma vez que ela não retorna nenhuma informação relevante para o processo de combinação.

É importante ressaltar que tanto o serviço aspectual de autenticação quanto o de cobrança poderão ser implementados e reutilizados livremente, utilizando qualquer tecnologia da COS. Além disso, com pequenas modificações na especificação dos pontos de junção, ambos os aspectos poderão facilmente ser combinados ao fluxo mensagens SOAP proveniente da interação entre outras aplicações e serviços.

3. Implementação de um Combinador de Aspectos para WSAL

O processo de combinação de aspectos utilizado em WSAL baseia-se na idéia de interceptar as mensagens SOAP trocadas entre as aplicações no nível de rede, de modo que a combinação dos aspectos aconteça num ambiente externo às aplicações, propiciando, assim, independência de plataforma e de linguagem ao processo de combinação.

Portanto, o combinador precisa atuar como um intermediário (ou um *proxy*) entre as aplicações provedoras e consumidoras de serviços. Para que isso aconteça, foi escolhido o WBI [Barret e Maglio 1998] como plataforma de interceptação, por já fornecer uma infraestrutura completa para o trabalho de interceptação e tratamento de requisições e respostas HTTP com o uso de mecanismos de extensão (*plugins*) da plataforma. Como HTTP também é o protocolo de transporte mais comum no mundo dos serviços web, o WBI se mostra uma opção flexível e adequada às necessidades do combinador de aspectos da WSAL.

O WBI é um *proxy programável* escrito em Java que promove a extensão através de *plugins* e *MEGs*. Os *plugins* são classes que podem ser dinamicamente implantadas na plataforma do WBI, desde que herdem de determinada classe abstrata da plataforma. Os *plugins* são acionados apenas uma vez, no momento de sua carga na plataforma WBI, e servem para configurar a condição de execução de um ou mais *MEGs*. *MEGs* são objetos especiais capazes de lidar com as requisições e respostas HTTP que forem interceptadas mediante as condições configuradas no *MEG* pelo plugin. MEG é a sigla para *Monitor, Editor and Generator*, significando o propósito de cada MEG. Por exemplo, apenas monitorar (*monitor*) o fluxo de requisições e respostas HTTP, alterar (*editor*) uma requisição ou uma resposta HTTP, ou ainda produzir uma nova resposta (*generator*). O WBI fornece classes básicas para que novas classes MEGs sejam criadas a partir delas por extensão, e possam ser utilizadas pelos plugins para que tratem o tráfego HTTP.

Com base nisso, tomou-se a decisão de que o combinador deveria ser construído na plataforma de interceptação de mensagens WBI e, portanto, na plataforma Java. Como o acoplamento do combinador a uma tecnologia externa à de serviços web é indesejável, algumas estratégias de componentização foram tomadas para garantir algum nível de reutilização do combinador em outras plataformas de interceptação. Essas estratégias serão discutidas nas subseções seguintes.

3.1. Geração e Combinação dos Aspectos

Primeiramente, foi preciso definir como os arquivos XML com as especificações de aspectos em WSAL seriam implantados no combinador, e como o combinador iria utilizá-los para produzir o efeito do aspecto. Duas possibilidade foram vislumbradas:

- Geração de código estático: mediante uma ferramenta externa que leria os aspectos e, usando *templates* (código pré-definido com alguns elementos que podem ser substituídos a fim de gerar um novo código estático), seria gerado o código de classes específicas para lidar com cada aspecto. Possivelmente, seria gerado o código de um *plugin* do WBI para cada aspecto.
- Código genérico dinâmico: seria criado um conjunto de classes capazes de carregar as definições de aspectos em memória e, em tempo de execução, o

combinador iria ler os aspectos, verificar dinamicamente as suas condições de execução e, se for o caso, aplicar o comportamento esperado.

Ambas as abordagens possuem vantagens e desvantagens, mas para a atual versão do combinador foi tomada a decisão de utilizar a segunda abordagem: a criação de um conjunto de classes genéricas e dinâmicas, no formato de uma API (*Application Programming Interface*) genérica de combinação de aspectos para a WSAL. O principal fator motivador para esta decisão foi a possibilidade de reutilização, que poderia fazer com que esta mesma API fosse utilizada em diferentes plataformas de interceptação de mensagens, e a independência em relação ao WBI (que não aconteceria da mesma forma, caso optássemos por gerar código estaticamente para esta plataforma).

3.2. Componentes da Arquitetura

Os elementos que compõem a API de combinação de aspectos incluem os mecanismos de implantação de aspectos, interpretação dos aspectos lidos em XML, validação e, por fim, a combinação propriamente dita. Além destes elementos genéricos, são necessários elementos específicos da plataforma de interceptação de mensagens, neste caso, o WBI.

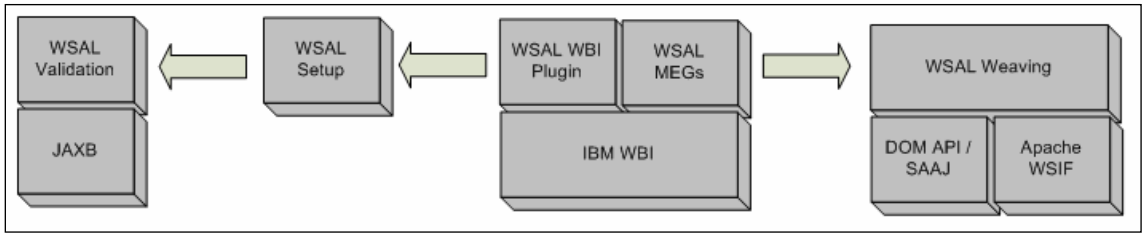


Figura 4. Componentes do combinador de aspectos para WSAL.

A Figura 4 mostra os principais componentes do combinador de aspectos para WSAL e os seus relacionamentos. A funcionalidade e alguns detalhes de implementação de cada um desses componentes são descritos a seguir:

- *WSAL Setup*: mecanismo simples, responsável pela carga dos arquivos XML de aspectos para que estes sejam levados ao componente de *parse* e validação. Funciona obtendo uma lista de arquivos que terminam com a extensão ".wsal.xml", e utilizando o componente de *parse* validação (*WSAL Validation*) para obter o aspecto em forma de objetos. A partir dos aspectos como objetos produzidos pelo componente *WSAL Validation*, pode fornecer outras informações úteis aos outros componentes;
- *WSAL Validation*: com os arquivos de aspectos carregados em memória, este conjunto de classes transforma os arquivos XML lidos em um conjunto de objetos usando uma tecnologia de *XML binding*, opção que é a mais apropriada para a situação, pois pretende-se percorrer a configuração XML como uma estrutura de objetos, podendo-se criar uma API utilizando estes objetos. Como boa prática, foi criado um conjunto de interfaces para isolar a estrutura de objetos do esquema XML de uma tecnologia de binding específica, de modo que ela pode ser alterada. A presente versão implementa essas interfaces utilizando JAXB [Sun 2006];
- *WSAL WBI Plugin*: trata-se de apenas uma classe que é responsável por utilizar o componente de validação, obter as configurações dos aspectos, instanciar e

configurar os MEGs da WSAL necessários para tratá-los;

- *WSAL MEGs*: composta pelos MEGs da WSAL, *WsalWeavingDocumentEditor*, *WsalWeavingGenerator*, e *WsalWeavingRequestEditor*. Cada um deles trata um tipo de adendo na WSAL, correspondendo ao tipo de MEG no WBI, conforme descrito abaixo:
    - *WsalWeavingRequestEditor*: como um MEG do tipo *RequestEditor*, ele é invocado a cada requisição HTTP, podendo alterá-la. Na WSAL, este MEG irá tratar os adendos do tipo *beforeRequest* e *uponRequest*;
    - *WsalWeavingDocumentEditor*: MEGs do tipo *DocumentEditor* lidam com as respostas HTTP, portanto eles são os ideais para tratar os adendos na WSAL do tipo *afterException*, *afterResponse*, *uponException* e *uponResponse*;
    - *WsalWeavingGenerator*: a semântica do adendo *around* é substituir completamente o uso de um serviço web pelo uso do serviço aspectual. Portanto, um MEG do tipo *Generator* é o mais adequado para este caso, por ser apropriado para produzir novo conteúdo;
- *WSAL Weaving*: este componente tem o papel fundamental de realizar toda a semântica da linguagem WSAL, avaliando pontos de junção, incluindo o comportamento dos aspectos nos eventos dos adendos e invocando os serviços aspectuais. Sua implementação usa componentes para lidar com as estruturas das mensagens SOAP (*DOM API/SAAJ*) e para invocação dinâmica de serviços (*WSIF*), como apresentado na Figura 4.

