Dagstuhl Seminar 9538
New Trends in the Intergration of Paradigms

Organized by:
Chris Hankin (Imperial College, London)
Hanne Riis Nielson (Aarhus University)

September 18-22,1995

Preface

A number of programming paradigms have been identified, including object-oriented, concurrent, applicative (functional), declarative (logic), as well as imperative. Some of these are more successful than others but no single paradigm seems adequate for all aspects of software production; an example being the meaningful exploitation of the massive paralellism offered by current and future hardware. There appear to be two ways forward: integration of paradigms taking the best from each or the development of coordination mechanisms which support inter-language working.

Integration of paradigms necessitates a deeper understanding of the design methodologies, the semantic basis, the techniques for analysis, transformation and implementation, and the application in the construction of sofware as well as the verification of the ensuing systems. These problems may be approached via the development of integrated languages or may be studied through the design of calculi that exhibit the essential features. In the past the study of the lambda-calculus and calculi for concurrency have helped in providing a breakthrough in our understanding of functional languages and concurrent systems and calculi for object-orientation are on their way.

Coordination languages are based on a model of concurrent computation in which agents cooperate via a shared data space, rather than message passing or shared variables. Examples range from collections of coordination primitives to be embedded in other languages, as in Linda, to pure coordination languages such as Gamma. Problems relating to semantics, (open systems) verification and static analysis are current research topics.

In both approaches, we anticipate that techniques of a deductive nature will prove essential for most theoretical studies: semantics is likely to be based on variations of operational semantics whose descriptive power surpasses that of denotational semantics and more recent semantic frameworks. Consequently analyses may profitably be based on inference systems (perhaps including type systems as special cases). To obtain efficient implementations the use of constraints seem beneficial. However, many problems still require unifying frameworks, not least those of transformation and implementation.

In order to ensure that there was a good interaction between the advocates of these two different approaches we started the meeting with a number of provocative talks related to *language design*. It was followed by discussions in small groups which later reported back in a plenary session. A major question that arose during these discussions concerned the need to identify concrete examples where integration of specific paradigms adds value; it was clearly felt that it did not make sense to integrate any arbitrary two paradigms. To introduce another dimension in the discussions we continued the meeting with a number of *semantics related* talks bringing in issues as analysis and verification of programs and systems. Again this was followed by a discussion session which partly was a continuation of the previous one and partly was concerned with the applicability of the semantic techniques to the various programming paradigms and their integration. The seminar concluded with a number of short talks showing a spectrum of other problems studied for multi-paradigmatic languages.

The Dagstuhl seminar provided an excellent opportunity for bringing together two hitherto disparate groups of scientists. Thanks to the engagement of the participants the seminar exposed a lot of interesting developments in this emerging and exciting area. By having ample time for discussions we improved our understanding of the differences and commonalities between the various approaches to integration of paradigms and we got valuable insights into how semantic issues are handled in the different settings.

Acknowledgement: We would like to thank Ian Mackie for helping us to collect the abstracts.

MONDAY 18th September 1995

09.00 - 09.15	Introduction	Chris Hankin
09.15 - 10.15	Opening Statements	Everyone
10.45 - 11.30	The Oz Programming Model	Gert Smolka
11.30 - 12.15	The Essence of Functional Logic Languages	Michael Hanus
14.00 - 14.45	Foundations of Interactive Computing	Peter Wegner
14.45 - 15.30	Modularity and Abstraction Mechanisms ...	Gul Agha
16.00 - 16.45	Declarative Programming	Herbert Kuchen
16.45 - 17.30	Gamma	Chris Hankin

TUESDAY 19th September 1995

09.00 - 10.30	Discussion Groups	
11.00 - 12.15	Plenary Session	
14.00 - 14.45	Rewriting Logic	José Meseguer
14.45 - 15.30	Abstract Interpretation	Alan Mycroft
16.00 - 16.45	Reasoning about Programs	Carolyn Talcott
16.45 - 17.30	Agent Programming ...	Bent Thomsen

