BRICS

Basic Research in Computer Science

Programming Languages: Design, Analysis, and Semantics

Anders B. Sandholm

BRICS Dissertation Series **DS-00-2**

ISSN 1396-7002 **February 2000**

See back inner page for a list of recent BRICS Dissertation Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone:	**+45 8942 3360**
Telefax:	**+45 8942 3255**
Internet:	**BRICS@brics.dk**

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `DS/00/2/`

Programming Languages: Design, Analysis, and Semantics

Anders Bækgaard Sandholm

Ph.D. Dissertation

Department of Computer Science
University of Aarhus
Denmark

Programming Languages: Design, Analysis, and Semantics

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfillment of the Requirements for the
Ph.D. Degree

by
Anders Bækgaard Sandholm

Abstract

This thesis contains three parts. The first part presents contributions in the fields of domain-specific language design, runtime system design, and static program analysis, the second part presents contributions in the field of control synthesis, and finally the third part presents contributions in the field of denotational semantics.

Domain-specific language design is an emerging trend in software engineering. The idea is to express the analysis of a particular problem domain through the design of a programming language tailored to solving problems within that domain. There are many established examples of domain-specific languages, such as LaTeX, flex, and yacc. The novelty is to use such techniques for solving traditional software engineering task, where one would previously construct some collection of libraries within a general-purpose language. In Part I, we start by presenting the design of a domain-specific language for programming Web services. The various domain-specific aspects of Web service programming, such as, Web server runtime system design, document handling, and concurrency control, are treated.

Having treated many of the Web related topics in some detail, we turn to the particular ideas and central design issues involved in the development of our server-side runtime system. We build a runtime system that not only shares the advantage of traditional CGI-based services, namely that of portability, we also overcome many of the known disadvantages, such as intricate and tedious programming and poor performance.

We then turn to the use of dynamic Web documents, which in our language relies heavily on the use of static program analysis. Static program analysis allows one to offer static compile-time techniques for predicting safe and computable approximations to the set of values or behavior arising dynamically at runtime when executing a program on a computer. Some of the traditional applications of program analysis are avoidance of redundant and superfluous computations. Another aspect is that of program validation, where one provides guarantees that the analyzed code can safely be executed. We present an example of such an analysis as explained in the following. Many interactive Web services use the CGI interface for communication with clients. They will dynamically create HTML documents that are presented to the client who then resumes the interaction by submitting data through incorporated form fields. This protocol is difficult to statically type-check if the dynamic document are

created by arbitrary script code using "printf" statements. Previous proposals have suggested using static document templates which trades flexibility for safety. We propose a notion of typed, higher-order templates that simultaneously achieve flexibility and safety. Our type system is based on a flow analysis of which we prove soundness. We also present an efficient runtime implementation that respects the semantics of only well-typed programs.

Having dealt with dynamic Web documents, we turn to the topic of control synthesis. In the design of safety critical systems it is of great importance that certain guarantees can be made about the system. We describe in Part II of this thesis a method for synthesizing controllers from logical specifications. First, we consider the development of the BDD-based tool Mona for translating logical formulae into automata and give examples of its use in small applications. Then we consider the use of Mona for control synthesis, that is, we turn to the use of Mona as a controller generator. First, in the context of our domain specific language for generating Web services, where concurrency control aspects are treated using our control synthesis technique. Second, in the context of LEGO robots, where we show how controllers for autonomous LEGO robots are generated using our technique and actually implemented on the physical LEGO brick.

Finally, in Part III we consider the problem of sequentiality and full abstraction in denotational semantics. The problem of finding an abstract description of sequential functional computation has been one of the most enduring problems of semantics. The problem dates from a seminal paper of Plotkin, who pointed out that certain elements in Scott models are not defineable. In Plotkin's example, the "parallel or" function cannot be programmed in PCF, a purely functional, sequential, call-by-name language. Moreover, the function causes two terms to be distinct denotationaly even though the terms cannot be distinguished by any program. The problem of modeling sequentiality is enduring because it is robust. For instance, changing the reduction strategy from call-by-name to call-by-value makes no difference. When examples like parallel or do not exist, the denotational model is said to be fully abstract. More precisely, full abstraction requires an operational definition of equivalence (interchangeability in all programs) to match operational equivalence. It has been proved that there is exactly one fully abstract model of PCF. Until recently, all descriptions of this fully abstract model have used operational semantics. New constructions using logical relations have yielded a more abstract understanding of Milner's model.

We consider in Part III how to adapt and extend the logical relations model for PCF to a call-by-value setting. More precisely, we construct a fully abstract model for the call-by-value, purely functional, sequential language FPC.

To my parents
Jytte and Thorkild Sandholm

Acknowledgments

By the time I hand in this thesis, I will have spent eight years studying computer science, mostly at the University of Aarhus. It has been an exciting and truly rewarding experience. I owe thanks to the many people who have made years so wonderful and memorable.

I am deeply grateful to my advisor Michael I. Schwartzbach for his support throughout the years. I admire his capability of combining (at times seemingly useless) theory with all kinds of neat practical applications. I thank him for his support and encouragement. Mogens Nielsen deserves thanks for being my advisor through the first year of my Ph.D.-studies and for enthusiastically teaching concurrency and verification.

I would also like to thank Henrik Reif Andersen, Anindya Banerjee, and Erik Meineche Schmidt for being on my thesis evaluation committee.

I am very grateful to Jon Riecke for giving me the opportunity to visit him in the Summer of 1996 at Bell Labs in Murray Hill, NJ and for making the writing of my first paper an educational and memorable experience. I would like to thank Nils Klarlund for managing the Mona project and for giving me the opportunity to visit him in the Summer of 1997 at AT&T Labs Research in Florham Park, NJ. Also, I thank Bobby Bhattacharjee, a fellow AT&T intern, for introducing Samuel Adams to me, for taking me to Boston, and for being a great friend. I thank Tom Ball for giving me the opportunity to spend the Fall of 1998 at Bell Labs in Naperville, IL. I am grateful to Tom, his family, and the people at Bell Labs for making my stay in Naperville such a great and memorable experience.

Special thanks go to Jesper Gulmann Henriksen for being a good friend. Attending mostly the same classes, working together, and being officemates have, over the years, given me many memorable moments. I also thank Claus Brabrand and Anders Møller for being great friends and colleagues. Working with you has been great. Thanks go to the <`bigwig`> programmers, Steffan Olesen and Mikkel Ricky Christensen, for making many of the <`bigwig`> ideas come alive. Also thanks to Thomas Hune for writing the LEGO paper with me. It was great fun.

I would also like to thank all the people at BRICS and DAIMI who where willing to discuss intriguing problems and everyday life often over a cup of coffee: Gerth Stølting Brodal, Ole Caprani, Olivier Danvy, Thomas Troels Hildebrandt, Thore Husfeldt, Rune Bang Lyngsø, Christian Nørgaard Storm Pedersen, Theis

Rauhe, Erik Meinecke Schmidt, Glynn Winskel, Peter Ørbæk, and many others. I thank Rune for taking me on the Thanksgiving trip in 1998 across most of the Mid-West and Southern USA. At the department, I would also like to thank the always helpful secretaries and the staff keeping the system running, and all the students that I have had the wonderful experience of teaching computer science over the years.

I thank my girlfriend through the years, Ane Helene Thorhauge, for her seemingly endless love and support. Also, my sister Bente and my brother Jakob deserve thanks. I am however, by far, most grateful to my parents, Jytte and Thorkild Sandholm, for their love, encouragement, and support, which gave me all the opportunities. To them I dedicate this thesis.

Anders Bækgaard Sandholm,
Århus, October 1999/October 2000.

Contents

Chapter 1

Introduction

Contents

1.1 Structure

This thesis contains three main parts, each presenting results of my research during my graduate study. The three parts are first briefly described and the publications on which they are based are listed. Afterwards, more detailed presentations of the three parts including introductions to the relevant fields as well as descriptions of our contributions are given.

The three parts present contributions in various areas of computer science that each in a natural way have their own introduction. However, formal requirements ask for an overall thesis introduction. Therefore, as the careful reader will observe, do certain parts of this introductory chapter reappear in part later in the respective introductions to the fields dealt with in this thesis.

Part I

Part I presents work on the design of a domain specific language for programming Web services, including a survey chapter, a description of its runtime system, and the presentation of a static analysis for dynamic document creation.

The chapters in Part I are based on publications as follows:

Chapter 2 [15]: *<bigwig> — a language for developing Web services*, with Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Submitted for publication.

Chapter 3 [16]: *A Runtime System for Interactive Web Services*, with Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. In Computer Networks, 31:1391–1401, 1999. Also in Proceedings of the Eighth International World Wide Web Conference.

Chapter 4 [117]: *A Type System for Dynamic Web Documents*, with Michael I. Schwartzbach. In Proceedings of Principles of Programming Languages, 2000 (POPL'00).

Part II

Part II presents work on control synthesis based on a particular verification tool, Mona. First, an early description of the tool along with simple example applications is presented. Then the general techniques involved in using this tool for control synthesis are described, followed by an application providing concurrency control for our Web service programming language <bigwig>. Finally, Part II is rounded off with the presentation of an application of the control synthesis techniques in the domain of embedded systems, more specifically in the domain of autonomous LEGO robots.

The chapters in Part II are based on publications as follows:

Chapter 5 [44]: *Mona: Monadic Second-Order Logic in Practice*, with J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, and T. Rauhe.

In Tools and Algorithms for the Construction and Analysis of Systems, TACAS '95, LNCS 1019, 1996.

Chapter 6 [115]: *Mona-Based Control Synthesis.* Unpublished manuscript, 1999.

Chapter 7 [116]: *Distributed Safety Controllers for Web Services*, with Michael I. Schwartzbach. In Fundamental Approaches to Software Engineering, FASE'98, Lecture Notes in Computer Science, LNCS 1382, pages 270–284. Springer-Verlag, March/April 1998.

Chapter 8 [50]: *A Case Study on Using Automata in Control Synthesis*, with Thomas Hune. In Fundamental Approaches to Software Engineering, FASE'00, Lecture Notes in Computer Science, LNCS 1783, pages 349–362. Springer-Verlag, March/April 2000.

Part III

Part III presents work on denotational semantics. More specifically we construct a fully abstract model for a purely functional, sequential, call-by-value language. The model is built from partial continuous functions further constrained to be uniform with respect to a class of logical relations.

The chapter in Part III is based on the following publication:

Chapter 9 [111]: *A Relational Account of Call-by-Value Sequentiality*, with Jon G. Riecke. To appear in Information and Computation, 2000. A preliminary version appeared in Proceedings of the Twelfth Annual IEEE Symposium on Logic in Computer Science.

1.2 Part I: <bigwig>

Part I presents contributions in the fields of domain specific language design, runtime system design, and static program analysis.

1.2.1 Domain specific languages

The <bigwig> language described in Chapter 2 is an exercise in domain specific language design, which is an emerging trend in software engineering. The idea is to express the analysis of a particular problem domain through the design of a programming language tailored to solving problems within that domain. There are many established examples of domain specific languages, such as LaTeX, flex, and yacc. The novelty is to use such techniques for solving traditional software engineering task, where one would previously construct some collection of (class) libraries within a general-purpose language.

There are many advantages to a domain specific language. A superficial one is nicer syntax, in that one is not restricted to a fixed notation for invoking library routines. A much deeper advantage is that the compiler may exploit domain specific knowledge for analysis and optimizations. A concrete example of this particular advantage is the contribution presented in Chapter 4. The disadvantages of domain specific languages are equally clear. First of all, they are presumably more expensive to develop, even though that may gradually change. Secondly, they introduce new syntax to learn and remember.

Thus, domain specific languages, that is, languages that are designed with a particular problem domain in mind, are more powerful and useful than traditional general purpose languages for solving problems in these specific problem domains. We argue, that the use of domain specific languages can increase the speed of software development in providing domain specific syntax, program analysis, and efficient runtime implementation.

The work presented in Part I in this dissertation supports this thesis through the design and implementation of a number of little languages for programming interactive Web services. These languages are all part of <bigwig> — a high-level programming language for developing interactive Web services.

1.2.2 Web service languages

There are numerous languages and systems for programming Web services. Some of the dominating ones are Active Server Pages (ASP) [34] and the non-commercial PHP [10]. Like many others, these systems are page-centered. That is, the development of a Web service is centered around the concept of a Web page. This view seems to be the optimal one when building services that do not involve too much complexity.

In the Mawl project [67] the view was changed from being page-centered to being centered around the service logic. That is, when building a Web service one focuses on what the service is supposed to do and not so much on exactly how its interface to the client should be designed. In a service-logic-based Web programming language the documents are thus just values in a computation. <bigwig> is an intellectual descendant of the Mawl project but is a completely new design and implementation with vastly expanded ambitions. Another interesting Mawl-inspired Web service programming language is Guide [73]. However, Guide has certain shortcomings, such as, no static checking of dynamic documents, which is a central notion in both Mawl and <bigwig>.

The <bigwig> language

The chapters in Part I and Chapter 7 in Part II are centered around the design and development of <bigwig>.

<bigwig> is a high-level programming language for developing interactive Web services. Complete specifications are compiled into a conglomerate of lower-level technologies such as CGI-scripts, HTML, JavaScript, Java applets, and plug-ins running on top the runtime system presented in Chapter 3. The <bigwig> language is really a collection of tiny domain specific languages focusing on different aspects of interactive Web services. To minimize the syntactic

burdens, these contributing languages are held together by a C-like skeleton language. Thus, <bigwig> has the look and feel of C-programs with special data- and control-structures. Chapter 2 gives a survey of the <bigwig> project.

HTML documents are first-class values that may be computed and stored in variables. A document may contain named gaps that are placeholders for either HTML fragments or attributes in tags. Such gaps may at runtime be plugged with concrete values. Since those values may themselves contain further gaps, this is a highly dynamic mechanism for building documents. A flow-sensitive type checker ensures that documents are used in a consistent manner. This concept of dynamic documents is treated further both later in this introduction as well as in Chapter 4.

A <bigwig> service executes a dynamically varying number of threads. To provide a means of controlling the concurrent behavior, a thread may synchronize with a central controller that enforces the global behavior to conform to a regular language accepted by a finite-state automaton. That is, the 'control logic' in <bigwig> consists of finite-state automata. The controlling automaton is not given directly, but is computed (by the MONA [60, 88] system) from a collection of individual concurrency constraints phrased in first-order logic. Extensions with counters and negated alphabet symbols add expressiveness beyond regular languages. This notion concurrency control in <bigwig> is also treated later in the introduction and further in Part II on control synthesis in Chapter 7.

1.2.3 Static program analysis

Static program analysis allows one to offer static compile-time techniques for predicting safe and computable approximations to the set of values or behavior arising dynamically at runtime when executing a program on a computer. Some of the traditional applications of program analysis are avoidance of redundant computations, such as reusing available results, and superfluous computations, such as result known at compile-time. Another aspect is that of program *validation*, where one provides guarantees that the analyzed code can safely be executed. Our contribution in Chapter 4 is in some sense an example of a static analysis being performed to validate Web service programs with respect to Web document computations performed by these programs.

Traditionally, one distinguishes between the following four main approaches to program analysis: data flow analysis, control flow analysis, abstract interpretation, and type and effect systems of which our contribution falls in the category of traditional data flow analyses. Though often considered different, the four approaches do have a lot in common as described in [91].

One of the commonalities is the notion of *approximation* which is almost always unavoidable in static program analysis. One cannot statically produce precise answers to the questions at hand. The requirement of precise information would simply lead to undecidability. However, what we can do, and what has traditionally been done in program analysis, is to provide a safe answer. That is, we expect the program analysis to produce a possibly larger set of possibilities than what will ever happen during execution. This notion of safe approximation is illustrated in Figure 1.1. Clearly the challenge is to have the program analysis

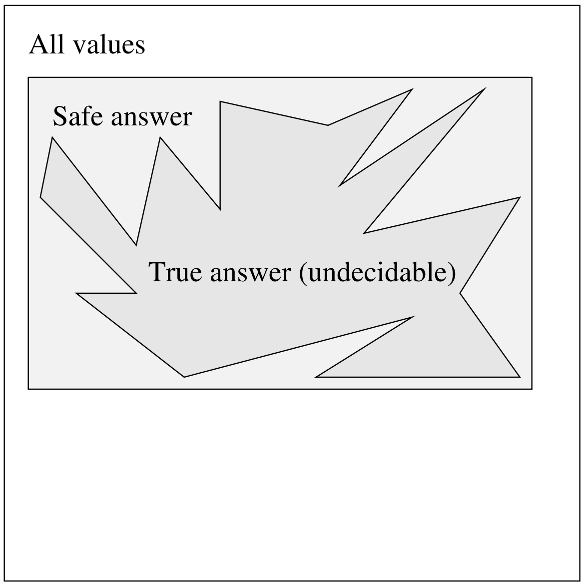


Figure 1.1: Safe approximation.

give safe answers that are as close to the precise answers as possible within the given limitations, such as limited memory or time.

Data flow analysis

The contribution in Chapter 4 is an example of a data flow analysis. We shall thus concentrate on briefly introducing the notion of data flow analysis and refer to [91] for a more thorough treatment of this and the other approaches to program analysis. Data flow analysis is the traditional form of program analysis which is described in many textbooks on compiler writing. Typical data flow analyses include available expressions, reaching definitions, very busy expressions and live variables. When working with data flow analysis, the program is roughly considered a graph, called the control flow graph, with the nodes being the basic block of the program and the edges the paths which the control might take during execution.

The way an analysis is performed is by extracting a number of constraints from the program text and then have a separate algorithm find the solution to the constraint set at hand. (Data flow analyses are also quite often given by a set of equations instead of constraints, but the two approaches are more or less equivalent.)

As in Chapter 4 one would normally have an operational semantics to the language in which the programs that are to be analyzed are written. The extraction of constraints can then conveniently be defined according to the monotone framework of [91], which will automatically provide a terminating algorithm for calculating the "right" solution, which in this case is the smallest solution with respect to the partial order, which is defined via the use of the monotone framework.

Typed dynamic Web documents

Many interactive Web services use the CGI interface for communication with clients. They dynamically create HTML documents that are presented to the client who then resumes the interaction by submitting data through incorporated form fields. This protocol is difficult to statically type check if the dynamic document are created by arbitrary script code using `printf`-like statements. Previous proposals [67, 73] have suggested using static document templates which trades flexibility for safety. We propose a notion of typed, higher-order templates that simultaneously achieve flexibility and safety. Our type system is based on a flow analysis of which we prove soundness. We present an efficient runtime implementation that respects the semantics of only well-typed programs. This work is fully implemented as part of the <`bigwig`> system.

Furthermore, the static analysis presented is rather domain specific. The parts of a program that actually perform analysis relevant computations are few and often spread around in the code. We show in Chapter 4 that in this case, it is worthwhile reducing the control graph before performing the data flow analysis. In fact, we get an execution time improvement of factor two.

1.3 Part II: Mona and Control Synthesis

Part II presents contributions in the field of control synthesis.

1.3.1 Tools for verification

Formal verification of reactive systems is based on the choice of a formal semantical description of the systems and techniques for proving that these descriptions have certain properties.

A predominant verification technique is known as *model checking*. In model checking one has a system or a model, M, which is to be checked against a specification ϕ typically written in some kind of logic, say linear time temporal logic [101]. If the model satisfies the formula (normally written as $M \models \phi$) then, when using M, one is guaranteed that it obeys the behavior described by ϕ. If, on the other hand, $M \not\models \phi$, then it is considered good practice for the model checker to provide a counter example, that is, the trace of a possible execution in M which does not satisfy ϕ.

There are different approaches to and thus various tools for doing model checking. For instance, the method may be based on tableaux [70, 125], state labeling and graph algorithms [25, 26, 86, 123], automata theory [65, 130], partial order semantics [39, 129], BDDs [18, 20], or partial model checking [6].

Mona

Our contribution, the design and development of the first BDD-based version of Mona, belongs to the practical end of the field of formal verification. Mona is in essence a tool for deciding the weak second-order theory of one or two successors. A (non-BDD-based) tool for monadic second-order logic on finite, binary trees has been developed at the University of Kiel [124]. Apparently, this tool has only been used for very simple examples.

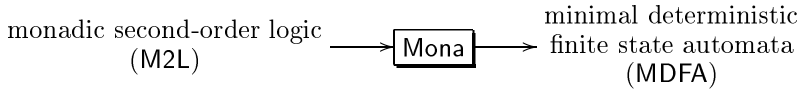


Figure 1.2: The functionality of Mona.

Through the Mona tool, our work provides a logic, automata, and BDD based mechanism for doing model checking as well as other kinds of verification [12, 44, 50, 54, 61, 63, 115, 116]. Furthermore, there are a number of papers (of which Chapter 5 is based on [44]) describing the concepts underlying the design of Mona and Fido [13, 44, 60, 62, 64]. Fido is a high-level language notation built on top of monadic second-order logic, which allows for easy formulation of complex properties. In its latest version, Mona itself has support for parts of this high-level notation, such as, recursive tree structures.

1.3.2 Control synthesis

In the design of safety critical systems it is of great importance that certain guarantees can be made about the system. Traditionally, model checking has served as a successful method for establishing such properties.

We describe in Chapter 6 a method for synthesizing controllers from logical specifications. That is, when implementing a system one can concentrate on the functionality of the system. Afterwards, appropriate safety requirements can be formulated separately and then compiled into a control agent that when run in conjunction with the system will guarantee that the stated requirements are met.

The control synthesis problem was first formulated by Church as a realization problem [24] and later solved by Büchi and Landweber using two-player games [22]. The use of automata for the control of reactive systems is not new. Ramadge and Wonham give a survey of traditional control theory in [103]. The use of Mona, that is, the use of logic in combination with automata has several advantages over traditional automata-based approaches:

- Requirements written in the specification language are *automatically* compiled into a control agent by use of the Mona tool.
- The specification language, based on monadic second-order logic on strings (M2L-Str), is *succinct* and *expressive* allowing complicated requirements to be formulated easily and compact.
- Specifications are easy to *maintain*, that is it is easier to add, remove, and modify parts of a logic-based specification.
- The generated control agent *efficiently* implements the safety requirements by means of the developed techniques for modular synthesis and decentralized control.

Monadic second-order logic on strings (M2L-Str) is a sub-logic expressible in the weak second-order theory of one or two successors, which is the input language of the new version of Mona. We base the design of the control specification language on M2L-Str and show how to use Mona as a powerful tool for doing computations in the logic and more specifically for doing control synthesis.

The use of formal methods in system design is gradually becoming more and more common. One of the major challenges in this field is to manage the large state spaces that naturally occur when dealing with independent requirements and many parallel processes. By focusing on the control synthesis problem rather than the traditional model checking and verification problem we show how to use symbolic representation techniques in a more extensive way than what is normally possible in the world of model checking and verification.

Controllers for Web services

The Web programming environment, with rapidly changing code, fast machines, and slow networks seems an ideal niche for this radical approach of synthesizing finite-state controllers from high-level specifications without suffering an unacceptable performance loss compared to hand-written code. The work presented in Chapter 7 provides the foundations for these ideas. For a diagram, see Figure 1.3

Controllers for embedded systems

In Chapter 8 we study a control program for a moving crane. For this example we have implemented the control synthesis method in the LEGO® RCX™ system [72]. (In the following we shall just be writing LEGO, RCX, and Mindstorms for LEGO®, RCX™, and Mindstorms™.)

The described method for *partial* synthesis differs from traditional synthesis as described in e.g. [76], in that the method is partly based on an existing control program. The aim of the partial synthesis is to restrict the behavior of the existing control program. The allowed executions are described using formulas over a set of I/O events. These events define the interface between the existing control program and the specification.

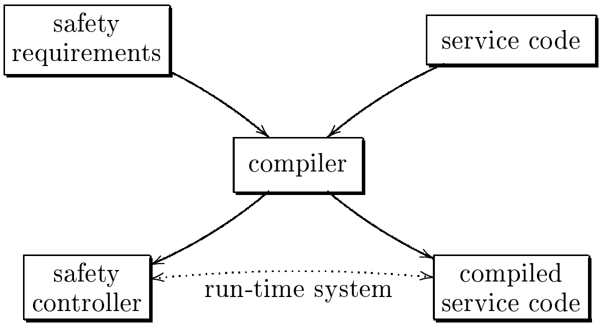


Figure 1.3: Integrated control synthesis.

We argue that using the LEGO RCX system is interesting for at least two reasons. First of all the environment of the RCX system and especially the programming language is quite restricted, so it is not obvious that it is feasible to implement the method at all. We show that it is indeed feasible. Secondly, with the LEGO RCX system one can build actual physical systems for testing the control programs.

1.4 Part III: Full Abstraction for FPC

Part III presents contributions in the field of denotational semantics.

1.4.1 Programming language semantics

When defining, implementing, or seriously using a programming language, one needs a complete and accurate understanding of the *semantics* of every construct of that language. However, what one will find in reference manuals and language standards is that, being based primarily on implementation techniques and intuition, the descriptions are quite often rather inadequate. Thus, we need to be able to give formal semantics to programming languages.

Traditionally, one distinguishes between three kinds of formal semantics, namely operational, axiomatic, and denotational semantics of which our contributions fall in the last category.

In *operational semantics* the meaning of a construct is roughly specified by the computation it induces when it is executed on a machine. That is, the meanings of programs can be described in terms of sequences of computational steps on an abstract or idealized computer. An early approach is [68]. Properties of and relations between operational semantics are discussed in [105] and the canonical reference for structural operational semantics is Plotkin's technical report from 1981 [99]. Semantic descriptions based on operational semantics are often obscured by representation detail. That is, they are often coupled too tightly with the syntax and implementation of the language.

Using a formal system for reasoning about program meanings as an implicit description of the semantics of the language is another more abstract approach to semantics, which has come to be known as *axiomatic semantics* [36, 45]. It is important that the system in which one is reasoning has the property of being *sound*. That is, if a formula can be derived in the formal system, it must also be true in the intended interpretation.

When turning to *denotational semantics*, the semantic interpretation of a formal language is essentially a valuation function mapping every well-formed phrase into its intended meaning in an appropriate domain. These domains of mathematical entities can be simple domains, such as numbers and truth values, or more complex ones, such as functions. Thus, in denotational semantics only the effect of executing a construct is of interest, not how it is obtained. A central concept in denotational semantics is that of compositionality: the valuation functions should be defined in a compositional manner, that is, the meaning of a composite construct should be expressed as a function of the meanings of its immediate sub-constructs. The approach of denotational semantics was pioneered by Strachey and Scott [120] who supplied the mathematical foundations. The influential paper by Scott [118, 119] started what we today know as (classical) domain theory.

There are several books on semantics of programming languages that treat operational, axiomatic, and denotational semantics some of which are [41, 92, 133].

1.4.2 Sequentiality and full abstraction

The problem of finding an abstract description of sequential functional computation has been one of the most enduring problems of semantics. The problem dates from a seminal paper of Plotkin [98], who pointed out that certain elements in Scott models are not definable. In Plotkin's example, the function

$$por(x,y) = \begin{cases} \mathit{true} & \text{if } x \text{ or } y = \mathit{true} \\ \mathit{false} & \text{if } x \text{ and } y = \mathit{false} \\ \perp & \text{otherwise} \end{cases}$$

where $\perp$ denotes divergence, cannot be programmed in the language PCF, a purely functional, sequential, call-by-name language with booleans and numbers as base types. The problem is called the "sequentiality problem" because, intuitively, the only way to program *por* involves evaluating boolean expressions in parallel. Of course, there are other elements in the Scott model of PCF that cannot be programmed: the domains have uncountably many elements. Nevertheless, the *por* function causes problems for reasoning about programs: it causes two terms in the language PCF to be distinct denotationally even though the terms cannot be distinguished by any program.

The problem of modeling sequentiality is enduring because it is robust. For instance, changing the reduction strategy of PCF from call-by-name to call-by-value makes no difference: versions of the *por* function reappear in the standard

Scott model (albeit at higher type). Even for languages that lack an explicit base type, such as, the polymorphic λ-calculus with recursion, the known Scott models contain parallel elements that cause a failure of full abstraction.

When examples like *por* do not exist, the denotational model is said to be **fully abstract**. More precisely, full abstraction requires an operational definition of equivalence (interchangeability in all programs) to match operational equivalence. Shortly after Plotkin's paper, Milner proved that there was exactly *one* fully abstract model of PCF meeting certain conditions [85]. Until recently, all descriptions of this fully abstract model have used operational semantics (see, for instance, [89, 126]). New constructions using logical relations [94, 121] have yielded a more abstract understanding of Milner's model. Game semantics [4, 52, 90] has also been used to give other semantic constructions of fully abstract models of PCF, even though it is still open whether these models are isomorphic to Milner's model.

The FPC language

FPC is a call-by-value, purely functional, sequential language. It includes a base type with one convergent value, strict products, strong (categorical) sums, functions, and recursive types. Full abstraction for FPC is interesting for at least two reasons. First, FPC can be regarded as the purely functional, non-polymorphic sublanguage of Standard ML [84]: recursive types and sums are the basis of `datatype` declarations, and both Standard ML and FPC are call-by-value. By studying FPC, we learn more about programming languages like Standard ML. Second, FPC can serve as an expressive metalanguage for denotational semantics [41, 100]. FPC, for instance, has enough expressive power to encode a call-by-value version of PCF (the base type of numbers can be encoded via a recursive type). Given a fully abstract translation [109] from a language into FPC, the model of FPC yields a fully abstract model of the language.

There is another, purely technical reason to be interested in FPC: since it contains only one trivial base type, we can learn more about the structure of Sieber's logical relations by studying FPC. Sieber's model of PCF, and the fully abstract model of PCF using Kripke relations [94], begin from a set of n-ary relations (for all n) at a **flat** base type—that is, where the convergent elements of the domain are unordered. PCF thus builds in a limited form of sum type, available only at the base type. When we decompose that base type into a sum of the primitive base type, we may learn more about the nature of sequential computation.

Our contribution

The work presented in Part III adapts and extends the logical-relations model for PCF to a call-by-value setting. More precisely, it constructs a fully abstract model for the call-by-value, purely functional, sequential language FPC. At the time of publication, it was the first such model.

Part I

<bigwig>

Chapter 2

A Survey on <bigwig>

Contents

In this chapter we give a description of "<`bigwig`>: a language for developing interactive Web services". The presentation is in preparation and a submission for publication is planned for the Fall of 1999. It is joint work with Claus Brabrand, Anders Møller, and Michael I. Schwartzbach.

<`bigwig`> is a high-level programming language and a compiler for developing interactive Web services. The overall goal of the language design is to remove many of the obstacles that face current developers of Web services in order to lower cost while increasing functionality and reliability. The <`bigwig`> compiler translates programs into a conglomerate of lower-level standard technologies such as CGI-scripts, HTML, JavaScript, and htaccess. This chapter describes the major facets of the <`bigwig`> language design and the techniques used in their implementation, and compares the design with alternative Web service technologies.

2.1 Introduction

<`bigwig`> is a high-level programming language for developing interactive Web services. Complete specifications are compiled into a conglomerate of lower-level standard technologies such as CGI-scripts, HTML, and JavaScript.

The aim of <`bigwig`> is to allow simple and inexpensive construction of advanced services. We are only concerned with specifying the behavior of the server; the graphical layout of the pages is left to other tools.

<`bigwig`> is committed to run on the lowest common denominator of the Web, that is, no special software is required for the browser or the server. Consequently, our services are CGI-based, use plain HTML for communication, and only exploit the subset of JavaScript that is common to the standard browsers.

The <`bigwig`> language is really a collection of individual domain-specific languages focusing on different aspects of interactive Web services. To minimize the syntactic burdens, these contributing languages are held together by a C-like skeleton language. Thus, <`bigwig`> has the look and feel of C-programs with special data- and control-structures.

The main aspects of the language are:

- the underlying *runtime model*, which in a sense constitutes the conceptual foundation of <`bigwig`>;
- the *session* concept, which allows sequential sessions with multiple interactions with the browser to be specified in ordinary programming notation;

- the safety-constraint specification language, which allows complex *concurrency control* requirements to be expressed;
- the *typed dynamic document* language, which provides simple and safe dynamic construction of HTML documents.

Other useful aspect involve databases, cryptologic security, extension of the HTML GUI model, applet integration, interaction with external services, management of services, monitoring of processes and data.

<`bigwig`> is designed to be used by average programmers rather than casual users. Thus, it requires programming skills to be a useful tool. In return, it allows the production of professional services without requiring detailed knowledge of low-level Web technology.

The language has superficially the look of C, but offers a strong type system and automatic garbage collection. It supports a sophisticated macro mechanism that allows the syntax to be extended quite easily. With this feature in mind, only primitives have been allowed in the core language—everything that could be reduced to something more basic using macros has been omitted.

2.1.1 The `WebBoard` example

A very typical kind of interactive Web service is the "Web board"[1]. A Web board is a Web-based forum for discussion where clients can post, read, and reply to messages. `WebBoard` is an example of such a service developed using <`bigwig`>. The center of a Web board service is the "message list", which typically is a hierarchical list showing the titles, authors, etc. of the available messages and their relations. Figure 2.1 shows an example of a message list from `WebBoard`.

The purpose of the `WebBoard` example is to show how easily a small interactive Web service can be generated using <`bigwig`>. The example has deliberately been kept small and simple, but—as we will argue for in the remainder of this chapter—it is relatively easy to extend the `WebBoard` making it a complex, professional quality Web service.

The following sections each describe an aspect of the <`bigwig`> language using the `WebBoard` example as illustration. In the appendix, the complete <`bigwig`> source code for a restricted version of the `WebBoard` is shown. The full version—along with many other example services—can be found on the <`bigwig`> home page [33].

2.2 Runtime System

There exists a variety of technologies for construction of interactive Web services. These can be divided into three fundamentally different approaches:

[1]See for instance `http://webboard.ora.com/`, `http://thinkofit.com/webconf/`, or `http://worldwidemart.com/scripts/wwwboard.shtml`.