4. Avaliação Preliminar

Esta seção apresenta uma análise preliminar do custo de desempenho imposto pela atual versão do combinador de aspectos de WSAL, implementado sobre a plataforma WBI, às aplicações envolvidas no processo de combinação. O intuito da análise é oferecer indícios do impacto do combinador na prática, propiciando, assim, subsídios para futuras análises mais abrangentes.

Foram realizados testes para a análise do impacto do aspecto de cobrança – apresentado na Figura 3(b) – no desempenho das aplicações afetadas, através da medição, em diferentes cenários, do tempo médio de resposta obtido quando do consumo do serviço web cobrado. Foram utilizadas três máquinas, todas conecadas à mesma rede local, onde a primeira executava a aplicação consumidora do serviço; a segunda executava o combinador de aspectos e o serviço aspectual associado ao adendo de cobrança; e a terceira executava o serviço web cobrado.

Os cenários analisados contemplam interações entre a aplicação consumidora e a aplicação provedora do serviço web, com a implementação do interesse de cobrança realizada de três maneiras distintas: dentro da própria aplicação provedora do serviço, ou seja, sem a combinação do aspecto de cobrança (Cenário 1); via combinação do aspecto de cobrança no modo de invocação assíncrono (Cenário 2); e via combinação do aspecto de cobrança no modo de invocação síncrono (Cenário 3).

Os experimentos foram feitos através da invocação de uma operação do serviço alvo que simplesmente converte os caracteres de um texto (*string*) recebido como

parâmetro de entrada para caracteres minúsculos. O texto convertido é então retornado para a aplicação consumidora como resultado da operação. Foram realizadas quatro seqüências de testes. Em cada seqüência, foram feitas 15 invocações do serviço, envolvendo mensagens SOAP de requisição e de resposta de um mesmo tamanho. A cada nova seqüência, o tamanho das mensagens foi incrementado, na seguinte ordem: 1Kb, 20Kb, 40Kb e 60Kb. Em cada invocação do serviço foi computado o tempo transcorrido desde o envio da requisição até o recebimento da resposta. Para amenizar os efeitos da latência da rede, foram descartadas as cinco primeiras invocações em cada seqüência.

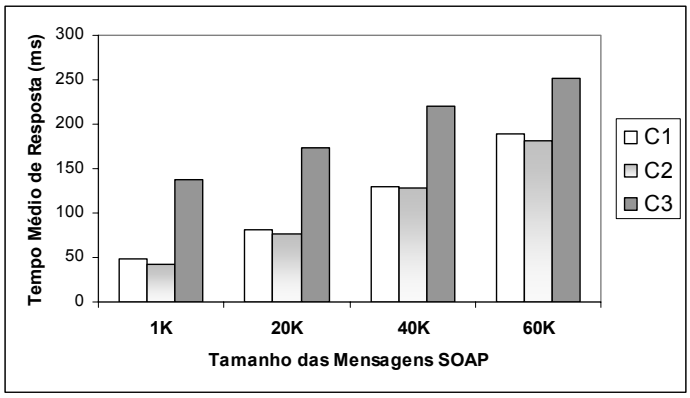


Figura 5. Tempo médio de resposta do serviço de cobrança.

A análise do resultado dos experimentos demonstra que o impacto de desempenho obtido com a combinação do aspecto neste serviço tende a ser, em relação ao cenário sem aspectos (Cenário 1), inferior a 100ms em média, quando o adendo é invocado no modo síncrono (Cenário 3), e praticamente nulo quando o adendo é invocado no modo assíncrono (Cenário 2), conforme mostram a Figura 5 e a Tabela 1.

	Tamanho da Mensagem SOAP			
Cenário	1K	20K	40K	60K
C2	-6 ms	-5 ms	-1 ms	-7 ms
C3	89 ms	92 ms	92 ms	62 ms

Tabela 1. Desempenho do serviço de cobrança com relação ao cenário C1.

Analisando mais detalhadamente o impacto no desempenho gerado pelo uso do aspecto de cobrança no Cenário 3, percebemos que o valor relativo deste impacto varia decrescentemente com o aumento do tamanho das mensagens SOAP interceptadas. O custo imposto pelo aspecto tende a ter um teto máximo, neste caso, representado por um aumento no tempo de resposta do serviço não superior a 100ms; diferentemente, o custo imposto pelo consumo das operações torna-se maior à medida que aumenta o tamanho das mensagens SOAP trocadas. Como resultado, o custo adicional advindo da combinação do aspecto torna-se percentualmente menor em relação ao custo total do consumo de uma operação do serviço web alvo que não use o aspecto, à medida que elevamos o tamanho das mensagens SOAP interceptadas pelo combinador. Além disso, considerando que, mesmo ao combinar o aspecto de cobrança, as operações ainda

apresentam tempos médios de resposta abaixo de 1s, a perda média de desempenho observada no Cenário 3 em relação ao Cenário 1, apesar de relativamente alta, ainda seria aceitável na medida que não comprometesse os requisitos de qualidade do serviço cobrado.

Apesar da perda não desprezível, os benefícios de reutilização e independência de plataforma e de tecnologia de implementação que esta abordagem traz para os desenvolvedores de aplicações orientadas a serviços, em nossa visão, em muitos casos podem compensar o custo que combinador de aspectos de WSAL certamente impõe ao desempenho das aplicações afetadas por ele.

5. Trabalhos Relacionados

Utilizar intermediários SOAP como forma de implementar interesses não funcionais de serviços não é uma idéia nova. Na verdade, ela faz parte da especificação SOAP desde suas primeiras versões [Gudin e Hadley 2003] e é suportada pela maioria dos frameworks de web services atuais. O modelo de serviços aspectuais adotado em WSAL leva essa idéia mais adiante, uma vez que ele se baseia em intermediários SOAP, não como tecnologia para implementar interesses não funcionais diretamente, mas como forma de trazer, com baixo acoplamento, a disciplina de POA para o desenvolvimento de aplicações orientadas a serviços.

Diversas outras abordagens também propõem a integração dos paradigmas de POA e COS [Verheecke e Cibrán 2003, Verspecht et al. 2003, Baligand e Monfort 2004]. Entretanto, todos eles se baseiam em um mecanismo de combinação de aspectos fortemente restrito a uma linguagem de programação ou plataforma de desenvolvimento específica. Por exemplo, as abordagens de [Verheecke e Cibrán 2003, Baligand e Monfort 2004] baseiam-se em mecanismos distintos, mas específicos para a plataforma Java, assim exigem que ambos, aspectos e aplicação, sejam desenvolvidos em Java. De forma semelhante, a abordagem [Verspecht et al. 2003] baseia-se num mecanismo de combinação específico para a framework .NET, limitando assim a implementação dos aspectos e aplicações ao contexto .NET e às linguagens suportadas por esse framework. Ao contrário do que ocorre nas abordagens acima citadas, em WSAL os aspectos também são implementados como serviços web fracamente acoplados que podem ser dinamicamente combinados ao fluxo de mensagens interceptado na rede. Embora nossa abordagem de combinação dinâmica reduza seu escopo de atuação aos eventos de interação que ocorrem fora do ambiente de execução das aplicações, isto torna possível implementar aspectos de maneira completamente independente de qualquer linguagem de programação ou plataforma de desenvolvimento específicos.