WEDNESDAY 20th September 1995

09.00 - 10.30	Discussion Groups	
11.00 - 12.15	Plenary Session	

THURSDAY 21st September 1995

09.00 - 09.30	Geometry of Interaction	Ian Mackie
09.30 - 10.00	Using CHAMS ...in Facile	Lone Leth
10.00 - 10.30	What is Coordination ...	Paolo Ciancarini
10.45 - 11.30	Performance Eval. in Systems Spec.	Roberto Gorrieri
14.00 - 14.45	The TAO Computation Model	António Porto
14.45 - 15.30	Reasoning about Higher-order Processes	Roberto Amadio
16.00 - 16.45	Integration of OO and FP	Laurent Dami
16.45 - 17.30	Towards Temporal Type Systems	Mads Dam
17.30 - 18.00	A general model for object-based systems	Holger Naundorf

FRIDAY 22nd September 1995

09.00 - 10.30	Closing discussion	

Contents

Gul A. Agha
Modularity and Abstraction Mechanisms for Specifying Concurrent Systems 6
Roberto M. Amadio and Mads Dam
Reasoning about Higher-OrderProcesses 7
Paolo Ciancarini
What is coordination and what has to do with integration 8
Mads Dam
Towards Temporal Type Systems 9
Laurent Dami
Integration of Functional and Object-Oriented Paradigms:
Achievements and Open Problems 10
Roberto Gorrieri
Performance Evaluation in Systems Specification 11
Chris Hankin
Gamma 13
Michael Hanus
The Essence of Functional Logic Languages 14
Herbert Kuchen
Declarative Programming 15
Lone Leth
Using CHAMs for the design of the distributed constructs in Facile 16
Ian Mackie
Geometry of Interaction 18
Holger Naundorf
A general Model for object based Systems 19
Oscar Nierstrasz
Towards Composition Languages 20
António Porto
The TAO computation model 21
Didier Rémy
Typechecking record concatenation using constrained type 22
Gert Smolka
The Oz Programming Model 23
Carolyn Talcott
Reasoning about Programs 24
Bent Thomsen
Agent Programming Needs Integration of Multiple Paradigms
or How to program the worlds largest computer 25
Peter Wegner
Foundations of Interactive Computing 26

Modularity and Abstraction Mechanisms for Specifying Concurrent Systems

Gul A. Agha

University of Illinois
agha@cs.uiuc.edu

Abstract

A central goal of our research is to address the complexity of building and maintaining large-scale software systems. We describe a number of programming language constructs which abstract over common coordination patterns as well as their representation. Such constructs include activators, actorspaces, synchronizers, and protocols. The constructs are abstractions built using autonomous concurrent objects (actors). Specifically, we show that the Actor model provides a flexible basis for building a meta-architecture for implementing coordination abstractions. The talk provides examples which illustrate how using these constructs simplifies the task of designing, implementing and modifying software systems without compromising computational efficiency.

Reasoning about Higher-Order Processes

Roberto M. Amadio
(joint work with Mads Dam)

Sophia Antipolis
amadio@cma.cma.fr

Abstract

We address the specification and verification problem for process calculi such as Chocs, CML and Facile where processes or functions are transmissible values. Our work takes place in the context of a static treatment of restriction and of a bisimulation-based semantics. As a paradigmatic and simple case we concentrate on (Plain) Chocs. We show that Chocs bisimulation can be characterized by an extension of Hennessy-Milner logic including a constructive implication, or function space constructor. This result is a non-trivial extension of the classical characterization result for labelled transition systems. In the second part of the paper we address the problem of developing a proof system for the verification of process specifications. Building on previous work for CCS we present a sound proof system for a Chocs sub-calculus not including restriction. We present two completeness results: one for the full specification language using an infinitary system, and one for a special class of so-called *well-described* specifications using a finitary system.

The results of this work are reported in [1].

What is coordination and what has to do with integration

Paolo Ciancarini

Dept. of Computer Science, Univ. of Bologna
Pza. di Porta S.Donato, 5 – 40127 Bologna - Italy
tel. +39 51 354506 fax. +39 51 354510 e-mail: cianca@cs.unibo.it
WWW Home page: http://www.cs.unibo.it/ cianca/index.html

Abstract

The birth of world-wide information systems like the WWW suggests the possibility of building on the Internet infrastructure (middleware) similarly open systems for coordination of people and their activities. Coordination languages and models were born for parallel programming tasks, but are more and more explored as tools for designing and building open and distributed software systems. A coordination model should clearly define its coordination entities, coordination media, and coordination laws, that should be implemented in the coordination languages which embed such a model. A coordination language is itself a tool for integration. At least at application level coordination languages have been used for "putting together" existing pieces of software; some efforts are in progress to use similar concepts also to combine and integrate specification notations of diverse natures.