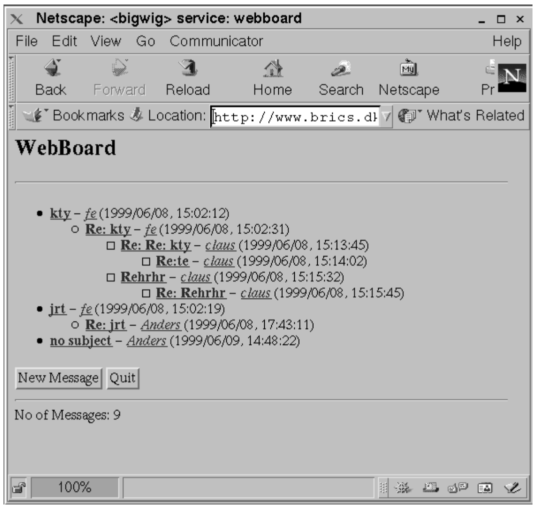


Figure 2.1: Screen dump from the `WebBoard`.

the script-centered — the traditional approach, typically based on CGI and Perl [40];

the page-centered — where the most well known languages are ASP (Active Server Pages) [34][2] and PHP (Personal Home Page) [10][3]; and

the session-centered — which is used in Mawl [9, 67], Guide [73], and in <`bigwig`>.

In this section we will describe the approaches in greater detail, exhibit some deficiencies of the script- and page-centered ones, and motivate the need for a session-centered approach. Also, some problems regarding implementation of the desired approach are addressed.

2.2.1 The session model

Traditional interactive Web services are composed of a number of loosely connected scripts. A script is started upon request from a client through the HTTP/CGI protocol [40]. It reads the CGI arguments, accesses a common database, dynamically generates a reply to the client, and then terminates. These scripts form a whole service by having each script insert links to other scripts in the reply page.

Although presumably still being the most widely used platform for writing interactive Web services, the scripting language Perl combined with a standard

[2] See also `http://www.aspsite.com/websites/`

[3] See also `http://www.php.net/`

CGI server has some significant limitations. If used naively, each client interaction causes a new process to be created on the server. Obviously, this has a serious impact on scalability. As the number of clients increases, many almost identical processes will be running at the same time and, furthermore, the overhead of starting new processes might become intolerable. Also, the programmer must manually take care of storing and restoring relevant state information every time a script starts or ends. These problems can to some degree be remedied using FastCGI[4] which allows script processes to run through whole sequences of interactions. A much larger problem is that the approach is inherently low-level and requires detailed knowledge of the underlying protocols—even when libraries such as `cgi-lib` are used. This well-known fact makes it difficult to write and maintain complex services this way.

In the page-centered approaches, such as ASP or PHP, the service code is embedded within the Web pages. When a client requests a page, a specialized Web server interprets the embedded code and replaces it with HTML code resulting from the interpretation. That is, instead of designating a script to build the whole page to be shown, the client designates a page which may contain pieces of embedded code that builds the dynamic parts of the page.

The basic intention of the page-centered approach is that it should be easier to add dynamics to static Web pages, and—compared to the Perl-like solutions—increase the level of abstraction and improve on the scalability issue. However, there still are some fundamental problems:

- Both the script-centered and the page-centered approaches are based on interpreted scripting languages (though compilers do exist), meaning that there typically are no static checks enforced by, for instance, strong type systems. All errors occur at runtime instead of being caught during development. Naturally, it would both reduce development time and improve stability of the service if errors were caught as early as possible.

- Neither Perl, ASP or PHP can be classified as domain-specific languages (although they do have Web-domain-specific object models or function libraries). This means that—even if there were static checks—these could never check consistency between the various parts of service code and the generated Web pages.

- The major problem with both the script-centered and the page-centered approach is however the following: It can be very difficult for the programmer to maintain a coherent view of the control-flow of the service and of the structure of the produced Web pages. Both control code and HTML code are scattered around in pieces. In particular, since it can be highly unclear how the various interactions are connected, it is difficult to get a clear picture of the overall flow of the service.

The page-centered approach has been claimed to be superior to the script-centered one, but experiences show that for services of just medium complexity,

[4]See `http://www.fastcgi.com/`

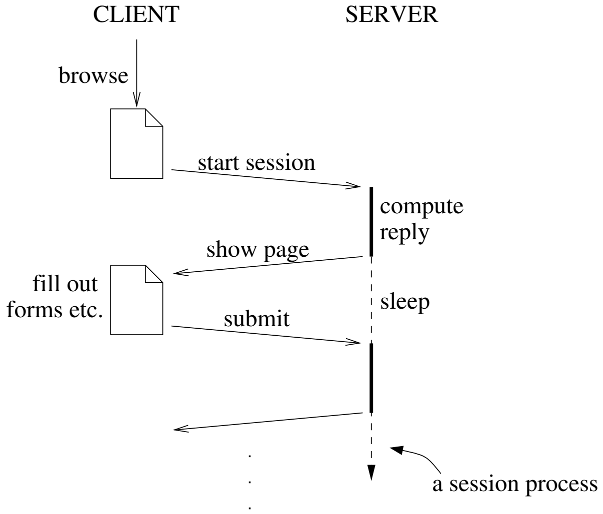


Figure 2.2: An interactive Web session.

the structure of the whole service source tends to converge towards that obtained using the script-centered approach. Often an ASP or PHP page consists of only one large chunk of session code which generates most of the page.

One might conclude that seen from a programming point of view page-centered languages are not much of an improvement, but they do improve on scalability and perhaps make it easier to get started on writing simple Web services.

To address the problems mentioned above, a language with the following characteristics is needed: It must be

- compilation-based—such that static checks can be performed;
- domain-specific—such that consistency of Web-specific concepts can be checked; and importantly,
- allow a structured separation of service code and HTML code.

The <bigwig> language is designed with these criteria in mind. Inspired by the Mawl project, <bigwig> is based upon the concept of a *session*. Instead of viewing the use of a Web service as clients browsing through a set of more or less independent pages, we want each client to be guided through a session of interactions controlled by a process on the server. The session concept is illustrated in Figure 2.2.

Using the session-centered approach, the session structure is directly reflected in the service code. Since a whole service in this way can be written as a single program instead of being spread around a number of Web pages or scripts, the service control flow can potentially become more clear—both to the developer who can get a better understanding of the service code, and to the compiler which can perform a better analysis to guarantee consistency of server code and HTML code. To support the latter, <bigwig> contains a domain-specific sublanguage for dealing with HTML fragments, which is the topic of Section 2.5.

By allowing a separation between service code and HTML code another benefit is obtained: the programmer need not know about Web page design, and the Web page designer need not know about programming.

Although the main flow of interactive Web services usually can be captured by the session model, it can for some parts of some services be rather cumbersome to use—both to the programmer and to the client. In Section 2.9.1 two extensions to the session model are introduced. One of them permits pages to be updated automatically without client interaction, and the other makes it possible to show a whole set of connected pages to be browsed by the client without server interaction.

2.2.2 Building on top of CGI

When raising the level of abstraction from scripts or Web pages to sessions, a number of technical issues need to be addressed in order to implement the session model using standard technology. By standard technology, we mean the HTTP/CGI protocol since this is the only standard supported by all common browsers and Web servers.

- A well-known problem is that the HTTP/CGI protocol is stateless in the sense that the CGI server does not maintain any information about session identity. Seen from the CGI server, there is no connection between the interactions belonging to a certain session. This means that the session state and identity needs to be modeled somehow.

- Since the sessions may access a common database, it does in general not make sense for a client to "step back in time" to a previous page. In other words, the history buffer in the client's browser gets filled up with URLs of obsolete pages. This might confuse the client, who might very well fill in and submit forms which no longer make sense and which the service does not expect to receive. If the service programmer wants the client to be able to step back to a previous page, it is fair to say that it should be programmed explicitly into the control flow.

- The client cannot use browser bookmarks to interrupt a session and then come back later. Adding a bookmark for a page generated directly by a CGI script will store the URL pointing to the script generating the page—not to the page itself. This has the effect that selecting such a bookmark will cause a previous form to be resubmitted, which in general makes the bookmarking functionality rather useless or in the worst case even hazardous.

The first of these problems is only relevant if using a standard HTTP/CGI server (and not for instance FastCGI). The other problems are not inherent to CGI but are also present when using languages that do not intrinsically support the session model but only through library functions (such as the ASP `Session` object).

In <`bigwig`>, each client is assigned a stand-alone process running residently on the server, according to the session model. The process communicates with the client using a standard HTTP/CGI server but, to overcome the lack of state of the CGI-protocol, a dedicated runtime system is used. Since the CGI protocol requires a new process for each single client interaction, a tiny transient process called a "connector" is used to connect the CGI server to the resident session process. The effect is reminiscent of FastCGI, which we choose not to use because our connector also will take care of other things as mentioned in the following.

Each session is associated a unique file containing the most recent page shown to the client. Instead of sending the reply for an interaction directly to the client through the CGI server, it is written to this file, which is visible from the client using a normal HTML URL. Since the URL for the reply page is fixed for the duration of the session, it can function as a "session identity". This solves both the problem with the history buffer and with the bookmarks. Pressing "back" will now bring the client back to the Web page where he started the session, which seems like a natural effect, and sessions can be temporarily suspended and subsequently continued using bookmarks.

Basically, using a URL as session identity fits better into the fundamentals of the Web than, for instance, storing session IDs in browser cookies as in ASP or in hidden input fields as seen in other approaches.

The combination of the connector and the reply-file method is extended with a "timeout" feature. If a reply to the client is not ready within a certain amount of time, a temporary reply explaining the reason for the delay is written to the file and shown to the client by the connector. This page will automatically reload itself such that the real response page will be shown to the client when it is ready. Without the explanation of such delays, the client may become impatient and lose interest or assume that the server or the connection is down.

The whole runtime system as illustrated in Figure 2.3 is implemented in <`bigwig`> using standard HTTP/CGI functionality. The "controller" component in the figure is part of the concurrency control feature described in Section 2.3. More details of the system can be found in [16].

One might argue that the runtime system as explained here and currently implemented does not scale to real-world scenarios. This can be remedied by combining the HTTP/CGI server and the connector, and using light-weight threads instead of a process per session. The resulting system is then close to being a dedicated Web server. Some overhead is caused by the indirect reply pages and the repetitive reloading of temporary replies. This can be argued to be negligible compared to the main network and server load, since it contributes relatively few and small pieces of information. The temporary reply reloading could of course be more efficiently implemented with real server-push technology,

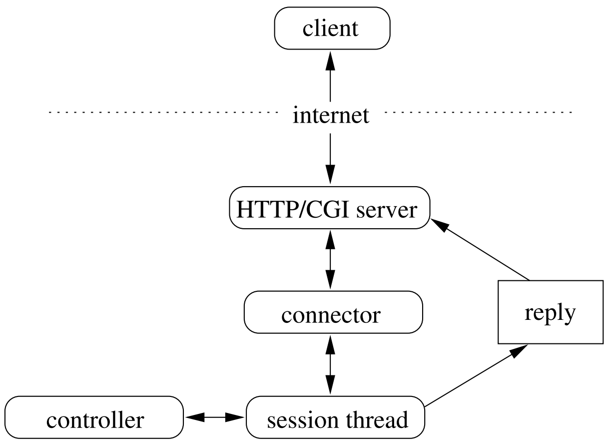


Figure 2.3: The runtime system. For each interaction there is one connector, and for each session there is one session process and one reply file.

but that would sacrifice the idea of basing the runtime system on standard CGI.

2.2.3 Benefits for the `WebBoard`

In the `WebBoard`, clients are required to log in prior to accessing the message list. Once logged in, they can change between viewing the message list, reading a particular message, writing a new message, and setting various preferences. This suggests the structure of the simple `WebBoard` <`bigwig`> source code shown in Figure 2.4.

A `service` can contain a number of `session` kinds. Here we define one session kind named `WebBoard`.

The imperative code inside sessions has a C or Java-like style, augmented with Web-domain-specific constructs.

Of course a small example like this cannot illustrate the whole power of the session-centered approach. However, it should be clear that the flow of the session as informally described above is directly reflected by the structure of the <`bigwig`> source code skeleton, making it simple to comprehend and to extend.

Once a session has been initiated it is protected from naive hijacking attacks using a low-level security scheme inherent to the underlying runtime system. In this primitive scheme, the client sends a “key string” (using a hidden form field) which is checked by the connector. This prevents “accidental” attacks but is of course highly vulnerable to eavesdropping. For services requiring “real” security <`bigwig`> also supports full cryptographic encryption and authentification as explained in Section 2.7.

Due to the reply page technique, `WebBoard` sessions can be bookmarked, and the client will never be shown obsolete pages caused by the browser history

```
service {
   session WebBoard() {
      show and validate login
      while (!leave) {
         switch (current) {
         case VIEW:
            show hierarchical message list
         case READ:
            show selected message
         case WRITE:
            show message composer
         case PREFERENCES:
            show preferences
         }
      }
      show goodbye page
   }
}
```

Figure 2.4: `WebBoard` source code structure

buffer. Though not crucial to this particular service, the feature explaining delays can be useful. If, for instance, the validation of login information is known occasionally to take a while, the client can be made aware of this.

2.3 Concurrency Control

With a number of session processes running concurrently, each interacting with a client and perhaps accessing shared data, there is an obvious need for concurrency control. Large Web services quickly require quite complicated control which is hard to implement and maintain if using semaphores or other low-level primitives.

The <bigwig> approach is to use a high-level declarative logic-based language for specification of behavioral constraints, thereby increasing expressibility and flexibility significantly. The compiler translates these constraints into an operational specification, which at runtime is used to enforce a conforming global service behavior by a centralized controller component running on the server (shown in Figure 2.3). Centralized components in a distributed system should generally be avoided, but since the typical Web scenario contains fast machines and slow networks, the controller will rarely be a performance bottleneck—at least for services of the size <bigwig> primarily aims for. Nevertheless, it is possible in certain circumstances to distribute the controller [116].

2.3.1 Constraining execution

A typical <`bigwig`> synchronization constraint would informally sound like: "don't allow a session to do A if B has occurred and C hasn't occurred since that—unless D at some point has occurred, or ...", where A, B, C, and D are some kind of events. An event in this sense is triggered by a session passing a checkpoint, called a `wait` statement, which has the following form (resembling that of a C `switch` statement):

```
wait {
   case A: ...
   case B: ...
   ...
   timeout t: ...
}
```

Executing such a `wait` statement causes the session to wait for the controller to grant access for one of the events A, B, etc. to occur. When permission has been granted the event is considered "occurred" and execution of the session is resumed at the corresponding location in the code. If a `timeout` t has been specified, the session waits at most t seconds. If the controller has not granted permission to continue within that time, the waiting is canceled and the session resumes execution at the code following the timeout specification. For convenience, there is a short version of the `wait` statement, which has the form "`wait` A" and is equivalent to "`wait {case` A`:}`".

The use of a whole service thus induces a sequence of events. It is the job of the controller to ensure that this sequence fulfills all specified constraints by appropriately detaining execution of the session processes.

2.3.2 Constraint logic and automata

Constraints are written in a linear temporal logic or, more precisely, in a dialect of Monadic Second-order Logic on Strings. This very expressive logic allows concise specification of safety properties, that is, properties stating that "something bad will never happen". By using logic instead of other formalisms, such as Colored Petri Nets or Message Sequence Charts,specification becomes declarative and succinct. Furthermore, this particular logic is distinguished by being constructively equivalent to finite-state automata [60], thereby allowing an operational use. That is, regular languages as recognized by finite-state automata are exactly those definable by Monadic Second-order Logic on Strings.

As an example, consider the following piece of <`bigwig`> session code:

```
...
wait ENTER;
critical region
wait LEAVE;
...
```

and the following constraint:

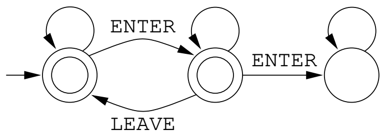
```
constraint {
   label ENTER, LEAVE;
   all t1,t3: (t1<t3 && ENTER(t1) && ENTER(t3)) =>
                is t2: t1<t2 && t2<t3 && LEAVE(t2);
}
```

The `label` line declares the two events, `ENTER` and `LEAVE`. The formula reads: "for any two `ENTER` events there is a `LEAVE` event in between". In other words, the formula ensures *mutual exclusion*: at any time, at most one session will be allowed in the "critical region".

Although quite natural to use, the constraint formulas become rather complex. The macro language described in Section 2.8 allows the programmer to wrap constraints (and other <bigwig> constructs) into layers of user-friendly syntax, thereby increasing the level of abstraction even further.

During compilation of the service, each individual constraint formula is transformed (by the MONA tool [60, 88]) into the minimal prefix-closed finite-state automaton which accepts the set of event sequences that satisfy the formula. The resulting set of automata can be seen as an implicit representation of the product automaton conjoining all the constraints. An explicit representation would very easily cause a state-space explosion as described in [116].

For the single constraint above, the corresponding minimal automaton is:



As can be seen, an `ENTER` event is not allowed if a session is already in the critical region.

The controller component in the runtime system (see Figure 2.3) contains the control automaton and a number of queues, one for each possible event. Initially all automata are in their start states. When a session process requests permission to pass a `wait` statement, the session ID is added to the queues for the designated events. An event is said to be *enabled* if the transitions corresponding to the event lead to accepting states in all control automata. Whenever a queue is non-empty and the corresponding event is enabled, the controller makes the event "occur" by

- following the transitions for the event in each automaton, and
- waking up the session at the front of the queue with information of the event and removing the ID of the session from all queues.

The controller is illustrated in Figure 2.5.

2.3.3 Expressiveness beyond regular languages

Although automata are a very general notion allowing extremely complex safety requirements to be expressed, they have the limitation of being finite-state only.

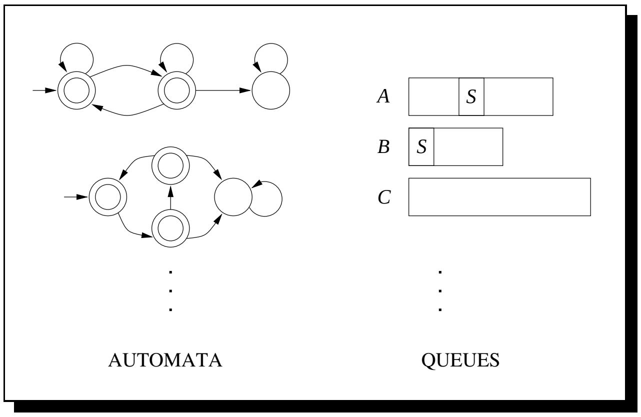


Figure 2.5: The inside of the runtime controller.

This can become a serious hindrance since the number of sessions varies over time and in principle can be arbitrarily high, and since each session by itself is not finite-state.

As an example, consider the following *reader-writer protocol*:

- A number of *readers* and *writers* want access to a critical region.
- An arbitrary number of readers may be in the region at the same time.
- When a writer is in the region, no other process may be there.

To enforce this protocol, the controller obviously somehow needs to remember the current number of readers in the region—something which cannot be done by finite-state automata, and hence cannot be expressed in the basic constraint language.

To overcome this limitation a notion of *triggers* and *counters* is introduced. A trigger is an event associated with a simple arithmetic formula expressing a relation between the number of events of the various kinds occurred so far. For the reader-writer protocol, the following trigger could be defined:

```
trigger NO_READERS when #READER_ENTER == #READER_LEAVE;
```

The meaning is that the event `NO_READERS` will occur once every time the equation becomes true, that is, when the number of `READER_ENTER` events occurred so far is equal to the number of `READER_LEAVE` events. Such trigger events are considered *uncontrollable*, meaning that the programmer cannot restrict them in any way. The expressions in trigger formulas may involve addition and subtraction of event counts and integer constants.

To implement this, the control automata are extended with a number of counters, one for each trigger. A counter increases and decreases when events occur such that it is zero precisely when the corresponding arithmetic formula is true. When this happens, the trigger event itself will occur. Since it is uncontrollable, it will always be enabled.

Using the `NO_READERS` trigger event, it is easy to formulate the reader-writer protocol in the constraint language. In general the constraint language extended with triggers has shown to be a simple, expressive, and efficient formalism for specification of concurrency control requirements. Nevertheless, as already mentioned, the average programmer will not have to deal with constraint formulas directly, but will typically use some convenient high-level macro packages (described in Section 2.8) hiding the inherent complexity.

The concurrency control aspect is described in greater detail in [116, 16, 14].

2.3.4 Concurrency control in `WebBoard`

In `WebBoard`, both the contents of the message list and the client preferences are shared between session processes and hence access to them need concurrency control. The reader-writer protocol is suitable for both. It is okay that several sessions simultaneously read the shared data, but when it needs to be updated, exclusive access is required.

Using constraints wrapped into some macros as described in Section 2.8, this becomes easy to enforce. Each of the two resources—the message list and the preference list—are declared using the `protected` macro, and the critical regions are protected simply by using the two macros `reader` and `writer`. All details are hidden in some layers of macro definitions.

The full version of the `WebBoard` contains an administration feature, which makes it possible for a privileged administrator to modify the whole message list and the user list. Using concurrency control, all access to the service by regular clients can be denied while performing these tasks, such that intermediate modifications are not seen.

Another application of concurrency control is described in Section 2.9.1. There, it is used together with an extension of the runtime model, called *seslets*, to implement server-push-like automatic updating of Web pages.

2.4 Relational Databases

Almost all web services require some form of persistent store for sharing information among individual sessions. Usually, this is achieved through files or by accessing an external database. Claiming to be a programming language for developing web services, <bigwig> clearly needs to address this issue explicitly. One solution, as in for instance PHP, is to support a wide range of SQL databases and have a vast number of SQL queries built into the language. For services with large quantities of information the best choice would be to rely on a real database's efficient storing, retrieval, and data rescue mechanisms. Acknowledging this, we have planned to integrate several SQL databases in <bigwig> in the near future.

However, since a large number of services operate on data sets of only small to medium size, we have aimed for a simpler and more lightweight solution. A solution that bridges the gap between the way internal and external data is accessed and modified. To this end, we have equipped <`bigwig`> with a small internal database capable of storing all of <`bigwig`>'s native values. In this way, external global variables are accessed and manipulated in the same way local ones are, presenting the programmer with one uniform concept of data manipulation.

Also, no substantial SQL queries are needed. However, as we shall see in Section 2.8, <`bigwig`>'s syntactic level macro abstraction mechanism can be used to camouflage our solution as standard SQL queries.

Clearly, the quantities of persistent information varies greatly from service to service, however, the organization hereof is relatively invariant and usually not very complex. The information in most services can typically be structured into records of customers, employees, products, transactions, etcetera. For this reason, we have chosen a simple relational approach. We have thus incorporated *tuples*, *relations*, and *vectors* into the language as native values. The advantages of this approach as compared to, for instance, pointers or object structures is, besides simplicity, that we avoid pointers or references and hence all related errors. Also, the concept of tuples parallel the role of *structs* in C. Should more complex structures be required, they can be implicitly encoded in relations using keys, as we shall see in an example in Section 2.5, where a tree structure is embedded in a relation. For reasons of simplicity we do not allow nesting of relations.

Our solution is based on one general operator: `factor` [71], which is briefly discussed in the following. For a more detailed account, we refer to <`bigwig`>'s online reference manual available from the <`bigwig`> homepage [33].

For our purposes, the advantages of our solution as opposed to the traditional SQL approach are numerous. It is simple—we only have one operator and one framework. The queries are fully integrated into the rest of the language, in the sense that queries are programmed operationally alike other code. Finally, the concept of tuples follow in a natural way, addressing issues we would otherwise have to invent other mechanisms for.

As shown in [71], `factor` can be implemented efficiently and is together with the cartesian product equivalent to relational algebra in the sense that semantics preserving translations exist in both directions.

2.4.1 Factor

Factor provides a means for iterating through the tuples of a relation and manipulate them appropriately. A `factor` expression is evaluated in three steps: The first step is to factorize the relational arguments. The second step is to perform simple computations on the smaller components obtained hereby. The third and final step is to combine the individual results from step two.

The general factor expression takes a number of relational arguments, followed by a list of identifiers used for projection purposes. In the following, however, we shall limit us to cases with only one relational argument and no identifiers. The body of the factor expression is a standard <bigwig> statement that may or may not return a value using the `return` statement. The semantics is that the body will be executed once for each tuple in the relation given as argument, denoting by "#" the value of the current tuple. All tuples or relations returned underway will be collected into one relation that will constitute the result of the entire factor expression.

The usages of factor can roughly be divided in two, depending on the presence of return statements. Either it is executed to produce a relation derived from the one given as argument or it is evaluated for its side-effects.

2.4.2 Database in WebBoard

Being a strongly typed language, the compiler must be able to determine the exact types of the components of all tuples, relations, and vectors used. We have thus invented a type definition construct, `schema`, for defining composite types, that are in turn used to declare *tuple*, *relation*, and *vector* variables. Variables in a service are either *local* or *global*. Local variables are private to a session, in the sense that all running threads have their own private copies. Global variables on the other hand are persistent and shared among all the sessions of a service. All declarations not prefixed by the modifier `global`, are taken to be local.

Below is an example from `WebBoard` declaring a global relation that will hold all the posted messages in the `WebBoard` along with their relationships. A type `Message` is defined and subsequently used to declare a global relation `board`.

```
schema Message {
   int id, reply_to;
   string name, email, subject, content;
   time time_stamp;
}

global relation Message board;
```

As for the manipulation of such values, we exhibit another example from `WebBoard`, where a function `getMessage` uses `factor` for fetching a message with a particular `id` from a `Message`-relation `B`.

```
tuple Message getMessage(int id, relation Message B) {
   tuple Message m;

   factor (B) {
      if (#.id==id) m = #;
   };
   return m;
}
```

Here the factor expression does not return a relation. Instead, the `Message`-relation argument `B` is traversed for its side-effects. The local variable `Message`-tuple, `m`, is updated to the value of the current tuple whenever the current tuple's `id` component equals that of the formal parameter, also named `id`. Finally, `getMessage` returns the tuple held in `m`.

2.5 Typed Dynamic Documents

When creating Web services using the traditional script-centered approach, Web documents are constructed dynamically at request-time typically using `printf`-like statements. This is a dynamic mechanism which allows for tailor-made documents for the client at hand. The one major drawback of this traditional approach is the lack of static safety. That is, given a Perl CGI-script it is by no means in general possible to determine whether the script will produce HTML which is readable by the clients browser, whether the form input fields in the document match what is expected by the receiving script receiving, and so on.

Furthermore, using `printf` the documents must often be constructed in a linear fashion from top to bottom as opposed to constructing (with focus on functionality) various parts of a document when the dynamic data is ready and then later (with focus on design) plug these parts together into a document structure. That is, it is hard to separate the various task in dynamically constructing Web documents. As mentioned in Section 2.2, separating these tasks has the advantage the Web page designer can concentrate on designing Web pages in, for instance, FrontPage, while the service logic programmer can concentrate on implementing the service logic.

It would thus be advantageous to raise the level of abstraction and let documents be first-class values that can be constructed and shown to the client during computation instead of flat strings that are written to standard output.

In the Mawl project [67] the above problems are addressed by introducing the notion of a *document template* which is a syntax-checked HTML document with named gaps. At runtime, gaps can be replaced by simple contents such as text strings or numbers. Using this approach, static safety guarantees can be given for documents constructed at runtime. However, the static safety is gained sacrificing the flexibility of traditional script-based document construction.

In <`bigwig`> we provide a mechanism for building documents which has the flexibility of script-based authoring and yet provides static safety. The mechanism is based on raising the notion of a template to higher-order templates in the sense that template gaps can be replaced by other templates which again can have gaps. This is a truly dynamic mechanism which calls for special care in the analysis checking the correctness of the created documents. The type checking of documents relies on a sound first-order forward data-flow analysis which keeps track of the named gaps and input fields occurring in the program.

2.5.1 Language constructs

The <bigwig> language has support for dynamic construction of HTML documents via use of higher-order templates. In this section we describe the syntax provided for this, that is, the extension of HTML which allows the use of named gaps, the plug operation which provides a way of combining higher-order templates, and the `show-receive` statement which shows a dynamically generated page to the client and reads the supplied input field values into the state of the running session.

Document templates First, let us introduce the syntax which is added to express higher-order templates. We introduce two kinds of named *gaps*:

```
This <[foo]> <b>is <[bar]></b> you know!
Some pictures <img src=[baz]> of funny ducks.
```

Here, `foo` and `bar` are names of gaps which can be plugged with HTML fragments, while `baz` is a gap which can be plugged with an attribute string value. This HTML extension is purely internal to <bigwig>; only standard documents are presented to the Web client.

To be able to provide static safety, we introduce the notion of well-formed documents. A document is *well-formed* if the following requirements are satisfied. All gap names occur at most once in each document. If an input field name occurs more than once, all the occurrences must be of the same type, which has to be either radio or checkbox. For instance, the following document is not well-formed:

```
This <[foo]> is good.
Last name: <input name="last" type="text">
Previous <[foo]> order:
   <input name="last" type="radio" value="coffee">
   <input name="last" type="radio" value="tea">
```

First of all, the document has two gaps named `foo` which is not allowed. Furthermore, it is has three occurrences of a `last` input field of kind `text`, `radio`, and `radio`, respectively. Removing the `text` kind occurrence of `last` would make the document satisfy the field part of the well-formedness requirements.

The plug operation In <bigwig>, a document expression can either be a document variable, a constant HTML document, or a plug operation. The result of the plug operation d_1`<[`g`=`d_2`]` is d_1 with its g gap replaced by d_2 as illustrated in Figure 2.6. Documents can thus be constructed from the higher-order template building blocks and dynamic data using the plug operation.

The `show` statement When a response in form of a well-formed HTML document has been composed, the `show` call is used to send the document to the clients browser and later retrieve the supplied input field values as explained in Section 2.2. Thus, the `show-receive` call

`show` d `receive[`$v_1 = f_1, \ldots, v_n = f_n$`]`

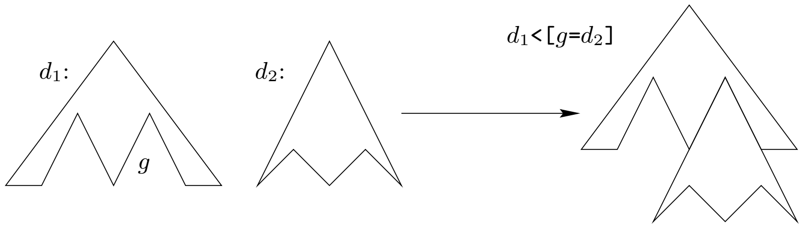


Figure 2.6: The plug operation: d_1<[g=d_2].

allows interaction with the client: The document d is shown to the client, and when the client submits the reply, the received values of the form fields named $f_1, \ldots, f_n$ are assigned to the program variables $v_1, \ldots, v_n$, respectively.

The following example illustrates a simple use of the plug operation and the `show-receive` statement:

```
html H = <html><body bgcolor=[color]>
             Hello <[what]>! <br>
             Enter your name: <input name="myname" type="text">
         </body></html>;
string s;
show H<[color="red",what="world"] receive [s=myname];
```

Here, `color` is a string gap and `what` is an HTML gap. The contents of the input field `myname` is assigned to the variable `s`.

2.5.2 Static safety

Using a specialized data-flow analysis, we can check programs and provide static safety. Once a program has been checked and found to be well-typed, we can provide static safety, that is, we can statically guarantee that

- only well-formed documents are ever constructed;
- the receiving document in a plug operation has the specified gap;
- the document shown with `show-receive` has the input fields expected in the `receive` part.

In other words, we guarantee statically that document related runtime errors cannot occur.

Static safety is provided by introducing a somewhat complicated notion of *document type*. However, these types are not explicitly written by the programmer, but are inferred by the compiler using a global flow analysis. A document type has two distinct components: a *gap map* and a *field map*. The gap map describes the gaps which are present in a document and, similarly, the field map describes its input fields.

An interprocedural monovariant first-order forward flow analysis associates sound and precise gap and field information for each document variable to each program point. Based on this information we can then determine if a program is type correct. If it is, the properties listed above will in fact hold. If not, a suitable error message is produced.

Runtime representation

By exploiting the fact that only well-formed documents are constructed at runtime due to the static analysis, an efficient implementation can be obtained. Using a technique resembling closures in functional languages, the <bigwig> implementation has a compact representation with sharing of the static parts of the documents, and at the same is able to perform both the plug operation and `show-receive` in low time complexity. The implementation is described in detail in [117].

2.5.3 Tuple input fields

In this section we describe an extra internal extension that has been added to the <bigwig> document syntax for convenience. It is the notion of *tuple input fields* which allows relations of dynamically varying sizes to be received on the server-side. The extension consists of a mechanism, called *tuples*, for collating input values. The following document shows an example of tuples:

```
<tuple name="order">
   Pizza Napoli <input type="hidden" name="pizza" value="napoli"><br>
   Amount <input type="text" name="amount"><br>
   Extra cheese?  <input type="checkbox" name="cheese" value="true">
</tuple> <p>
<tuple name="order">
   Pizza Roma <input type="hidden" name="pizza" value="roma"><br>
   Amount <input type="text" name="amount"><br>
   Extra cheese?  <input type="checkbox" name="cheese" value="true">
</tuple>
```

The tuples, which are denoted by `tuple` tags, are viewed as a single input field named `order`. This extension allows multiple occurrences of the same input field name to appear in a document. When the field values are received, they are automatically transformed into tuples in a relation with schema {`pizza`, `amount`, `cheese`}. As explained in Section 2.4 relations are present in <bigwig> as native values.

<bigwig> does not have support for higher-order relations, that is, <bigwig> tuples cannot be nested but can only hold basic values.

Again, we stress that the extensions are purely internal to <bigwig>; only standard HTML documents are presented to the Web client.