Dois trabalhos mais recentes seguem uma linha um pouco diferente, integrando conceitos da POA e da COS em nível de composição de serviços [Charfi e Mezini 2004, Courbis e Finkelstein 2004]. Ambos estendem a linguagem BPEL4WS [BEA Systems et al.], uma linguagem de composição de processos baseados em serviços, com novas construções para especificar aspectos, e com um novo mecanismo para combinar os aspectos aos processos BPEL4WS originais. No que diz respeito ao mecanismo de combinação que age em nível de composição de serviço, esses trabalhos, como o nosso, suportam a implementação de aspectos de forma independente de linguagem e de ambiente. No entanto, os seus mecanismos de combinação possuem escopo mais limitado, sendo restritos aos pontos de junção dinâmicos expressos em termos de

eventos capazes de serem capturados pela máquina de execução BPEL4WS. Isso torna a implementação de aspectos fortemente acoplada à máquina de execução BPEL4WS.

6. Conclusões

Este artigo apresentou uma nova linguagem de POA, chamada WSAL, cujo objetivo é prover um mecanismo natural para integrar os paradigmas da computação orientada a aspectos e da computação orientada a serviços. Em WSAL, os aspectos são implementados como serviços web fracamente acoplados, cujas operações podem ser combinadas dinamicamente, em pontos de junção especificados em termos de propriedades de rede e do conteúdo de mensagens SOAP trocadas entre aplicações provedoras e consumidoras de serviços web. Comparada a outras abordagens POA baseadas em serviços, WSAL oferece o importante benefício de viabilizar a aplicabilidade de aspectos ao contexto da COS na web, independentemente de plataforma de desenvolvimento ou tecnologia de implementação.

Também foram apresentados o funcionamento, a implementação inicial e uma avaliação preliminar de desempenho dos mecanismos de validação, implantação e combinação dinâmica de aspectos definidos em WSAL, integrados através de uma tecnologia de intermediários HTTP existente (WBI).

Como sugestões para trabalhos futuros, destacamos:

- Finalizar a implementação do combinador de aspectos, implementando todos os cenários possíveis na linguagem WSAL em uma situação de combinação de aspectos, e também realizar mais testes objetivando o uso das ferramentas em sistemas em produção.
- Testar o uso das APIs genéricas desenvolvidas para a WSAL em outra plataforma que não o WBI, como, por exemplo, em manipuladores (*handlers*) de serviços web, para que elas sejam aprimoradas e que tenham o seu caráter de reutilização confirmado.
- Implementar ferramentas de suporte e monitoramento do combinador de aspectos, de modo a tornar todo o processo de especificação, implantação e monitoramento dos aspectos algo automatizado e mais simples.
- Realizar novos e mais abrangentes testes de desempenho para o combinador proposto, de modo a validar de forma mais consistente as decisões que foram tomadas e, se necessário, realizar alterações que melhorem seu desempenho.

Referências

Barret, R. Maglio, P. (1998), Intermediaries: New Places for Producing and Manipulating Web Content. Computer Networks and ISDN Systems, 30(1-7):509-18.

Bryan, D., Draluk, V., Ehnebuske, D., Glover, T, et al. (2002), Universal Description, Discovery and Integration version 2.04. Disponível em: http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm.

Cauldwell, P., Chawla, R., Chopra, V., et al. (2001), Professional XML Web Services, Wrox Press, Birminghan, USA.

Charfi, A. and Mezini, M. (2004), Aspect-Oriented Web Service Composition with

AO4BPEL, In Proc. of the Eur. Conf. on Web Services (ECOWS'04), Volume 3250 of LNCS, Springer-Verlag, pp. 168-182.

Christensen, E., Curbera, F., Meredith, G., et al. (2001), Web Services Description Language (WSDL) 1.1, W3C Note. Disponível em: http://www.w3.org/TR/wsdl.

Courbis, C., Finkelstein, A. (2004), Towards an Aspect Weaving BPEL Engine. In Proc. of the 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'01), Lancaster, U.K.

Gudin, M., Hadley, M., Mendelsohn, N., Moreau, J. and Nielsen, H. F. (2003), "SOAP Version 1.2", W3C Recommendation. Disponível em: http://www.w3.org/ TR/soap12.

Henkel, M., Boström, G., Wäyrynen, J. (2005), Moving from Internal to External Services using Aspects. In Proc. of the 1st Int. Conf. on Interoperability of Enterprise Software and Applications (ICIESA'05), Genebra, Suíça.

IBM, "WBI Web Site", Disponível em: http://www.almaden.ibm.com/cs/wbi/.

Kiczales, G. J., Mendhekar, L. A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J. (1997), Aspect-Oriented Programming. In Proc. of the 11th Eur. Conf. on Object-Oriented Programming (ECOOP), volume 1241 of LNCS, Springer-Verlag, pages 220–242.

Lafferty, D. C., Cahill, V. (2003), Language-Independent Aspect-Oriented Programming. In Proc. of the 18th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03), California, USA.

Mendonça, N. C., Silva. C. F. (2005), Aspectual Services: Unifying Service- and Aspect-Oriented Software Development. In Proc. of the Int. Conf. on Next Generation Web Services Practices (NWeSP'05), Seul, Coréia do Sul. IEEE Computer Society Press.

Mendonça, N. C., Silva. C. F. (2006), A Unified Model for Service- and Aspect-Oriented Software Development. International Journal of Web Services Practices, 2(1-2):59-67.

Papazoglou, M. P. and Georgakopoulos, D. (2003), Service-Oriented Computing. Communication of the ACM, Vol. 46, No. 10, pp. 25-28.

Sun (2006), Java Enterprise Edition 5. Disponível em http://java.sun.com/javaee/5/.

Verheecke, B., Cibrán, M.A. (2003), AOP for Dynamic Configuration and Management of Web Services. In Proc. of the Int. Conf. on Web Services – Europe 2003 (ICWS – Europe '03), Erfurt, Alemanha.

Verspecht, D., Vanderperren, W., Suvée, D., Jonckers, V. (2003), JasCo.NET: Unraveling Crosscutting Concerns in .NET Web Services. Vrije Universiteit Brussel, Pleinlaan 2, Bruxelas, Bélgica.

Programação avançada em Common Lisp: Linguagens de Domínio Específico e Meta-programação

Pedro Kroger[1]

[1]Grupo de Pesquisa GENOS – Escola de Música – Universidade Federal da Bahia (UFBA)
Parque Universitário Edgard Santos, Canela – Salvador – Bahia – 40110-150 – Brazil

pedro.kroger@gmail.com

Resumo. *Lisp é uma linguagem avançada que é menos usada do que deveria. Isso se dá, em parte, a mitos que foram criados e ao fato de que a exposição inicial à linguagem muitas vezes acontece de forma incompleta e puramente teórica, geralmente em cursos de "comparativos de linguagens" ou "programação funcional". Mas Common Lisp é uma linguagem rica definida pelo padrão ANSI com listas, arrays, vetores, números racionais, complexos, etc. Além disso, Lisp é multi-paradigma possuindo abstrações para programação funcional, procedural, imperativa, aspectos, um sistema avançado e flexível de programação orientada a objetos, e um caráter dinâmico sem precedentes. Meta-programação e a criação de linguagens de domínio específico (LDE) são maneiras poderosas de resolver problemas específicos e possibilitar a escrita de código mais curto e elegante. O objetivo desse tutorial é mostrar como é possível desenvolver LDE em Common Lisp de uma maneira simples e elegante, sem ter que recorrer a mecanismos externos (como XML).*

1 Introdução

1.1 Duração

O tutorial terá a duração de 3 horas, preferivelmente divididas em duas seções.