Towards Temporal Type Systems
Or
Reasoning About Open Distributed Systems

Mads Dam
SICS
mfd@sics.se

Abstract

Interesting properties of open distributed systems mix functional and temporal aspects. Systems with components that are incompletely specified, or which spawn processes locally or remotely require new compositional techniques for their verification, at least as far as their temporal properties are concerned. Such techniques are crucial to the development of future type systems for high-level distributed programming languages that incorporate more information about the dynamic, and interactive, behaviour of systems. We present the first proof system that attempts to solve this problem in a general fashion. Specifically we present a compositional proof system for checking processes against formulas in the modal μ-calculus, capable of handling general infinite-state processes, and hence process spawning. The proof system is obtained in a systematic way from the operational semantics of the underlying process algebra. A non-trivial proof example is given, and the proof system is shown to be sound in general, and complete for finite-state processes.

Integration of Functional and Object-Oriented Paradigms: Achievements and Open Problems

Laurent Dami

CUI, University of Geneva
dami@cui.unige.ch

Abstract

Both functional and object-oriented languages have interesting mechanisms for software composition and reuse. Functional languages have parametric polymorphism and type inference; object-oriented languages have subtype polymorphism and various constructs for incremental construction, like inheritance or delegation. Although clearly different, these mechanisms are not orthogonal, and are not easily combined. Some attempts have been made to design new languages which borrow syntactic constructs from both paradigms; however, such attempts, lacking a formal semantics, fail to fully understand the interaction between these constructs, and often lose many properties of the original FP or OO paradigms. So, in our view, successful integration can only proceed from a sound formal model supporting both paradigms. Efforts are currently being made in this direction, both in the ML and in the Haskell community, based on some extensions of the lambda-calculus with either records or with "primitive objects" la Abadi/Cardelli. We show how our Lambda-N calculus, a lambda-calculus with "name-based interaction", i.e. parameter passing by keywords, is also appropriate, and probably simpler, for this formal foundation. The open problems then consist in "coming back to the surface", i.e. finding the best combination of high-level constructs which make such integration convenient for programmers, and amenable to reasonable implementations. Some aspects are already quite clear: for example, most researchers seem to agree that the proper way to combine type inference, parametric polymorphism and subtyping is to use *recursively constrained types*, i.e. types with collections of subtyping constraints on their free type variables. On the other hand, there is still no obvious answer for a number of design aspects, like for example how to combine pattern matching with OO data abstraction.

Performance Evaluation in Systems Specification

Roberto Gorrieri

University of Bologna
gorrieri@cs.unibo.it

Abstract

The need of integrating the performance analysis of a concurrent system into the design process of the system itself has been widely recognized and stimulated many researchers. The problem is that, in the case when the performance aspect is neglected, time-critical concurrent systems, such as real-time systems and communication protocols, cannot be modeled in a completely satisfactory manner. And, more important, there is no way of estimating the performance of the concurrent systems under consideration. It often happens that a concurrent system is first fully designed and tested for functionality, and afterwards tested for efficiency. As a consequence, if the performance is detected to be poor, the concurrent system has to be redesigned, thus negatively affecting both the design costs and the delivery at a fixed deadline.

In the last two decades a remarkable effort has been made in order to enhance the expressiveness of well-established formalisms developed in the theory of concurrency, such as Petri nets and process algebras, based on the introduction of the concept of time. Among the results of this effort, we shall focus our attention on stochastic Petri nets and stochastic process algebras.

Stochastic Petri nets constitute a very mature field in which it is possible to model and analyze concurrent systems both from the qualitative and quantitative point of view. However, the use of stochastic Petri nets implies some drawbacks related to the lack of both linguistic and semantic compositionality as well as the fact that a real integrated analysis does not exist since the functional and the stochastic models obtained from each net are separately studied. This problems could be solved by putting a stochastic process algebra on top of the stochastic Petri net formalism.