2.5.4 Dynamic documents in WebBoard

Being a central notion in <bigwig>, dynamic documents are used extensively in WebBoard. The most sophisticated use of dynamic documents is when the hierarchical message list in WebBoard is transformed into a document. The transformation is shown in a simplified version below. It is at the same time succinct, concise, and yet inherently complex.

```
schema Message {int id; int replyto; string subject};
global relation Message Board;

html Entry = <html><li><[id]>:<[sub]><ul><[replies]></ul></html>;
html Cons = <html><[reply]><[replies]></html>;

html View(int id, string subject) {
    int i;
    html H = Entry<[id=id,sub=subject];
    vector Message R;
    R = sort(select * from Board where (#.replyto==id); id);
    for (i=0; i<|R|; i++)
       H = H<[replies = Cons<[reply=View(R[i].id,R[i].subject)]];
    return H<[replies=""];
}

show View(0,"");
```

A schema Message is defined and a relation Board with schema Message declared. The idea is that each entry in WebBoard has its own unique ID, it is a reply to some other entry in WebBoard, and it has a subject. For simplicity, we here abstract away from the actual content of the entry and other attributes such as name and email address of the contributor. To build documents that reflect the hierarchical structure in Board, we define the constant templates Entry and Cons for building new entries and listing replies, respectively. The generation of the document is initiated by the statement "show View(0,"")" which calls the recursive function View with the top-level entry 0 as argument. The View function builds HTML for the entry id using the template Entry and the given arguments id and subject. Then all the replies to the current entry are found, sorted, and afterwards added to the document one by one using a for loop and the template Cons. A recursive call is made for each of the replies to include the transitive closure of its own replies.

For the programmer, the static safety analyzer comes in handy checking both well-formedness of document templates and existence of gaps in plug operations.

Again, note the possibility of separating Web design from service logic. The template Entry could be much more sophisticated with respect to design without affecting the complexity and readability of the service logic code.

2.5.5 Reverse-engineering Web pages

Document templates can also be used to automatically extract pieces of information from pages fetched from the Web. A document template is then used as a pattern where the gaps correspond to the pieces to be extracted.

The `match` construct attempts to pattern-match a string with a document template such that the non-gap parts of the document match completely. If a match is possible, the parts of the string that correspond to the gaps are assigned to a list of variables.

As an example, after executing the following code,

```
html H = <html>This <[foo]> <b>is <[bar]></b> you know!</html>;
string s = "This thing <b>is cool</b> you know!";
string x,y;
match (s,H) [x=foo,y=bar];
```

the values of the variables `x` and `y` are "`thing`" and "`cool`", respectively.

This operation makes it possible to write services that build on top on existing Web services (that do not necessarily have anything to do with <`bigwig`>).

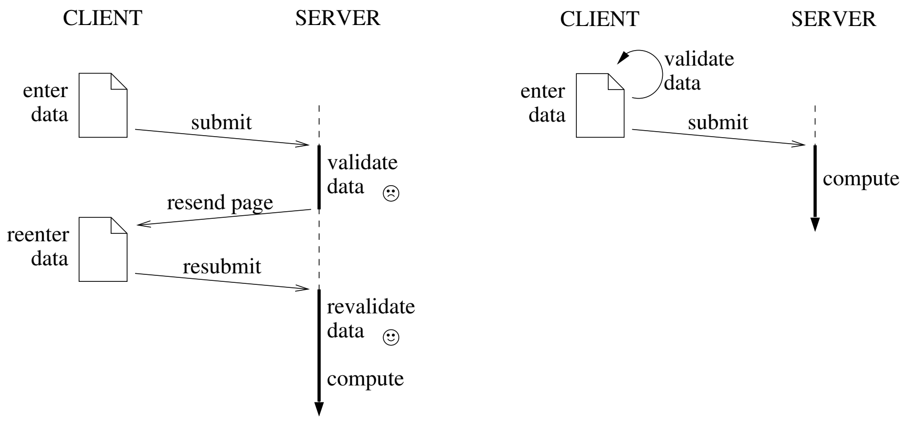
2.6 Form Input Validation

It is often the case that data entered by the client in a text input field is required to comply with a certain format. A field might for instance expect a valid number, date, or email address to be entered. Usually, this is achieved through *server-side input validation*. When the page is submitted, the program on the server determines whether the entered data is of the required form. If this is not the case, the program outputs a page containing appropriate error messages along with the erroneous input fields, allowing the client to correct them. This process, as illustrated in Figure 2.7(a), is repeated until all input fields contain appropriate data. Although widely used, the approach has some considerable drawbacks:

- it takes time, thereby annoying the client;
- it causes excess network traffic; and
- it requires tedious explicit programming.

Our solution is to automate the validation and move it from the server-side to the client-side. This approach, as illustrated in Figure 2.7(b), overcomes all three mentioned drawbacks.

In the following we give a detailed explanation of our solution and illustrate it with examples taken from the `WebBoard`. As a corollary, we mention how the technique also can be applied to other text pattern matching tasks.



(a) Traditional, server-side.

(b) <`bigwig`>, client-side.

Figure 2.7: Input validation.

2.6.1 Formats and client-side validation

We allow the <`bigwig`> programmer to define formats and attach them to input fields of type `text`. Submission of form input to the server is prohibited while data entered in the input fields does not comply with the attached formats. This way, the client can only continue the session when all input fields contain appropriate data. This validation takes place entirely on the client side, saving the time and traffic resulting from the round-trip to the server for validation. Furthermore, client-side validation has the benefit that it can be performed incrementally—that is, as the client enters the data. In this way, errors can be signaled as soon as possible making the service even more convenient to use.

The formats are specified by the programmer as standard regular expressions enhanced with intersection, complement, ASCII character ranges, and integer intervals as syntactic sugar. During service compilation, all format expressions are translated into minimal deterministic finite-state automata. These automata are then run incrementally on the client as data is entered in the fields. Thus, at any given moment, an input field is associated with a state, namely the state of its automaton run on its data. The data in an input field complies with its format if and only if its state is accepting. When a document is submitted, all that needs to be done in terms of validation is to inspect the states of all input fields to see whether they are all accepting. Notice how this use of finite-state automata is reminiscent of the technique used for concurrency control described in Section 2.3.

Since it is possible for a malicious client to bypass the client-side validation process, the data is double checked on the server side by rerunning the automata on the data received. If any inconsistencies are discovered, a runtime error is issued.

Why regular expressions?

There are several reasons for choosing regular expressions rather than other specification formalisms, such as context-free grammars or Turing complete languages. They are well known and widely used for other text pattern-matching purposes. Due to their declarative nature, no programming in the operational sense is required on behalf of the service programmer. This makes them easy to understand, and simple to use and modify. Also, it can be efficiently decided whether a string is in the language defined by a regular expression through the use of an equivalent automaton. Our experiences show that the expressive power of regular expressions suffices for common validation purposes, such as validating dates, email addresses, and zip-codes. Anything requiring expressiveness beyond regularity is deferred to the server side, but this is rarely needed.

Implementation

A number of technologies can be considered for implementing automaton-based client-side input validation. For the reasons described in Section 2.1 we shall disregard all browser specific technologies, such as Plug-ins or ActiveX. This leaves us with JavaScript and Java applets as candidates.

JavaScript has the advantage of being lightweight. No Java Virtual Machine is required to start, which may take a considerable amount of time, but this of course only happens once. Also, the JavaScript implementation in current browsers is more stable than that of Java.

Java, on the other hand allows us to catch more events and to control much more of the appearance of the text displayed. This extra event and display control could be exploited to catch help and auto-completion keys to display various help messages and show auto-completed text in inversed colors, as known from many other GUIs. Text not conforming to the format could be displayed in red. Unfortunately, Java text fields have another look-and-feel than that of HTML text input fields, and we have experienced problems with choosing and scaling the fonts to match the appearance of the surrounding HTML text.

We have experimented extensively with both technologies and chosen JavaScript for the reliability and appearance aspects.

To compensate for the lack of display control, we have added a feature to <bigwig>, the usage of which is optional. An image is displayed next to all input fields with attached formats. This image is dynamically changed to always reflect the automaton's current state, which is either an *accept*, *reject*, or *crash* state. By default, the <bigwig> compiler will use a "traffic light" to display the verdicts of the automaton. When the text entered is in the language defined by the regular expression, the light is green. It is yellow when the input is not OK, but perhaps later will be—that is, it is in the prefix closure of this language. Finally, the light is red when the input is definitely wrong—the automaton is in the crash state. These images can be redefined and accompanied by appropriate text messages in the browser's status bar.

The automata are automatically generated by the compiler during service compilation and placed in files on the server for the client to fetch on demand across the Web. This may seem as an expensive thing to do. However, the format files are usually quite small and are often used repeatedly and hence get cached by the browser or a proxy server and are thus typically only fetched once.

2.6.2 An example from the `WebBoard`

When a client logs in to `WebBoard`, he is prompted for his name, age, and email address. In the example below, some formats are defined and subsequently attached to input fields in a document:

```
format Digit = range('0','9');
format Number = plus(Digit);
format Alpha = union(range('a','z'),range('A','Z'));
format Word = plus(union(Digit,Alpha));
format Email = concat(Word, "@",Word,star(concat(".",Word)));

html EnterData = <html>
   Enter your name: <input type="text" name="name" format="Word">
   Enter your age: <input type="text" name="age" format="Number">
   Enter your email: <input type="text" name="email" format="Email">
</html>
```

The `name` field is required to consist of letters only, the `age` field must be a number, and the `email` field must consist of a word followed by a "`@`"-symbol and a number of dot-separated words.

As the example shows, we have for reasons of readability chosen a rather elaborate syntax for specifying the regular expressions as opposed to the perhaps more frequently used operator-like syntax with asterisks and plus signs etc.

When the document `EnterData` is shown, the client can only submit the page and continue when he has filled in a legal name, age, and email address.

2.6.3 Other format applications

Formats are also used for other text pattern-matching purposes in <`bigwig`>, paralleling the use of regexps in Perl. A declared format can be applied to a string to give the longest prefix of the string that is contained in the language defined by the format. Equivalent operational solutions can be tedious and hard to read, write, or modify. This feature is also used in the `WebBoard`, from which the following example is taken.

```
format Re = concat("[Re: ", anything, "]");

string addReply(string s) {
   if (Re(s)) return s;
```

```
    else return "[Re: " + s; + "]"
}
```

The function `addReply` will wrap the string "`[Re: ...]`" around a reply string if it not already has been. In the `WebBoard`, this is used to present a sensible default reply text to someone replying to a message.

A similar format application is for file scanning. The <`bigwig`> construct `scan` takes two arguments, namely a file identifier and a format. The operation then returns as much of the file from the current file position that is in the regular language defined by the format and advances the file pointer accordingly.

As the examples show, formats provide a powerful mechanism for text pattern-matching. They make it easy to perform high-level client-side form input validation and are also useful internally in the service code.

2.7 Cryptographic Security

Most CGI services execute with hardly any security. Client requests and server responses may be eavesdropped, exposing sensitive information. A client's session interaction with the server may be hijacked by a third party. Also, there is no support for authentication of any kind. The solution to these problems is concealment of sensitive information through encryption and establishment of identity and integrity using digital signatures.

Usually, the lack of such precaution measures can be attributed to the complexity and amount of low-level details involved in designing and adapting using such protocols for the services at hand. Furthermore, a lot of the services that do address these issues, often do it unsoundly.

Having a compiler generate the web service code from high-level specifications, as in <`bigwig`>, gives us a unique opportunity for addressing these issues "once and for all". The appropriate levels of security, depending on the individual services, can thus be generated automatically by the compiler.

2.7.1 Implementation

Recall from Section 2.2, that we use a random key string to in the identity of a session preventing users from guessing it. However, this is considered low-level since malignant users resort to more than guessing.

A variety of security modifiers are available in <`bigwig`> for manipulating the level of security of the services produced. All of them can be applied at different levels in a service. They can be made to affect an entire service, a particular session, or even individual show-calls. In fact, even pieces of a document can be protected, with the effect that no matter how it is plugged into other documents, as explained in Section 2.5, its security level is respected.

htaccess can be used to specify a unix password-file that is to be used for login-password verification through the browser/web-server built-in `htaccess` protocol.

singular is used when the server wants to communicate with the same client throughout execution. The modifier causes the service to verify that the IP number of the client does not change.

selective allows one to specify which types of IP addresses are serviced and which are not.

signed will cause documents to be digitally signed using the RIPE-MD160 algorithm.

secret will encrypt documents using triple-DES and RSA.

So far only the modifiers `htaccess` and `singular` have been implemented, but rest will be follow imminently. The selective modifier will either be implemented directly or compiled to `htaccess`. The more advanced modifiers `signed` and `secret` will most likely be implemented using SSL or S-HTTP.

The above mentioned modifiers are the ones currently planned for <`bigwig`>, but as new security technologies arrive, they will be incorporated.

Programmers no longer need to worry about complex security issues, but simply add the appropriate modifiers to their <`bigwig`> programs to obtain the required levels of security. In this way, advanced security is available to everyone.

2.8 Macro Mechanism

Most programming languages have a general mechanism that allow the programmer to extend the language with new constructs representing relevant concepts and thereby increase the abstraction level. Typical such mechanisms are macros and classes. A class mechanism, as known from object-oriented languages to be very well suited for this purpose, has—seen from a <`bigwig`> perspective—the drawback that it requires a whole paradigm in order to be used properly. Also, since <`bigwig`> contains relatively many syntactic categories which all could benefit from abstractions, a class mechanism may become too restrictive for this purpose. Macros, as known for instance from the C preprocessor, are usually though of as a low-level mechanism. They typically operate on the lexical level meaning that they are completely independent of the language syntax. It is well known that this makes it hard to use macros to build layers of advanced abstractions. On the other hand, macros have the benefit that it is a simple mechanism in the sense that there is no semantics involved—only substitution of syntax.

In <`bigwig`>, we have chosen to use a macro mechanism. The limitations of lexical-level macro mechanisms are addressed by using a novel *syntax-level* macro mechanism. The remainder of this section describes this mechanism and shows a number of small but useful applications. These examples make it evident that syntax-level macros provide a simple but valuable abstraction mechanism for many different aspects of the <`bigwig`> language.

2.8.1 Syntax-level macros

A lexical-level macro mechanism works by performing substitutions in the source code before the actual parsing takes place. The problems that inevitably arise with this approach are: 1) Unless being very careful when defining a macro it is possible that the expanded code is parsed differently than expected. The standard example of this is the following. Consider a C macro defined by "`#define square(x) x*x`". When this seemingly innocent macro is applied to `y+1` the result is parsed as `y+(1*y)+1` and not as `(y+1)*(y+1)`—a bug which might be hard to find. 2) Macro definitions cannot be syntax checked separately. Also, when syntax or type errors occur, the error messages are generated from the expanded code instead of the original code. This makes it hard to track down errors and, in the end, limits the possibility of building layers of macros.

For these reasons we move the macro abstraction mechanism from a lexical level to a syntactic level of operation. Instead of piecing together lexical tokens, they operate on parse trees, allowing the construction of legal parse trees only. In order to ensure this, all macros are given a "return type" which is a nonterminal from the language grammar with which the macro body is required to comply. Thus, instead of having a macro body as a sequence of arbitrary characters, it is now parsed according to its return type nonterminal that hence determines where the macro can be invoked. A parse error is issued if the body and the return type do not correspond. Similarly, all formal parameters of macros are also typed with nonterminals. When a macro is applied, it is required that the actual parameters comply with the parameter types. All nonterminals in the language grammar are potential return types and parameter types.

This move from the lexical level to the syntactic level corresponds in many ways to moving from an untyped language to a typed language. Adding types causes more errors to be caught early on and also makes it easier to read and understand the code.

The example below illustrates how the <`bigwig`> macro mechanism works. More realistic examples are given later.

```
macro <stm> repeat <stm S> until ( <exp E> ) ; ::= {
   {
      bool first = true;

      while (first || !E) {
         S
         first = false;
      }
   }
}
```

This example shows the definition of a `repeat-until` construct known from other languages. Its return type is `stm`, which is the nonterminal for statements, and it has two parameters, a statement `S` and an expression `E`. Of course, the construct could also have been defined as "`S; while (!E) S`". Defining it this way though would cause a blow-up in the size of the resulting parse tree, because

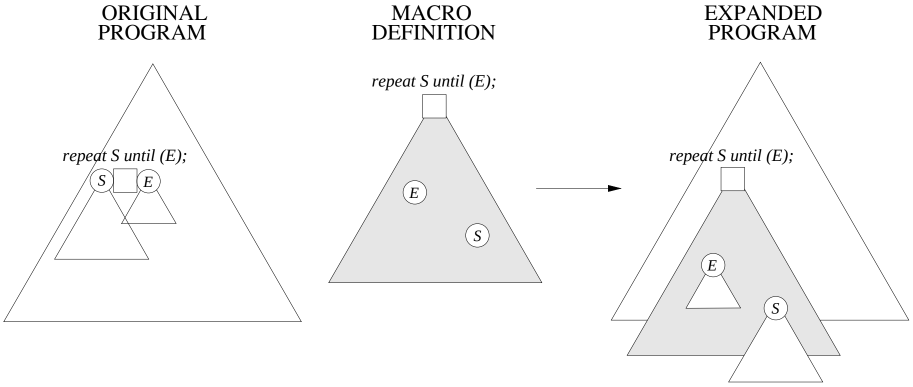


Figure 2.8: Macros—operators on parse trees.

argument S is mentioned twice in the body.

The effect of invoking the `repeat-until` macro in a program is depicted in Figure 2.8. The white parts are written by the service programmer and the grey parts by the macro programmer.

The benefits compared to using a lexical-level macro are numerous: Since substitutions take place on the parse tree, situation like in the `square` example above can no longer occur. By using types, the macro definition can be syntax checked separately, and when it is applied it can be checked that the arguments fit into the macro and that the expanded macro fits into the place of application. Defining macros is almost like dynamically extending the language, adding new productions to its grammar.

The <`bigwig`> macro mechanism contains some additional features described in the following.

Arbitrary syntax The <`bigwig`> macro mechanism allows the invocation syntax to be designed by the macro programmer. This flexibility has two major advantages. First, the syntax may help convey the meaning of a macro abstraction. Second, it allows the underlying language to be transparently extended with new constructs that appear as if they were inherent parts of the language.

This has been used in the example above to give the `repeat-until` construct the look-and-feel of a C control structure with the `until` delimiter, the parentheses around the conditional expression, and with the semicolon terminator. This syntax is enforced in the sense that it would constitute a parse error if one were to deviate from it in subsequent invocations.

In order to allow efficient parsing, there are limitations though. In the current implementation, macros must begin with a unique identifier, but this restriction can to some degree be loosened up without losing efficiency.

Naturally, the very liberal approach of allowing arbitrary syntax makes it even more important to use macros cautiously. If overused, the source code can become completely unreadable.

Alpha conversion Using macros—especially when using them in layers—identifier clashes often occur. Consider for instance the `repeat-until` example. If such a construct is used nested inside another, the uses of the `first` variable will be bound improperly. The solution to this problem is to perform *alpha conversion*. We alpha convert, that is, rename, all identifiers in the body of a macro each time it is invoked. In this way, the body of a macro becomes a closed scope and identifier clashes can no longer occur.

Error reporting By operating on syntax trees instead of token sequences it becomes possible to obtain sensible error messages—even when macros are used in many layers. In the implementation, error messages are threaded back through macro expansion. Thus, along with an error we list the complete trail of macro expansions. This allows the programmer to locate and correct errors as when programming without macros.

Pretty printing Another feature in our macro implementation is *pretty printing*. The compiler contains a general pretty printer that is capable of printing the code in HTML. We take advantage of the hypertext mechanisms in HTML to insert jumps from identifier usages to definitions, from macro invocations to their definitions, from keywords to their specification in the reference manual, and so on. Furthermore, it can also be instructed to expand all macros in the resulting output.

2.8.2 Examples from the WebBoard

The macro mechanism presented, although designed for <bigwig>, is a language independent mechanism that is in no way <bigwig> specific. It can be applied to most other programming languages. However, it is especially useful when there are many sublanguages involved as in <bigwig>. In the examples below we show how our macro mechanism can help glue the different components together in a uniform way. Also, many of the more complex tasks such as controlling concurrency and creating database queries, can be defined as simple high-level abstraction that appear as if originally in <bigwig>.

Concurrency control macros

As promised in Section 2.3, the typical programmer does not have to worry about complex logic formulas. He will simply use the high-level concurrency control abstractions presented to him through the macro mechanism. Recall the mutual exclusion example from Section 2.3. The mutual exclusion predicate can conveniently be defined as a macro:

```
macro <formula> mutex ( <id ENTER> , <id LEAVE> ) ::= {
   forbid ENTER when is t: ENTER(t) && all tt: t<tt => !LEAVE(tt)
}
```

The `forbid-when` construction used is in fact defined as another macro in a "standard macro library".

The example in Section 2.3 also exhibited how to obtain exclusive access to critical regions by wrapping them into two `wait` statements, waiting on the two `ENTER` and `LEAVE` events. In practice, however, the programmer should not have to define the two events, explicitly add the mutex requirement and the two related `wait` statements each time exclusive access is required. Instead, using macros it is possible to lift this to an even higher level of abstraction and completely hide away all details of events and constraints:

```
macro <toplevel> region ( <id R> ) ; ::= {
   constraint {
      label R~ENTER, R~LEAVE;
      mutex(R~ENTER, R~LEAVE);
   }
}

macro <stm> exclusive ( <id R> ) <stm S> ::= {
   {
      wait R~ENTER;
      S
      wait R~LEAVE;
   }
}
```

The tilde character "~" is a means for concatenating identifiers. All the programmer needs to do is to "declare" a region by using the region macro. Subsequent uses of the `exclusive` statement with the same "declared" identifier will then operate on the same region, granting exclusive access to their statements. It is almost as if we have extended <`bigwig`> with a whole new concept that was not there before—the concept of regions.

Similar techniques are used to create even more sophisticated concurrency abstractions. The `WebBoard` uses a *reader–writer* protocol where writers have priority over readers for protecting various global resources. Three macros that build on top of the `region` and `exclusive` macros are used to ensure this protocol:

```
macro <toplevel> protected <type T> <id V> ; ::= {...}
macro <stm> reader ( <id V> ) <stm S> ::= {...}
macro <stm> writer ( <id V> ) <stm S> ::= {...}
```

(Since rather complex, the bodies of these macros have here been omitted.) Everything the service programmer needs to do is to prepend his global variable declarations with the "modifier" `protected`. With this done, the programmer can now obtain read and write access to his variables through the statements `reader` and `writer`. Thus, the average programmer does not have to worry about logic and constraints, but just use the simple and powerful constructs provided by these macros.

Database macros

As with concurrency control, the macros can be used to heighten the level of abstraction for database queries. Also, the flexible syntax properties of the macros can be exploited to camouflage the macros, making them appear as if they were real SQL queries. Various SQL queries such as `union`, `join`, `project`, and `select` are defined and used in the `WebBoard` example. The `select` query is defined as follows:

```
macro <exp> select * from <exp rel> where <exp cond> ::= {
   factor (rel) {
      if (cond) return #;
   }
}
```

It abstracts away all the details of the <bigwig> query language with the `factor` operator and allows the programmer to use this SQL query as if it was already in <bigwig>.

Format macros

The `WebBoard` uses a regular expression constructor `plus` to define email formats as in Section 2.6. This construct, allowing one-or-more occurrences of a given regular expression is, in fact, not part of the <bigwig> language, but a macro:

```
macro <regexp> plus ( <regexp R> ) ::= {
   concat(R, star(R))
}
```

Similarly, one can define a regular expression constructor `optional`, for specifying zero-or-one instances of a given regular expression.

Cryptographic security macros

One can write a macro collecting the various security requirements present within an organization. For instance, if a number of services, such as WebBoard, are to be used only within BRICS, the macro "brics only" shown below could be defined in a common BRICS macro library and used as a special securify modifier in all BRICS services:

```
macro <mode_list> brics only ::= {
   htaccess("/etc/passwd") singular selective("*.brics.dk")
}
```

This could easily be applied to `WebBoard` if it is to be used as a message bulletin board within a company.

2.8.3 Domain specific abstractions

One could imagine a library of domain specific macros tailored for certain types of services, further increasing the level of abstraction. For instance, a macro library dedicated to the development of Web shopping sites, with shopping baskets, customers, shelves and products as language constructs. Another example could be a library with specialized two-party communication primitives, applicable to a wide range of services.

Further information about <`bigwig`> macros together with more examples of macros building layers of abstractions on top of the various sublanguages of <`bigwig`> can be found in [14] and on the home page [33].

2.9 Plans for the future

2.9.1 Extending the runtime model

The session based runtime model described in Section 2.2 has been successful in capturing the main flow of most service kinds. However, not all kinds of interactions fit elegantly into the simple model. A general design principle is to separate the main service code from user-interaction code with the purpose of keeping the code easily comprehensible to the programmer. Some interactions cannot be programmed in the model without violating this design principle or making the interaction awkward to the client. In this section some examples of this are given, and some powerful extensions of the runtime model are described to solve the problems.

Document clusters

Many services contain interactions where a whole "cluster" of linked documents are shown and browsed by the client independently of the main service flow. In the current runtime model the service programmer is forced to mix code dealing with the browsing with the main service code. Inevitably this becomes both tedious to program and complicated to maintain.

In order to separate the browsing from the main service code the runtime model can be extended as explained in the following. A cluster is formed in the service code using a construct resembling the plug operation (described in Section 2.5), but instead of plugging one document into another it forms a link between them. When a `show` is issued, a reply file is generated for each document reachable from the given document. The client can then browse through the cluster without involving the session process at all. Document clusters are illustrated in Figure 2.9.

There is however a small complication. Recall that in the presented setup the name of the reply file serves as the identity of the session and hence must be fixed for the duration of the session. To retain this we need a small modification of the runtime system: the reply files are no longer retrieved directly by the HTTP server but via a connector process. This connector receives the ID of the session

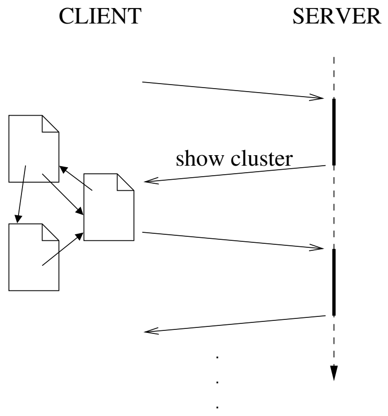


Figure 2.9: Document clusters. (Compare with Figure 2.2.)

in the CGI query string together with the document number, and returns the corresponding page.

Seslets

The session-based runtime model has the inherent limitation that client interaction is required every time a page needs to be updated. In `WebBoard` for instance, the client must click "Refresh" to see whether any new messages have been posted. Many services have similar features where lack of automatic updating becomes annoying to the client.

A different example illustrates a similar problem. Consider a "table viewer" service which allows clients to browse relational database tables. If the tables are large it could be desirable that a whole table is not shown at once but rather scrolled through by the client such that only relevant parts needed to update the page are sent across the network. This can be programmed into the runtime model from Section 2.2, but the main flow in the service code quickly gets cluttered with code dealing with the table scrolling, thereby violating the design principle mentioned above.

Both problems—the annoying "Refresh" interactions, and the cluttered main code—can be elegantly solved using *seslets*, a kind of light-weight sessions introduced in the following.

A seslet is a session not initiated directly by a client as normal sessions are, but instead by a Java applet located in a page shown to a client. Such a session can do anything a normal session can do with the following exceptions: 1) it cannot perform a `show` (since the client is not waiting for a normal reply page when a seslet is running), and 2) instead it can perform a `return` which returns a value to the applet.

Initiating a seslet in an applet resembles invocation of a remote procedure call on the server. Seslets can be used to

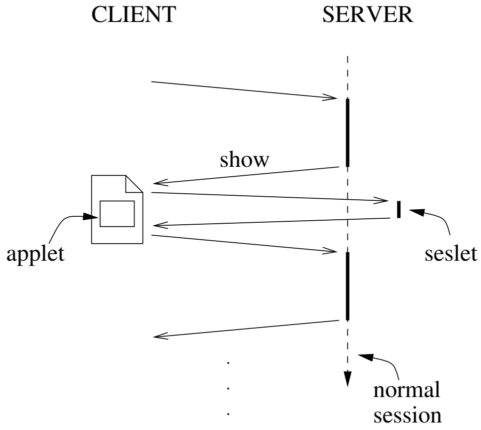


Figure 2.10: Seslets—light-weight sessions.

- inspect and modify the global data on the server, and
- communicate with the safety controller (see Section 2.3) to wait for certain events to occur or to perform events and thereby affect the behavior of other sessions in the service.

Figure 2.10 illustrates the use of seslets.

A page can now be updated automatically without client interaction using a simple seslet which waits until a session belonging to another client causes a "time to update" event. When this happens, the applet on the page is informed and it simulates a click on "Refresh". Alternatively, the seslet itself sends all information required to update the page. Notice how this example shows how a kind of "server push" can be implemented.

The table-viewer example can now also be expressed succinctly. The browsing is simply handled by an applet which uses a seslet to fetch the parts of the table required to update the page.

These examples should illustrate that the notion of seslets is a powerful and elegant technique enhancing the runtime model significantly. Some Java applet programming is required, though, when using seslets. To facilitate that, a set of Java stubs taking care of all parameter marshaling and unmarshaling are automatically generated by the <`bigwig`> compiler.

2.9.2 Service monitoring

We also plan to equip <`bigwig`> services with a service monitor capable of surveying an entire service. Global data, sessions and the controller can be inspected and modified. Basically the monitor can tell you all the information available about the actual state of a service plus previously logged information.

Chapter 3

Runtime System

Contents

This chapter describes a runtime system for interactive web services. The work has been published in Computer Networks (The International Journal of Computer and Telecommunications Networking) [16] and appeared in the proceedings of The Eight International World Wide Web Conference. It is joint work with Claus Brabrand, Anders Møller, and Michael I. Schwartzbach.

Interactive web services are increasingly replacing traditional static web pages. Producing web services seems to require a tremendous amount of laborious low-level coding due to the primitive nature of CGI programming. We present ideas for an improved runtime system for interactive web services built on top of CGI running on virtually every combination of browser and HTTP/CGI server. The runtime system has been implemented and used extensively in <bigwig>.

3.1 Introduction

An interactive web service consists of a global shared state (typically a database) and a number of distinct sessions that each contain some local private state and a sequential, imperative action. A web client may invoke an individual thread of one of the given session kinds. The execution of this thread may interact with the client and inspect or modify the global state.

One way of providing a runtime system for interactive web services would be to simply use plain CGI scripts [40]. However, being designed for much simpler tasks, the CGI protocol by itself is inadequate for implementing the session concept. It neither supports long sessions involving many user interactions nor any kind of concurrency control. Being the only widespread standard for running web services, this has become a serious stumbling stone in the development of complex modern web services.

We present in this chapter a runtime system built on top of the CGI protocol that among other features has support for sessions and concurrency control. First, we motivate the need for a runtime system such as the one presented here. This is done by presenting its advantages over a simple CGI script based solution. Afterwards, a description of the runtime system, its different parts, and its dynamic behavior is given. We round off with a discussion of related work, a conclusion, and directions for future work.

In the appendices, we briefly describe an implementation of the suggested runtime system. Also, we give a short presentation of <bigwig> [33], which is a tool for producing interactive web services that makes extensive use of the self-contained runtime system package.

3.2 Motivation

The technology of plain CGI scripts lacks several of the properties one would expect from a modern programming environment. In the following we discuss various shortcomings of traditional CGI programming and motivate our solution to these problems, namely the design of an improved runtime system built on top of the standard CGI protocol.

3.2.1 The session concept

First, we will describe and motivate the concept of an interactive web service.

The HTTP protocol was originally designed for browsing *static* documents connected with hyperlinks. CGI together with forms allows *dynamic* creation of documents, that is, the contents of a document are constructed on the server at the time the document is requested. Dynamic documents have many advantages over static documents. For instance, the contents of the documents can be *tailor-made*, and *up-to-date*.

A natural extension of the dynamic-document model is the concept of *interactive* services, which is illustrated in Figure 3.1. Here the client does not

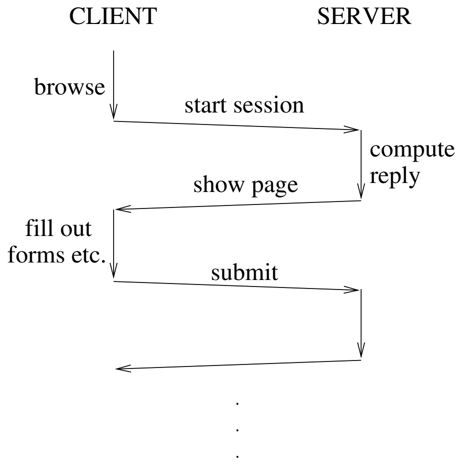


Figure 3.1: An interactive web session

browse a number of more or less independent statically or dynamically generated pages but is guided through a *session* controlled by a session thread on the server. This session can involve a number of user interactions. The session is initiated by the client submitting a "start session" request. The server then starts a thread controlling the new session. This thread generates a reply page which is sent back to the client. The page typically contains some input fields that are filled in by the client. That information is sent to the server, which then generates the next reply, and so on, until the session terminates.

This session concept allows a large class of services to be defined. However, a number of practical problems needs to be solved in order to implement this model on top of the CGI model.

3.2.2 CGI scripts and sequential session threads

As explained above, a web service session consists of a sequential computation that along the way presents information to the client and waits for replies. However, CGI is a state-less protocol, meaning that execution of a CGI script only lasts until a page is shown to the web client. This fact makes it rather tedious to program larger web services involving many client interactions. The sequential computation has to be split up into the small bits of computation that happen between client interactions. Each of these small bits will then constitute a CGI script or an instance of a CGI call.

Furthermore, to achieve persistency of the local state, one has to store and restore it explicitly between CGI-calls, for instance "hidden" in the web page sent to the client. For simple services where the full session approach is not needed this stateless-server approach might be preferable, but it is clearly inadequate in general.

Thus, the problem of forced termination of the CGI script at each client interaction is two-fold:

- Having to deal with many small scripts makes the *writing* and *maintenance* of a web service rather difficult because the control-flow of the service tends to become less clear from the program code.

- Starting up a whole new process every time a client interaction is performed is expensive in itself. On top of this a complete image of the local state has to be stored and restored each time a client interaction is required. The local state can potentially hold a lot of data, such as database contents. Thus one gets a *substantial overhead* in the execution of a web service.

We provide a simple solution which splits CGI scripts into two components, namely *connectors* and *session threads*. A connector is a tiny transient CGI script that redirects input to a session thread, receives the response from that thread, and redirects it back to the web client. The session threads are persistent processes running residently on the web server. They survive CGI calls and can therefore implement a long sequential computation involving several client interactions. The use of transient connectors and persistent session threads decreases the difficulty of writing and maintaining web services. Furthermore, it improves substantially on the overhead of the web server during execution of a service.