1.2 Justificativa

Em geral apenas um subconjunto de Lisp é visto em cursos de graduação, geralmente em disciplinas de "comparativos de linguagens". Contudo, Lisp é uma linguagem poderosa com um suporte avançado para meta-programação e criação de linguagens de domínio específico (LDE). Acreditamos que esse tutorial será útil a estudantes de graduação, pós-graduação, e mesmo aqueles que não desejem programar em Lisp, já que as técnicas mostradas podem ser utilizadas em outras linguagens.

Outro fator que justifica esse tutorial é que Lisp tem uma tradição rica em meta-programação e LDE, contudo, artigos como [van Deursen et al. 2000] e [Mernik et al. 2005] que deveriam supostamente ser abrangentes, nem mencionam essa tradição, documentada em diversos lugares como [Abelson and Sussman 1987], [Graham 1993], e [Abelson and Sussman 1996].

1.3 Esquema do tutorial

1. Introdução geral a Lisp
2. Abstração usando macros
3. Meta-programação usando macros
4. Criando linguagens de domínio específico

1.4 Curriculum Vitæ

Pedro Kröger é Doutor em composição pela Universidade Federal da Bahia/University of Texas at Austin e professor adjunto da Escola de Música da UFBA. Ele participou de diversos projetos de Software Livre como LyX, LDP-br, debian-br-cdd e LilyPond. Seus conhecimentos de computação são aplicados principalmente no desenvolvimento de protótipos para pesquisas acadêmicas relacionadas à música e computação. Dentre as linguagens que utiliza estão Tcl/TK, Itcl/iwidgets, Python, e principalmente Lisp. Ele é um dos fundadores do Grupo de Usuários Lisp do Brasil. Desde 2006 ele tem ministrado a disciplina MAT052—"Paradigmas de Linguagens de Programação" no Departamento de Computação da Universidade da Bahia utilizando Lisp para demonstrar os diferentes tipos de paradigmas.

1.5 Agradecimentos

O autor agradece aos pareceristas anônimos pelos comentários, sugestões e críticas construtivas.

2 Linguagens de domínio específico

Linguagens de domínio específico são dedicadas a resolver problemas específicos de um domínio. Elas trocam generalidade por expressividade nesse domínio [Sloane et al. 2003] e permitem desenvolver programas mais curtos e fáceis de manter [Graham 1993]. A literatura sobre LDE é vasta e uma análise completa dos diversos tipos, implementações e usos de LDE está fora do escopo desse documento. Contudo [Shivers 1996], [Sloane et al. 2003], [van Deursen et al. 2000], e [Spinellis 2001] devem servir como um bom ponto de partida dentro do contexto desse tutorial.

3 Implementação de linguagens de domínio específico

Uma LDE pode ser implementada de basicamente duas maneiras; através de um compilador ou interpretador, ou através da extensão de uma linguagem de propósito geral [van Deursen et al. 2000]. A primeira é a maneira mais comum de implementar novas linguagens, especialmente como "pequenas linguagens" no Unix. Essa abordagem é conhecida como LDE fechada [Kieburtz 2000]. Makefiles, troff, e arquivos de configuração como os do sistema de janelas xorg são exemplos. A segunda maneira, a criação de "linguagens embutidas" é a mais comum na comunidade Lisp e tem a grande vantagem de usar a linguagem *host* como base, sendo desnecessário a criação de um compilador ou interpretador do nada. Uma desvantagem dessa abordagem é que a LDE está limitada aos mecanismos sintáticos da linguagem *host*. "In many cases, the optimal domain-specific notation has to be compromised to fit the limitations of the base language" [van Deursen et al. 2000, p. 4]. Essa é a principal razão que acreditamos que para criar uma LDE embutida, é fundamental utilizar uma linguagem poderosa e expressiva como base, que permita modificações sintáticas. Na seção 6 vamos fornecer exemplos de como isso pode ser feito com Lisp.

Apesar das vantagens e desvantagens gerais dos tipos de implementação serem conhecidos, nem sempre é clara qual a maneira de implementação mais apropriada para determinado domínio. Por exemplo, Make [Stallman and Mcgrath 1998] é o utilitário clássico para construção de programas e é implementado como uma LDE fechada. Scons

[Knight 2004], um outro programa para construção, por outro lado, é implementado como uma LDE embutida em Python.

4 Meta-programação

Meta-programação é a técnica de escrever programas que escrevem ou manipulam programas. O exemplo canônico é um compilador. Em geral, meta-programação é vista como algo complicado e que não pode ser aplicado aos problemas do "mundo real" [Lugovsky 2004]. Contudo, usando as ferramentas corretas, meta-programação pode ser simples a ajudar a aumentar a clareza e simplicidade do código [Graham 1993].

A prática de meta-programação se originou em Lisp, e apesar de Lisp não ser a única linguagem programável existente, é reconhecido o fato dela representar mais fortemente essa prática [Fowler 2005] e de uma maneira simples, elegante e integrada à linguagem. A prática de meta-programação é muitas vezes usada em Lisp para criar linguagens de domínio específico. "In Lisp, you don't just write your program down toward the language, you also build the language up toward your program." [Graham 1993]

Abelson sumariza em [Abelson and Sussman 1987] porque Lisp é tão bem adaptada para meta-programação e linguagens de domínio específico:

> People who first learn about Lisp often want to know for what particular programming problems Lisp is "the right language." The truth is that Lisp is not the right language for any particular problem. Rather, Lisp encourages one to attack a new problem by implementing new languages tailored to that problem. Such a language might embody an alternative computational paradigm, as in the rule language. Or it might be a collection of procedures that implement new primitives, means of combination, and means of abstraction embedded within Lisp, as in the Henderson drawing language. A linguistic approach to design is an essential aspect not only of programming but of engineering design in general. Perhaps that is why Lisp, although the second-oldest computer language in widespread use today (only FORTRAN is older), still seems new and adaptable, and continues to accommodate current ideas about programming methodolgoy.

Ou mais sucintamente, na famosa citação de Guy Steele, "If you give someone Fortran, he has Fortran. If you give someone Lisp, he has any language he pleases."

5 Lisp

Essa seção oferecerá uma introdução geral ao Common Lisp, de modo que os leitores possam acompanhar os exemplos do final do artigo. Naturalmente, uma visão aprofundada da linguagem está fora do escopo desse documento, mas sugestões de leitura são providas na seção 8.

5.1 Introdução

Lisp é uma linguagem de programação inventada no final dos anos 50 por John McCarthy, um dos pioneiros em Ciência da Computação. Ele cunhou o termo "inteligência artificial", inventou alocação dinâmica de memória, expressões condicionais, dentre outras coisas.

Em geral o termo Lisp se refere a uma família de linguagens, onde os dialetos mais conhecidos atualmente são Common Lisp, Scheme e Emacs Lisp. Nesse documento Lisp e Common Lisp serão usados de maneira intercalada.