In this talk we present an integrated approach for modeling and analyzing concurrent systems based on stochastic process algebras and stochastic Petri nets, which relates different points of view of concurrent systems (centralized vs. distributed) as well as different aspects of their behavior (qualitative vs. quantitative). The approach is divided into two phases.

The first phase, interfaced with the system designer, consists of representing the concurrent system as a term of the stochastic process algebra. Because of compositionality, the system designer is allowed to develop the algebraic representation of the system in a modular, stepwise refinement manner. If the stochastic process algebra is equipped with an interleaving semantics accounting for both the qualitative and quantitative part of the system behavior, the interleaving model of the algebraic representation of the system can be projected on a functional model and a performance model which can be analyzed by means of tools like Concurrency Workbench and SHARPE, respectively.

The second phase consists of automatically obtaining from the algebraic representation of the system an equivalent distributed representation. In order for it to be implemented, the stochastic process algebra is required to have a distributed semantics as well. A suitable distributed model might be a stochastic Petri net, because stochastic Petri nets constitute

a mature field in which it is possible the description and the analysis of concurrent systems both from the functional and the performance viewpoint, assisted by tools like GreatSPN. The net representation of the concurrent system is derived from the algebraic one without intervention of the system designer, in order to avoid overhead concerning graphical complexity and absence of compositionality. The net representation turns out to be useful in the case when a less abstract representation is required highlighting dependencies and conflicts among system activities, and helpful to detect some properties (e.g., partial deadlock) which can be easily checked only in a distributed setting.

This integrated approach is instantiated to the case of the stochastic process algebra Extended Markovian Process Algebra (EMPA), as it is a stochastic process algebra fulfilling the needed requirements. In fact, it is supplied with an interleaving semantics, from which a functional and a performance semantics can be derived, as well as a distributed semantics. The name of the algebra stems from the fact that action durations are mainly expressed by means of exponentially distributed random variables (hence Markovian), but it is also possible to express actions having duration zero (hence Extended).

Gamma

Chris Hankin

Imperial College, UK
clh@doc.ic.ac.uk

Abstract

Gamma is a concurrent programming model which is based on the chemical reaction metaphor. A Gamma program may be viewed as a conditional multi-set rewrite system. Rules in the program interact by generative communication via the multi-set.

In this talk we give an overview of the current developments of the Gamma formalism. We start with some simple examples. We then discuss the semantics of Gamma programs; we start with an intuitive structural operational semantics for Gamma and then develop a resumption-style denotational semantics. The latter is used to derive a program logic for Gamma using notions from Abramsky's Domain Theory in Logical Form. The soundness of the logic follows from the general theory. The logic is a variant of Brookes' Transition Assertion Logic; it is related to a logic for Gamma that has been previously studied by Errington, Hankin and Jensen. We conclude by considering higher-order extensions to the basic Gamma formalism.

This talk is based on joint work with David Cohen, Simon Gay, Daniel Le Métayer, Juarez Muylaert Filho and David Sands.

The Essence of Functional Logic Languages

Michael Hanus

Informatik II, RWTH Aachen
D-52056 Aachen, Germany
hanus@informatik.rwth-aachen.de

Abstract

Declarative programming languages like functional or logic languages have raised an increasing interest during recent years. The expressive power of such languages results in smaller and more readable programs and requires less development time. Moreover, the execution of functional or logic programs can compete with imperative programs due to efficient implementation techniques which have been essentially improved during the last years. The application of such languages has raised the demand for one language that integrates the advantages of functional and logic programming languages. In this talk we discuss the basic operational principles of such integrated functional logic languages. These languages are usually based on narrowing, a combination of the reduction principle of functional languages and the resolution principle of logic languages. Due to the inefficiency of simple narrowing, a lot of narrowing strategies have been proposed. We discuss sophisticated narrowing strategies for different classes of functional logic programs. The main difficulty of combining functional and logic languages is the reasonable combination of the search paradigm of logic languages with the efficient deterministic reduction principle of functional languages. We show that this is possible using recent results in this area. For inductively sequential programs (programs with non-overlapping left-hand sides), needed narrowing is an evaluation strategy which is optimal w.r.t. the length of computed derivations and the number of computed solutions. In the presence of rules with overlapping left-hand sides, the addition of a simplification process between lazy narrowing steps can largely reduce the search space. Using these recent strategies, functional logic programs have a deterministic behavior if ground expressions are evaluated and non-deterministic steps are performed only if logical variables occur at run time. This shows that a sound, complete and efficient combination of search and deterministic reduction is possible in modern functional logic languages.