3.2.3 Other CGI shortcomings

Traditionally, reply pages from session threads are sent directly to the client. That is, the session thread (or the connector if using the system described above) writes the page to standard-output and the web server sends it on to the client browser. This basic approach imposes some annoying problems on the client:

- The client is not able to use "bookmarks" to identify the session, since selecting a bookmark might imply resending an old query to the server while the server expects a reply to a more recent interaction. It would be natural to the client if selecting a bookmarked session would continue the session from its current state. Obviously, this requires the server to always keep some kind of backup of the latest page sent to the client.

- In the session concept described in the previous section, it does not make sense to roll back execution of a session thread to a previous state. A thread can only be continued from its current point of execution. As a result of sending pages directly using the standard-output method, every new page shown to the client gets stacked up in the client's browser. This means that the stack of visited pages becomes filled up with references to outdated pages. One result is that the "back" button in the browser becomes rather useless.

We suggest a simple solution where—instead of sending the reply itself—the session thread writes its reply to a file visible to the client and then sends to the client a *reference* to the reply file. By choosing the same URL for the duration of the session, this reference can then function as an identification of that particular session. This solves both the problem with bookmarks and with the "back" button. Pressing "back" will now bring the client back to the web page where he started the session, which seems like a natural effect.

This method also opens up for an easy solution to another problem. Sometimes the server requires a long time to compute the information for the next page to be shown to the client. Naturally, the client may become *impatient* and lose interest in the service or assume that the server or the connection is down if no response is received within a certain amount of time. If confirmation in the form of a temporary response page is sent, the client will know that something is happening and that waiting will not be in vain.

This extra feature is implemented in the runtime system as follows. If a response is not ready within for instance 8 seconds, the connector responds with a reference to a temporary page (for instance saying "please wait") and terminates. This page will then automatically be loaded by the clients web browser and reload itself, say every 5 seconds. Once the session thread finishes its computation and the real response page is ready, the thread just replaces the temporary page with the real response page. This will have the effect that next time the page is reloaded, the real response page will be shown to the client.

This reloading can be done with standard HTML functionality. Of course the reloading causes some extra network traffic, but using this method is probably as close as one gets to server pushing in the world of CGI programming.

3.2.4 Handling safety requirements consistently

Another serious problem with traditional CGI programming is that concurrency control, such as synchronization of sessions and locking of shared variables, gets handled in an ad-hoc fashion. Typically, this is done using low-level semaphores supplied by the operating system.

As a result, web services often implement these aspects incorrectly resulting in unstable execution and sometimes even damaging behavior.

Our solution allows one to put safety requirements, such as mutual exclusion or much more complex requirements, separately in a centralized supervising process called the controller. This approach significantly simplifies the job of handling safety requirements. Also, since each of the requirements can be formulated separately, the solution is much more robust towards changes in various parts of the code.

It is generally considered inefficient and unsafe to have centralized components in distributed systems. However, in this case the bottleneck is more likely to be the HTTP/CGI server and the network than the safety controller. In spite of that, we do try to distribute the functionality of our safety controller as discussed in Section 3.5.

3.3 Components in the Runtime System

At any time there will be a number of *web clients* accessing the *HTTP/CGI server* through the CGI protocol. On the server side we will have a *controller* and a number of *session threads* running. The session threads access the global data and produce response pages for the web clients. From time to time a *connector* will be started as the result of a request from a web client. The connector will make contact with the running session thread. A connector is shut down again after having delegated the answer from a session thread back to the web client.

In the following we give a more detailed description of these components. For an overview of the components in the runtime system, see Figure 3.2.

Web clients Web clients are the users of the provided web service. They make use of the service essentially by filling in forms and submitting requests (HTTP/CGI) using a browser.

The HTTP/CGI server The HTTP/CGI server handles the incoming HTTP/CGI requests by retrieving web pages and starting up appropriate CGI scripts, in our case connectors. It also directs response pages back to the web clients.

Session threads Session threads are the resident processes running on the web server surviving several CGI calls. They represent the actual service code that implements the provided web service. They do calculations, search databases, produce response web pages, etc.

Connectors When a web client makes a request through the server, a connector is started up. If this request is the first one made, the controller starts

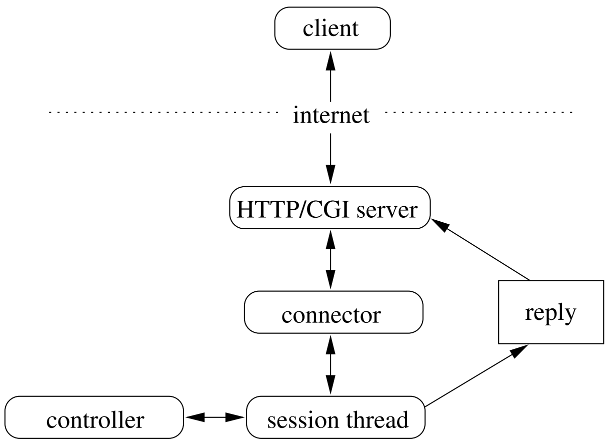


Figure 3.2: The runtime system

up a new session thread corresponding to the request made by the web client. Otherwise—that is, if the web client wants to continue execution of a running session thread—the connector notifies the relevant session thread that a request has been made and forwards the input to that thread.

Reply pages Each session thread has a designated file which contains the current web page visible to the client of the session. When writing to this file, the whole contents is through a buffer updated atomically since the client may read the file at any time.

The controller The controller is a central component. It supervises session threads and has the possibility of suspending their execution at various points. This way it is ensured that the stated safety requirements are satisfied.

Furthermore, the runtime system also contains a *global-state database* (could be the file-system or a full-fledged database), and a *service manager*, which takes care of garbage-collecting abandoned session threads and other administrative issues.

3.4 Dynamics of the Runtime System

In this section we describe the dynamic behavior of the runtime system. We start by explaining the overall structure of the execution of a session thread. Starting from this, we present each of the possible thread transitions.

First, it is described how a session thread is started. Then, transitions involving interaction with a web client, that is, showing web pages and getting replies, are dealt with. Finally, the transitions involving interaction with the controller are presented.

For each transition we give a description of the components involved and their interaction.

3.4.1 Execution of a thread

The lifetime of a session thread is depicted in the diagram in Figure 3.3. When

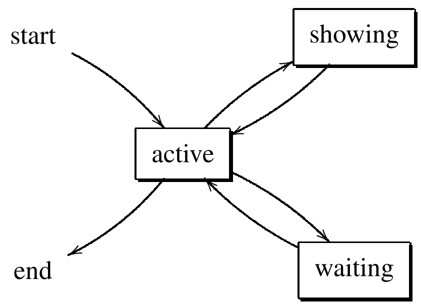


Figure 3.3: Possible states and transitions for a session thread

a thread is first started, it enters the state *active*. Here it can do all sorts of computations.

Eventually it reaches a point where it has composed a response HTML page. This page is shown to the web client and the thread enters the state *showing*. Here it waits for the web client to respond via yet another HTTP/CGI request. Upon re-submission the thread reenters the state *active* and resumes execution.

Note that in the world of naive CGI programming when moving from *active* to *showing* and back one would have to store a complete image of the local state before terminating the script. Then, when started again a new process would be started and the local state would have to be reconstructed from the image that was saved. This substantial overhead of saving and restoring local state is avoided completely by the use of transient connectors and resident threads.

While in state *active* a thread can get to a point in execution where safety critical computation, such as accessing a shared resource, needs to be carried out. When reaching such a point the thread asks the controller for permission to continue and enters the state *waiting*. When permission is granted from the controller the thread reenters the *active* state and continues execution.

With a traditional approach one would have to merge the code implementing the intricate details dealing with concurrency control with the service code. This intermixing would in addition to substantially reducing the readability of the code also increase the risk of introducing errors. Our solution separates the code dealing with concurrency control from the service code.

When the session is complete, the thread will leave the state *active* and end its execution.

3.4.2 Starting up a session thread

This section describes the transition from *start* to *active*.

When a new web client makes an HTTP/CGI request, the server will start up a new connector as a CGI script. Since this request is the first one made by the web client, a new thread is started according to the session name given in the request. As will be described later, a response page will be sent back to the client when the thread reaches a show call or a certain amount of time, for instance 8 seconds, has passed.

When a session thread is initiated or when it moves from *showing* to *active*, the contents of the reply file is immediately overwritten by a web page containing a "reply not ready—please wait" message and a "refresh" HTML command. The "refresh" command makes the browser reload the page every few seconds until the temporary reply file is overwritten by the real reply as described in the following section. The default contents of the "please wait" page can be overridden by the service programmer by simply overwriting the reply file with a message more appropriate for the specific situation.

3.4.3 Interaction with the client

During execution of a running thread the service can show a page to the web client and continue execution when receiving response from the client. In the following we describe these two actions.

Showing a page

This section describes the transition from *active* to *showing*.

During execution of a session thread one can do computations, inspect the input from the client, produce response documents, etc. When a response document has been constructed and the execution reaches a point where the page is to be shown to the client, the following actions will be taken:

1. First, the document to be shown is written to the reply file as indicated in Figure 3.2. This file always contains a "no cache" pragma-command, so that the client browser always fetches a new page even though the same URL is used for the duration of the whole session. Unfortunately we thereby lose the possibility of browser caching, but being restricted to building on top of existing standards we cannot get it all.

2. If the connector, that is, the CGI script started by the web client, has not already terminated due to the 8 second timeout, the session thread tells it that the reply page is ready. After this, the thread goes to sleep.

3. When the connector either has been waiting the 8 seconds or it receives the "reply ready" signal from the session thread, the connector writes a location-reference containing the URL for the reply page onto standard-output (using the CGI "location" feature), and then dies.

4. Finally, the HTTP/CGI server will transmit the URL back to the web clients browser which then will fetch the reply page through the HTTP/CGI server and show it to the client.

In Figure 3.2, these actions describe a flow of data starting at the session thread and ending at the client.

Receiving client response

This section describes the transition from *showing* to *active*.

While the session thread is sleeping in the showing state, the web client will read the page, fill out appropriate form fields, and resubmit. This will result in the following flow of data from the client to the session thread (see Figure 3.2):

1. First, a request is made by the client via the CGI protocol. This request can be initiated either by clicking on a link or by pressing a submit button.

2. As a result, the HTTP/CGI server starts up a CGI script, that is, a connector.

3. The connector will then see that the client is already associated with a running thread and thus wake up that sleeping session thread and supply its new arguments.

3.4.4 Interaction with the controller

The controller allows the programmer to restrict the execution of a web service in such a way that stated safety requirements are satisfied.

Threads have built-in checkpoints at places where safety critical code is to be executed. At these checkpoints the thread must ask the controller for permission to continue. The controller, in turn, is constructed in such a way that it restricts execution according to the safety requirements and only allow threads that are not about to violate the requirements to continue.

In the following we describe in further detail the controller itself, what happens when session threads ask for permission, and how permission is granted by the controller.

The controller

The controller consists of three parts: some control logic, a number of checkpoint-event queues, and a timeout queue. Figure 3.4 gives an overview of the controller.

The control logic The control logic is the actual component representing the safety requirements. It controls whether events are enabled, and hence when the various session threads may continue execution at checkpoints. One could imagine various approaches, such as, the use of finite state machines or petri-nets. For that reason, the internals of the control logic are not specified

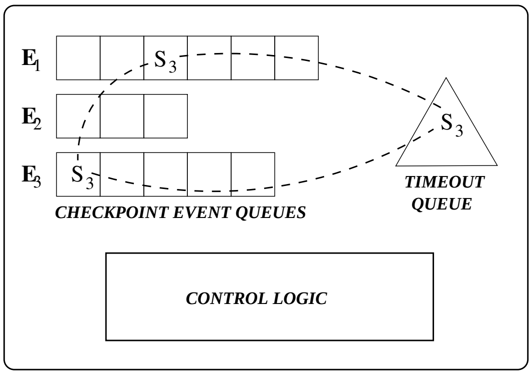


Figure 3.4: Components of the controller

here. The only requirement is that the interface must contain the following two functions available to the runtime system:

- `check_enabled` — takes a checkpoint-event ID as argument and replies whether that event is currently enabled.
- `event_occurred` — takes the ID of an enabled checkpoint-event as argument and updates the internal state of control logic with the information that the event has occurred.

We explain in the following how these functions are used in the controller.

Checkpoint-event queues The *checkpoint-event queues* form the interface to the running threads of the service. There is a queue for each possible checkpoint event. When a thread reaches a checkpoint it asks the controller for permission to continue by adding its process-ID onto the queues corresponding to the events it wants to wait for at the checkpoint.

Timeout queue As an extra feature one can specify a *timeout* when asking the controller for permission to continue. For this purpose the controller has a timeout queue. If permission is not granted within the specified time bound, the controller wakes up the thread with the information that permission has not been granted yet, but a timeout event has occurred. The specified timeouts are put in the special timeout queue (which is implemented as a priority queue).

Asking for permission at checkpoints

This section describes the transition from *active* to *waiting*.

As mentioned earlier, one has the possibility of adding checkpoints to session code where critical code is to be executed. The runtime system interface makes some functions available to the service programmer for specifying checkpoints. Conceptually, the programmer uses them to specify a “checkpoint statement” as illustrated with an example in Figure 3.5. This example would have the effect

```
wait {
  case E1:
    ...
  case E3:
    ...
  timeout 20:
    ...
}
```

Figure 3.5: A checkpoint example

that whenever a thread instance of this session reaches this point it will do the following:

1. First, it will tell the controller that it waits for either an E_1 event, an E_3 event, or a timeout of 20 seconds.
2. Having sent this request to the controller, the thread goes to sleep waiting for a response.

Controller actions

When the controller is up and running, it loops doing the following:

- If it receives a request to pass a checkpoint from a client, the controller pushes the ID of the client onto the appropriate queues. These entries are chained so that later, when permission is granted, they can all be removed at once. Figure 3.4 illustrates the effect of the example from Figure 3.5 where entries belonging to a session, S_3, are in the E_1, E_3 and TIMEOUT queues.
- If a timeout has occurred, the controller deletes the affected entries in the queues and informs the involved thread.
- Otherwise, it will look for an enabled event using the `check_enabled` function from the control logic. If the queue corresponding to an enabled event is non-empty then the controller makes the event occur by doing the following:
  1. It removes the linked entries with the thread-ID of the enabled event from the respective queues,
  2. tells the control logic that the event has occurred using the function `event_occurred`, and
  3. wakes up the involved thread with a "permission granted" signal containing the name of the event.

If several events become enabled, a token-ring scheduling policy is used. This ensures fairness in the sense that if a thread waits for an enabled event, it will at some point be granted permission to continue.

Permission granted

This section describes the transition from *waiting* to *active*.

Having sent a request for permission to continue the thread is sleeping, waiting for the controller to make a response. If a "permission granted" signal is sent to the thread, it wakes up and continues, branching according to the event signaled by the controller. In the example checkpoint in Figure 3.5, if the controller grants permission for an $\mathtt{E}_1$ event, execution is continued at the code following `case` $\mathtt{E}_1$. If the controller sends a "timeout" signal, execution continues after `timeout`.

3.5 Extending the Runtime System

The runtime system described in the previous sections can be extended in several ways. The following extensions either have been implemented in an experimental version of the runtime system package or will be in near future. With these extensions, we believe that we begin reaching the limits of what is possible with the standard CGI protocol and the current functionality of standard browsers.

Distributed safety controller

To smoothen presentation, we have so far described the controller as one centralized component. In most cases it is possible to divide the control logic into independent parts controlling disjoint sets of checkpoint events. The controller can then be divided into a number of distributed control processes [116]. This way the problem of the controller being a bottleneck in the system is successfully avoided.

Service monitors

Using the idea of connectors and controllers, one can construct a "remote service monitor", that is, a program run by a super-client, which is able to access logs and statistics information generated by the connectors and controllers, and to inspect and change the global state and the state of the control logic in the controllers. This can be implemented by having a dedicated *monitor process* for each service.

Secure communication

The system presented here is quite vulnerable to hostile attacks. It is easy to hijack a session, since the URL of the reply file is enough to identify a session. A simple solution is to use random keys in the URLs, making it practically impossible to guess a session ID. Of course, all information sent between the clients browser and the server, such as the session ID and all data written in forms, can still be eavesdropped. To avoid this, we have been doing experiments with cryptography, making all communication completely secure in practice. This requires use of browser plug-ins, which unfortunately has not been standardized. The protocols being used in the experiments are RSA, DES3, and RIPE-MD160. They prevent hijacking, provide secure channels, and verify user ID—all transparently to the client.

Document clusters

In the session concept illustrated in Figure 3.1, only one page is generated and shown to the client at a time. However, often the service wants to generate a whole "cluster" of linked documents to the client and let the client browse these documents without involving the session thread. With the current implementation, a solution would be to program the possibility of browsing the cluster into the service code—inevitably a tedious and complicated task.

Document clusters can be implemented by simply having a reply file for each document in the cluster. Recall, however, that in the presented setup, the name of the reply file was fixed for the duration of a session. That way, the history buffer of the browser got a reasonable functionality. Therefore, to get that functionality we need a somewhat different approach: the reply files are not retrieved directly by the HTTP server but via a connector process. This connector receives the ID of the session thread in the CGI query string and the document number in a hidden variable.

Single process model

If all server processes (the session threads, safety controllers, etc.) are running on the same machine, that is, the possibility of distributing the processes is not being exploited, they might as well be combined into a single process using lightweight threads. This decreases the memory use (unless the operating system provides transparent sharing of code memory) and removes the overhead of process communication. The resulting system becomes something very close to being a dedicated web server. The important difference being that it still builds upon the CGI protocol.

3.6 Related Work

The idea of having persistent processes running residently on the server is central in the FastCGI [80] system. One difference is that FastCGI requires platform- and server-dependent support, while our approach works for all servers that support CGI. Also, our runtime system is tailored to support more specific needs.

A more detailed and formal description of how one can make use of safety requirements written separately in a suitable logic can be found in [14, 116]. A language for writing safety requirements is presented, the compilation process into a safety controller is described, and optimizations for memory usage and flow capacity of the controller are developed.

The Mawl language [9, 28, 67] has been suggested as a domain-specific language for describing sequential transaction-oriented web applications. Its high-level notation is also compiled into low-level CGI scripts. Mawl directly provides programming constructs corresponding to global state, dynamic document, sessions, local state, imperative actions, and client interactions. This system shows great promise to facilitate the efficient production of reliable web services. While Mawl thus offers automatic synthesis of many advanced concepts, it still relies on standard low-level semaphore programming for concurrency control. Also, it does not have a FastCGI-like solution but in instead it is possible to compile a service into a dedicated server for that particular service. Though being faster than using simple CGI scripts this solution is, as opposed to using a FastCGI-like solution, not easily ported between different machine architectures.

3.7 Conclusions and Future Work

The implementation as briefly described in Section 3.8 constitutes the core of the <bigwig> tool which currently is being developed at BRICS. In the <bigwig> tool, the runtime system we propose here has shown to provide simple and efficient solutions to problems occurring more and more often due to the increased use of interactive web services. Furthermore, the session concept seems to constitute a framework which is very natural to use for designing complex services. By basing the design of the runtime system on very widely used protocols, the system is easy to incorporate. The further development of the runtime system can be followed on the <bigwig> homepage [33].

3.8 Implementation

A UNIX version of the runtime system has been implemented (in C) as a package "runwig" containing the following components (corresponding to Figure 3.2):

- The *connector*. It provides connection between the other components and the clients through the HTTP/CGI server.
- The *safety controller*, which handles syncronization and concurrency control. For the reasons described in Section 3.4.4, the control-logic is not included in the package but needs to be supplied separately.
- The *runtime library*, which is linked into the service code. It provides functions for easy interaction with the other components.

An experimental version of the runtime package implements the extensions described in Section 3.5. The runwig package—including all source code, detailed documentation, and examples—is available at `http://www.brics.dk/bigwig/runwig/`.

Chapter 4

Typed Dynamic Documents

Contents

In this chapter we describe a type system for dynamic Web documents. Part of the presentation in this chapter appears in the proceedings of Principles of Programming Languages, 2000 (POPL'00). It is joint work with Michael I. Schwartzbach.

Many interactive Web services use the CGI interface for communication with clients. They will dynamically create HTML documents that are presented to the client who then resumes the interaction by submitting data through incorporated form fields. This protocol is difficult to statically type-check if the dynamic document are created by arbitrary script code using `printf`-like statements. Previous proposals have suggested using static document templates which trades flexibility for safety. We propose a notion of typed, higher-order templates that simultaneously achieves flexibility and safety. Our type system is based on a flow analysis of which we prove soundness. We present an efficient runtime implementation that respects the semantics of only well-typed programs. This work is fully implemented as part of the <`bigwig`> system for defining interactive Web services.

4.1 Introduction

The HTTP protocol was originally designed for browsing *static* documents connected with hyperlinks. CGI together with HTML forms allows *dynamic* creation of documents whose contents are constructed on the server at the time the document is requested [40].

Interactive Web services make intensive use of dynamic documents when communicating with Web clients. A standard way of implementing such services is through CGI-scripts (often written in Perl [131]) that dynamically generate documents simply by printing their contents directly to standard output.

This technique is fraught with dangers, since it is impossible to statically verify that an arbitrary script will output a legal and consistent HTML document. First, the presented document may be garbled if the generated output does not contain properly balanced HTML tags. Second, each presented document is typically an HTML form that prompts the client for input and contains a submit button that allows the service to resume execution. If the input fields are dynamically generated, then they might not correspond to those that the resuming service expects.

There are alternatives to the CGI protocol, such as ASP [34], PHP [10], and Java Servlets [51], but the protocol remains popular since it is the only mechanism that is ubiquitously supported by all servers on all platforms. Thus, it remains relevant to provide static safety for CGI-programs.

The domain-specific language MAWL provides high-level mechanisms for programming interactive Web services [9, 67]. The problems with dynamic documents are addressed through the introduction of static document templates that behave quite similarly to typed functions. The arguments are integer or string values that are plugged into named gaps in the template before it is presented. The result from the client is a tuple of simple values corresponding to the input fields that the client is requested to fill in. The simple type system of MAWL then guarantees that the templates are used correctly.

On the downside, MAWL has sacrificed much flexibility in order to obtain safety. For many uses the document templates are too static, since they are essentially constant documents apart from the fixed collection of gaps that can be plugged with simple values. MAWL partially alleviates this shortcoming by allowing special iteration gaps that can be filled with lists of values. However, MAWL cannot construct a dynamic document that uses nested lists to display an unbounded hierarchical structure such as the active threads in a bulletin board or the directory of a file system.

The language Guide [73], which is inspired by MAWL, uses a similar template mechanism with gaps that are plugged with simple values, but without static type checking.

In this chapter we propose a notion of *higher-order* document templates that generalize those of MAWL and Guide while preserving static safety.

Documents are first-class values that may be computed and stored in variables. A document may contain named gaps that are placeholders for either HTML fragments or string attributes in tags. Such gaps may at runtime be plugged with concrete values that are themselves templates. Since those values may contain further gaps, this is a highly dynamic mechanism for building documents reminiscent of higher-order functions [56]. As input fields may now appear dynamically, they are much harder to keep track of.

To allow a sufficiently flexible use, we introduce a sophisticated notion of document types and employ a flow-sensitive type checker. As a further benefit, we introduce an efficient runtime representation that is sound for only well-typed programs, for which it represents document values with almost perfect sharing in such a way that the plug operation takes constant time only while the show command remains efficient.

Our proposal has been fully implemented in the <`bigwig`> [15] project which like MAWL provides a high-level language for specifying interactive Web services. This chapter also provides a case study in using the flow analysis framework of [91] in a domain-specific setting.

In Section 4.2 we give a stylized version of a language with dynamic documents; Section 4.3 describes the notion of document types; Section 4.4 presents and proves correctness of the flow-based type checker; and Section 4.5 specifies the efficient runtime implementation.

4.2 The Language

We consider a simple imperative language DynDoc which corresponds to the minimal fragment of <`bigwig`> that deals with dynamic documents. In examples, we will allow ourselves to use a richer syntax drawn from the full language, which uses a C-like syntax.

4.2.1 Syntax

We first give a description of the extended HTML syntax that we use. From that we define the central notion of a well-formed document. Then, the abstract syntax of our simple imperative language DynDoc is given.

Gaps The HTML syntax that we use is extended in two ways. First, we introduce two kinds of named *gaps*:

```
This <[foo]> <b>is <[bar]></b> you know!
Some pictures <img src=[baz]> of funny ducks.
```

Here, `foo` and `bar` are names of gaps that can be plugged with HTML fragments, while `baz` is a gap that can be plugged with an attribute string value.

An HTML document constant, *const*, is thus ideally a tree structure conforming to the following abstract syntax:

$$
\begin{aligned}
\mathit{const} &::= \texttt{<}\mathit{tag}\ \mathit{attr}^{*}\texttt{>}\ \mathit{const}^{*}\ \texttt{</}\mathit{tag}\texttt{>} \mid \texttt{<[}y\texttt{]>} \mid s \\
\mathit{attr} &::= \mathit{name}\texttt{=}\mathit{value} \mid \mathit{name}\texttt{=[}y\texttt{]}
\end{aligned}
$$

where *tag* ranges over HTML tags (the opening and closing tags must match), *name* over attribute names, *value* over attribute values, and y over gap names. An HTML document is thus a tree structure where the nodes consist of a tag name, an attribute list of named values or gaps, and a number of sub-trees. The leaves are either strings or named gaps. Note that in the examples, the concrete syntax is not quite as restrictive, such as the example above, where the `img` tag does not have a closing tag.

Tuples Second, we extend the HTML form input fields with a mechanism for collating input values:

```
<tuple name="order">
  Pizza Napoli <input type="hidden" name="pizza" value="napoli"><br>
  Amount <input type="text" name="amount"><br>
  Extra cheese?
  Yes <input type="radio" name="cheese" value="yes">
  No <input type="radio" name="cheese" value="no">
</tuple>
<p>
<tuple name="order">
  Pizza Roma <input type="hidden" name="pizza" value="roma"><br>
  Amount <input type="text" name="amount"><br>
  Extra cheese?
  Yes <input type="radio" name="cheese" value="yes">
  No <input type="radio" name="cheese" value="no">
</tuple>
```

The tuples are viewed as a single input field named `order`. This extension allows multiple occurrences of the same input field name to appear in a document.

When the field values are received, they are automatically transformed into tuples in a relation with schema {`pizza`, `amount`, `cheese`}. (If the tuple occurs only once in a document, the field values are transformed into a single tuple.) Tuples cannot be nested and are not allowed to contain checkboxes or HTML gaps.

These two HTML extensions are purely internal to the DynDoc language (and to <`bigwig`>); only standard documents are presented to the Web client.

Well-formed HTML Throughout this chapter we will be working with the notion of well-formed documents. A document, *const*, is *well-formed* if the following requirements are satisfied. All gap names occur at most once in *const*. If an input field name occurs more than once, all the occurrences must be of the same kind, which has to be either radio, checkbox, or tuple. If of kind tuple, the occurrences must have the same associated schema. In the following **HTML** denotes the set of well-formed HTML documents.

Abstract syntax For the formal presentation we introduce the following syntactic categories:

$b \in \mathbf{BExp}$	boolean expressions	$d \in \mathbf{DExp}$	document expressions
$s \in \mathbf{StrExp}$	string expressions	$S \in \mathbf{Stm}$	statements

Boolean and string expressions will not be further defined whereas **DExp** and **Stm** will be defined later by a grammar.

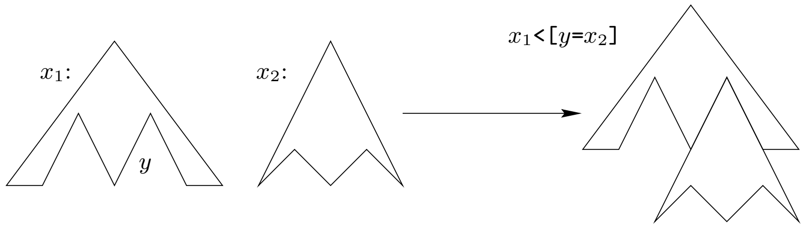
For a given program S_* the following finite sets of program variables, document variables, gap names, and field names are given:

$v \in \mathbf{PVar}$	program variables	$y \in \mathbf{GName}$	gap names
$x \in \mathbf{DVar}$	document variables	$z \in \mathbf{FName}$	field names

The syntax of DynDoc is completed by the following *abstract syntax* defining document expressions and statements:

$$\begin{array}{rl} d ::= & x \mid \mathit{const} \mid x_1\texttt{<[}y\texttt{=}x_2\texttt{]} \mid x\texttt{<[}y\texttt{=}s\texttt{]} \\ S ::= & [x = d]^l \mid [\texttt{skip}]^l \mid S_1\,\texttt{;}\,S_2 \mid \texttt{if}\,[b]^l\,S_1\,\texttt{else}\,S_2 \mid \texttt{while}\,[b]^l\,S \\ \mid & [\texttt{show}\;x\;\texttt{receive}\,\texttt{[}v_1 = z_1, \ldots, v_n = z_n\texttt{]}]^l \end{array}$$

A document expression can be a document variable, a constant HTML document with zero or more named gaps, or a plug operation. The result of the plug operation $x_1\texttt{<[}y\texttt{=}x_2\texttt{]}$ is the value of x_1 where its y gap is replaced by the value of x_2:



There is no problem in having the plug operation accept document expressions rather than document variables. However, to simplify the presentation we have chosen to work with simpler, normalized document expressions.

A statement is either a document variable assignment, a `skip` statement, a sequence of statements, a conditional, a `while` loop, or a `show` call. The `show` call [`show` x `receive`[$v = z$]]^l allows interaction with the client: The contents of the document variable x is shown to the client, and when the client submits the reply, the received value of the form field named z is assigned to the program variable v. Again, we require for simplicity that the argument to `show` is a document variable rather than a document expression.

Later, we will associate document type flow information with assignment statements, tests (in conditionals and loops), `skip`, and `show` calls. For simplicity, we will assume that these *elementary blocks* are initially assigned distinct labels, $l \in \mathbf{Lab}$. The occurrence of $[\cdot]^l$ in the syntax above denotes this explicit labeling.

Note that this is not a complete language, but merely a sketch suitable for formal analysis. The `skip` statement can be thought of as representing other statements that are not relevant for type-checking documents.

Examples A simple example of the plug operation in <`bigwig`> syntax is:

```
{ const html H = <html bgcolor=[color]>Hello <[what]>!</html>;
  show H<[color="red",what="world"];
}
```

Here, `color` is a string gap and `what` is an HTML gap. Another standard example builds an HTML table from a vector of tuples:

```
{ schema Student {string name; int age};
  vector Student S;
  const html Row = <html><tr><td><[name]></td>
                             <td><[age]></td>
                         </tr>
                         <[row]>
                   </html>;
  html T = <html><table><[row]></table></html>;
  int i;
  for (i=0; i<|S|; i++)
    T = T<[row = Row<[name=S[i].name,age=S[i].age]];
  show T;
}
```

4.2.2 Semantics

For the purpose of giving a formal small-step operational semantics of DynDoc we shall first define a number of categories and functions.

First define a *state* as a mapping from document variables to well-formed HTML documents:

$$\sigma \in \mathbf{State} = \mathbf{DVar} \to \mathbf{HTML}.$$

Note that for simplicity, program variables are not considered part of state. To deal with boolean, string, and document expressions we require the following three semantic functions

$$\begin{aligned}\mathcal{B} &: \mathbf{BExp} \to \mathbf{State} \to \{\mathsf{true}, \mathsf{false}\},\\ \mathcal{S} &: \mathbf{StrExp} \to \mathbf{State} \to \mathbf{String}, \text{ and}\\ \mathcal{D} &: \mathbf{DExp} \to \mathbf{State} \hookrightarrow \mathbf{HTML},\end{aligned}$$

where **String** denotes the set of strings. We assume $\mathcal{B}$ and $\mathcal{S}$ are given and define the partial map $\mathcal{D}$ below.

Semantics of document expressions To give semantics to document expressions we first need to define functions that will allow us to inspect the gaps and input fields of a document. For this we define the set $\mathbf{GKind} = \{\mathsf{nogap}, \mathsf{html}, \mathsf{string}, \mathsf{error}\}$ to be the set of gaps kinds. From that we can define the function

$$gap : \mathbf{HTML} \to \mathbf{GName} \to \mathbf{GKind}$$

which maps well-formed HTML documents into a map that associates gap names with gap kinds. The *gap* function is defined inductively as follows:

$$\begin{aligned}&gap(\texttt{<}tag\ name\texttt{=[}y\texttt{]>}\ const_1 \dots const_n\ \texttt{</}tag\texttt{>}) =\\ &\qquad [y \mapsto \mathsf{string}] \oplus_g gap(const_1) \oplus_g \cdots \oplus_g gap(const_n)\\ &gap(\texttt{<}tag\ \dots\texttt{>}\ const_1 \dots const_n\ \texttt{</}tag\texttt{>}) =\\ &\qquad [\,] \oplus_g gap(const_1) \oplus_g \cdots \oplus_g gap(const_n)\\ &gap(\texttt{<[}y\texttt{]>}) = [y \mapsto \mathsf{html}]\\ &gap(s) = [\,]\end{aligned}$$

where $[\,]$ denotes the function that maps everything to nogap, $[y \mapsto kind]$ the function that maps everything but y (which is mapped to $kind$) to nogap, where $kind$ is any gap kind, and $\oplus_g$ on gap kinds is defined as follows (and is lifted pointwise to gap maps — that is, $\oplus_g$ is overloaded):

$$a \oplus_g b = \begin{cases} a & \text{if } b = \mathsf{nogap},\\ b & \text{if } a = \mathsf{nogap},\\ \mathsf{error} & \text{otherwise.}\end{cases}$$

Note that by definition of well-formedness, *gap* is a well-defined function, that is, the error case of $\oplus_g$ will never occur for *gap*. Thus for a given well-formed document d and a gap name y, $gap(d)(y)$ gives the unique kind (string or html) of the y gap in d and if d does not have a y gap, the result is nogap.