Uma série de mitos e concepções incorretas em relação a Lisp foram acumulados durante os anos. O primeiro é que Lisp é uma linguagem de programação funcional, quando na verdade dialetos modernos como Common Lisp enfatizam um uso multiparadigma. Common Lisp permite o uso de diversos paradigmas como funcional, procedural, imperativo, aspectos, e um sistema de programação orientado a objetos poderoso e sofisticado. O segundo mito é que Lisp é uma linguagem interpretada e lenta. Apesar de isso ter sido verdade no passado, atualmente existem compiladores otimizados capazes de gerar código nativo com velocidade comparável a linguagens como C e C++ [Verna 2006b, Verna 2006a, Svingen 2006]. Common Lisp usa tipagem dinâmica mas permite a declaração de tipos, permitindo o compilador gerar código mais rápido.

5.2 Notação pré-fixa

Lisp é baseado em expressões delimitadas por parênteses (chamadas de expressões simbólicas, ou *sexp*) e todas as expressões retornam um valor. Além disso Lisp usa a notação pré-fixa, garantindo que as expressões serão escritas de uma maneira não-ambígua. Por exemplo, a expressão `2 * 3 + 4 - 3` pode ter 7 ou 8 como resultado, dependendo de quais elementos são calculados primeiro (exemplo 5.1). Por outro lado, qualquer ambiguidade é removida com o uso da notação pré-fixa (ex. 5.2).

Exemplo 5.1 Expressões com notação infixa

```
(2 * 3) + (4 - 3) => 6 + 1 => 7
2 * ((3 + 4) - 3) => 2 * (7 - 3) => 2 * 4 => 8
```

Exemplo 5.2 Expressões com notação pré-fixa

```
(+ (* 3 2)
   (- 4 3)) => 7

(* 2
   (- (+ 3 4)
      3)) => 8
```

5.3 Aplicação de funções

Em Lisp cada expressão é composta por uma lista onde o primeiro elemento é um *operador* que é aplicado a um ou mais *operandos* (ex. 5.3). Uma das vantagens dessa notação é que as funções primitivas e funções definidas pelo usuário tem a mesma sintaxe. E, de fato, todas as funções no exemplo 5.3 poderiam ter sido definidas pelo usuário, inclusive +.

Exemplo 5.3 Aplicação de funções

```
(+ 1 3 4)
(raiz-quadrada 25)
(show-gui)
```

5.4 Formas especiais

Common Lisp usa avaliação estrita, ou seja, os operandos são computados antes de serem passados para o operador. Desse modo, se `if` fosse definido como uma função todos os termos da expressão `(if (= x 0) (foo) (bar))` seriam computados, mas queremos que as funções `foo` e `bar` só sejam executadas dependendo do valor de `x`.

Formas especiais são expressões que não seguem as regras normais de avaliação. Elas seguem as regras da forma especial específica. Os termos não são computados antes de serem passados para a forma especial, então `if` pode ser implementado como uma forma especial.

5.5 Macros

Uma das coisas que diferencia Lisp das demais linguagens é que podemos definir nossas próprias formas especiais usando macros. Apesar de terem o mesmo nome, macros em Lisp tem pouca ou nenhuma semelhança com macros em C. Macros em C fazem substituições de caracteres enquanto macros em Lisp operam em expressões. Por exemplo, uma expressão condicional para fazer alguma coisa quando o valor da condição não for verdadeiro pode ser descrita em termos de `if` e `not` (ex. 5.4).

Exemplo 5.4 Expressão condicional

```
(if (not <condição>)
    <corpo>)
```

Esse padrão pode ser abstraido com o comando `unless`. Caso `unless` não fosse definido em Lisp, poderia ser facilmente implementado com uma macro (ex. 5.5). Tendo definido `unless`, ele pode ser usada como se tivesse sido definido como um elemento primitivo da linguagem como na expressão `(unless (= x 0) (foo))`. Como observamos anteriormente, não há diferença aparente se `unless` foi implementado pelo usuário ou como elemento primitivo. O significado de `‘` e `,` no exemplo 5.5 será visto na seção 5.7.

Exemplo 5.5 Implementando unless como uma macro

```
(defmacro unless (condicao corpo)
  ‘(if (not ,condicao)
       ,corpo))
```

5.6 Código como dado

Um dos recursos que distingue Lisp de outras linguagens é a capacidade de tratar código como dado e vice-versa de uma maneira uniforme. O primeiro elemento de uma expressão

entre parênteses é tratado como uma função ou forma especial a menos que algo impeça a avaliação da expressão. O operador especial `quote` impede a avaliação de uma expressão e retorna ela literalmente. Por exemplo, `(quote (2 + 3))` retorna `(2 + 3)`, enquanto a expressão sem `quote`, `(2 + 3)`, retorna um erro, já que 2 não é uma função. Geralmente se usa `quote` em sua forma abreviada, com um apostrofo antes da expressão, como `'(2 + 3)`.

Expressões com `quote` podem ser usadas para a computação simbólica com a qual Lisp é normalmente associado. O exemplo 5.6 mostra alguns exemplos simples, onde o resultado da expressão é mostrado depois do símbolo `=>`. No primeiro exemplo duas listas são concatenadas. No exemplos seguintes, o primeiro e terceiro elementos da lista são retornados, respectivamente.

Exemplo 5.6 Computação simbólica

```
(append '(Maria da Silva) '(Santos de Oliveira))
=> (MARIA DA SILVA SANTOS DE OLIVEIRA)

(first '(MARIA DA SILVA SANTOS DE OLIVEIRA))
=> MARIA

(third '(MARIA DA SILVA SANTOS DE OLIVEIRA))
=> SILVA
```

Um exemplo um pouco mais sofisticado, contudo ainda simples, pode ser visto no exemplo 5.7. A função `infixo` é definida para aceitar uma expressão infixa na forma `(infixo '(2 + 3))` e retornar o resultado dessa expressão. O operador especial `funcall` aplica uma função a n argumentos. Na função `infixo`, a função que `funcall` irá aplicar é o segundo elemento da lista, e os operandos são o primeiro e terceiro elementos da lista, respectivamente. É interessante observar que `infixo` é genérica o suficiente para permitir expressões como `(infixo '(2 * 3))` ou `(infixo '(7 - 3))`.

Exemplo 5.7 Função simples para expressão infixa

```
(defun infixo (expressao)
  (funcall (second expressao) (first expressao) (third expressao)))
```

5.7 Mais sobre macros

Macros em Lisp são funções que aceitam expressões simbólicas arbitrárias e as transformam em código Lisp executável. Por exemplo, suponhamos que queremos ter um comando que atribui o mesmo valor a duas variáveis, como `(set2 x y (+ z 3))`. Quando `z` for 2 o valor de `x` e `y` deverá ser 5. Naturalmente não podemos implementar isso como uma função, já que os valores de `x` e `y` seriam computados antes de serem passados para a função, e ela não teria nenhum conhecimento de que variáveis devem ser atribuídas. Na verdade, queremos que quando Lisp veja a expressão `(set2` v_1 v_2 e`)` ele a trate como equivalente a `(progn (setq` v_1 `e) (setq` v_2 e`))`. Uma macro em Lisp nos permite fazer exatamente isso, transformar o padrão de entrada de `(set2` v_1 v_2 e`)` para `(progn`

(setq v_1 e) (setq v_2 e)). A implementação de set2 pode ser vista no exemplo 5.8. Uma macro não computa nenhum valor, ela apenas transforma uma expressão em algo que pode ser entendido pelo compilador. No exemplo 5.8 a macro está literalmente construindo listas que para formar a expressão desejada.

Exemplo 5.8 Uma macro simples

```
(defmacro set2 (v1 v2 e)
  (list 'progn (list 'setq v1 e) (list 'setq v2 e)))
```

Contudo, é fácil ver que o uso explícito de list dificulta a escrita e leitura de expressões complexas. Geralmente se usa o recurso de *backquote*, que indica que na expressão que segue, cada sub-expressão precedida por uma vírgula deve ser computada, enquanto cada sub-expressão sem a vírgula deve ser retornada sem avaliação (i.e. como com quote). Uma nova implementação de set2 usando *backquote* pode ser vista no exemplo 5.9. O uso de *backquote* permite uma visualização muito mais clara da expressão resultante pela macro.