Declarative Programming

Herbert Kuchen

RWTH Aachen, Lehrstuhl für Informatik II
D-52056 Aachen, Germany
herbert@informatik.rwth-aachen.de

Abstract

The idea of declarative programming is that the user specifies an application on a high level and in such a way that the specification can be executed directly. Thus, a proof showing the correctness of some implementation is not necessary. This approach is only feasable, if the specification language is restricted enough. Mainly three paradigms for declarative programming have been proposed which have this property: logic programming, functional programming, and functional logic programming.

Logic programming is based on Horn clauses and uses unification for parameter passing and (SLD-)resolution as execution mechanism. It offers logic variables and hence partial data structures (like difference lists) as well as a built-in search mechanism. Predicates can be used in such a way that, given (the shapes) of some arguments, corresponding remaining arguments can be computed. Logic programming enables an easy integration of constraint solving and hence domain specific search strategies. Unfortunately, determinism is hard to spot and to exploit in logic programs. Moreover, logic languages are mostly untyped, causing "typing errors" to show up at runtime. The most popular logic language Prolog contains a lot of impure features which spoil the declarative nature of the language, e.g. the cut (an ad-hoc way to express determinism), assert and retract (to store global information and to implement an implication), call (an unsafe and tedious way of simulating higher order predicates), and I/O-predicates (which cause (non-backtrackable) side effects).

Functional programming is based on (recursive) function definitions. It uses pattern matching for parameter passing and (graph) reduction as execution mechanism. It provides higher order functions (and thus user defined control structures), lazy evaluation (enabling infinite data structures), nested function calls, (typically) a polymorphic type system (offering the reuse of code and disabling runtime type errors), and purely declarative I/O. Functional languages lack built-in search and logic variables.

Functional logic languages are based on function definitions and use either residuation or narrowing as operational semantics. Narrowing is a combination of pattern matching for parameter passing and reduction as execution mechanism. Functional logic programming combines the advantages of purely functional and purely logic languages, i.e. higher order functions, lazy evaluation, polymorphic typing, partial data structures, and (mostly) built-in search. The impure features of Prolog are no longer needed. In particular, determinism is expressed by the use of functions (rather than "cut").

Using CHAMs for the design of the distributed constructs in Facile

Lone Leth
(Joint work with Bent Thomsen)

ECRC
Lone.Leth.Thomsen@ecrc.de

Abstract

In this talk we use the chemical abstract machine (CHAM) framework for discussing various semantics for the Facile programming language and for formalising (parts of) its implementations.

The basic idea of the CHAM is that the state of a system is like a chemical solution where molecules float around. These molecules can interact with each other according to reaction rules. Solutions can be heated to break complex molecules into smaller ones, and solutions can be cooled to rebuild heavy molecules from components. Molecules can contain subsolutions enclosed in membranes to deal with abstraction and hierarchical programming. CHAMs all obey a simple set of structural laws. Each particular machine is defined by adding a set of simple rules that specify how to generate new molecules from old ones. The framework has been used to specify CHAMs for CCS and the γ-calculus, for LCCS, for the π-calculus, for LO and for the weak λ-calculus with sharing.

We use the formal CHAM semantics to give informal arguments about implementability of various constructs based on the observation that the more complicated CHAM machinery needed to describe the semantics of constructs the harder it is to give a "real" implementation, since each molecule will correspond to some data structure or even some thread of control. Furthermore, we use the CHAM framework to formalise (parts of) the current implementations of Facile developed at ECRC by specifying some of the internal representations and at the same time abstract from other details.

We also use this formalisation to demonstrate how different implementation strategies could be used and what would be the consequence of such an implementation choice in a distributed implementation.

We take the Facile language as source for discussion, but the results also apply to several other new languages such as CML and Poly/ML. Characteristic for all these languages is that they combine ideas from the λ-calculus and process algebra, such as CCS, to support high level constructs for programming concurrent, parallel and/or distributed systems and mobile agents.

The work may also be seen as a case study in comparing semantic descriptions using structural operational semantics and the chemical abstract machine framework

This is a step towards a complete formalisation of the language implementation.