Similarly we consider the set of input field kinds:

nofield	field name not associated with a field
ordinary	select menus, text, textarea, and hidden fields
radio	radio buttons
checkbox	check boxes and select multiple
$\textsf{tuple}(F_i)$	single tuple field with schema F_i
$\textsf{rel}(F_i)$	more than one tuple field with schema F_i

where $F_1, \ldots, F_n$ are the schemas occurring in S_*. Again, we shall use the error kind to indicate lack of well-formedness. Let **FKind** denote the set of field kinds. We can then define the function

$$\mathit{field} : \mathbf{HTML} \to \mathbf{FName} \to \mathbf{FKind}$$

such that *field*(d)(z) is the field kind of the input field named z in the document d. More precisely, *field* is defined inductively as follows:

$$\mathit{field}(\texttt{<input name="}z\texttt{" type="radio"></input>}) = [z \mapsto \textsf{radio}]$$
$$\mathit{field}(\texttt{<input name="}z\texttt{" type="checkbox"></input>}) = [z \mapsto \textsf{checkbox}]$$
$$\mathit{field}(\texttt{<input name="}z\texttt{" type=...></input>}) = [z \mapsto \textsf{ordinary}]$$
$$\begin{aligned}&\mathit{field}(\texttt{<select name="}z\texttt{">}\ \mathit{const}_1 \ldots \mathit{const}_n\ \texttt{</select>})\\ &\qquad = [z \mapsto \textsf{ordinary}] \oplus_f \mathit{field}(\mathit{const}_1) \oplus_f \cdots \oplus_f \mathit{field}(\mathit{const}_n)\end{aligned}$$
$$\begin{aligned}&\mathit{field}(\texttt{<tuple name="}z\texttt{">}\ \mathit{const}_1 \ldots \mathit{const}_n\ \texttt{</tuple>})\\ &\qquad = [z \mapsto \textsf{tuple}((\mathit{field}(\mathit{const}_1) \oplus_f \cdots \oplus_f \mathit{field}(\mathit{const}_n))^{-1}(\{\textsf{ordinary}, \textsf{radio}\}))]\end{aligned}$$
$$\mathit{field}(\texttt{<}\mathit{tag}\texttt{>}\ \mathit{const}_1 \ldots \mathit{const}_n\ \texttt{</}\mathit{tag}\texttt{>}) = \mathit{field}(\mathit{const}_1) \oplus_f \cdots \oplus_f \mathit{field}(\mathit{const}_n)$$
$$\mathit{field}(\texttt{<[}y\texttt{]>}) = \mathit{field}(s) = [\,]$$

where $h^{-1}(y) = \{\, x \mid h(x) = y \,\}$ and the $\oplus_f$ operation on field kinds is defined as follows (again we can easily lift to field maps and thus also overload $\oplus_f$):

$$a \oplus_f b = \begin{cases} \textsf{rel}(F_i) & \text{if } a, b \in \{\textsf{tuple}(F_i), \textsf{rel}(F_i)\}, \\ a & \text{if } a = b \in \{\textsf{radio}, \textsf{checkbox}\}, \\ a & \text{if } b = \textsf{nofield}, \\ b & \text{if } a = \textsf{nofield}, \\ \textsf{error} & \text{otherwise.} \end{cases}$$

Again, by definition of well-formedness $\mathit{field}(d)(z)$ will never be error for a well-formed document d. Notice that if there are several occurrences of a tuple z (with schema F), the kind supplied by the *field* function for z is $\textsf{rel}(F)$. Otherwise, that is, if there is only one occurrence, the kind is $\textsf{tuple}(F)$. The remaining cases (nofield, ordinary, radio, and checkbox) should be straightforward.

With the *gap* and *field* functions in place, we can now define the semantic function $\mathcal{D}$:

$$\begin{aligned}
\mathcal{D}[\![x]\!]\sigma &= \sigma(x) \\
\mathcal{D}[\![const]\!]\sigma &= const \\
\mathcal{D}[\![x_1\texttt{<[}y\texttt{=}x_2\texttt{]}]\!]\sigma &= plug(\sigma(x_1), y, \sigma(x_2)) \\
\mathcal{D}[\![x\texttt{<[}y\texttt{=}s\texttt{]}]\!]\sigma &= strplug(\sigma(x_1), y, \mathcal{S}[\![s]\!]\sigma),
\end{aligned}$$

where $plug(d_1, y, d_2)$ is defined if and only if

- $gap(d_1)(y) = \mathsf{html}$;
- $\forall y' : (gap(d_1)[y \mapsto \mathsf{nogap}] \oplus_g gap(d_2))(y') \neq \mathsf{error}$; and
- $\forall z : (field(d_1) \oplus_f field(d_2))(z) \neq \mathsf{error}$.

If defined $plug(d_1, y, d_2)$ is the well-formed document resulting from replacing the y gap in d_1 with d_2. Intuitively, the first requirement ensures that the gap to be replaced is in fact there and the remaining two requirements ensure that the resulting document will be well-formed. Similarly, $strplug(d, y, s)$ is defined if and only if $gap(d)(y) \in \{\mathsf{html}, \mathsf{string}\}$ in which case $strplug(d, y, s)$ is just d with its y gap replaced by s.

Thus the meaning of a document variable is its associated value in the current state. The meaning of a constant is just the well-formed document constant itself. The meaning of $x_1\texttt{<[}y\texttt{=}x_2\texttt{]}$ is defined at σ only if $\sigma(x_1)$ has an HTML gap named y and if the resulting document is well-formed, that is, if the maps combining the gap and field maps of the two operands are total maps, that is, if they do not map any values to error. Similarly, $x\texttt{<[}y\texttt{=}s\texttt{]}$ is defined on σ if and only if $\sigma(x)$ has a y gap.

Semantics of statements Before we can define the configurations of our semantics and the semantic rules for statements we need to define the map:

$$typeof : \mathbf{PVar} \rightarrow \mathbf{PType}$$

which assigns a type to each of the program variables, that is, to each of the non-document variables of the program. The set of program variable types is the following: $\mathbf{PType} = \{\mathsf{string}, \mathsf{tuple}(F_1), \mathsf{rel}(F_1), \ldots, \mathsf{tuple}(F_n), \mathsf{rel}(F_n)\}$. Such types are native to the <`bigwig`> language.

We choose to use a small step semantics. This will allow us to reason about intermediate stages in a program execution (even for non-terminating programs). A *configuration* of the semantics is either a pair consisting of a statement and a state or it is simply a state in which case the configuration is called terminal. The *transitions* of the semantics, expressing how the configuration is changed by one step of computation, are of the form $\langle S, \sigma\rangle \rightarrow \langle S', \sigma'\rangle$ and $\langle S, \sigma\rangle \rightarrow \sigma$. Non-terminal configurations with no enabled transitions are called *stuck* configurations. In the semantics presented below, stuck configurations are reached only when errors occur in the execution. We would thus like to avoid stuck configurations, that is, for the type system that we will introduce later, we expect the property that type-checked programs will never enter stuck configurations.

The detailed definition of the semantics of statements is defined in Figure 4.1. We write $\sigma[x \mapsto d]$ for the state that is as σ except that x is mapped to d and

$$
\begin{array}{lll}
[asn] & \langle [x = d]^l, \sigma\rangle \to \sigma[x \mapsto \mathcal{D}[\![d]\!]\sigma] & \text{if } \mathcal{D}[\![d]\!]\sigma \text{ is defined} \\
[skip] & \langle [\mathtt{skip}]^l, \sigma\rangle \to \sigma & \\
[seq_1] & \dfrac{\langle S_1, \sigma\rangle \to \langle S_1', \sigma'\rangle}{\langle S_1\,;\, S_2, \sigma\rangle \to \langle S_1'\,;\, S_2, \sigma'\rangle} & \\
[seq_2] & \dfrac{\langle S_1, \sigma\rangle \to \sigma'}{\langle S_1\,;\, S_2, \sigma\rangle \to \langle S_2, \sigma'\rangle} & \\
[if_1] & \langle \mathtt{if}\,[b]^l\, S_1\,\mathtt{else}\, S_2, \sigma\rangle \to \langle S_1, \sigma\rangle & \text{if } \mathcal{B}[\![b]\!]\sigma = \mathsf{true} \\
[if_2] & \langle \mathtt{if}\,[b]^l\, S_1\,\mathtt{else}\, S_2, \sigma\rangle \to \langle S_2, \sigma\rangle & \text{if } \mathcal{B}[\![b]\!]\sigma = \mathsf{false} \\
[wh_1] & \langle \mathtt{while}\,[b]^l\, S, \sigma\rangle \to \langle (S\,;\,\mathtt{while}\,[b]^l\, S), \sigma\rangle & \text{if } \mathcal{B}[\![b]\!]\sigma = \mathsf{true} \\
[wh_2] & \langle \mathtt{while}\,[b]^l\, S, \sigma\rangle \to \sigma & \text{if } \mathcal{B}[\![b]\!]\sigma = \mathsf{false} \\
[show] & \langle [\mathtt{show}\; x\; \mathtt{receive}\, [v_1 = z_1, \ldots, v_n = z_n]]^l, \sigma\rangle \to \sigma & \\
 & \quad \text{if } \forall i:\ \mathit{typeof}(v_i) \lessdot \mathit{field}(\sigma(x))(z_i) \text{ and} & \\
 & \quad \forall z \in \mathbf{FName} \setminus \{z_1, \ldots, z_n\}:\ \mathit{field}(\sigma(x))(z) = & \mathsf{nofield}
\end{array}
$$

Figure 4.1: The semantics of statements.

the relation $\lessdot \subseteq \mathbf{PType} \times \mathbf{FKind}$ is defined as follows: $\mathsf{string} \lessdot \mathsf{radio}$, $\mathsf{string} \lessdot \mathsf{ordinary}$, $\mathsf{tuple}(F) \lessdot \mathsf{tuple}(F)$, $\mathsf{rel}(F) \lessdot \mathsf{nofield}$, $\mathsf{rel}(F) \lessdot \mathsf{tuple}(F)$, $\mathsf{rel}(F) \lessdot \mathsf{rel}(F)$, and $\mathsf{rel}(\{\mathit{name}\}) \lessdot \mathsf{checkbox}$. Note also, that due to the fact that program variables are not considered part of the state, the state σ remains the same after the execution of a `show-receive` statement in $[show]$.

That is, if the receiving variable is of type string, the corresponding field must be of kind radio or ordinary. If the variable is of type tuple, the field must be so as well, whereas if the variable is of type relation, then the field can either not appear in the document or appear as a tuple or a relation. If the variable is of kind relation with a singleton schema, the corresponding field can be a checkbox.

Notice, that the stuck configurations are caused either by $\mathcal{D}[\![d]\!]\sigma$ not being defined in $[asn]$, that is, by an illegal plug operation, or by non-matching program variables and document fields in $[show]$.

Intuitively, we wish to devise a type checker that allows as liberal a use of dynamic documents as possible, while guaranteeing that no errors occur, that is, while guaranteeing the following requirements:

- in x_1`<[`y`=`x_2`]`, x_1 must have an HTML gap named y;
- in x`<[`y`=`s`]`, x must have a gap named y;
- all constructed documents must be well-formed; and
- in a `show` call, the document fields must match the `receive` program variables.

4.3 Document Types

We introduce a somewhat complicated notion of document type. These types are not explicitly written by the programmer, but are inferred by the compiler using a global flow analysis.

A document type has two distinct components: a *gap map* and a *field map*. The gap map describes the gaps that are present in a document and, similarly, the field map describes its input fields. To facilitate the subsequent flow analysis, these maps are defined as points in two finite lattices.

4.3.1 Gap Maps

To each gap name in **GName** we associate for a given document value a gap kind. This is html if this name denotes an HTML gap, string if it denotes a string gap, and nogap if it is not associated with a gap. Furthermore, we extend **GKind** with the extra kinds $\perp$ (no information available yet) and error (an error has occurred). We impose a partial ordering on gap kinds thus making **GKind** a finite lattice (see Figure 4.2).

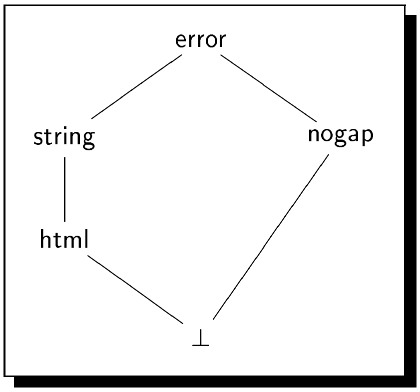


Figure 4.2: **GKind**, the finite lattice of gap kinds.

This particular ordering is carefully chosen to ensure that functions defined in Section 4.4 become monotone; intuitively, higher values describe more restricted capabilities of gaps. From **GKind** we define the notion of a *gap map*: **GMap** =

$\mathbf{GKind}^{|\mathbf{GName}|}$. Though elements of **GMap** are |**GName**|-tuples, we shall sometimes use the more intuitive function notation. That is, a gap map can be considered a function from **GName** to **GKind**. Gap maps are associated with document values and variables.

4.3.2 Field Maps

To each field name in **FName** we associate for a given document value a field kind that describes its presence and capabilities. It should be clear that our approach is quite flexible; for example, we also capture the `tuple` extension that we have introduced in <`bigwig`>.

We impose a partial order on the field kinds described earlier extended with a bottom element, $\perp$. Again, the ordering is carefully chosen to ensure that functions defined in Section 4.4 become monotone. The field kinds thus form a finite lattice, **FKind** (see Figure 4.3). As for gaps, we lift to field maps, $\mathbf{FMap} = \mathbf{FKind}^{|\mathbf{FName}|}$.

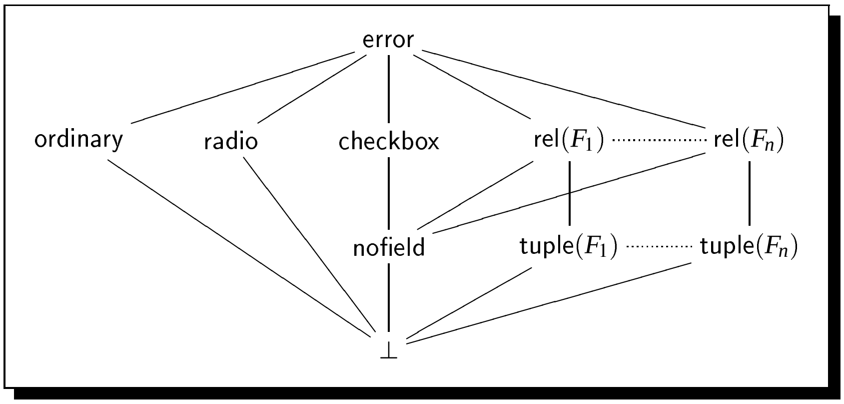


Figure 4.3: **FKind**, the finite lattice of field kinds.

4.3.3 Type environments

We shall define $\mathbf{DType} = \mathbf{GMap} \times \mathbf{FMap}$ to be the set of document types. A document type is thus a pair of a gap and a field map. Document types are associated with document values and variables. Given a document type, we need projection maps to extract the gap map and field map of the type: $\pi_g : \mathbf{DType} \to \mathbf{GMap}$ and $\pi_f : \mathbf{DType} \to \mathbf{FMap}$. Also, we shall denote by $gap \otimes field$ the map $(gap \otimes field)(d) = (gap(d), field(d))$.

For our analysis we need to associate information with each program point, that is, each labeled elementary block. For this, we introduce the notion of a *document type environment*: $\mathbf{DEnv} = \mathbf{DType}^{|\mathbf{DVar}|}$. Thus, a document type environment can be considered a map from the set **DVar** of document variables to the set of document types, **DType**.

Note that **GKind** and **FKind** are finite lattices. Since **GName**, **FName**, and **DVar** are all finite sets, both **GMap**, **FMap**, and **DEnv** inherit the property of being a finite lattice (with the obvious point-wise orderings).

4.3.4 Type Correctness

Our flow analysis will for each program point compute a document type environment that later is proven to soundly describe the possible values of document variables. The gap and field information is represented via a function solving the constraint set produced by the analysis:

$$A_{\text{entry}} : \mathbf{Lab} \rightarrow \mathbf{DEnv}.$$

Based on this function, we can determine if a program is type correct. The following overall condition ensures that only well-formed document values have been constructed: no gap or field map may contain the kind error. Furthermore, gap information is used to check plug operations:

- in $[x = x_1\texttt{<[}y\texttt{=}x_2\texttt{]}]^l$, we must have $\pi_g(A_{\text{entry}}(l)(x_1))(y) = \mathsf{html}$, that is, when executing the operation, x_1 must have an HTML gap named y;
- in $[x = x'\texttt{<[}y\texttt{=}s\texttt{]}]^l$, we must have $\pi_g(A_{\text{entry}}(l)(x'))(y) \in \{\mathsf{string}, \mathsf{html}\}$, that is, x' must have a gap named y;

Finally, the field information imposes the following type requirement on the statement $[\texttt{show}\ x\ \texttt{receive[}v_1 = z_1, \ldots, v_n = z_n\texttt{]}]^l$:

$$\mathit{typeof}(v_i) \lessdot \pi_f(A_{\text{entry}}(l)(x))(z_i) \text{ for all } i.$$

(The relation $\lessdot$ is defined in the section on semantics of statements.)

Example The following example in <bigwig> syntax is type correct and builds an unbounded hierarchical structure that corresponds to the active threads in a bulletin board:

```
{ schema Message {int id; int replyto; string subject};
  relation Message Board;
  const html X = <html><li><[id]>:<[sub]><ul><[replies]></ul></html>;
  const html Y = <html><[reply]><[replies]></html>;

  html View(int id, string subject) {
    int i;
    html H = X<[id=id,sub=subject];
    vector Message R;
    R = sort(select * from Board where (#.replyto==id); id);
    for (i=0; i<|R|; i++)
      H = H<[replies = Y<[reply=View(R[i].id,R[i].subject)]];
```

```
    return H;
  }

  show View(0,"");
}
```

4.4 Flow Analysis

The overall structure of our first-order forward data-flow analysis is quite standard. We generate a set of constraints for S_*. Through this system we define entry and exit environments for each labeled program point, relate them with monotone operations, and appeal to the general fixed-point theorem for finite lattices to obtain a minimal solution [91].

We present only an analysis for the simple imperative language DynDoc, but the techniques generalize in a straightforward manner to the monovariant interprocedural flow analysis that is required for the full <`bigwig`> language.

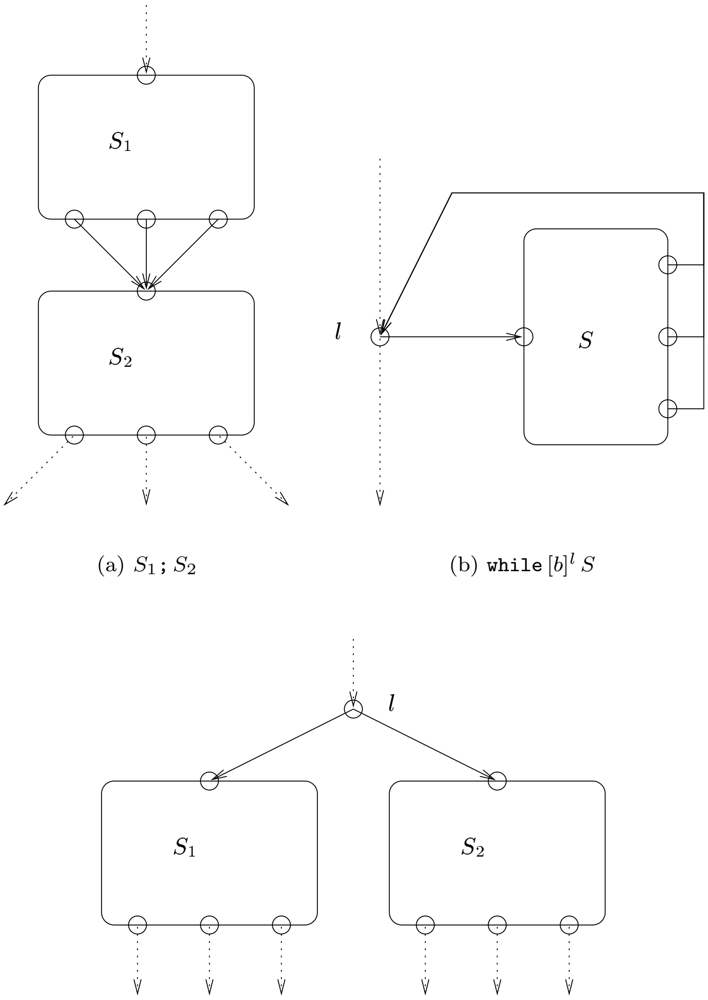
4.4.1 Terminology

In our type analysis we will be referring to the *initial label*, *init*(S), of a statement S. The initial label of a sequence, $(S_1\,;\,S_2)$, is the initial label of S_1. The initial label of the other statement kinds is the unique label literally occurring in that statement.

Furthermore, we shall define for a statement, S, the set of *final labels*, *final*(S). The final labels of a sequence are those of the second statement. The set of final labels for a conditional is the union of those of the two branches. The final label of a `while` statement is the label associated with the test. The final label of the other statement kinds is the unique label literally occurring in that statement.

Similarly, we define the set of *associated labels*, *labels*(S). The set of labels associated with a sequence is just the union of the labels of the statements in the sequence. The labels associated with a conditional $\texttt{if}\,[b]^l\,S_1\,\texttt{else}\,S_2$ is the union of $\{l\}$, *labels*(S_1), and *labels*(S_2). For the while statement, $\mathit{labels}(\texttt{while}\,[b]^l\,S) = \{l\} \cup \mathit{labels}(S)$. The label associated with an assignment, a skip, and a show statement is just the label literally occurring in that statement. As mentioned earlier, we let **Lab** denote the set of labels associated with the program of interest, that is, $\mathbf{Lab} = \mathit{labels}(S_*)$.

Finally, we assume to have at our disposal the flow graph, G, of the program of interest S_*. We have $G = (\mathbf{Lab}, \mathit{flow}(S_*))$, where $\mathit{flow}(S) \subseteq \mathbf{Lab} \times \mathbf{Lab}$ is the set of flow edges associated with the statement S. More specifically, the set of flow edges associated with an assignment, a skip statement, and a show statement is simply the empty set. For sequences, conditionals, and while loops, we define *flow*() as follows (see Figure 4.4):



(a) S_1 `;` S_2

(b) `while` $[b]^l\ S$

(c) `if` $[b]^l\ S_1$ `else` S_2

Figure 4.4: The flow of sequencing, while statements, and conditionals.

$$\begin{aligned}
\mathit{flow}(S_1\,;\,S_2) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{\, (l, \mathit{init}(S_2)) \mid l \in \mathit{final}(S_1) \,\} \\
\mathit{flow}(\texttt{if}\,[b]^l\, S_1\, \texttt{else}\, S_2) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{(l, \mathit{init}(S_1)), (l, \mathit{init}(S_2))\} \\
\mathit{flow}(\texttt{while}\,[b]^l\, S) &= \mathit{flow}(S) \cup \{(l, \mathit{init}(S))\} \cup \{\, (l', l) \mid l' \in \mathit{final}(S) \,\}
\end{aligned}$$

Since we will be defining a forward analysis, we shall make the simplifying assumption that the program of interest S_* has an *isolated entry*, that is, $\mathit{init}(S_*)$ has no incoming flow-edges. This can be ensured by prepending S_* with a `skip` statement.

4.4.2 The analysis

We introduce a constraint system via the functions $\mathsf{A}_{\text{entry}}$ and A_{exit} that map labels to environments, that is:

$$\mathsf{A}_{\text{entry}}, \mathsf{A}_{\text{exit}} : \mathbf{Lab} \to \mathbf{DEnv}$$

For $l = \mathit{init}(S_*)$, we require that $\mathsf{A}_{\text{entry}}(l)$ corresponds to the start state σ_0 of S_*. That is, for any x we have $\mathsf{A}_{\text{entry}}(l)(x) \sqsupseteq (\mathit{gap} \otimes \mathit{field})(\sigma_0(x))$. For all other l, $\mathsf{A}_{\text{entry}}(l)$ must be greater than or equal to each of the exit environments from incoming flow-edges, that is:

$$\mathsf{A}_{\text{entry}}(l) \sqsupseteq \bigsqcup \{\, \mathsf{A}_{\text{exit}}(l') \mid (l', l) \in \mathit{flow}(S_*) \,\}$$

For labels, l, associated with `skip` statements, conditionals, `while` loops, or `show` calls, we simply require $\mathsf{A}_{\text{exit}}(l) \sqsupseteq \mathsf{A}_{\text{entry}}(l)$. If a label l is associated with an assignment $[x = d]^l$, we impose:

$$\mathsf{A}_{\text{exit}}(l) \sqsupseteq \mathsf{A}_{\text{entry}}(l)[x \mapsto \mathcal{A}[\![d]\!](\mathsf{A}_{\text{entry}}(l))]$$

where the map $\mathcal{A}[\![\cdot]\!] : \mathbf{DExp} \to \mathbf{DEnv} \to \mathbf{DType}$ is defined as follows (using the function notation):

$$\begin{aligned}
\mathcal{A}[\![x]\!]\mathit{denv} &= \mathit{denv}(x) \\
\mathcal{A}[\![\mathit{const}]\!]\mathit{denv} &= (\mathit{gap} \otimes \mathit{field})(\mathit{const}) \\
\mathcal{A}[\![x_1\texttt{<[}y\texttt{=}x_2\texttt{]}]\!]\mathit{denv} &= \mathit{denv}(x_1)[y \mapsto \mathsf{nogap}] \oplus \mathit{denv}(x_2) \\
\mathcal{A}[\![x\texttt{<[}y\texttt{=}s\texttt{]}]\!]\mathit{denv} &= \mathit{denv}(x)[y \mapsto \mathsf{nogap}]
\end{aligned}$$

The notation $dt[y \mapsto \mathsf{nogap}]$ simply means the document type that acts as dt except for the gap map part which maps x to nogap. The monotone operators $\oplus_g$ and $\oplus_f$ are extended from the previous definitions to **GKind** and **FKind**, respectively, as follows:

$$a \oplus_g b = \begin{cases} \mathsf{error} & \text{if } a, b \in \{\mathsf{string}, \mathsf{html}\} \\ a & \text{if } b = \mathsf{nogap} \\ b & \text{if } a = \mathsf{nogap} \\ a \sqcup b & \text{otherwise} \end{cases} \qquad a \oplus_f b = \begin{cases} \mathsf{error} & \text{if } a = b = \mathsf{ordinary} \\ \mathsf{rel}(F) & \text{if } a = b = \mathsf{tuple}(F) \\ a & \text{if } b = \mathsf{nofield} \\ b & \text{if } a = \mathsf{nofield} \\ a \sqcup b & \text{otherwise} \end{cases}$$

They can easily be lifted to **GMap** and **FMap**, respectively. For **DType**, we define

$$(g_1, f_1) \oplus (g_2, f_2) = (g_1 \oplus_g g_2, f_1 \oplus_f f_2).$$

4.4.3 Solving the constraint system efficiently

Since we are only working with finite lattices and all our transfer functions are monotone, we can use standard techniques to find the smallest solution to the constraints generated from the program S_*. In this section, we describe how the so-called worklist algorithm is used to compute the smallest solution in terms of the least fixed point of a monotone function. Then we present an optimization that reduces the size of the flow graph substantially and thus the time needed to compute a solution. The optimization is successful mainly because we are performing a domain-specific analysis leaving large parts of the program and thus large parts of the flow-graph irrelevant for the analysis.

Classical fixed point computation In general we consider the two functions A_{entry} and A_{exit} as elements in $\mathbf{Lab} \to \mathbf{DEnv}$. If substituting the variables $\mathsf{A}_{\text{entry}}$ and A_{exit} with the elements A_{entry} and A_{exit} makes the constraint set true, that is, if $A \models \mathsf{A}^{\sqsubseteq}(S_*)$, we have found a solution. What we want is the smallest such solution. The smallest solution is found simply by considering the generated constraints as the definition of a monotone function, F, from the finite lattice $(\mathbf{Lab} \to \mathbf{DEnv}) \times (\mathbf{Lab} \to \mathbf{DEnv})$ to itself. Starting with the bottom element, that is, the pair consisting of two functions that both map every label to the bottom element of **DEnv**, we can iterate using F and find the smallest fixed point as $\bigsqcup F^n(\perp)$. By monotonicity, we can show by induction that $F^n(\perp) \sqsubseteq F^{n+1}(\perp)$. Thus, by finiteness the our lattice we have a safe way of computing the smallest solution to our constraint set. More specifically, we find the fixed point using the following standard worklist algorithm:

```
for all l ∈ Lab {A_entry(l) = A_exit(l) = ⊥_DEnv};
W = Lab;
while (W ≠ ∅) {
    l = remove_one(W);
    new_entry = ⊔{ A_entry(l') | (l', l) ∈ flow(S_*) };
    if (new_entry ≠ A_entry(l)) {
        A_entry(l) = new_entry;
        new_exit = F_l(new_entry);
        if (new_exit ≠ A_exit(l)) {
            A_exit(l) = new_exit;
            W = W ∪ { l' | (l, l') ∈ flow(S_*) };
        }
    }
}
```

Flow-graph reduction Due to the fact that our analysis is concerned only with document template computations, we can collapse the flow-graph substantially. More, specifically we make use of the four sound flow-graph transformation rules in Figure 4.5. where a single circled node indicates that the node could be associated with any program point and a double circled node denotes an "identity node", that is, a node in which the entry and exit maps are identical. Using these four rules as long as they apply will as we indicate through a number of examples below reduce the size of the flow-graph considerably. An efficient strategy for using these four rules is the following (where $V = \mathbf{Lab}$ and $E = \mathit{flow}(S_*)$):

```
for all id nodes l ∈ V {
   if ((l,l) ∈ E) apply rule one;
   if ({ l' | (l,l') ∈ E } ≠ ∅) {
      while ({ l' | (l',l) ∈ E } ≠ ∅) apply rule two;
      while ({ l' | (l,l') ∈ E } ≠ ∅) apply rule three;
   }
   if ({ l' | (l,l') ∈ E ∨ (l,l') ∈ E } = ∅) apply rule four;
}
```

That is, we consider the nodes in the graph one by one and then for each identity node apply rule one if possible and then if the out degree of the node is positive, we iteratively use rule two until the in degree of the node is zero. After this rule three is used repeatedly until the out degree reaches zero, and we can finally use rule four to remove the by now possibly redundant identity node. We then move on to the next identity node.

Using the sound graph reduction technique presented above we can reduce the size of the flow-graph to be analyzed by a factor of three allowing for a speedup of the analysis of roughly 50% as indicated by the diagrams in Figure 4.6(a) and 4.6(b) for the test programs `appointment`, `wwwboard`, `calendar`, `cdshop`, and `tournament`.

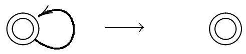
4.4.4 Correctness of the analysis

In this section we shall prove that the gap and field analysis defined previously in fact safely collects information about the gaps and fields occurring during computation. Before proving the correctness results we shall establish the following property of the semantics:

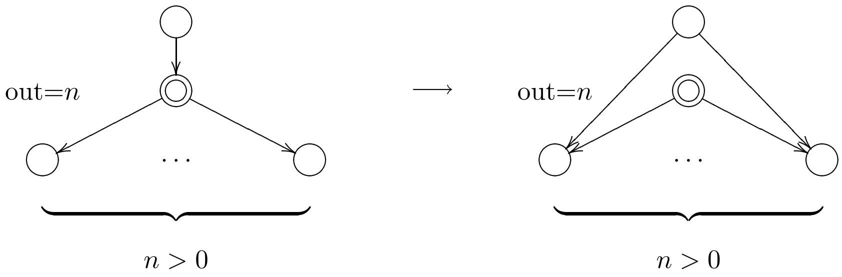
Lemma 4.4.1 *If* $\langle S, \sigma\rangle \rightarrow \langle S', \sigma'\rangle$ *then*

$$\mathit{final}(S) \supseteq \mathit{final}(S'),\ \mathit{flow}(S) \supseteq \mathit{flow}(S'),\ \textit{and } \mathit{labels}(S) \supseteq \mathit{labels}(S').$$

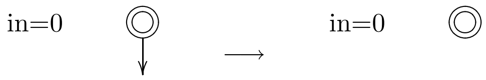
Proof: (A similar proof appears in [91]). First, let us show that $\mathit{final}(S) \supseteq \mathit{final}(S')$. The proof is by case analysis on the sematic rule used to establish



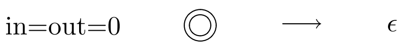
(a) Rule one.



(b) Rule two.

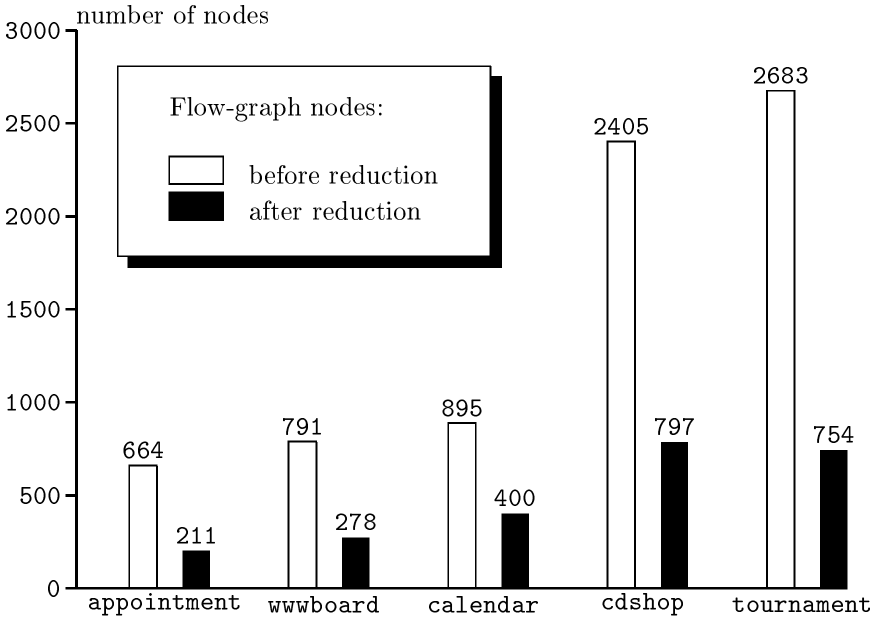


(c) Rule three.