Exemplo 5.9 A macro simples com *backquote*

```
(defmacro set2 (v1 v2 e)
  `(progn (setq ,v1 ,e) (setq ,v2 ,e)))
```

5.8 Programação orientada a objetos

Em Common Lisp classes são definidas com defclass e métodos com defmethod. Uma diferença em relação a maioria das linguagens de programação orientada a objetos é que os métodos não pertencem a uma classe específica. O exemplo 5.10 mostra a implementação da função genérica soma, que especializa em diferentes tipos de objetos como cadeia de caracteres, listas, e vetores. Uma visão mais aprofundada do sistema de objetos do Common Lisp pode ser vista em [Seibel 2004].

6 Exemplos de Meta-programação em Lisp

6.1 Notação pós-fixa

Conforme visto Lisp permite a criação de novas formas especiais que permitem mudar o sentido da sintaxe. Por exemplo, como modificar a sintaxe da linguagem para usar a notação posfixa? Uma solução para diversas linguagens de programação seria definir a expressão como uma cadeia de caracteres e fazer uma função para parsear-la, que seria usada como le_dados("3 4 5 +").

Em Lisp podemos definir uma macro com duas linhas (ex. 6.1). Em ambos exemplos uma espécie de extensão do compilador teve que ser criada. Em outras linguagens um *parser* tem que ser criado pelo próprio programador e não tem (necessariamente) conexão com a linguagem. Por outro lado, Lisp provê meios de estender a linguagem.

Apesar de curta, a macro postfix permite que expressões posfixas sejam aninhadas, como (postfix (2 (postfix (3 4 *)) +)). Contudo, o resultado é deselegante e

Exemplo 5.10 Exemplo de métodos com diferentes despachos

```
(defmethod soma ((a string) (b string))
  (concatenate 'string a b))

(defmethod soma ((L1 list) (L2 list))
  (append L1 L2))

(defmethod soma ((L1 list) (S1 string))
  (soma (apply #'concatenate 'string (mapcar #'princ-to-string L1))
        S1))

(defmethod soma ((x vector) (y vector))
  (concatenate 'vector x y))

(defmethod soma (X Y)
  (error (format nil "tipo de dado ~a não definido nessa função" (type-of X))))
```

Exemplo 6.1 Notação posfixa

```
(defmacro postfix (expr)
  (reverse expr))

(postfix (2 3 4 +))
```

prolixo. Seria mais interessante se pudessemos escrever algo como `[2 3 5 +]` e o sistema reconhecesse automaticamente que se trata de uma expressão postfixa. Naturalmente, isso é fácil em Lisp. Tudo que precisamos é definir que uma expressão delimitada por colchetes corresponde à uma chamada de `postfix` (ex. 6.2). Esse tipo de macro é conhecido como macro de leitura (*reader macros*). Desse modo podemos escrever expressões como `[10 30 40 +]`, `[30 50 14.3 *]`, e `[2 [4 9 -] *]`, como se esse recurso estivesse disponível na linguagem desde o começo.

Exemplo 6.2 Definindo macros de leitura

```
(defun open-bracket (stream char)
  `(postfix ,(read-delimited-list #\] stream t)))

(set-macro-character #\[ #'open-bracket)
(set-macro-character #\] (get-macro-character #\)))
```

Algumas explicações para auxiliar no entendimento do código no exemplo 6.2: os caracteres em Lisp são precedidos por `#\`, então “a” é representado por `#\a`, “b” por `#\b` e abre colchete por `#\[`. A notação `#'` é uma abreviação da forma especial `function`, de modo que `#'open-bracket` e `(function open-bracket)` são equivalentes. A forma especial `(function <nome>)` retorna a função associada com o nome `<name>`. No exemplo 6.2 a função `open-bracket` está sendo passada como parâmetro para `set-macro-character`.

Sem o `#' open-bracket` seria uma variável.

6.2 Compreensão de listas

Em Haskell o algoritmo para o *quick sort* pode ser expressado de maneira simples e elegante, com 5 linhas, (ex. 6.3) enquanto em Lisp temos uma implementação mais prolixa, ainda que elegante, com 11 linhas, (ex. 6.4). (Veja o apêndice A para mais exemplos de *quicksort* em Lisp).

Exemplo 6.3 Quicksort em Haskell

```
qsort []     = []
qsort (x:xs) = qsort elts_lt_x ++ [x] ++ qsort elts_greq_x
                 where
                   elts_lt_x   = [y | y < - xs, y < x]
                   elts_greq_x = [y | y <- xs, y >= x]
```

Exemplo 6.4 Quicksort em Lisp

```
(defun qsort (lst)
  (labels ((partition (left right acc)
             (if (null acc)
                 (append (qsort left)
                         (cons (car lst) (qsort right)))
                 (if (<= (first acc) (first lst))
                     (partition (cons (first acc) left) right (rest acc))
                     (partition left (cons (first acc) right) (rest acc))))))
    (if (null (rest lst))
        lst
        (partition nil nil (rest lst)))))
```

A versão em Haskell é concisa porque usa uma abstração apropriada (compreensão de lista). Algumas linguagens como Haskell, Python, e Miranda tem compreensão de lista embutida da linguagem, o que não é o caso de Common Lisp. Em Miranda a compreensão de listas pode ser escrita como `[x | x <- xs ; odd x]`. No seguinte exemplo vamos implementar compreensão de listas em Lisp de modo a poder usar diretamente em Lisp uma notação como `[x (x <- xs) (oddp x)]`. É importante observar que essa sintaxe é bem diferente da sintaxe normal de Lisp, mas ainda assim pode ser implementada facilmente usando macros, enquanto que para implementar um recurso sintático como esse em outras linguagens o compilador teria que ser modificado. Em Lisp, um recurso sintático maior como esse pode ser implementado em menos de 30 linhas de código, como demonstrado em [Lapalme 1991]. O código completo para a implementação de compreensão de listas pode ser visto no exemplo 6.5. Apesar de uma análise completa da implementação estar fora do escopo desse documento, é fácil ver que a mesma técnica geral usada para a implementação da sintaxe posfixa 6.2 é usada aqui. A função `set-macro-character` associa os caracteres "[" e "]" às funções `open-bracket` e `closing-bracket`, respectivamente. A maior parte do trabalho é efetuado pela macro de 17 linhas `comp`.

Exemplo 6.5 Compreensão de listas em Lisp

```
(defun open-bracket (stream ch)
  (defmacro comp ((e &rest qs) l2)
    (if (null qs) `(cons ,e ,l2)
        (let ((q1 (car qs))
              (q (cdr qs)))
          (if (not (eq (cadr q1) `<-))
              `(if ,q1 (comp (,e ,@q),l2) ,l2)
              (let ((v (car q1))
                    (l1 (third q1))
                    (h (gentemp "H-"))
                    (us (gentemp "US-"))
                    (us1 (gentemp "US1-")))
                `(labels ((,h (,us)
                            (if (null ,us) ,l2
                                (let ((,v (car ,us))
                                      (,us1 (cdr ,us)))
                                  (comp (,e ,@q) (,h ,us1))))))
                   (,h ,l1)))))))
  (do ((l nil)
       (c (read stream t nil t)(read stream t nil t)))
      ((eq c `|]|) `(comp ,(reverse l) ()))
    (push c l)))

(defun closing-bracket (stream ch) `|]|)

(eval-when (compile load eval)
  (set-macro-character #\[ #'open-bracket)
  (set-macro-character #\] #'closing-bracket))
```

Tendo implementado compreensão de listas o algoritmo para o *quick sort* pode ser expressado de uma maneira bem semelhante a Haskell, como visto no exemplo 6.6. O ponto principal desse exemplo é que uma abstração diferente pode ser implementada em Lisp como se fizesse parte da linguagem. E porque ela captura a idéia básica da abstração, serve não apenas para um caso particular, mas para diferentes fins onde essa abstração pode ser empregada.