We do not propose to implement Facile directly by implementing a given CHAM. Although the CHAM framework is intended as a framework for describing concurrent abstract machines it seems to be too abstract for direct implementation. The CHAM seems to be advantageous when describing internal system architecture and behaviour. Compared with an SOS a CHAM semantics is much simpler since a CHAM factors out so-called structural rules. Inference rules become replaced by rewrite rules, e.g. parallel composition will be translated into molecules, and solutions of molecules are naturally associative and commutative. This allows us to

rapidly write down rules for prototype semantics which are shorter and clearer than their SOS counterparts. The CHAM framework has also allowed us to work in an incremental way when moving from one semantics to another. It seems to be difficult to use this approach when designing SOS.

Geometry of Interaction: Semantics and Implementation

Ian Mackie

LIX, Ecole Polytechnique, France
mackie@lix.polytechnique.fr

Abstract

The purpose of this talk is to try to provide some intuitions and basic concepts underlying the Geometry of Interaction and Game Semantics that have recently been introduced to model the *dynamics* of computation.

Traditionally, semantics has been divided into two schools of thought: operational (step by step syntax rewriting) and denotational (modeling answers in some mathematical universe). The Geometry of Interaction and Game semantics are attempts to find a middle ground between these paradigms. The result is a semantics that we can use to reason about step by step computation in a (traditional) mathematical setting. Moreover, the steps of computation are *linear* and *reversible*, i.e. really atomic.

We conclude the talk by showing how to apply this idea to functional programming implementations (the Geometry of Interaction Machine) and hint at some current research directions including: automatic compiler generation, evaluation strategies, parallel computation, communication, logic programming and term rewriting systems.

A general Model for object based Systems

Holger Naundorf

University of Paderborn, Germany
snoo@plato.uni-paderborn.de

Abstract

The object based paradigm will play a key rôle in integration of paradigms, because it is more a coordination than a programming language paradigm – encapsulation hides the concrete language used to implement an object.

To formalize this coordination aspect it is important to use a model that really abstracts from the internals of objects. A suitable means are abstract time systems known from abstract system theory – a generalization of traces. An arbitrary number of objects – even infinite many objects – can form an object based system. The objects communicate with each other and the outside world via a transmission unit – itself an object described by an abstract time system. Since objects communicate by messages only they cannot create new objects; instead the creation of objects is modeled by activating objects (therefore object based systems with infinite many objects are needed).

An object based system itself defines an abstract time system in a canonical way, hence can be viewed as an object and thus this model is hierachical. Unfortunately a so defined object can expose strange behavior like lack of causality. But with relatively weak conditions – the transmission unit needs some time to propagate messages – the defined abstract time system is very well behaved.

Towards Composition Languages

Oscar Nierstrasz

University of Berne
Institut für Informatik (IAM)
Neubrückstrasse 10, CH-3012 Bern, Switzerland
oscar@iam.unibe.ch
WWW: http://iamwww.unibe.ch/∼oscar

Abstract

Present-day Object-Oriented Languages emphasize programming over composition: in general, it is not possible to build applications from an object-oriented class library or framework by simply composing and linking objects instantiated from pre-existing classes. One must always program new classes that use, or are derived from those provided. In this sense, object-oriented technology fails to raise the level of abstraction from programming with core language concepts to composition with domain specific components.

We posit the need for a so-called composition language, which would support both the composition of applications from domain-specific components, as well as the definition of the component frameworks themselves. Composition languages would be used at two distinct levels: the framework level and the application level. At the Framework Level, a composition language would support the specification of: (i) generic software architectures, (ii) standard component interfaces and protocols, (iii) composition mechanisms (or "glue"), (iv) open, generic software components, (v) high-level syntax for compositions. At the Application Level, a composition language would support the specification of: (i) elaboration and instantiation of generic components, (ii) applications as compositions, (iii) explicit, manipulable software architectures, (iv) meta-level reasoning (access to framework level to support dynamic composition).

A composition language would support an integrated object/component model based on a rigorous semantic foundation. Such a foundation - or object calculus - could well be based on a variant of the π calculus. Challenges for such a language include the modeling of the foundational software abstractions (components, active objects, etc.), the development of an appropriate type system with type inference, support for interoperability with existing languages and component libraries, modeling of abstractions for concurrency and distribution, and modeling of reflective capabilities to support evolution of long-lived distributed systems.