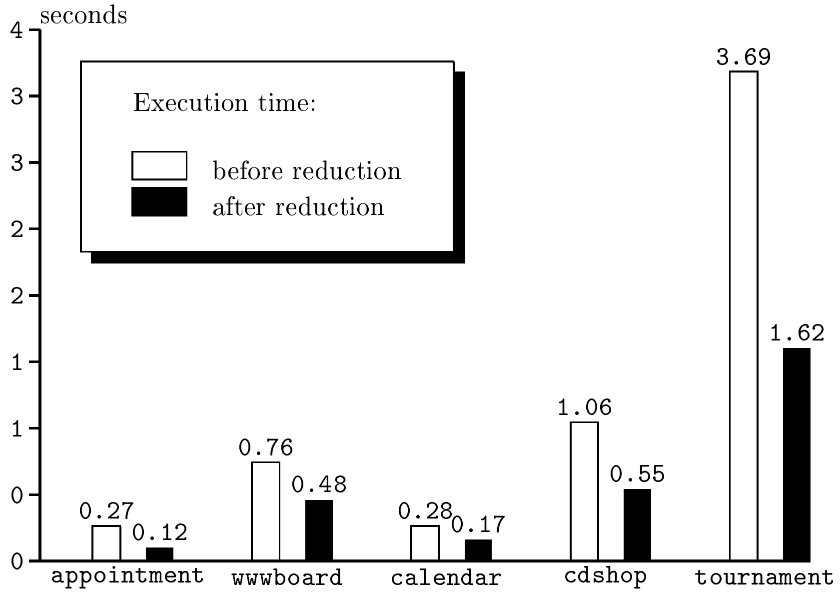


(d) Rule four.

Figure 4.5: Flow-graph transformation rules.



(a) Reduction in number of flow-graph nodes.



(b) Reduction in execution time.

Figure 4.6: Flow-graph reductions.

$\langle S, \sigma\rangle \to \langle S', \sigma'\rangle$. Consulting Figure 4.1, we see that there are five non-vacuous cases:

The case [seq_1]: Assuming $\langle S_1\,;\, S_2, \sigma\rangle \to \langle S_1'\,;\, S_2, \sigma'\rangle$ we get:

$$\mathit{final}(S_1\,;\,S_2) = \mathit{final}(S_2) = \mathit{final}(S_1'\,;\,S_2).$$

The case [seq_2]: Then $\langle S_1\,;\, S_2, \sigma\rangle \to \langle S_2, \sigma'\rangle$ and we get:

$$\mathit{final}(S_1\,;\,S_2) = \mathit{final}(S_2).$$

The case [if_1]: Thus $\langle \texttt{if}\,[b]^l\,S_1\,\texttt{else}\,S_2, \sigma\rangle \to \langle S_1, \sigma\rangle$ and we get:

$$\mathit{final}(\texttt{if}\,[b]^l\,S_1\,\texttt{else}\,S_2) = \mathit{final}(S_1) \cup \mathit{final}(S_2) \supseteq \mathit{final}(S_1).$$

The case [if_2] is similar to the case [if_1].

The case [wh_1]: We have $\langle \texttt{while}\,[b]^l\,S, \sigma\rangle \to \langle (S\,;\,\texttt{while}\,[b]^l\,S), \sigma\rangle$ and get that:

$$\mathit{final}(S\,;\,\texttt{while}\,[b]^l\,S) = \mathit{final}(\texttt{while}\,[b]^l\,S).$$

This completes the proof that $\mathit{final}(S) \supseteq \mathit{final}(S')$.

Now, let us turn to the proof that $\mathit{flow}(S) \supseteq \mathit{flow}(S')$ which is established by induction on the shape of the inference tree used to establish $\langle S, \sigma\rangle \to \langle S', \sigma'\rangle$. Again, there are five non-vacuous cases:

The case [seq_1]: Assuming $\langle S_1\,;\, S_2, \sigma\rangle \to \langle S_1'\,;\, S_2, \sigma'\rangle$ because $\langle S_1, \sigma\rangle \to \langle S_1', \sigma'\rangle$ we get (using the induction hypothesis and $\mathit{final}(S_1) \supseteq \mathit{final}(S_1')$):

$$\begin{aligned}
\mathit{flow}(S_1\,;\,S_2) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{\, (l, \mathit{init}(S_2)) \mid l \in \mathit{final}(S_1)\,\} \\
&\supseteq \mathit{flow}(S_1') \cup \mathit{flow}(S_2) \cup \{\, (l, \mathit{init}(S_2)) \mid l \in \mathit{final}(S_1)\,\} \\
&\supseteq \mathit{flow}(S_1') \cup \mathit{flow}(S_2) \cup \{\, (l, \mathit{init}(S_2)) \mid l \in \mathit{final}(S_1')\,\} \\
&= \mathit{flow}(S_1'\,;\,S_2).
\end{aligned}$$

The case [seq_2]: Then $\langle S_1\,;\, S_2, \sigma\rangle \to \langle S_2, \sigma'\rangle$ because $\langle S_1, \sigma\rangle \to \sigma'$ and we get:

$$\begin{aligned}
\mathit{flow}(S_1\,;\,S_2) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{\, (l, \mathit{init}(S_2)) \mid l \in \mathit{final}(S_1)\,\} \\
&\supseteq \mathit{flow}(S_2).
\end{aligned}$$

The case [if_1]: Thus $\langle \texttt{if}\,[b]^l\,S_1\,\texttt{else}\,S_2, \sigma\rangle \to \langle S_1, \sigma\rangle$ and we get:

$$\begin{aligned}
\mathit{flow}(S_1\,;\,S_2) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{(l, \mathit{init}(S_1)), (l, \mathit{init}(S_2))\} \\
&\supseteq \mathit{flow}(S_1).
\end{aligned}$$

The case [if_2] is similar to the case [if_1].

The case [wh_1]: We have $\langle \texttt{while}\,[b]^l\,S, \sigma\rangle \to \langle (S\,;\,\texttt{while}\,[b]^l\,S), \sigma\rangle$ and get that:

$$\begin{aligned}
\mathit{flow}(S\,;\,\texttt{while}\,[b]^l\,S) &= \mathit{flow}(S) \cup \mathit{flow}(\texttt{while}\,[b]^l\,S) \cup \{\, (l', l) \mid l' \in \mathit{final}(S)\,\} \\
&= \mathit{flow}(S) \cup (\mathit{flow}(S) \cup \{(l, \mathit{init}(S))\} \cup \{\, (l', l) \mid l' \in \mathit{final}(S)\,\}) \\
&\qquad \cup \{\, (l', l) \mid l' \in \mathit{final}(S)\,\} \\
&= \mathit{flow}(S) \cup \{(l, \mathit{init}(S))\} \cup \{\, (l', l) \mid l' \in \mathit{final}(S)\,\} \\
&= \mathit{flow}(\texttt{while}\,[b]^l\,S).
\end{aligned}$$

This completes the proof that $\mathit{flow}(S) \supseteq \mathit{flow}(S')$.

The proof of $\mathit{labels}(S) \supseteq \mathit{labels}(S')$ is similar to that of $\mathit{flow}(S) \supseteq \mathit{flow}(S')$ and we thus omit the details.

Previously we showed how to define a gap and field constraint system for a labeled program S_*; we will refer to this system as $\mathsf{A}^{\sqsubseteq}(S_*)$. Given functions $A_{\text{entry}}, A_{\text{exit}}$, we say that A solves $\mathsf{A}^{\sqsubseteq}(S_*)$, and write $A \models \mathsf{A}^{\sqsubseteq}(S_*)$ if the functions satisfy the constraints.

The next lemma shows that if we have a solution to the constraint system corresponding to some statement S, then it will also be a solution to the constraint system obtained from a sub-statement S'; this result will be essential in the proof of the correctness result.

Lemma 4.4.2 *If $\mathit{flow}(S) \supseteq \mathit{flow}(S')$, $\mathit{labels}(S) \supseteq \mathit{labels}(S')$, and $A \models \mathsf{A}^{\sqsubseteq}(S)$ then $A \models \mathsf{A}^{\sqsubseteq}(S')$.*

Proof: Assume $\mathit{flow}(S) \supseteq \mathit{flow}(S')$, $\mathit{labels}(S) \supseteq \mathit{labels}(S')$, and $A \models \mathsf{A}^{\sqsubseteq}(S)$. The result in the lemma then follows from the observation that if $A \models \mathsf{A}^{\sqsubseteq}(S)$, that is, if A_{entry} and A_{exit} satisfy the constraints in $\mathsf{A}^{\sqsubseteq}(S)$, then they must also satisfy the smaller set of constraints, $\mathsf{A}^{\sqsubseteq}(S')$. Thus $A \models \mathsf{A}^{\sqsubseteq}(S')$ as was needed.

We now have the following corollary expressing that a solution to the constraints of $\mathsf{A}^{\sqsubseteq}$ are preserved during computation.

Corollary 4.4.3 *If $\langle S, \sigma \rangle \to \langle S', \sigma' \rangle$ and $A \models \mathsf{A}^{\sqsubseteq}(S)$ then $A \models \mathsf{A}^{\sqsubseteq}(S')$.*

Proof: The corollary follows from Lemma 4.4.1 and 4.4.2.

We shall define the correctness relation, $\lesssim \subseteq \mathbf{State} \times \mathbf{DEnv}$, as follows:

$$\sigma \lesssim \mathit{denv} \text{ if and only if for all } x : (\mathit{gap} \otimes \mathit{field})(\sigma(x)) \sqsubseteq \mathit{denv}(x).$$

Notice, that if $\mathit{denv} \sqsubseteq \mathit{denv}'$ then by definition of $\lesssim$, $\sigma \lesssim \mathit{denv}$ implies $\sigma \lesssim \mathit{denv}'$.

We have the following lemma expressing that the $\mathcal{D}$ and $\mathcal{A}$ functions preserve the property of being related by the correctness relation.

Lemma 4.4.4 *If $\sigma \lesssim \mathit{denv}$ and $\mathcal{D}[\![d]\!]\sigma$ is defined then*

$$(\mathit{gap} \otimes \mathit{field})(\mathcal{D}[\![d]\!]\sigma) \sqsubseteq \mathcal{A}[\![d]\!]\mathit{denv}.$$

Proof: The proof is by case analysis on the document expression kind. Thus assume $\sigma \lesssim \mathit{denv}$ and $\mathcal{D}[\![d]\!]\sigma$ is defined. There are four cases to consider:

The case $d = x$: From $\sigma \lesssim \mathit{denv}$ we get $(\mathit{gap} \otimes \mathit{field})(\sigma(x)) \sqsubseteq \mathit{denv}(x)$ for all x. Now, since $\mathcal{D}[\![x]\!]\sigma = \sigma(x)$ and $\mathcal{A}[\![x]\!]\mathit{denv} = \mathit{denv}(x)$ we get:

$$(\mathit{gap} \otimes \mathit{field})(\mathcal{D}[\![d]\!]\sigma) \sqsubseteq \mathcal{A}[\![d]\!]\mathit{denv}.$$

The case $d = const$: Since $\mathcal{D}[\![const]\!]\sigma = const$ and $\mathcal{A}[\![const]\!]denv = (gap \otimes field)(const)$, we immediately get that:

$$(gap \otimes field)(\mathcal{D}[\![const]\!]\sigma) = (gap \otimes field)(const) = \mathcal{A}[\![const]\!]denv.$$

The case $d = (x_1\texttt{<[}y\texttt{=}x_2\texttt{]})$: We get:

$$\begin{aligned}
(gap \otimes field)(\mathcal{D}[\![d]\!]\sigma) &= (gap \otimes field)(\mathcal{D}[\![x_1\texttt{<[}y\texttt{=}x_2\texttt{]}]\!]\sigma) \\
&= (gap \otimes field)(plug(\sigma(x_1), y, \sigma(x_2))) \\
&= ((gap \otimes field)(\sigma(x_1)))[y \mapsto \mathsf{nogap}] \oplus (gap \otimes field)(\sigma(x_2)) \\
&\sqsubseteq denv(x_1)[y \mapsto \mathsf{nogap}] \oplus denv(x_2) \\
&= \mathcal{A}[\![x_1\texttt{<[}y\texttt{=}x_2\texttt{]}]\!]denv \\
&= \mathcal{A}[\![d]\!]denv
\end{aligned}$$

where the second line follows from the assumption that $\mathcal{D}[\![d]\!]\sigma$ is defined, third line from the fact that:

$$\begin{aligned}
gap(plug(c_1, y, c_2)) &= gap(c_1)[y \mapsto \mathsf{nogap}] \oplus_g gap(c_2) \text{ and} \\
field(plug(c_1, y, c_2)) &= field(c_1) \oplus_f field(c_2)
\end{aligned}$$

if $plug(c_1, y, c_2)$ is defined, and the fourth line from the assumption $\sigma \lesssim denv$.

The case $d = (x_1\texttt{<[}y\texttt{=}s\texttt{]})$ is similar to (but simpler than) the case $d = (x_1\texttt{<[}y\texttt{=}x_2\texttt{]})$ and is thus omitted.

This completes the proof that $\sigma \lesssim denv$ implies $(gap \otimes field)(\mathcal{D}[\![d]\!]\sigma) \sqsubseteq \mathcal{A}[\![d]\!]denv$.

We are now ready for the main result. It states how semantically correct information is preserved under each step of the execution.

Theorem 4.4.5 *Assume $A \models \mathsf{A}^{\sqsubseteq}(S)$ and $\sigma \lesssim A_{entry}(init(S))$.*

- *If $\langle S, \sigma\rangle \to \langle S', \sigma'\rangle$ then $\sigma' \lesssim A_{entry}(init(S'))$.*
- *If $\langle S, \sigma\rangle \to \sigma'$ then $\sigma' \lesssim \bigsqcup\{\, A_{exit}(l) \mid l \in final(S)\,\}$.*

Proof: The proof is by induction on the shape of the inference tree used to establish $\langle S, \sigma\rangle \to \langle S', \sigma'\rangle$ and $\langle S, \sigma\rangle \to \sigma'$, respectively. Thus assume $A \models \mathsf{A}^{\sqsubseteq}(S)$ and $\sigma \lesssim A_{\text{entry}}(init(S))$. Consulting Figure 4.1 we see that there are nine cases to consider.

The case $[asn]$: Thus $\langle [x = d]^l, \sigma\rangle \to \sigma[x \mapsto \mathcal{D}[\![d]\!]\sigma]$. What we need to show is that $\sigma' \lesssim A_{\text{exit}}(l)$, where $\sigma' = \sigma[x \mapsto \mathcal{D}[\![d]\!]\sigma]$. Since $A \models \mathsf{A}^{\sqsubseteq}([x = d]^l)$, we get $A_{\text{exit}}(l) \sqsupseteq A_{\text{entry}}(l)[x \mapsto \mathcal{A}[\![d]\!](A_{\text{entry}}(l))]$. Thus it is enough to show $(gap \otimes field)(\mathcal{D}[\![d]\!]\sigma) \sqsubseteq \mathcal{A}[\![d]\!](A_{\text{entry}}(l))$. But this follows from Lemma 4.4.4 since $\sigma \lesssim A_{\text{entry}}(l)$. We conclude that $\sigma' \lesssim A_{\text{exit}}(l)$.

The case $[skip]$: Then $\langle [\texttt{skip}]^l, \sigma\rangle \to \sigma$. Since $A \models \mathsf{A}^{\sqsubseteq}([\texttt{skip}]^l)$ we get that $A_{\text{exit}}(l) \sqsupseteq A_{\text{entry}}(l)$. From the fact that $final([\texttt{skip}]^l) = \{l\}$ and the assumption $\sigma \lesssim A_{\text{entry}}(l)$ we then conclude that $\sigma \lesssim \bigsqcup\{\, A_{\text{exit}}(l) \mid l \in final(S)\,\}$ as needed.

The case [seq_1]: Then $\langle S_1\,;\,S_2, \sigma\rangle \rightarrow \langle S_1'\,;\,S_2, \sigma'\rangle$ because $\langle S_1, \sigma\rangle \rightarrow \langle S_1', \sigma'\rangle$. From the assumption $A \models \mathsf{A}^{\sqsubseteq}(S_1\,;\,S_2)$ and the fact that $\mathit{flow}(S_1\,;\,S_2) \supseteq \mathit{flow}(S_1)$, it follows using Lemma 4.4.2 that $A \models \mathsf{A}^{\sqsubseteq}(S_1)$. By assumption,

$$\sigma \lesssim A_{\text{entry}}(\mathit{init}(S_1\,;\,S_2)) = A_{\text{entry}}(\mathit{init}(S_1)).$$

Thus using the induction hypothesis we get

$$\sigma' \lesssim A_{\text{entry}}(\mathit{init}(S_1')) = A_{\text{entry}}(\mathit{init}(S_1'\,;\,S_2))$$

as needed.

The case [seq_2]: Then $\langle S_1\,;\,S_2, \sigma\rangle \rightarrow \langle S_2, \sigma'\rangle$ because $\langle S_1, \sigma\rangle \rightarrow \sigma'$. Again, using the induction hypothesis we get:

$$\sigma' \lesssim \bigsqcup\{\, A_{\text{exit}}(l) \mid l \in \mathit{final}(S_1)\,\}.$$

Since $\{\,(l, \mathit{init}(S_2)) \mid l \in \mathit{final}(S_1)\,\} \subseteq \mathit{flow}(S_1\,;\,S_2)$ and $A \models \mathsf{A}^{\sqsubseteq}(S_1\,;\,S_2)$ we get $A_{\text{entry}}(\mathit{init}(S_2)) \sqsupseteq \bigsqcup\{\, A_{\text{exit}}(l) \mid l \in \mathit{final}(S_1)\,\}$. We conclude that $\sigma' \lesssim A_{\text{entry}}(\mathit{init}(S_2))$ as needed.

The case [if_1]: Then $\langle \texttt{if}\,[b]^l\, S_1\, \texttt{else}\, S_2, \sigma\rangle \rightarrow \langle S_1, \sigma\rangle$. Since $A \models \mathsf{A}^{\sqsubseteq}(\texttt{if}\,[b]^l\, S_1\, \texttt{else}\, S_2)$ and $(i, \mathit{init}(S_1)) \in \mathit{flow}(\texttt{if}\,[b]^l\, S_1\, \texttt{else}\, S_2)$ we get $A_{\text{entry}}(l) \sqsubseteq A_{\text{entry}}(\mathit{init}(S_1))$. Thus, $\sigma \lesssim A_{\text{entry}}(\mathit{init}(S_1))$ as needed.

The case [if_2] is similar to the case [if_1] and is thus omitted.

The case [wh_1]: Then $\langle \texttt{while}\,[b]^l\, S, \sigma\rangle \rightarrow \langle (S\,;\,\texttt{while}\,[b]^l\, S), \sigma\rangle$. Since A is a solution to the constraint set and $(l, \mathit{init}(S)) \in \mathit{flow}(\texttt{while}\,[b]^l\, S)$, we get that $A_{\text{entry}}(\mathit{init}(S)) \sqsupseteq A_{\text{exit}}(l) \sqsupseteq A_{\text{entry}}(l)$. Thus since by assumption $\sigma \lesssim A_{\text{entry}}(l)$, we get $\sigma \lesssim A_{\text{entry}}(\mathit{init}(S)) = A_{\text{entry}}(\mathit{init}(S\,;\,\texttt{while}\,[b]^l\, S))$.

The cases [wh_2] and [$show$] are similar to the case [$skip$] and are thus omitted.

This completes the proof.

Using $\sigma_0 \lesssim A_{\text{entry}}(\mathit{init}(S_*))$ for the base case and Lemma 4.4.3 for the induction step, simple induction on the length of the derivation sequences

$$\langle S, \sigma_0\rangle \rightarrow^* \langle S', \sigma'\rangle \text{ and } \langle S, \sigma_0\rangle \rightarrow^* \sigma'$$

will extend the above result to arbitrary computation sequences.

Using the correctness result for the analysis, we can show that type-checked programs never encounter runtime errors:

Corollary 4.4.6 *Assume S_* type checks. Then the execution of S_* will never enter a stuck configuration.*

Proof: The result follows from the correctness of the gap and field analysis and the close relation between type correctness and semantics of DynDoc.

Assume that our program of interest, S_* has been analyzed for gaps and fields and based on that checked to be well-typed. We show that during execution of

S_* we will never enter a stuck configuration. Assume on the contrary, that we have reached a stuck configuration. By inspecting the semantics of statements, we see that this can either be due to an assignment in which an illegal plug operation is performed or due to a show call in which the form fields of the shown document do not match the types of the receiving program variables.

In the former case ($[x = x_1\texttt{<[}y\texttt{=}x_2\texttt{]]}^l$) there are two possible reasons why the plug operation is illegal: either there is no y gap in x_1 or the resulting document is not well-formed. We can rule out the first immediately since type correctness requires $\pi_g(A_{\text{entry}}(l)(x_1))(y) = \mathsf{html}$ and by correctness of the analysis, x_1 *must* have an HTML gap named y in the current state. The second we can rule out by the following argument: since the program type-checked, there are no occurrences of the error kind in the analysis. More specifically, $A_{\text{exit}}(l)$ does not have any error kinds associated. Thus, by correctness of the analysis and the close relation between semantics of the plug operation and the definition of the analysis, the resulting document must be well-formed.

Now, in the case of the `show-receive` call, we can rule out the possibility of a mismatch between the form fields of the shown document and the types of the receiving program variables by referring to the correctness of the analysis and the direct connection between the semantics of show-receive and the definition of type correctness that both use the $\lessdot$ relation.

Experience has shown that almost all correct programs do in fact type-check. That is, our analysis is not only sound but also precise enough to be useful in a practical setting.

4.4.5 Extensions

In this section we treat the implemented extension to full monovariant interprocedural flow analysis and type-checking, and we describe how a slight change in the ordering on gap kinds along with implicit gap closing can conveniently be used to extend the set of programs accepted by our analysis.

Interprocedural analysis In our implementation of the presented type system, the flow analysis is extended in a standard way to a full monovariant *interprocedural* first-order forward data-flow analysis. For simplicity, we have presented the analysis and correctness results for the core language only, but the results of course extend to the full interprocedural analysis.

Implicit gap closing If we change the ordering on **GKind** slightly then the compiler can implicitly close gaps occurring in some but not all incoming flow edges to a given program point. This implicit *coercion* allows for more flexibility in the programming of Web services without sacrificing the important possibility of checking static safety. The implicit closing described in this section is part of our implementation.

As mentioned, we change the ordering on **GKind** slightly. More precisely, the ordering is changed to the total ordering:

$$\perp \sqsubseteq \mathsf{error} \sqsubseteq \mathsf{html} \sqsubseteq \mathsf{string} \sqsubseteq \mathsf{nogap}.$$

By examination of the transfer function for assignments, it is clear that the transfer function is still monotone and thus that the flow analysis is still well-defined. This slight change will allow for more programs to type-check. For instance, the following code fragment would not type-check in the original type system:

```
string name, title, univ;
html Intro = <html> <[title]> <[name]> <[affil]> </html>;
html Affil = <html>got the Ph.D. degree from <[univ]> </html>;
...collect name, title, etc. ...
if (title=="Doctor") Intro = Intro<[affil=(Affil<[univ=univ])];
show Intro<[name=name,title=title];
```

The code will not type-check because after the conditional, `Intro` might or might not have an `affil` gap. This problem could of course be solved simply by changing the conditional to

```
if (title=="Doctor") Intro = Intro<[affil=(Affil<[univ=univ])]
else Intro = Intro<[affil=""];
```

However, experience has shown that when writing a lot of code, the drag of dealing with such technicalities becomes rather annoying. With our new ordering on **GKind**, the above code fragment would in fact type-check. At the end of the conditional, the least upper bound of incoming flow-edges would instead of an error result in a nogap for the gap name `affil` in the document variable `Intro`.

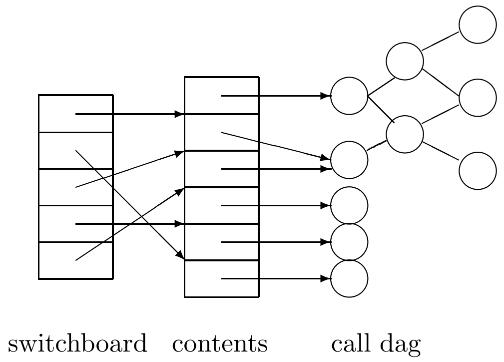
The crucial observation then, is that a program which type-checks with the new ordering can via use of the information inferred by the analysis easily be transformed into a program which type-checks in the original system and which has the expected behavior.

Note that the only new things that can be accepted with the changed ordering are merging of flow-edges in which for one or more gaps, some incoming flow edges deliver gaps and some do not. In the old system this would give rise to the top element error. Also, this observation explains how the program transformation should change programs to make them type check in the original type system: for each node in which the gap maps of incoming flow edges differ, insert explicit gap closing along the appropriate paths such that in fact all incoming flow edges will have the same gaps. The transformed program will then type-check with the old ordering on **GKind**. Thus relying on our correctness results we can still statically check the program for errors in document computations, that is, we can provide static safety if the program is well-typed.

4.5 Runtime Implementation

A naive implementation of dynamic documents would at runtime represent each document as a complete parse tree. Each document would then require memory proportional to its printed size, and plugging a document of size n into one of size m would require time $O(n+m)$. For well-typed programs, we can do much better than that: documents are represented with almost complete sharing, and the plug operation only takes time $O(1)$ while preserving efficiency of the show operation.

The key point it to represent a document value as a *closure*, which is essentially a table of function calls. The general structure of a closure is as follows:



In a document with k gaps, the *switchboard* has k entries and the *contents* table has $k+1$ entries. The switchboard is a map from gap names in lexicographic order to the order in which they occur in the document. The contents table defines the contents between gaps by pointing to nodes in a *call dag* (directed acyclic graph). Each node in the call dag has a reference counter. Internal nodes branch in two, while each leaf consists of either a string value or a function pointer with an integer argument. To be quite explicit, the corresponding type declaration in C is:

```
typedef struct dynDoc {
  int size;
  int *switchboard;
  struct callDag **contents;
} dynDoc;

typedef struct callDag {
  int refcount;
  enum {stringK,funcK,nodeK} kind;
  union {
    char *stringD;
    struct {void (*p)(int); int arg;} funcD;
    struct {struct callDag *left,*right;} nodeD;
  } val;
}
```

The reference count in the call dag nodes is used for automatic garbage collection (there are no cycles in the representation). The switchboard is required since document types do not specify the order of gaps, and two flow edges may choose different permutations. For instance, in a page with addresses, the appearance of street name and number would be different in a US page and a Continental European page.

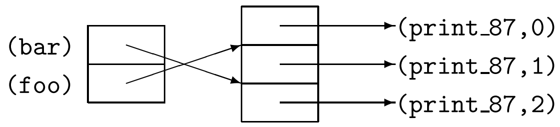
For each document constant we construct a function which on input i prints the contents between gaps number $i-1$ and i. For the fragment:

```
This <[foo]> <b>is <[bar]></b> you know!
```

the constructed function could be:

```
void print_87(int i) {
  switch (i) {
    case 0: printf("This"); break;
    case 1: printf("<b>is"); break;
    case 2: printf("</b> you know!"); break;
    }
  }
```

where the index `87` is some identification of the constant fragment. The corresponding document value has the obvious switchboard and the i'th entry in its contents table is a function node with function pointer `print_87` and argument i. For the example above, the representation is:



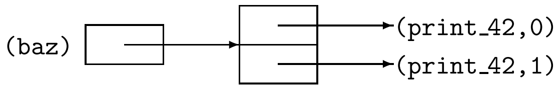
The textual contents of a closure can be printed by a simple in-order traversal of the dag which for each leaf node either calls the given function with the given argument or prints the given string constant. Because of the dag structure, the textual contents may be exponentially larger than the runtime representation. Note that this printing technique implicitly plugs every remaining gap with the empty document or string. The plug operation is realized by a function:

```
dynDoc *docPlugHTML(dynDoc *a, int i, dynDoc *b, int *merge)
```

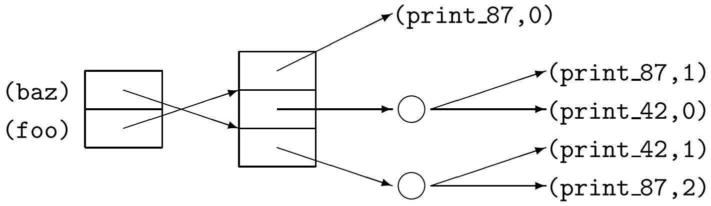
which plugs `b` into the HTML gap in `a` whose lexicographic index is `i`. Note that the value of `i` can be determined from the statically computed document type of `a`. The created document will have array size `(a->size)+(b->size)-1`, so its two tables are allocated accordingly.

The contents table is constructed as follows. Assume that the switchboard of `a` sends `i` to point between entries j and $j+1$ in the contents table. The contents table of `a` is cut in two after entry j. The contents table of `b` is inserted in between. At the two joints the neighboring elements are combined by creating new internal dag nodes, which leaves a contents table of the correct length.

The switchboard table is now created, using the `merge` argument which for every entry in the switchboard table of `b` indicates the position in the the switchboard table of `a` after which it must be merged to preserve the lexicographical order. Note that `merge` can be computed from the static document types of `a` and `b`. If the switchboard of `a` contains the gap names `(a,b,e,f)` and that of `b` contains `(c,d,g)` then the `merge` list would be `(2,2,4)`. If in the above example document we plug the gap named `bar` with the document:



then the result becomes:



Since the maximal length of a switchboard is a constant determined from the program text, the plug operation executes in time $O(1)$. A similar, but simpler, function is defined for plugging string gaps.

4.6 Conclusions

Interactive Web services based on the CGI protocol and HTML forms can be made safe, convenient, and efficient by using the typed higher-order templates that we propose.

Part II

Mona and Control Synthesis

Chapter 5

Mona: M2L in Practice

Contents

This chapter is on Mona: Monadic Second-Order Logic in Practice. The work has been published in the proceedings of Tools and Algorithms for the Construction and Analysis of Systems, TACAS'95 [44]. It is joint work with Jesper G. Henriksen, Jakob Jensen, Michael Jørgensen, Nils Klarlund, Robert Paige, and Theis Rauhe.

We introduce Monadic Second-order Logic as a practical means of specifying regularity. The logic is a highly succinct alternative to the use of regular expressions. We have built a tool MONA, which acts as a decision procedure and as a translator to finite-state automata. The tool is based on new algorithms for minimizing finite-state automata that use binary decision diagrams (BDDs) to represent transition functions in compressed form. A byproduct of this work is an algorithm that matches the time but improves the space of Sieling and Wegener's algorithm to reduce OBDDs in linear time.

The potential applications are numerous. We discuss text processing, Boolean circuits, and distributed systems. Our main example is an automatic proof of properties for the "Dining Philosophers with Encyclopedia" example by Kurshan and MacMillan. We establish these properties for the parameterized case *without* the use of induction.

Our results show that, contrary to common beliefs, high computational complexity may be a desired feature of a specification formalism.

5.1 Introduction

In computer science, *regularity* amounts to the concept that a class of structures is recognized by a finite-state device. Often phenomena are so complicated that their regularity either

- may be overlooked, as in the case of parameterized verification of distributed finite-state systems with a regular communication topology; or
- may not be exploited, as in the case when a search pattern in a text editor is known to be regular, but in practice inexpressible as a regular expression.

We argue that the *Monadic Second-Order Logic* or *M2L* can help in practice to identify and to use regularity. In M2L-Str, one can directly mention positions and subsets of positions in the string. This feature distinguishes the logic from regular expressions or automata. Together with quantification and Boolean connectives, an extraordinary succinct formalism arises.

Although it has been known for thirty-five years that M2L-Str defines regular languages (see [128]), the translator from formulas to automata that we describe in this article appears to be one of the first implementations.

The reason such projects have not been pursued may be the staggering theoretical lower-bound: any decision procedure is bound to sometimes require as much time as a stack of exponentials that has height proportional to the length of the formula.

It is often believed that the lower the computational complexity of a formalism is, the more useful it may be in practice. We want to counter such beliefs in this article — at least for logics on finite strings.

5.1.1 Why use logic?

Some simple finite-state languages easily described in English call for convoluted regular expressions. For example, the language L_{2a2b} of all strings over $\Sigma = \{a, b, c\}$ containing at least two occurrences of a *and* at least two occurrences of b seems to require a voluminous expression, such as

$$\begin{array}{ll} & \Sigma^* a \Sigma^* a \Sigma^* b \Sigma^* b \Sigma^* \\ \cup & \Sigma^* a \Sigma^* b \Sigma^* a \Sigma^* b \Sigma^* \\ \cup & \Sigma^* a \Sigma^* b \Sigma^* b \Sigma^* a \Sigma^* \\ \cup & \Sigma^* b \Sigma^* b \Sigma^* a \Sigma^* a \Sigma^* \\ \cup & \Sigma^* b \Sigma^* a \Sigma^* b \Sigma^* a \Sigma^* \\ \cup & \Sigma^* b \Sigma^* a \Sigma^* a \Sigma^* b \Sigma^*. \end{array}$$

If we added $\cap$ to the operators for forming regular expressions, then the language L_{2a2b} could be expressed more concisely as $(\Sigma^* a \Sigma^* a \Sigma^*) \cap (\Sigma^* b \Sigma^* b \Sigma^*)$. Even with this extended set of operators, it is often more convenient to express regular languages in terms of positions and corresponding letters. For example, to express the set $L_{a\mathrm{after}b}$ of strings in which every b is followed by an a, we would like a formal language allowing us to write something like

> "for every position p, if there is a b in p then for some position q after p, there is an a in q."

The extended regular languages do not seem to allow an expression that very closely reflects this description — although upon some reflection a small regular expression can be found. But in M2L-Str we can express $L_{a\mathrm{after}b}$ by a formula

$$\forall p : {}'b'(p) \;\Rightarrow\; \exists q : \; p < q \;\wedge\; {}'a'(q)$$

(Here the predicate $'b'(p)$ means "there is a b in position p".) In general, we believe that many errors can be avoided if logic is used when the description in English does not lend itself to a direct translation into regular expressions or automata. However, the logic can easily be combined with other methods of specifying regularity since almost any such formalism can be translated with only a linear blow-up into M2L-Str.