Exemplo 6.6 *Quicksort* usando compreensão de listas

```
(defun qsort (ax)
  (and ax (let ((a (car ax))
                (x (cdr ax)))
            (append (qsort [y (y <- x) (< y a)])
                    (list a)
                    (qsort [y (y <- x) (>= y a)])))))
```

7 Implementando linguagens de domínio específico

7.1 Uma linguagem para gerar documentos

Nessa seção vamos criar uma linguagem de domínio especifico para gerar HTML. Ou seja, poder escrever algo como no exemplo 7.1 ter o código HTML gerado automaticamente. É importante salientar que os exemplos dessa seção são meramente demonstrativos. Uma implementação real usaria técnicas diferentes.

Exemplo 7.1 LDE para gerar HTML

```
(html
  (h1 "Título do Artigo")
  (h2 "Autor")
  (p "Primeiro parágrafo e um " (b "negrito")))
```

A maneira mais simples para implementar essa LDE é através do uso funções, onde criamos uma função para cada marcação HTML (ex. 7.2). Desse modo a LDE no ex. 7.1 gera o código HTML no ex. 7.3.

O problema com o exemplo 7.2 é que todas as funções são praticamente iguais. Em geral, repetição de código é um convite à refatoração. Uma possível solução é capturar a abstração básica em uma função (ex. 7.4). Dessa maneira podemos não apenas substituir todas as funções que foram definidas no exemplo 7.2 por uma única função, como ela permite que qualquer marcador seja usado.

Conceitualmente o exemplo 7.4 é melhor e mais coeso, mas para usá-lo teríamos que escrever o documento usando o nome da função antes de cada chamada (ex. 7.5). Isso remove a idéia básica de usar uma linguagem de domínio específico, além de seu uso ser menos ortogonal, limpo e claro que originalmente pretendido em 7.1.

Nós precisamos de uma maneira de capturar a abstração básica (como em 7.4) e manter a facilidade de escrever o código (como em 7.1). Para isso podemos criar as

Exemplo 7.2 Funções para gerar HTML

```
(defun html (&rest text)
  (format nil "<html>~{~a~}</html>~%" text))

(defun h1 (&rest text)
  (format nil "<h1>~{~a~}</h1>~%" text))

(defun h2 (&rest text)
  (format nil "<h2>~{~a~}</h2>~%" text))

(defun p (&rest text)
  (format nil "<p>~{~a~}</p>~%" text))

(defun b (&rest text)
  (format nil "<b>~{~a~}</b>" text))

(defun i (&rest text)
  (format nil "<i>~{~a~}</i>" text))
```

Exemplo 7.3 HTML gerado

```
<html>
<h1>Título do Artigo</h1>
<h2>Autor</h2>
<p>Primeiro parágrafo e um <b>negrito</b></p>
</html>
```

Exemplo 7.4 Função para gerar marcações HTML

```
(defun tag (tag &rest text)
  (format nil "<~a>~{~a~}</~a>~%" tag text tag))
```

Exemplo 7.5 Uso da função tag

```
(tag 'html
 (tag 'h1 "Título do Artigo")
 (tag 'h2 "Autor")
 (tag 'p "Primeiro parágrafo e um"
      (tag 'b (tag 'i "negrito itálico"))))
```

funções do exemplo 7.2 automaticamente usando macros. A macro `make-tag` cria uma função cujo nome é o valor do argumento formal da macro (ex. 7.6). Por exemplo, a expressão `(make-tag html)` vai gerar a função vista no exemplo 7.7.

Exemplo 7.6 Macro para criar funções

```
(defmacro make-tag (name)
  `(defun ,name (&rest text)
     (format nil "<~a>~{~a~}</~a>~%" ',name text ',name)))
```

Exemplo 7.7 Função gerada pela macro

```
(defun html (&rest text)
   (format nil "<~a>~{~a~}</~a>~%" 'html text 'html))
```

As funções de marcação podem ser criadas com código como `(make-tag html)`, `(make-tag h1)`, e assim por diante para cada marcação. Claro que é mais fácil criar uma função para fazer isso automaticamente a partir de uma lista (ex. 7.8). Tendo isso as marcações podem ser definidas com algo como `(make-all '(html h1 h2 h3 h4 p i em))`.

Exemplo 7.8 Função para criar diversas marcações

```
(defun make-all (lst)
   (dolist (f lst)
     (eval `(make-tag ,f))))
```

Desse modo temos um código que é mais genérico que os anteriores e mais extensível. E podemos não apenas escrever html na forma proposta no exemplo 7.1 como estender o exemplo para criar uma LDE apropriada para descrever documentos (ex. 7.9).

Com uma pequena alteração em `make-tag` podemos especificar que a marcação html pode ser diferente do nome da marcação usada (ex. 7.10). Então `(make-tag html)` vai gerar a função `html` que gera a marcação HTML `html` como antes, enquanto `(make-tag documento html)` vai gerar a função `documento` mas que gera a marcação HTML `html`.

Finalmente, precisamos alterar `make-all` para quando as marcações estiverem dentro de uma lista como `(documento html)`, o primeiro elemento será o nome da marcação do documento e o segundo a marcação HTML que deverá ser gerada (ex 7.11). Com a nova versão de `make-all` podemos definir marcações como visto no exemplo 7.12.

Resumindo, em menos de 15 linhas definimos código suficiente para criar uma linguagem de domínio específico extensível para gerar documentos. Um exemplo de utilização dessa linguagem pode ser vista no exemplo 7.13.

7.2 Uma linguagem de domínio específico

Nessa seção vamos demonstrar uma solução usando Lisp para a LDE proposta em [Fowler 2005]. Nesse artigo ele usa XML e C# para criar uma LDE. Rainer Joswig de-

Exemplo 7.9 LDE para documentos

```
(documento
 (titulo "Título do Artigo")
 (autor "Autor")
 (p "Primeiro parágrafo e um" (b (i "negrito itálico"))))
```

Exemplo 7.10 Modificação em make-tag

```
(defmacro make-tag (tag-name &optional html-tag)
  (let ((name (if html-tag html-tag tag-name)))
    `(defun ,tag-name (&rest text)
       (format nil "<~a>~{~a~}</~a>~%" ',name text ',name))))
```

Exemplo 7.11 Modificação em make-all

```
(defun make-all (lst)
   (dolist (f lst)
     (print f)
     (eval `(make-tag ,@(if (listp f)
                            f
                            (list f))))))
```

Exemplo 7.12 Nova definição de marcações

```
(make-all '(html h1 h2 h3 h4 b p i em
           (documento html) (titulo h1) (autor h2) (secao p)))
```

Exemplo 7.13 Exemplo de uso da LDE

```
(documento
 (titulo "Meu Artigo")
 (autor "Pedro Kröger")
 (secao "Bla bla bla" (b "foo") (i "bar")))
```

monstrou em [Joswig 2005] que uma versão em Lisp é incrivelmente mais simples, curta e fácil de entender. É essa versão que demonstraremos nessa seção.

No exemplo proposto por Fowler é necessário criar objetos e classes a partir de dados de entrada, onde cada linha se relaciona a uma classe diferente. Um exemplo de entrada de dados pode ser visto abaixo no exemplo 7.14. Os pontos representam dados que não interessam ao exemplo. A primeira linha indica a posição onde se espera encontrar o tipo de dados. SVCL indica uma chamada de serviço, USGE um registro de uso. Os caracteres em seguida representam os dados para o objeto, de modo que os caracteres da posição 4 a 18 em uma chamada de serviço indicam o nome do cliente.