The TAO computation model

António Porto

Departamento de Informática
Universidade Nova de Lisboa
2825 Monte da Caparica
Portugal
ap@@fct.unl.pt

Abstract

We present the *TAO* computation model, which aims at providing an abstract view of computation, of general applicability, directly supporting high-level abstractions, and based on a set of orthogonal primitive computational ingredients.

The state of an agent is considered to consist of three components: the task, representing actions to be performed, the database, representing contingent truths, and the rules, defining procedural abstractions. The task specifies a composition (parallel, sequential, choice, synchronous) of elementary actions (queries and commands) or named tasks. These are decomposed into tasks using the rules, in a recursive manner, eventually getting to elementary actions. The operational semantics defines a computational step as a synchronous (atomic) execution of a query and a command. Queries may only be executed when certain truths are entailed by the database, whereas commands always execute and as an effect may (non-monotonically) change the database. Commands are assumed to have a deterministic nature, and we abstractly characterize the restrictions needed for this. This basic model allows for coordination among concurrent tasks through queries and updates on their shared database. We further consider scoped variables in tasks and substitutions, for direct communication among tasks. Finally we introduce systems of named agents with separate rules and databases, with dynamic agent creation and task delegation among agents.

Typechecking record concatenation using constrained types

Didier Rémy

INRIA-Rocquencourt
B.P. 105, F-78153 Le Chesnay Cedex
Didier.Remy@inria.fr

Abstract

We use the framework of contrained types to proposed a type system for record concatenation with type inference. The main application is multiple inheritance in object-oriented languages. Another interest is to show the flexibility of type inference with constrained types.

The talk is composed of three parts. We first review type inference with contrained types. In a second part we consider record concatenation, which raises two problems: the need for exact information about the presence of fields is solved by enriching the structure of types and refining the subtype relation; the encoding of the intersection type of concatenation into contrained types is solved by the addition of a new kind of constraints. In a third part, we compare two different approaches to record types, subtyping vs. row variables, and we show that both mechanisms are orthogonal and can be added together to provide a richer type system.

The Oz Programming Model

Gert Smolka

Programming Systems Lab
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
66123 Saarbrücken, Germany
smolka@dfki.uni-sb.de
http://ps-www.dfki.uni-sb.de:80/~smolka/

Abstract

The Oz Programming Model (OPM) is a concurrent programming model that subsumes functional and object-oriented programming as facets of a general model. This is particularly interesting for concurrent object-oriented programming, for which no comprehensive and formal model existed until now. There is a conservative extension of OPM providing the problem-solving capabilities of constraint logic programming. OPM has been developed together with a concomitant programming language Oz designed for applications that require complex symbolic representations, organization into multiple agents, and soft real-time control. An efficient, robust, and interactive implementation of Oz is freely available.

Reasoning about Programs

Carolyn Talcott

Stanford University
clt@sail.stanford.edu

Abstract

This talk gives a rather personal perspective in which I will consider reasoning about programs exhibiting a spectrum of paradigms/facilities. I will begin with a brief summary of early work in which methods were developed to reason about programs that could be modelled as first order functions, or as state transformers. Then I will trace a lambda thread, following the view of Landin that programming languages are lambda expressions plus operators providing desired computational facilities (lambda – the ultimate integrating mechanism). I will focus on three rather different choices of primitives:

functional – lambda-fun = lambda + if + numbers and pairs

mutable data – lambda-mk = lambda-fun + cell creation, access, update

concurrent – lambda-act = lambda-fun + actor primitives

For each situation, questions to consider are:

What are we reasoning about? What is the mental or semantic model? What sorts of assertions are being made?

What are some methods/reasoning principles?

As one adds facilities to the language what breaks? what persists? what tools continue to work? what new ones have been developed?

I will take an operational approach to reasoning about programs, beginning with a study of operational equivalence. Program calculi provide a particularly robust means of proving equations, as the proofs remain valid when new facilities are added. In particular, the laws of the computational lambda calculus form a kernel theory that is valid in all the cases considered. One might take this as a criteria for an acceptable integration of primitives. However, program calculi are generally too weak to serve as the only tool for proving observational equivalence. We discuss some general approaches to proving equivalence and how they are modified in the different settings. In the case of lambda-act we also consider equivalence based on interaction semantics, which appears to be amenable to modular reasoning about components and their combinations.