Often regularity is identified by means of *projections*. For example, if L_{trans} is regular on a cross-product alphabet $\Sigma \times \Sigma$ (e.g. describing a parameterized transition relation, see Section 5) and L_{start} is a regular language on Σ describing a set of start strings, then the set of strings that can be reached by a transition from a start string is $\pi_2(L_{trans} \cap \pi_1^{-1}(L_{start}))$, where π_1 and π_2 are the projections from $(\Sigma \times \Sigma)^*$ to the first and second component. Such language-theoretic operations can be very elegantly expressed in M2L-Str.

5.1.2 Our results

In this article, we discuss applications of M2L-Str to text processesing and the description of parameterized Boolean circuits. Our principal application is a new proof technique for establishing properties about parameterized, distributed finite-state systems with regular communication topology. We illustrate our method by showing safety and liveness properties for a non-trivial version of the Dining Philosophers' problem as proposed in [66] by Kurshan and MacMillan.

We present Mona, which is our tool that translates formulas in M2L-Str to finite-state machines. We show how BDDs can be used to overcome an otherwise inherent problem of exponential explosion. Our minimization algorithm works very fast in practice thanks to a simple generalization of the unary apply operation of BDDs.

5.1.3 Comparisons to other work

Parameterized circuits are described using BDDs in [43]. This method relies on formulating inductive steps as finite-state devices and does not provide a single specification language. The work in [108] is closer in spirit to our method in that languages of finite strings are used although not as part of a logical framework. In [11], another approach is given based on iterating abstractions. The parameterized Dining Philosopher's problem is solved in [66] by a finite-state induction principle.

A tool for M2L-Str on finite, binary trees has been developed at the University of Kiel [124]. Apparently, this tool has only been used for very simple examples.

In [27], a programming language for finite domains based on a fixed point logic is described and used for verification of non-parameterized finite systems.

5.1.4 Contents

In Section 2, we explain the syntax and semantics of M2L-Str on strings. We recall the correspondence to automata theory in Section 3. We give several applications of M2L-Str and the tool in Section 4: text patterns, parameterized circuits, and equivalence testing. Our main example of parameterized verification is discussed in Section 5. We give an overview of our implementation in Section 6.

5.2 The Monadic Second-order Logic on Strings

Let Σ be an alphabet and let w be a string over Σ. The semantics of the logic determines whether a closed M2L-Str formula ϕ holds on w. The language $L(\phi)$ denoted by ϕ is the set of strings that make ϕ hold. Assume now that w has length n and consists of letters $a_0 a_1 ... a_{n-1}$. The *positions* in w are then $0,...,n-1$. We can now describe the three syntactic categories of M2L-Str on strings.

A *position term* t is either

- the constant 0 (which denotes the position 0);
- the constant \$ (which denotes the last position, that is $n-1$);
- a position variable p (which denotes a position i);
- of the form $t \oplus i$ (which denotes the position $j+i \bmod n$, where j is the interpretation of t); or
- of the form $t \ominus i$ (which denotes the position $j-i \bmod n$, where j is the interpretation of t);

(Position terms are only interpreted for non-empty strings).
A *position set term* T is either

- the constant $\emptyset$ (which denotes the empty set);
- the constant **all** (which denotes the set $\{0, ..., n-1\}$);
- a position set variable P (which denotes a subset of positions);
- of the form $T_1 \cup T_2$, $T_1 \cap T_2$, or $\complement T_1$ (which are interpreted in the natural way);
- of the form $T+i$ (which denotes the set of positions in T shifted right by an amount of i); or
- of the form $T-i$ (which denotes the set of positions in T shifted left by an amount of i);

A *formula* ϕ is either of the form

- $'a'(t)$ (which holds if letter a_i in $w = a_0 a_1 \cdots$ is a, where i is the interpretation of t);
- $t_1 = t_2$, $t_1 < t_2$ or $t_1 \leq t_2$ (which are interpreted in the natural way);
- $T_1 = T_2$, $T_1 \subseteq T_2$, or $t \in T$ (which are interpreted in the natural way);
- $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \Rightarrow \phi_2$, or $\phi_1 \Leftrightarrow \phi_2$ (where ϕ_1 and ϕ_2 are formulas, and which are interpreted in the natural way);
- $\exists p : \phi$ (which is true, if there is a position i such that ϕ holds when i is substituted for p);
- $\forall p : \phi$ (which is true, if for all positions i, ϕ holds when i is substituted for p);
- $\exists P : \phi$ (which is true, if there is a subset of positions I such that ϕ holds when I is substituted for P); or
- $\forall P : \phi$ (which is true, if for all subsets of positions I, ϕ holds when I is substituted for P);

5.3 From M2L to Automata

In this section, we recall the method for translating a formula in M2L-Str to an equivalent finite-state automaton (see [128] for more details). Note that any formula ϕ can be interpreted, given a string w and a *value assignment* $\mathcal{I}$ that fixes values of the free variables. If ϕ then holds, we write $w, \mathcal{I} \models \phi$. The key idea is that a value assignment and the string may be described together as a word over an extended alphabet consisting of Σ and extra binary tracks, one for each variable. By structural induction, we then define for each formula an automaton that exactly recognizes the words in the extended alphabet corresponding to pairs consisting of a string and an assignment that satisfy the formula.

Example 5.3.1 Assume that the free variables are $\mathcal{P} = \{P_1, P_2\}$ and that $\Sigma = \{a, b\}$. Let us consider the string $w = abaa$ and value assignment

$$\mathcal{I} = [P_1 \mapsto \{0,2\}, P_2 \mapsto \emptyset].$$

The set $\mathcal{I}(P_1) = \{0,2\}$ can be represented by the bit pattern 1010, since the numbered sequence

$$\begin{array}{cccc} 1 & 0 & 1 & 0 \\ {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} \end{array}$$

defines that 0 is in the set (the bit in position 0 is 1), 1 is not in the set (the bit in position 1 is 0), etc. Similarly, the bit pattern 0000 describes $\mathcal{I}(P_2) = \emptyset$.

If these patterns are laid down as extra "tracks" along w, we obtain an *extended word* α, which may be depicted as:

a	b	a	a
1	0	1	0
0	0	0	0

Technically, we define $\alpha = \alpha_0 \cdots \alpha_3$ as the word $(a,1,0)(b,0,0)(a,1,0)\,(a,0,0)$ over the alphabet $\Sigma \times \mathbb{B} \times \mathbb{B}$ of *extended letters*, where $\mathbb{B} = \{0,1\}$ is the set of truth values.

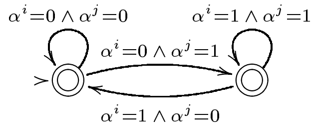
This correspondence can be generalized to any w and any value assignment for a set of variables $\mathcal{P}$ (which can all be assumed to be second-order).

By structural induction on formulas, we construct automata $A^{\phi,\mathcal{P}}$ over alphabet $\Sigma \times \mathbb{B}^k$—where $\mathcal{P} = \{P_1, \cdots, P_k\}$ is any set of variables containing the free variables in ϕ—satisfying the *fundamental correspondence*:

$$w, \mathcal{I} \models \phi \text{ iff } (w, \mathcal{I}) \in L(A^{\phi,\mathcal{P}}).$$

Thus $A^{\phi,\mathcal{P}}$ accepts exactly the pairs $(w, \mathcal{I})$ that make ϕ true.

Example 5.3.2 Let ϕ be the formula $P_i = P_j + 1$. Thus when ϕ holds, P_i is represented by the same bit pattern as that of P_j but shifted right by one position. This can be expressed by the automaton $A^{\phi,\mathcal{P}}$:



In this drawing, α^i refers to the ith extra track. Thus, the automaton checks that the ith track holds the same bit as the jth track the instant before.

5.4 Applications

5.4.1 Text patterns

The language L_{2a2b} of strings containing at least two occurrences of a and two occurrences of b can be described in M2L-Str by the formula

$$\begin{array}{l}(\exists p_1, p_2 : {'a'}(p_1) \;\wedge\; {'a'}(p_2) \;\wedge\; p_1 \neq p_2) \wedge \\ (\exists p_1, p_2 : {'b'}(p_1) \;\wedge\; {'b'}(p_2) \;\wedge\; p_1 \neq p_2)\end{array}$$

Our translator yields the minimal automaton, which contains nine states, in a fraction of a second.

The language $L_{a\mathrm{after}b}$ given by the formula

$$\forall p : {'b'}(p) \;\Rightarrow\; \exists q :\; p < q \;\wedge\; {'a'}(q)$$

is translated to the minimal automaton, which has two states, in .3 seconds.

A far more complicated language to express is $L_{<1\mathrm{apart}}$ consisting of every string over $\{a, b\}$ such that for any prefix the number of a's and b's are at most one apart. When using regular expressions or M2L-Str, one needs to struggle a bit, but in M2L-Str there is a strategy for describing the functioning of the finite-state machine that comes to mind.

We observe that a position p may be used to designate a prefix; for example, 0 denotes the prefix consisting of the first letter and \$ (the last position) denotes the whole input string. We may now recognize a string in $L_{<1\mathrm{apart}}$ by identifying three sets of positions: the set P_0 corresponding to prefixes with an equal number of a's and b's, the set P_{+1} corresponding to prefixes where the number of a's is one greater than the number of b's, and the set P_{-1} corresponding to prefixes where the number of a's is one less than the number of b's:

$$\begin{array}{l}\exists P_0, P_{+1}, P_{-1} : P_0 \cup P_{+1} \cup P_{-1} = \mathbf{all} \\ \qquad \wedge\; 0 \notin P_0 \\ \qquad \wedge\; 0 \in P_{+1} \Leftrightarrow {'a'}(0) \\ \qquad \wedge\; 0 \in P_{-1} \Leftrightarrow {'b'}(0) \\ \qquad \wedge\; \forall p : (p > 0 \;\Rightarrow \\ \qquad\qquad p \in P_0 \;\Leftrightarrow ({'a'}(p) \;\wedge\; p \ominus 1 \in P_{-1}) \\ \qquad\qquad\qquad \vee\; ({'b'}(p) \;\wedge\; p \ominus 1 \in P_{+1}) \\ \qquad\qquad \wedge\; p \in P_{+1} \;\Leftrightarrow\; {'a'}(p) \;\wedge\; p \ominus 1 \in P_0 \\ \qquad\qquad \wedge\; p \in P_{-1} \;\Leftrightarrow\; {'b'}(p) \;\wedge\; p \ominus 1 \in P_0)\end{array}$$

The resulting four-state automaton is calculated in a fraction of a second.

Text editors and operating systems often allow the user to the select pieces of text according to a *pattern*. For example, in Emacs we may specify text that contains two words separated by a "," using a regular expression

\w+ , \w+

where \w matches a character in a word, + makes the preceeding expression match one or more times and "," matches a comma.

Unfortunately, it is often exceedingly cumbersome to specify search patterns using regular expressions. Let us consider an example of modifying all declarations `int* ch` within structure declarations except if the structure declaration is within a procedure declaration. Thus in

```
struct {
  int* ch;
  int   x;
} inf;

void foo() {
  int y;
  struct {int* ch} inf;
  . . .
}
```

we want the first occurrence only of `int* ch` to be replaced. Let us express the problem in logic. Say that an *interval* I is a set of contiguous positions in the text. An interval can be designated by the lowest and highest position it contains, so intervals can easily be treated in M2L-Str. Assume we have already defined predicates *struct_decl*, *proc_decl*, and *int_ch* such that

- *struct_decl* (I) is satisfied if and only if the interval I holds a structure declaration,
- *proc_decl* (I) is satisfied if and only if the interval I holds a procedure declaration, and
- *int_ch* (I) is satisfied if and only if the interval I holds the string "`int* ch`".

We can write *struct_decl* and *proc_decl* in M2L-Str under the assumption that the nesting of braces is bounded. Then an interval I holds a declaration to be replaced if and only if the formula

$$\begin{aligned} &\mathit{int\_ch}(I) \wedge \exists J : J \supseteq I \wedge \mathit{struct\_decl}(J) \wedge \\ &\qquad\qquad \neg(\exists K : K \supsetneq J \wedge \mathit{proc\_decl}(K)) \end{aligned} \tag{5.1}$$

is satisfied. This formula states that I must hold an `int* ch` declaration and that there must be some interval J containing I such that J holds a structure declaration but is not properly contained in any interval K containing a procedure declaration.

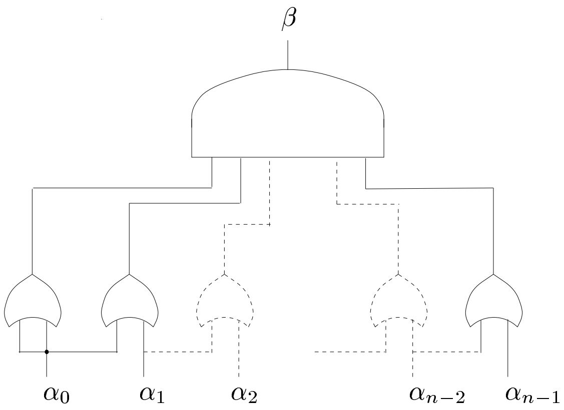


Figure 5.1: A parameterized circuit.

5.4.2 Parameterized circuits

Assume that we are given a drawing as in Figure 5.1 denoting a parameterized Boolean function.

How do we describe the language $L_{ex} \subseteq \mathbb{B}^*$ of input bit patterns that make the output true? From the drawing, no immediate description as a regular expression or finite-state automaton is apparent. In M2L-Str, however, it is easy to model the outputs of the n or-gates as a second-order variable Q, which allows the language to be described from a direct interpretation of the drawing. Note that the or-gate at position $p > 0$ is true if either there is a 1 at $p-1$ or p, or in other words: $p \in Q \Leftrightarrow {}'1'(p \ominus 1) \vee {}'1'(p)$. Since the output is 1 if and only if all or-gates are 1, that is if $Q = \mathbf{all}$, the language L_{ex} is given by the formula

$$\begin{array}{ll} \exists Q : (\forall p : & (p = 0 \Rightarrow p \in Q \Leftrightarrow {}'1'(p)) \wedge \\ & (p > 0 \Rightarrow (p \in Q \Leftrightarrow {}'1'(p \ominus 1) \vee {}'1'(p))) \wedge Q = \mathbf{all}) \end{array}$$

The resulting automaton has three states and accepts the language $(1 \cup 10)^*$, which is the regular expression that one would obtain by reasoning about the circuit. For more advanced applications to hardware verification, see [12].

5.4.3 Equivalence testing

A closed formula ϕ is a *tautology* if $L(\phi) = L(\Sigma^*)$, that is if all strings over Σ satisfy ϕ. The equivalence of formulas ϕ and ψ then amounts to whether $\phi \Leftrightarrow \psi$ is a tautology.

Example. That a set P contains exactly the even positions in a non-empty input string may be expressed in M2L-Str by the following two rather different approaches: either by the formula *even1*$(P) \equiv$

$$0 \in P \ \wedge \ \forall p : ((p \in P \ \wedge \ p < \$ \Rightarrow p \oplus 1 \notin P) \\ \wedge \ (p \notin P \ \wedge \ p < \$ \Rightarrow p \oplus 1 \in P)),$$

or as a formula $\mathit{even2}(P) \equiv$

$$P \cup (P+1) = \mathbf{all} \ \wedge \ P \cap (P+1) = \emptyset \ \wedge \ P \neq \emptyset$$

To show the equivalence of the two formulas, we check the truth value of the bi-implication:

$$\forall P : \mathit{even}1(P) \Leftrightarrow \mathit{even}2(P)$$

The translation of this formula does indeed produce an automaton accepting Σ^*, and thus verifies our claim.

5.5 Dining Philosophers with Encyclopedia

A distributed system is *parameterized* when the number n of processes is not fixed a priori. For such systems the state space is unbounded, and thus traditional finite-state verification methods cannot be used. Instead, one often fixes n to be, say two or three. This yields a finite state space amenable to state exploration methods. However, the validity of a property for $n = 2$, 3 does not necessarily imply that the property holds for all n.

A central problem in verification is automatically to validate parameterized systems. One way to attack the problem is to formulate induction principles such that the base case and the inductive steps can be formulated as finite-state problems. Kurshan and MacMillan [66] used such a method to verify safety and liveness properties of a non-trivial version of the Dining Philosophers example.

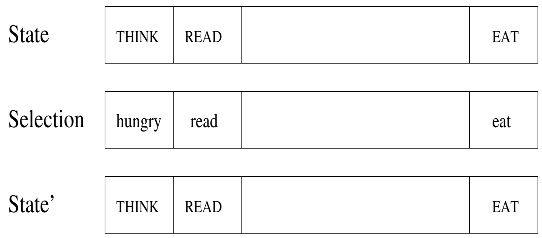


Figure 5.2: Dining Philosophers with Encyclopedia

In this system, symmetry is broken by an encyclopedia that circulates among the philosophers. Thus each philosopher is in one of three states: EAT, THINK, or READ. The global state can be described as a string *State* of length n over the alphabet $\Sigma_{\mathrm{State}} = \{\mathrm{EAT}, \mathrm{THINK}, \mathrm{READ}\}$, see Figure 5.2.

The system makes a transition according to external events that constitute a *selection* . Each process is presented with an event in the alphabet

$$\Sigma_{\mathrm{Selection}} = \{\mathrm{eat}, \mathrm{think}, \mathrm{read}, \mathrm{hungry}\}.$$

Thus the selection can be viewed as a string *Selection* over $\Sigma_{\text{Selection}}$, see Figure 5.2. As shown, all processes make a synchronous transition to a new global State' on a selection according to a transition relation $\text{trans}(\mathit{State}, \mathit{State}', \mathit{Selection})$, which is shown in Figure 5.3 (We use '#' in the beginning of a line to indicate that this line is a comment.) together with an auxiliary predicate blocking ($\mathit{Selection}$) used in its definition. Thus the new state of each process is dependent on its old state and on the selection events presented to itself and its neighbors. The transition relation is so complicated that it is hard to grasp the functioning of the system.

Fortunately, the parameterized transition relation can be translated into basic M2L-Str on strings. For example, we encode *State* using two second-order variables P and Q with the convention that

$$\begin{aligned}
\text{EAT}_p(\mathit{State}) &\equiv p \in P \ \wedge\ p \in Q \\
\text{READ}_p(\mathit{State}) &\equiv p \notin P \ \wedge\ p \in Q \\
\text{THINK}_p(\mathit{State}) &\equiv p \notin P \ \wedge\ p \notin Q
\end{aligned}$$

Similarly, State' and *Selection* can also each be encoded using two second-order variables. Thus, the predicate $\text{trans}(\mathit{State}, \mathit{State}', \mathit{Selection})$ becomes a formula with six free second-order variables.

For this distributed system there are two important properties to verify:

- *Safety Property*: The encyclopedia is neither lost nor replicated. Thus there is always exactly one process in state READ.
- *Liveness Property*: If no process remains in state EAT forever, then the encyclopedia is passed around over and over.

In [66] both properties are proved in terms of a complicated induction hypothesis. This hypothesis is itself a distributed system, where each process has four states. (The Liveness Property in [66] is technically different since it is modeled in terms of selections.)

Our strategy is fundamentally different. We cannot directly verify liveness properties. But we can easily verify properties about the transition relation in the parameterized case and *without* induction as follows.

Let ϕ be an M2L-Str formula about the global state. For example, we might consider the property that if a philosopher eats, then his neighbors do not:

$$\phi_{\text{mutex}}(\mathit{State}) \equiv \forall p : \text{EAT}_p(\mathit{State}) \Rightarrow \neg\text{EAT}_{p\ominus 1}(\mathit{State}) \ \wedge\ \neg\text{EAT}_{p\oplus 1}(\mathit{State})$$

A property given as a formula ϕ can be verified using the *invariance principle*:

$$\begin{aligned}
&\forall \mathit{State}, \mathit{State}', \mathit{Selection} : \\
&\quad \phi(\mathit{State}) \ \wedge\ \text{trans}(\mathit{State}, \mathit{State}', \mathit{Selection}) \Rightarrow \phi(\mathit{State}'),
\end{aligned}$$

which is also a formula in M2L-Str. In this way, we have verified for the parameterized case that both ϕ_{mutex} and the Safety Property that exactly one philosopher reads, that is $\exists! p : \text{READ}_p(\mathit{State})$, are invariant. Mona verifies such a formula in approximately 3 seconds on a Sparc 20.

$$
\begin{array}{ll}
\text{blocking}(\mathit{Selection}) \equiv & \wedge \\
\text{eat}_{p\oplus 1}(\mathit{Selection}) \ \vee \ \text{hungry}_{p\ominus 1}(\mathit{Selection}) & \#\text{EAT} \rightarrow \text{EAT}: \\
\vee \ \text{eat}_{p\ominus 1}(\mathit{Selection}) & (\text{EAT}_p(\mathit{State}) \ \wedge \ \text{EAT}_p(\mathit{State}') \Rightarrow \\
& \text{eat}_p(\mathit{Selection})) \\
\text{trans}(\mathit{State}, \mathit{State}', \mathit{Selection}) \equiv & \\
\forall p: & \wedge \\
& \#\text{EAT} \rightarrow \text{READ}: \\
\#\text{THINK} \rightarrow \text{THINK}: & (\text{EAT}_p(\mathit{State}) \ \wedge \ \text{READ}_p(\mathit{State}') \Rightarrow \\
(\text{THINK}_p(\mathit{State}) \ \wedge \ \text{THINK}_p(\mathit{State}') \Rightarrow & \text{think}_p(\mathit{Selection}) \ \wedge \ \text{read}_{p\ominus 1}(\mathit{Selection})) \\
\text{think}_p(\mathit{Selection}) \ \wedge \ \neg(\text{read}_{p\ominus 1}(\mathit{Selection})) & \\
\vee & \wedge \\
\text{hungry}_p(\mathit{Selection}) \ \wedge \ \text{blocking}(\mathit{Selection})) & \#\text{READ} \rightarrow \text{THINK}: \\
& (\text{READ}_p(\mathit{State}) \ \wedge \ \text{READ}_p(\mathit{State}') \Rightarrow \\
\wedge & \text{read}_p(\mathit{Selection}) \ \wedge \ \text{think}_{p\oplus 1}(\mathit{Selection})) \\
\#\text{THINK} \rightarrow \text{EAT}: & \\
(\text{THINK}_p(\mathit{State}) \ \wedge \ \text{EAT}_p(\mathit{State}') \Rightarrow & \wedge \\
\text{hungry}_p(\mathit{Selection}) \ \wedge \ \neg(\text{blocking}(\mathit{Selection}))) & \#\text{READ} \rightarrow \text{EAT}: \\
& (\text{READ}_p(\mathit{State}) \ \wedge \ \text{EAT}_p(\mathit{State}') \Rightarrow \\
\wedge & \text{false}) \\
\#\text{THINK} \rightarrow \text{READ}: & \\
(\text{THINK}_p(\mathit{State}) \ \wedge \ \text{READ}_p(\mathit{State}') \Rightarrow & \wedge \\
\text{think}_p(\mathit{Selection}) \ \wedge \ \text{read}_{p\ominus 1}(\mathit{Selection})) & \#\text{READ} \rightarrow \text{READ}: \\
& (\text{READ}_p(\mathit{State}) \ \wedge \ \text{READ}_p(\mathit{State}') \Rightarrow \\
\wedge & \text{read}_p(\mathit{Selection}) \ \wedge \ \neg(\text{think}_{p\oplus 1}(\mathit{Selection}))) \\
\#\text{EAT} \rightarrow \text{THINK}: & \\
(\text{EAT}_p(\mathit{State}) \ \wedge \ \text{THINK}_p(\mathit{State}') \Rightarrow & \\
\text{think}_p(\mathit{Selection}) \ \wedge \ \neg(\text{read}_{p\ominus 1}(\mathit{Selection}))) &
\end{array}
$$

Figure 5.3: The transition relation

Note that this method does not rely on a state space exploration (which is impossible since the state space is unbounded). Instead, it is based on the Invariance Principle: to show that a property holds for all reachable states, it is sufficient to show that it holds for the initial state and is preserved under any transition.

5.5.1 Establishing the liveness property

The Liveness Property can be expressed in Temporal Logic as

$$\Box(\text{READ}_{p\ominus 1} \ \Rightarrow \ \Diamond\text{READ}_p), \tag{5.2}$$

that is, it always holds that if philosopher $p\ominus 1$ reads, then eventually philosopher p reads. We must prove this property under the assumption that no philosopher eats forever:

$$\Box(\text{EAT}_p \ \Rightarrow \ \Diamond\neg\text{EAT}_p). \tag{5.3}$$

So assume that $\text{READ}_{p\ominus 1}$ holds. We must prove that $\Diamond\text{READ}_p$ holds. There are two cases as follows.

- Case EAT_p holds. By assumption (5.3), there is an instant when $\text{EAT}_p \wedge \neg \bigcirc \text{EAT}_p$ holds. Thus if

$$\text{READ}_{p\ominus 1} \;\wedge\; \text{EAT}_p \;\wedge\; \neg \bigcirc \text{EAT}_p \Rightarrow \bigcirc\text{READ}_p \tag{5.4}$$

  is a valid property of the transition system, $\Diamond\text{EAT}_p$ holds. In fact, we verified using Mona that (5.4) indeed holds.

- Case $\neg\text{EAT}_p$ holds. If EAT_p becomes true, then use the previous case. Otherwise, $\neg\text{EAT}_p$ continues to hold. Now, by the assumption (5.3) at some point $\neg\text{EAT}_{p\oplus 1}$ will hold. We then use the property

$$\text{READ}_{p\ominus 1} \;\wedge\; \neg\text{EAT}_p \;\wedge\; \neg \bigcirc \text{EAT}_{p\oplus 1} \Rightarrow \bigcirc\text{READ}_p \;\vee\; \bigcirc\text{EAT}_p, \tag{5.5}$$

  which we have also verified using Mona, to show that eventually READ_p holds (or eventually EAT_p holds, which contradicts the assumption that $\neg\text{EAT}_p$ continues to hold).

5.6 Implementation.

Mona is our implementation of the decision procedure, which translates formulas of M2L-Str to finite-state automata as outlined in Section 3. Our tool is implemented in Standard ML of New Jersey. A previous version of Mona was written in C with explicit garbage collection and based on representing transition functions in a conjunctive normal form. Our present tool runs up to 50 times faster due to improved algorithms.

5.6.1 Representation of automata

Since the size of the extended alphabet grows exponentially with the number of variables, a straightforward implementation based on explicitly representing the alphabet would only work for very simple examples. Instead, we represent the transition relation using Binary Decision Diagrams (BDDs) [19, 18]. In this way, the alphabet is never explicitly represented. For the external alphabet of ASCII-characters, we choose an encoding based on seven extra tracks holding the binary representation. Thus, character classes such as `[a-zA-Z]` become represented as very simple BDDs.

A deterministic automaton A is represented as follows. The state space is $Q = \{0, 1, \ldots, n-1\}$, where n is size of the state space; $\mathbb{B}^k$ is the extended alphabet; $i_0 \in Q$ is the initial state; $\delta : Q \times \mathbb{B}^k \to Q$ is the transition function; and $F \subseteq Q$ is the set of accepting states. We use a bit vector of size n to represent F and an array containing n pointers to roots of multi-terminal BDDs representing δ. A leaf of a BDD holds the integer designating the next state. An internal node v is called a *decision node* and contains an *index* denoted $v.index$,

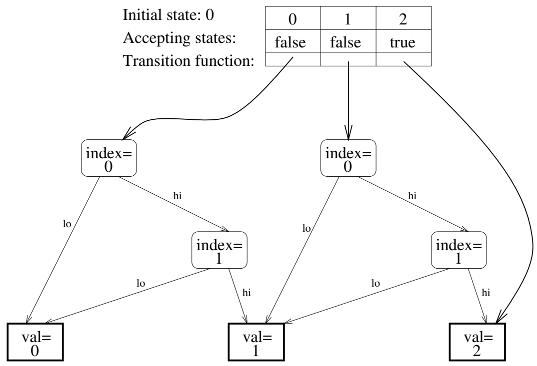
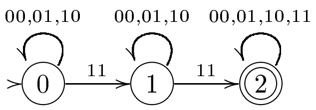


Figure 5.4: BDD automaton representation

where $0 \leq v.index < k$, and high and low successors $v.hi$ and $v.lo$. If b is a sequence of k bits, that is $b \in \mathbb{B}^k$, then $\delta(q, b)$ is found by looking up the qth entry in the array and following the decision nodes according to b until a leaf is reached (node v is followed by selecting the high successor if the $v.index$th component of b is 1 and the low successor if it is 0).

For example, the following finite automaton accepting all strings over $\mathbb{B}^2$ with at least two occurrences of the letter "11"



could be represented as in Figure 5.4.

The use of BDDs makes the representation very succinct in comparison to our earlier attempt to handle automata with large alphabets [53]. In most cases, we avoid the exponential blow-up associated with an explicit representation of the alphabet. We shall see that all operations on automata needed can be performed by means of simple BDD operations.

Another possibility would have been to use a two-dimensional array of ordinary BDDs. But that would complicate the operations on automata, because many more BDD operations would be needed.

5.6.2 Rewriting formulas

The first step in the translation consists of rewriting formulas so as to eliminate nested terms. Then all terms are variables and all formulas are among a small number of basic formulas.

5.6.3 Translating formulas

The translation is inductive. All automata corresponding to basic formulas have a small number of states (less than five!).

The composite formulas are translated by use of operations on automata. For $\neg\phi$, $\phi_1 \wedge \phi_2$ and $\exists P : \phi$, which are the ones left after rewriting, we need the operations of complement, product, projection, and determinization.

Complement Complementation is done by simply negating the bit vector representing the set of final states.

Product The product automaton A of two automata A_1 and A_2 is

$$(Q_1 \times Q_2, \mathbb{B}^k, (i_1, i_2), \delta, F_1 \times F_2),$$

where $\delta((q_1, q_2), b) = (\delta_1(q_1, b), \delta_2(q_2, b))$. We are careful, however, to consider only those states of A that are reachable from (i_1, i_2).

When considering a new state (q_1, q_2), we need to construct the BDD representing the corresponding part of the transition function δ. We use the binary apply operation on the BDDs corresponding to q_1 and q_2. For each pair of states (q', q'') encountered in a pair of leaves, we associate a unique integer in the range $\{0, 1, \ldots N-1\}$, where N is the number of different pairs considered so far. In this way, the new BDDs created conform with the standard representation.

Projection and determinization Projection is the conversion of an automaton over $\mathbb{B}^{k+1}$ to a nondeterministic automaton over $\mathbb{B}^k$ necessary for translating a formula of the form $\exists P : \phi$. On any letter $b \in \mathbb{B}^k$, there are two transitions possible in the nondeterministic automaton corresponding to whether the P-track is 0 or 1. Therefore this automaton is not hard to construct using the projection (restriction) operation of BDDs.

Determinization is done according to the subset construction. The use of the apply operation is similar to that of the product construction except that leaves hold subsets of states.

5.6.4 Minimizing

Minimization seems essential in order to obtain an effective decision procedure. For example, if a tautology occurs during calculations, then it is obviously a good idea to represent it using a one-state automaton instead of an automaton with e.g. 10,000 states.

The difficulty in obtaining an efficient minimization algorithm stems from the requirement to keep our shared BDDs in reduced form. Recall that a reduced BDD has no duplicate terminals or nonterminals. Such a BDD is just a specialized form of directed acyclic graph that has been compressed by combining structurally isomorphic nodes (see Aho, Hopcroft, and Ullman [5] or Section 3.4 of Cai and Paige [23]). In addition, a reduced BDD has no redundant tests [19].

Such a BDD is obtained by repeatedly pruning every internal vertex v that has both outedges leading to the same vertex w, and redirecting all of v's incoming edges to w.

Suppose that the shared BDD had all duplicate terminals and nonterminals eliminated, but did not have any of its redundant tests eliminated. Then it would be easy to treat the deterministic finite automaton combined with its BDD machinery as a single automaton whose states were the union of the BDD nodes and the original automaton states, and whose alphabet were zero and one. If this derived automaton had n states, then it could be minimized in $O(n \log n)$ steps using Hopcroft's algorithm [47]. Unfortunately, such an automaton would be too big.

For our purposes, the space savings due to redundant test removal is of crucial importance. But the important 'skip' states that arise from redundant test removal complicates minimization. Our algorithm combines techniques based on [5] with new methods adapted for use with the shared BDD representation of the transition function. For a finite automaton with n states and a transition function represented by m BDD nodes, the algorithm presented here achieves worst-case running time $O(\max(n, m)n)$.

Terminology A *partition* $\mathcal{P}$ of a finite set U is a set of disjoint nonempty subsets of U such that the union of these sets is all of U. The elements of $\mathcal{P}$ are called its *blocks*. A *refinement* $\mathcal{Q}$ of $\mathcal{P}$ is a partition of U such that any block of $\mathcal{Q}$ is a subset of a block of $\mathcal{P}$. If $q \in U$, then $[q]_{\mathcal{P}}$ denotes the block of partition $\mathcal{P}$ containing the element q, and when no confusion arises, we drop the subscript.

Let $A = (Q,\ \mathbb{B}^k,\ i_0,\ \delta,\ F)$ denote a deterministic finite automaton, and let $\mathcal{P}$ be a partition of Q, and $\mathcal{Q}$ a refinement of $\mathcal{P}$. A block B of $\mathcal{Q}$ *respects* the partition $\mathcal{P}$ if for all $q, q' \in B$ and for all $b \in \mathbb{B}^k$, $[\delta(q, b)]_{\mathcal{P}} = [\delta(q', b)]_{\mathcal{P}}$. Thus, δ cannot distinguish between the elements in B relative to the partition $\mathcal{P}$. A partition $\mathcal{Q}$ *respects* $\mathcal{P}$ if every block of $\mathcal{Q}$ respects $\mathcal{P}$. A partition is *stable* if it respects itself. The *coarsest, stable partition* $\mathcal{Q}$ *respecting* $\mathcal{P}$ is a unique partition such that any other stable partition respecting $\mathcal{P}$ is a refinement of $\mathcal{Q}$.