Exemplo 7.14 Entrada de dados

```
#123456789012345678901234567890123456789012345678901234567890
SVCLFOWLER         10101MS0120050313.......................
SVCLHOHPE          10201DX0320050315.......................
SVCLTWO           x10301MRP220050329.......................
USGE10301TWO          x50214..7050329.......................
```

Fowler sugere usar uma LDE para mapear que posições estão relacionadas aos campos e em que classes (ex. 7.15). Além dessa sintaxe, ele sugere outro exemplo de LDE definida em XML. No exemplo 7.16 nós definimos uma sintaxe "lispficada" da LDE proposta por Fowler.

Exemplo 7.15 LDE para mapear posições

```
mapping SVCL dsl.ServiceCall
  4-18:  CustomerName
  19-23: CustomerID
  24-27: CallTypeCode
  28-35: DateOfCallString

mapping USGE dsl.Usage
  4-8:   CustomerID
  9-22:  CustomerName
  30-30: Cycle
  31-36: ReadDate
```

Nós vamos implementar `defmapping` como macros. Isso significa que `defmapping` vai criar as classes `service-call` e `usage` de acordo com os mapeamentos `SVCL` e `USGE`. É interessante observar que enquanto a versão de Fowler tem 70 linhas, a versão de Joswig tem apenas 12 linhas! A definição de `defmapping` pode ser vista no exemplo 7.17. A macro `defmapping` é simples no sentido de que ela "apenas" gera uma classe, cujo nome corresponde ao primeiro parâmetro da macro, e dois métodos, `find-mapping-class-name` e `parse-line-for-class`. O método `parse-line-for-class` é especializado para a classe definida em `name`, ou seja, `service-call` e `usage`.

O exemplo 7.18 demonstra um exemplo de uso. Os métodos `parse-line-for-class` e `find-mapping-class-name` são usados para cada classe.

Exemplo 7.16 LDE em Lisp

```
(defmapping service-call "SVCL"
  (04 18 customer-name)
  (19 23 customer-id)
  (24 27 call-type-code)
  (28 35 date-of-call-string))

(defmapping usage "USGE"
  (04 08 customer-id)
  (09 22 customer-name)
  (30 30 cycle)
  (31 36 read-date))
```

Exemplo 7.17 Definição de defmapping

```
(defmacro defmapping (name mapping &body fields)
  `(progn
     (defclass ,name ()
       ,(loop for (nil nil slot) in fields collect slot))
     (defmethod find-mapping-class-name ((mapping (eql ',(intern mapping))))
       ',name)
     (defmethod parse-line-for-class (line (class-name (eql ',name)))
       (let ((object (make-instance class-name)))
         (loop for (start end slot) in ',fields
            do (setf (slot-value object slot)
                     (subseq line start (1+ end))))
         object))))
```

Exemplo 7.18 Exemplo de uso para a LDE

```
(defparameter *test-lines*
"SVCLFOWLER         10101MS0120050313.....................
SVCLHOHPE          10201DX0320050315.....................
SVCLTWO           x10301MRP220050329.....................
USGE10301TWO          x50214..7050329.....................")

(with-input-from-string (stream *test-lines*)
  (loop for line = (read-line stream nil nil)
     while line
     collect (parse-line-for-class line (find-mapping-class-name
                                         (intern (subseq line 0 4))))))
```

8 Conclusão e discussão

Nesse tutorial ilustramos o poder de expressão e elegância da meta-programação em Lisp e o seu uso na criação de linguagens de domínio específico. Também ilustramos como Lisp torna esse tipo de técnica simples através do uso de macros.

Achar exemplos adequados e didáticos é um dos problemas em se demonstrar recursos como macros. Por exemplo, em algumas linguagens com avaliação preguiçosa como Haskell, é desnecessário o uso de macro para definir coisas como expressões condicionais como `if`. Desse modo os usuários dessas linguagens podem, baseado nos exemplos simples, achar que um recurso como macro é desnecessário na sua linguagem. Naturalmente, é fora do escopo desse documento tratar de casos avançados de macros, meta-programação e LDE. Para um tratamento mais avançado do assunto, ver [Graham 1993] e [Norvig 1992]. [Seibel 2004] é um excelente livro direcionado para programadores experientes que mostra recursos modernos de Common Lisp de uma maneira prática.

Alguns itens e problemas relacionados a macros como captura de variáveis, macros higiênicas, e distinção de tempo de expansão de macros e tempo de execução estão fora do escopo desse documento, contudo a literatura pode ser consultada sobre esses assuntos.

A Quicksort em Lisp

A seção 6.2 demonstrou o uso de macros para implementar compreensão de listas em Lisp. É importante observar que na verdade Lisp tem uma forma de compreensão de lista na macro `loop`. Uma implementação do *quicksort* usando `loop` pode ser vista no exemplo A.1.

Exemplo A.1 Quicksort usando loop

```
(defun qsort (lst)
  (when lst
    (let* ((x  (car lst))
           (xs (cdr lst))
           (lt  (loop for y in xs when (< y x) collect y))
           (gte (loop for y in xs when (>= y x) collect y)))
      (append (qsort lt) (list x) (qsort gte)))))
```

Uma outra maneira de implementar *quicksort* em Lisp é através do uso de filtros como `remove-if` (ex. A.2). Essa versão é particularmente elegante, apesar de não ser a mais rápida.

Exemplo A.2 Quicksort usando remove-if

```
(defun qsort (ax)
  (and ax
      (let ((a (car ax)) (x (cdr ax)))
        (append (qsort (remove-if #'(lambda (y) (< a y)) x))
                (list a)
                (qsort (remove-if #'(lambda (y) (>= a y)) x))))))
```

Referências

Abelson, H. and Sussman, G. J. (1987). Lisp: A language for stratified design. Technical Report AI Lab Memo AIM-986, MIT AI Lab.

Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. The MIT Press.

Fowler, M. (2005). Language workbenches: The killer-app for domain specific languages? Disponível em `www.martinfowler.com/articles/languageWorkbench.html`.

Graham, P. (1993). *On Lisp: Advanced Techniques for Common Lisp*. Prentice Hall.

Joswig, R. (2005). Martin fowler talks about lisp. Disponível em `groups.google.com/group/comp.lang.Lisp/msg/4fe888b58ffa83b8?hl=en`.

Kieburtz, R. B. (2000). Implementing closed domain-specific languages. In *SAIG '00: Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 1–2, London, UK. Springer-Verlag.

Knight, S. (2004). Scons user guide. Disponível em `www.scons.org`.

Lapalme, G. (1991). Implementation of a "lisp comprehension" macro. *SIGPLAN Lisp Pointers*, IV(2):16–23.

Lugovsky, V. S. (2004). Using a hierarchy of domain specific languages in complex software systems design.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344.

Norvig, P. (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Seibel, P. (2004). *Practical Common Lisp*. Apress.

Shivers, O. (1996). A universal scripting framework or lambda: The ultimate 'little language'. In *Asian Computing Science Conference*, pages 254–265.

Sloane, T., Mernik, M., and Heering, J. (2003). When and how to develop domain-specific languages.

Spinellis, D. (2001). Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99.

Stallman, R. M. and Mcgrath, R. (1998). *GNU make: a program for directing recompilation*. Free Software Foundation, Boston, MA.

Svingen, B. (2006). When lisp is faster than c. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 957–958, New York, NY, USA. ACM Press.

van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36.

Verna, D. (2006a). Beating C in scientific computing applications. In *Third European Lisp Workshop*, Nantes, France.

Verna, D. (2006b). How to make Lisp go faster than C. Hong Kong.