Going beyond equational reasoning we discuss a framework for specifying and reasoning about programs based on properties viewed as collections of values (which include closures/procedures) defined by formulae.

Agent Programming Needs Integration of Multiple Paradigms or How to program the worlds largest computer

Bent Thomsen
(Joint work with: Lone Leth, Frederick Knabe and Pierre-Yves Chevalier)

ECRC
Bent.Thomsen@ecrc.de

Abstract

As the Internet swells today past 30 million users and commercial networks such as CompuServe and America OnLine gain in popularity, a broad range of information and services is becoming available on the emerging global information infrastructure (GII). In this extremely diverse and rapidly changing environment, flexibility and adaptability are key characteristics that the new distributed systems for accessing and providing services must have. An emerging technology, "mobile agents", offers significant potential in providing these capabilities to developers of the GII.

In this talk we give a short introduction to mobile agents, discuss their potential and describe a proof-of-concept prototype implementation done at ECRC. We then discuss the role of integrated multiple paradigm programming in the construction of systems based on mobile agents, in particular we discuss the use of the Facile multi-paradigm programming language.

Facile's basis on formal process models for mobility (e.g., the π-calculus and CHOCS) and on functional programming languages has allowed it to inherit features and a basic approach that provide an edge in developing a significant class of mobile applications.

Foundations of Interactive Computing

Peter Wegner

Brown University
pw@cs.brown.edu

Abstract

Computing paradigms are evolving from a focus on algorithms in the 1960s and 1970s to a focus on interaction in the 1990s. Interaction machines, defined by extending Turing machines with input and output actions, are shown to be more expressive than algorithms in capturing the behavior of objects, software systems, and intelligent agents. We develop fundamental concepts and models of interactive computing, show the direct relevance of these models to software technology and artificial intelligence, and show that interaction machines provide a precise framework for empirical computer science. Interaction is shown to be nonalgorithmic even in the absence of concurrency and distribution, while noninteractive concurrency and distribution are shown to be algorithmic. Interaction machines cannot compute more powerful (noncomputable) functions, but express "comptable nonfunctions" that handle nonfunctional properties like time.

Interaction machines cannot be specified by sound and complete logics: they are incomplete in the sense shown by Godel for the integers (the set of all true statements is not formally specifiable). Church's thesis that the intuitive notion of computing corresponds to formal computing by Turing machines is seen to be invalid or at least inapplicable, since interaction machines determine a very natural notion of computing more powerful than Turing machines. The Chomsky hierarchy of machines is extended beyond Turing machines to synchronous and asynchronous interaction machines, but mathematical characterization of machine behavior by sets or formal grammars cannot be similarly extended, showing that machines can specify more powerful forms of behavior than mathematical notations.

Since interaction machines cannot be completely specified by a formal system, they are specified by partial interface specifications that determine desired modes of use but not all possible behaviors. Practical software design techniques such as the object modeling technique OMT, use-case analysis, and Microsoft's Componet Object Model COM are seen to be natural instances of fundamental interactive models rather than ad-hoc kludges that will later be replaced by tidy algorithmic formal models. Multiple interfaces are shown to be a more flexible framework than object-oriented inheritance as a model for interactive systems of software components. The paradigm shift in artificial intelligence away from models based on logic and search towards agent-oriented models is likewise seen to be consistent with fundamental principles of interactive modeling. The Turing test is reinterpreted to include interaction machines with multiple interfaces and we conclude that interaction machines can plausibly be said to think, since they not only have richer behavior than Turing machines but also invalidate Searle's intensional and Penrose's extensional arguments that machines cannot think.

References

[1] R. Amadio and M. Dam. Reasoning about higher-order processes. In *Proc. CAAP 95*, Aarhus, 1995.

[2] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994. Available at http://www.sics.se/

[3] M. Hanus. Combining lazy narrowing and simplification. In *Proc. of the 6th International Symposium on Programming Language Implementation and Logic Programming*, pages 370–384. Springer LNCS 844, 1994.