The refinement algorithm The minimal automaton A' recognizing $L(A)$ is isomorphic to the automaton defined by the coarsest stable partition $\mathcal{Q}^A$ of Q respecting the partition $\{F, Q \setminus F\}$. The states of A' are $\mathcal{Q}^A$, the transition function δ' is defined by $\delta'([p], b) = [\delta(p, b)]$, the initial state is $[i_0]$, and the set of final states is $F' = \{[f] | f \in F\}$.

Now we are ready to sketch our minimizing algorithm, which works by gradually refining a current partition.

- First split Q into an initial partition $\mathcal{Q} = \{F, Q \setminus F\}$. Note that $\mathcal{Q}^A$ is a refinement of this partition.

- Now let $\mathcal{P}$ be the current partition. We construct the new current partition $\mathcal{Q}$ so that it respects $\mathcal{P}$ while $\mathcal{Q}^A$ remains a refinement of $\mathcal{Q}$.

For each state q in Q consider the functions $f_q : \mathbb{B}^k \to \mathcal{P}$ defined by $f_q(b) = [\delta(q,b)]_{\mathcal{P}}$ for all q and b. Now let the equivalence relation $\equiv$ be defined as $q \equiv q' \Leftrightarrow (f_q = f_{q'} \wedge [q]_{\mathcal{P}} = [q']_{\mathcal{P}})$. The new partition $\mathcal{Q}$ then consists of the equivalence classes of $\equiv$. By definition of the f_q's, $\mathcal{Q}$ respects $\mathcal{P}$ and is the coarsest such partition implying the invariant.

We repeat this process until $\mathcal{P} = \mathcal{Q}$.

It can be shown that the final partition $\mathcal{Q}$ is obtained in at most n iterations and equals $\mathcal{Q}^A$. The preceding algorithm is an abstraction of the initial naive algorithm presented in Section 4.13 of [5].

The difficult step in the above algorithm is the splitting according to the functions f_q. However, we can here elegantly take advantage of the shared BDD representation. The idea is to construct a BDD representing the functions f_q for each state. We represent a partition of the states Q, by associating with each state $q \in Q$ a *block id* identifying its block. The BDD for f_q is calculated by performing a unary apply on the collection of shared BDDs, where the value calculated in a leaf is the block id. By a suitable generalization of the standard algorithm, it is possible to carry out these calculations while visiting each node at most once (assuming that hashing takes constant time). Thus the split operation requires time $O(\max(n,m))$. Since we use shared BDDs, we may use the results of the apply operations directly as new block ids.

The splitting step without hashing An alternative implementation of the splitting step is possible that achieves the same worst case time bound $O(\max(n,m))$ without hashing. It is instructive to first consider the case in which the shared BDDs are reduced only by eliminating redundant nodes but not by eliminating redundant tests. In this case the BDD may be regarded as an acyclic deterministic automaton D whose states are the BDD nodes, and whose alphabet is zero and one. Consider a partition $\mathcal{P}'$ of the BDD nodes defined by equivalence classes of the following relation. Two BDD leaves are equivalent iff their next states belong to the same block of partition $\mathcal{P}$. All decision nodes of the BDD are equivalent. The coarsest stable partition $\mathcal{Q}'$ that respects $\mathcal{P}'$ for automaton D can be solved in $O(m)$ worst case time by Revuz [104] and Cai and Paige [23], Sec. 3.4. Finding the equivalence classes of states in Q that point to BDD roots belonging to the same block of $\mathcal{Q}'$ (that is, finding the coarsest partition $\mathcal{Q}$ that respects $\mathcal{P}$) solves the splitting step in the original automaton in time $O(n)$.

In the case of fully reduced BDDs, the splitting step is somewhat harder, and a closer look at the BDD structure is needed. For each decision node v, $v.index$ represents a position in a string of length k such that $v.index < (v.lo).index \wedge v.index < (v.hi).index$. For each BDD leaf v we have $v.index = k$, and let $v.lo = v.hi$ be an automaton state belonging to Q. For each BDD node v we define function $f_v : \mathbb{B}^k \to \mathcal{P}$ much like the way functions f_q were defined earlier on automaton states. For each nonleaf v, f_v is defined by the rule $f_v(b) = f_{v.lo}(b)$ if $b_{v.index} = 0$; $f_v(b) = f_{v.hi}(b)$ if $b_{v.index} = 1$. For each leaf v, f_v is a constant function that maps every argument into an element (that is, a block) of partition $\mathcal{P}$.

If $q \in Q$ is an automaton state that points to a BDD root v, then, clearly, $f_q = f_v$. It is also not hard to see that for any two nonleaf BDD nodes v and v', $f_v = f_{v'}$ iff either of the following two conditions hold:

1. $v.index = v'.index \wedge f_{v.hi} = f_{v'.hi} \wedge f_{v.lo} = f_{v'.lo}$, or

2. $f_{v.hi} = f_{v.lo} = f_v \wedge v.hi = v'$.

This leads to the more concrete equivalence relation $\equiv$ on BDD nodes defined as $v \equiv v'$ iff $f_v = f_{v'}$ iff either,

1. $v.index = v'.index = k \wedge [v.lo]_{\mathcal{P}} = [v'.lo]_{\mathcal{P}}$, or

2. $v.index = v'.index < k \wedge v.hi \equiv v'.hi \wedge v.lo \equiv v'.lo$, or

3. $v.index < k \wedge v.lo \equiv v.hi \equiv v'$.

Note that two BDD nodes of different index can be equivalent only by condition (3). Note also, that we can strengthen condition (2) with the additional constraint $v.hi \not\equiv v.lo$ without modifying the equivalence relation. These two observations allow us to construct the equivalence classes inductively using a bottom-up algorithm that processes all BDD nodes of the same index in descending order, proceeding from leaves to roots. The steps are sketched just below.

1. In a linear time pass through all of the BDD nodes, place each node in a bucket according to its index. An array of $k + 1$ buckets can be used for this purpose.

2. Next, distribute the BDD leaves (contained in the bucket associated with index k) into blocks whose nodes all have lo successors that belong to the same block of $\mathcal{P}$. This takes time proportional to the number of leaves.

3. For $j = k-1, ..., 0$ examine each node v with $v.index = j$. Both nodes $v.lo$ and $v.hi$ have already been examined, and have been placed into blocks. Hence, a streamlined form of multiset sequence discrimination [23] can be used to place v either in an old block (according to condition (3)) or a new block (according to condition (2)) for nodes whose children belong pair-wise to the same old block.

The preceding algorithm computes the equivalence classes as the final set of blocks in $O(m)$ time. As before, we can use these equivalence classes to find the coarsest partition $\mathcal{Q}$ that respects $\mathcal{P}$, which solves the splitting step in the original automaton, in time $O(n)$. Thus, the total worst-case time to solve the splitting step is $O(\max(n, m))$ (without hashing).

In an efficient implementation of finite-state automaton minimization, when the splitting algorithm above is is performed repeatedly, we only need to perform the first step of that algorithm (that is, sorting BDD nodes according to index) once. Thus, the full DFA minimization algorithm runs in worst case time $O(\max(n, m)n)$ without hashing.

BDD reduction without hashing Sieling and Wegener[122] were the first to compress an arbitrary BDD into fully reduced form in linear time. Their result depended on a radix sort, which is closely related to the multiset discrimination technique that we use. However, their algorithm needs to maintain integer representations of BDD nodes, and it utilizes two arrays of size m. We can show how our algorithm just described can be modified to fully reduce an arbitrary BDD in worst case time linear in the number of BDD nodes (without hashing), but with expected auxiliary space k times smaller than Sieling and Wegener's algorithm.

Let $\mathcal{Q}'$ be the partition of BDD nodes produced by the algorithm. The states of the reduced BDD are the blocks in $\mathcal{Q}'$. For each block $B \in \mathcal{Q}'$, $B.index$ is the largest index of any BDD node contained in B. Let v' be any node belonging to B of maximum index. If v' is a BDD leaf, then B is a leaf in the reduced BDD (that is, $B.index = k$), and $B.lo = B.hi = v'.lo$. Otherwise, $B.lo = [v'.lo]_{\mathcal{Q}'}$ and $B.hi = [v'.hi]_{\mathcal{Q}'}$. The hi and lo successor blocks can be determined during the multiset sequence discrimination pass when a new block is first created. The index of the first node placed in a newly created block is the index for that block.

What distinguishes our algorithm from that of Sieling and Wegener is that our buckets in steps (2) and (3) are associated with actual BDD nodes (inside the main BDD data structure). Their buckets are associated with components of two auxiliary arrays of size m each. If we replaced each equivalence class by a single witness (as they do) each iteration of step (3), then our auxiliary space would be bounded by the maximum number of BDD nodes that have the same index. If BDD nodes were uniformly distributed among indexes, then this number is m/k, which would give us a k-fold advantage in auxiliary space over their algorithm. We expect a minor constant factor advantage in time as well, because our BDD nodes are represented by their locations instead of by computed integer values, and because we avoid array access in favor of less expensive list and pointer processing.

Work is in progress for exploring the "processing the smaller half" idea found in e.g. [96]. We should mention, however, that the current implementation of the minimization algorithm in practice seems to run faster than the procedures for constructing product and subset automata.

5.7 Current version of Mona

Mona is a continuing research project at BRICS and has since this publication been improved substantially in many ways. The basic ideas, such as the BDD representation of automata are still the same. For an update on the capabilities of the Mona tool we refer to [60].

Chapter 6

Control Synthesis

Contents

In this chapter we describe a practical framework for synthesizing controllers for discrete-event systems. The work has been published as a BRICS Technical Report [115]. Our approach is based on Mona [44, 62, 60, 88], the tool for deciding the weak second-order theory of one or two successors described in Chapter 5. To make the approach practical and applicable to real world problems we make use of techniques for modular synthesis and decentralized control. Finally, we describe how to effectively combine control synthesis, sequence testing, and model checking in the Mona-based framework.

Our method has been implemented as part of the <bigwig> project and used extensively in the application domain of interactive web services. Furthermore, we have used the method to generate controllers for embedded systems, more specifically to generate controllers for LEGO Mindstorms robots

Our work shows that the described techniques along with Mona as the leveraging engine comprise a promising practical approach to traditional theoretically well-studied logic-based control synthesis.

6.1 Introduction

In the design of safety critical systems it is of great importance that certain guarantees can be made about the system. Traditionally, model checking has served as a successful method for establishing such properties.

We describe a method for synthesizing controllers from logical specifications. That is, when implementing a system one can concentrate on the functionality of the system. Afterwards, appropriate safety requirements can be formulated separately and then compiled into a control agent that when run in conjunction with the system will guarantee that the stated requirements are met.

The control synthesis problem was first formulated by Church as a realization problem [24] and later solved by Büchi and Landweber using two-player games [22]. The use of automata for the control of reactive systems is not new. Ramadge and Wonham give give a survey of traditional control theory in [103]. The use of logic in combination with automata has several advantages over traditional automata-based approaches.

- Requirements written in the specification language are *automatically* compiled into a control agent by use of the Mona tool.
- The specification language, based on monadic second-order logic on strings (M2L-Str), is *succinct* and *expressive* allowing complicated requirements to be formulated easily and compact.
- Specifications are easy to *maintain*, that is it is easier to add, remove, and modify parts of a logic-based specification.
- The generated control agent *efficiently* implements the safety requirements by means of the developed techniques for modular synthesis and decentralized control.

The Mona tool is an efficient translator from the weak second-order theory of one or two successors to finite state automata. Monadic second-order logic on strings (M2L-Str) is a sub-logic expressible in the weak second-order theory of one or two successors. The existence of Mona allows one to base the design of the control specification language on M2L-Str and yet have a powerful tool at hand for doing computations in the logic.

When dealing with control synthesis it is important that formulating and changing the control requirements is relatively easy. Using a logic as the core of ones specification language therefore seems like a much more natural choice than the traditional use of automata. First of all logic is in its nature much closer to the language in which informal requirements are written. Furthermore, as noted above it is very easy to add, remove, and change parts of a logic-based specification whereas an automata-based one is very sensitive to even small local changes.

The use of formal methods in system design is gradually becoming more and more common. One of the major challenges in this field is to manage the large state spaces that naturally occur when dealing with independent requirements and many parallel processes. By focusing on the control synthesis problem rather than the traditional model checking and verification problem we can use symbolic representation techniques in a radical and much more extensive way than what is normally possible in the world of model checking and verification.

In the rest of the chapter we first introduce some terminology. Then, Mona, the engine of our control synthesis process is described along with the specification language, M2L-Str. The general structure of a control specification is described and it is explained how a specification is processed and turned into a control agent. After this, techniques for modular synthesis and decentralized control are developed. Finally, we describe applications of our method to the domain of control synthesis for interactive web services. We conclude with a discussion of related and future work.

6.2 Terminology

In standard control technology one has a plant (the object to be controlled), and an agent or supervisor doing the controlling. Here, we consider only abstract plants in the sense that they are modeled by labeled transition systems. Our job is to synthesize, from a set of requirements, the code for the agent doing the controlling in such a way that the agent together with the plant satisfies the stated requirements.

In order to be able to control a labeled transition system, one partitions the event set Σ into *uncontrollable* and *controllable* events: $\Sigma = \Sigma_u \cup \Sigma_c$. The controllable events can be disabled by the control agent at any time, whereas the uncontrollable ones are performed autonomously by the plant without any possible interference by the control agent.

Let $pre(L)$ denote the prefix closure of a language L, and let $unc(L)$ denote closure of L under concatenation of uncontrollable events. That is, let

$$pre(L) = \{ v \in \Sigma^* \mid \exists w \in \Sigma^* : vw \in L \} \text{ and}$$
$$unc(L) = \{ vw \in \Sigma^* \mid v \in L \wedge w \in \Sigma_u^* \}.$$

A language, L over $\Sigma = \Sigma_u \cup \Sigma_c$ is called a *control language* if it satisfies the two properties $pre(L) = L$ and $unc(L) = L$. Note, that given two control languages L and L', then $L \cup L'$ and $L \cap L'$ are also control languages. That is, control languages are closed under union and intersection. This follows trivially from closedness under union and intersection of each of the properties $pre(L) = L$ and $unc(L) = L$.

An *automaton* is a structure $A = (Q, q^{\text{in}}, \Sigma, \rightarrow, F)$, where Q is a set of states with initial state $q^{\text{in}} \in Q$, Σ is a finite set of events, $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation, and $F \subseteq Q$ the set of acceptance states. We shall use $q_1 \xrightarrow{\sigma} q_2$ to denote $(q_1, \sigma, q_2) \in \rightarrow$. A sequence $w = \sigma_0\sigma_1 \ldots \sigma_{n-1} \in \Sigma^*$ is said to be *accepted* by the automaton A if there exists a run of A that reads the sequence w and ends up in an accepting state q, that is, if there exist $q_1, \ldots, q_{n-1} \in Q$ and $q \in F$, such that $q^{\text{in}} \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_{n-2}} q_{n-1} \xrightarrow{\sigma_{n-1}} q$. We shall denote by $L(A)$ the *language* recognized by an automaton, that is,

$$L(A) = \{ w \in \Sigma^* \mid A \text{ accepts } w \}.$$

In the following a control agent, C, will be represented by an automaton, A_c. Furthermore, the language of a control agent is required to be a control language, that is, it must be prefix closed and respect the partition of events into uncontrollable and controllable ones.

The *disabling* of controllable events is done using the transition relation of the control agent. If an event, σ, is not enabled in a given state, that is, if the σ transition from that state goes to a non-accepting state (if minimized, control agents have by prefix closedness at most one non-accepting state), then the plant cannot perform the σ action until the control agent changes its state.

Respecting the partition of events into uncontrollable and controllable ones is done by ensuring that an uncontrollable event is always allowed by the controller. At the level of automata this is done by requiring that a control agent must have in each state an enabled outgoing transition for each uncontrollable event.

6.3 Mona — the synthesis engine

The sub-logic of the weak second-order theory of one or two successors that we are using is M2L-Str. It is a very expressive logic in which several other logics can be encoded. For example, interval logic and linear time temporal logic can easily be encoded in M2L-Str. A control specification consists of a number of M2L-Str formulae. The specifics of the overall structure of a control specification is dealt with in the section on the specification langauge.

M2L-Str is a very succinct logic. In fact one can write M2L-Str-formulae that have corresponding automata of non-elementary size. That is, if a formula is of size n, then its corresponding automaton can be of size $2^{\cdot^{\cdot^{\cdot^{2^{n}}}}}$. Seen from a complexity point of view this is very bad news. But seen from the point of view of language design, it is useful to have a succinct logic as the basis of ones specification language. A succinct logic will imply shorter specifications which are often easier to write.

One might argue in favor of other specification formalisms such as high-level Petri Nets [55] or Message Sequence Chart diagrams [79]. Being a logic formalism, however, M2L-Str has the advantage that specifications can be developed iteratively, that is, one can easily add, delete, and modify parts of a specification. Furthermore, M2L-Str is very powerful and yet just simple enough to actually subject it to calculations using Mona.

More specifically, the abstract syntax of M2L-Str is given by the following grammar:

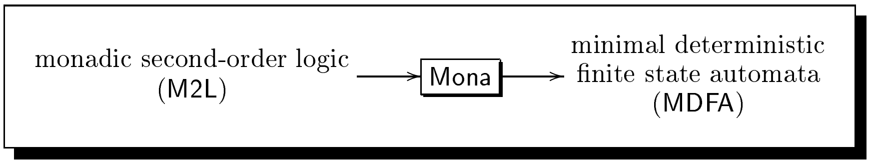
$$\begin{array}{rcl} \phi & ::= & \exists_1 p : \phi' \mid \exists_2 P : \phi' \mid \neg\phi' \mid \phi' \wedge \phi'' \mid \phi' \vee \phi'' \mid \phi' \Rightarrow \phi'' \\ & \mid & t \in T \mid \sigma(t) \mid \ldots \\ t & ::= & t + 1 \mid \ldots \\ T & ::= & T' \cup T'' \mid \ldots \end{array}$$

That is, the logic has universal and existential quantifications over both first order variables (ranging over positions in the sequence of events) and monadic second-order variables (ranging over sets of positions). Also, it has standard boolean connectives such as negation, conjunction, disjunction, and implication as well as operations on first order terms such as given a position t one can point out its successor $(t+1)$, operations on second-order terms such as taking the union of two sets, plus the basic formula that tests membership of a position in a position set. Furthermore, M2L-Str has basic formulae, $\sigma(t)$, to test whether an event σ can be found at position t.

A formula ϕ in M2L-Str over the event set Σ will – when interpreted over a finite sequence of events w – either evaluate to true or to false and we shall write $w \models \phi$ or $w \not\models \phi$, respectively. The *language* associated with ϕ is

$$L(\phi) = \{\, w \in \Sigma^* \mid w \models \phi \,\}.$$

Such a language is guaranteed to be regular. In fact, it has been known since the sixties that M2L-Str characterizes regularity [21, 32]. The Mona tool implements the constructive proof of the fact that for each M2L-Str formula there is a minimal deterministic finite state automaton accepting the language of the formula. That is, Mona translates M2L-Str formulae into minimal deterministic finite state automata (MDFA).

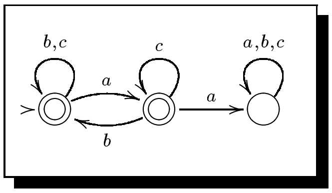


Example 6.3.1 Let $\Sigma = \{a, b, c\}$. The M2L-Str formula

$$\forall p, p'' : (p < p'' \wedge a(p) \wedge a(p'')) \Rightarrow \exists p' : p < p' < p'' \wedge b(p')$$

is true only when interpreted over a sequence in which any two occurrences of a will have an occurrence of b in between.

Using Mona we have a method for easily producing from the above formula an MDFA that accepts exactly those sequences that make the formula true. The MDFA corresponding to the formula thus looks as follows.



6.4 Control Specification Langauge

The language for writing control requirements is essentially the logic M2L-Str that is accepted as input to Mona except for a few things that have been added to the syntax. Mona does not have support for the definition of an input alphabet, that is, the basic formulae $\sigma(t)$ are not part of the Mona syntax. However, in the definition of control requirements one declares an input alphabet and the formulae $\sigma(t)$, are part of the syntax. The high-level notion of input alphabet gets compiled into Mona syntax using second-order variables. Furthermore, when writing control requirements one can make use of control specific syntactic sugar such as the forbid construct,

$$\textbf{forbid } \boldsymbol{\sigma} \textbf{ when } \boldsymbol{\phi} \equiv \forall \text{ now} : \neg(\boldsymbol{\sigma}(\text{now}) \wedge \textbf{restrict}(\boldsymbol{\phi}, \text{now})),$$

where the function restrict returns the given formula, $\boldsymbol{\phi}$ with all quantifications, ∀t or ∃t, restricted by t≤now.

Example 6.4.1 A mutex control requirement allowing only one take event in between give events could then be defined as follows.

$$\text{mutex(a,b)} = \textbf{forbid} \text{ take } \textbf{when } \exists \text{t: take(t)} \wedge \forall \text{t': t<t'} \Rightarrow \neg\text{give(t')}.$$

The resulting automaton would be the three state mutex automaton of Example 6.3.1 with a and b replaced by take and give, respectively.

The overall structure of a control specification is as follows. First, it has declarations of controllable and uncontrollable events. Then, possibly preceeded by definitions of predicates and macros, a number of named control requirements, $\phi_1, \ldots, \phi_n$, follow.

```
controllable {e_1, ..., e_k}
uncontrollable {e_{k+1}, ..., e_m}
requirements {
   r_1 : φ_1;
      ⋮
   r_n : φ_n;
}
```

6.4.1 Checking and compiling basic control specifications

The way a control specification gets checked and compiled is as follows. First, the formulae $\phi_1, \ldots, \phi_n$ get parsed, desugared, and the labels get encoded using second-order variables. Then, they are given to Mona. The resulting automata, $A_1, \ldots, A_n$, are then checked to be legal control agents, that is, it is checked that each of the $L(A_i)$'s are control languages. Since Mona always responds with minimal automata, this property can be checked as follows.

Prefix closedness is ensured by inspecting the set of states. All states except for at most one looping error state must be accepting states. Another possibility is to use the Mona operator *pre* which given a formula gives the prefix closure of that formula. Then one can just check whether

$$\bigwedge_{1 \leq i \leq n} (pre(\phi_i) \Rightarrow \phi_i)$$

is a tautology, that is, whether Mona returns the automaton representing Σ^* on this question.

Closedness under concatenation of uncontrollable events is checked by inspection of the transition table for accepting states. All transitions going from an accepting state labeled with an uncontrollable event must go to an accepting state.

If the property of controllability fails for one or more of the control requirements, appropriate error messages are given. Otherwise, Mona is asked to produce the automaton corresponding to $\phi_1 \wedge \cdots \wedge \phi_n$ which by closedness of control languages under intersection is guaranteed to be a legal control agent.

6.5 Improvements on Controller Generation

In the following we present two improvements on the generation of controllers. The first improvement concerns the *state space explosion* problem. Via the use of *modular synthesis* the state space explosion can often be reduced significantly. The second improvement increases parallelism of the system via *inference of independence information*. This allows for distributed or *decentralized control*.

Before going into details about how to accomplish modular synthesis and decentralized control we introduce in this section the notion of distributed automata on which the two improvements are based.

A *distributed alphabet* $\tilde{\Sigma} = (\Sigma_1, \dots, \Sigma_K)$ with $K \geq 1$ is a finite collection of finite, non-empty alphabets. We shall denote by Σ the union of the not necessarily disjoint alphabets Σ_i.

A *distributed automaton* over $\tilde{\Sigma}$ is a structure $\mathcal{A} = (A_1, \dots, A_K)$ consisting of finite state automata $A_1, \dots, A_K$, with $A_i = (Q_i, q_i^{\text{in}}, \Sigma_i, \rightarrow_i, F_i)$. The derived behavior of $\mathcal{A}$ will be as the behavior of the finite state automaton

$$A = (Q_1 \times \cdots \times Q_K, (q_1^{\text{in}}, \dots, q_K^{\text{in}}), \Sigma, \rightarrow, F_1 \times \cdots \times F_K),$$

where $(q_1, \dots, q_K) \xrightarrow{\sigma} (q_1', \dots, q_K')$ if and only if when $\sigma \in \Sigma_i$ then $q_i \xrightarrow{\sigma}_i q_i'$, otherwise $q_i = q_i'$. By $L(\mathcal{A})$ we shall denote the language recognized by the distributed automaton $\mathcal{A}$ which is just the language recognized by A, that is, $L(\mathcal{A}) = L(A)$. Note that given a distributed automaton $\mathcal{A} = (A_1, \dots, A_K)$ over $\tilde{\Sigma}$ we also have

$$L(\mathcal{A}) = \{\, w \in \Sigma^* \mid \forall i \in \text{Loc} : (w|\Sigma_i) \in L(A_i) \,\},$$

where $w|\Sigma_i$ denotes the projection of the string w onto the ith alphabet Σ_i. Note that for $K = 1$ the notion of a distributed automaton coincides with the notion of a simple finite state automaton.

The kind of distributed automata that we will be using in the first part to reduce the state space will be automata over distributed alphabets where $\Sigma_1 = \cdots = \Sigma_K$, that is, the derived behavior of the distributed automaton will simply be as the behavior of the product automaton $A_1 \times \cdots \times A_K$. Later on – when we want to increase parallelism – we will consider general distributed alphabets where the sub-alphabets are not necessarily equal to each other.

6.5.1 Modular synthesis and state space explosion

The state space explosion problem occurs when one wants to generate from the list of control requirements a control agent which can be used to supervise the execution of the plant. While each of the control requirements might correspond to simple automata, their conjunction can still have a corresponding automaton of potentially exponential size.

Example 6.5.1 Consider the example of having a plant consisting of many parallel processes trying to enter a total number of n critical regions. The control requirement is that we must have mutual exclusion in each of the critical regions.

Entering and leaving critical region i is signaled by the events take_i and give_i, respectively. The take_i events are controllable as opposed to the give_i events which are not. In order to obey the requirement of mutual exclusion in the critical regions the following list of control requirements are needed in the control specification.

```
controllable {take_1, ..., take_n}
uncontrollable {give_n, ..., give_n}
requirements {
    m_1: mutex(take_1,give_1);
        ⋮
    m_n: mutex(take_n,give_n);
}
```

The corresponding control agent will have a total of $2^n + 1$ states. It will look like the nth dimensional cube plus the error state.

We shall attack the state space explosion problem by using distributed automata. The crucial observation that makes it possible to easily use distributed automata in the context of control synthesis is the following. A control specification has the form of a big conjunction of control requirements, that is, it is a collection of requirements that all have to be satisfied. Thus, one can build and use the symbolic representation of the product (the distributed automaton) all the way through. In comparison, when doing model checking one most often needs to compute the product state space at one point or another. More precisely, the symbolic representation is achieved as follows.

Proposition 6.5.2 *Let $C = \{\, \phi_i \mid 1 \leq i \leq n \,\}$ be a set of control requirements, and let $C_1, \ldots, C_K$ be a partition of C with corresponding automata $A_1, \ldots, A_K$. Then the corresponding distributed automaton $\mathcal{A} = (A_1, \ldots, A_K)$ will behave exactly as the automaton corresponding to the full set C of conjuncts because*

$$\begin{aligned}
L(\bigwedge_{\phi \in C} \phi) &= L(\bigwedge_{1 \leq i \leq K} (\bigwedge_{\phi \in C_i} \phi)) \\
&= L(\bigwedge_{\phi \in C_1} \phi) \cap \cdots \cap L(\bigwedge_{\phi \in C_K} \phi) \\
&= L(A_1) \cap \cdots \cap L(A_K) \\
&= L(A_1 \times \cdots \times A_K) \\
&= L(\mathcal{A}).
\end{aligned}$$

By the above proposition we can thus use the distributed automaton $\mathcal{A}$ as our control agent. If the partition is chosen appropriately, modular synthesis can lead to considerable reduction in the size of the state space.

Note, however, that the hard part when using modular synthesis is to come up with the right partition. Here, the use of logic comes in very handy in the sense that a control specification in terms of a big conjunction is a good starting point for choosing the right partition.

For instance, in Example 6.5.1 we partition the control requirements such that each requirement constitutes a class in itself. Then, the controller will consist of n copies of the three state mutex control agent and thus have only a linear number of states ($3n$) instead of the original exponential number of states ($2^n + 1$).

The modular synthesis technique has been implemented in <`bigwig`>, where it has been used with great success. In fact, one of the advantages of the technique is that is will almost always yield a smaller state space, the drawback being that more synchronization is needed inside the control agent at runtime.

6.5.2 Decentralized control and independence analysis

It was shown in the previous section that the use of distributed automata can reduce the size of the state space in frequently occurring cases. In this section we will improve on the fact that the control is performed by a central component. Having a central control agent can potentially slow down the system. This problem of the control agent being a bottleneck can in many cases be avoided by exploiting independence in the control requirements. First, a bit of terminology.

We shall call a transition $q \xrightarrow{\sigma} q'$ *state preserving* if $q = q'$. An event σ is said to be *dead* if all σ-transitions are state preserving, that is, if

$$\rightarrow \cap (Q \times \{\sigma\} \times Q) \subseteq \{\, q \xrightarrow{\sigma} q \mid q \in Q \,\}.$$

A σ-transition is *dead* if σ is dead. In a distributed automaton $\mathcal{A} = (A_1, \ldots, A_K)$ an event or transition is said to be *locally dead* in A_i if it is dead in A_i. One can now make the following crucial observation.

Proposition 6.5.3 *Given a distributed automaton $\mathcal{A} = (A_1, \ldots, A_K)$. Let $\mathcal{A}'$ be the distributed automaton $\mathcal{A}$ where locally dead events and transitions have been removed. If $\Sigma' = \Sigma$ then $L(\mathcal{A}') = L(\mathcal{A})$.*

That is, removing locally dead events and transitions is a language preserving operation on distributed automata. (Only, when removing the very last occurrence of a label we must note that it is always enabled and does not change the state.)

The above proposition easily extends to an algorithm where we iterate through the automata $A_1, \ldots, A_K$ and for each $i \in \{1, \ldots, K\}$ remove all locally dead events and transitions in A_i. Running this algorithm on a distributed automaton $\mathcal{A}$ will result in a new distributed automaton $\mathcal{A}'$ with exactly the same overall behavior, but with minimized requirements regarding synchronization between the different components.

To achieve decentralized control, consider the undirected graph $G = (V, E)$ with nodes and edges

$$V = \{A_1, \ldots, A_K\}, \qquad E = \{\, (A_i, A_j) \mid \Sigma_i \cap \Sigma_j \neq \emptyset \,\}.$$

Let $C_1, \ldots, C_n$ be the connected components of G. Since each C_i is a collection of automata it can be considered a distributed automaton. These distributed automata have pairwise disjoint alphabets. Therefore, they can be run as control agents completely in parallel without violating any of control requirements.

Furthermore, each of the distributed automata $C_1, \ldots, C_K$ consist of finite state automata that due to the removal of dead events and transitions are loosely coupled and thus need very little internal synchronization.

Example 6.5.4 Using the analysis described above, the control requirements of Example 6.5.1 can be shown to be completely independent. That is, using n separate autonomous control agents, one for each of the mutex requirements, is a valid way of controlling the plant.

Inference of independence information via search for locally dead events and transitions will often lead to a substantial increase of parallelism in the controller and whenever possible decentralized control is obtained. This increase of parallelism will improve on the overall performance of the system, that is, on the flow capacities of the supervised plant.

6.6 Possible Extensions

Going from the basic use of formulae as control requirements, we now turn to some possible extensions of the framework. First, we explain how partial information about the behavior of the plant can be used to sometimes reduce the state space of the generated controller. Then, we discuss two extra features that can easily be integrated into our framework, namely checking for containment of example traces and linear time model checking.

6.6.1 Partial information about plant behavior

In some cases partial information about the behavior of the plant to be controlled is available either in terms of an automaton or a formula. By constructing the conjunct of the control specification and the partial system information one often ends up with a controller with a smaller state space. Thus this extra information can be of great use later on when checking test runs and specifications. Note, that the information given (though not precise) should be safe. That is, the formula should describe a superset of the possible executions of the plant. Otherwise, the resulting controller would be too restrictive.

Now, let in the following ϕ_{all} denote the conjunction of all the control requirements and the plant behavior formula (supposing the plant behavior information is supplied using a formula), that is,

$$\phi_{\text{all}} \equiv (\bigwedge_{1 \leq i \leq n} \phi_i) \wedge \phi_p.$$

6.6.2 Test sequences

One problem with writing control requirements is that only safety requirements can be stated. This is due to the fact that our model operates only on finite strings. Alternatively, one could base the language on monadic second-order logic on *infinite* strings and then compile into Büchi automata. This would allow one to define liveness properties as well. However appealing this scheme might seem, it will not work in practice, we claim. The main reason why the Mona tool works at all, is that one can always minimize the various automata that are computed along the way. Without this property of having canonical representations, the automata would very soon be of unmanageably large size and only very simple calculations would be doable. Thus one has to, at least in this setup, stick with safety requirements.

Therefore, when debugging a set of control requirements, it is often helpful to be able to at least check that certain traces or sequences of events are in fact possible executions in the plant under supervision of the control agent generated from the control requirements. Test sequences could also be constructed using Message Sequence Chart Diagrams.

Checking test sequences The test sequences $w_1, \ldots, w_l$ can be checked as follows. Compute the automaton for ϕ_{all} using Mona. Then run this automaton on each of the sequences $w_1, \ldots, w_l$ to check whether it ends up in an accepting state. If a sequence does not pass the checking, the longest prefix that does pass the test is displayed along with the full sequence and an error message.

The checking of test sequences can be adopted to the modular synthesis approach described earlier, that is, test sequences can be checked without necessarily computing the automaton for the full conjunction of ϕ_{all}. The test sequences can be checked in the following alternative way.

Compute the automaton for each of the control requirements $\phi_1, \ldots, \phi_n$ as well as the plant predicate ϕ_p using Mona. Then run each of these automata on each of the sequences $w_1, \ldots, w_l$ to check whether one ends up in an accepting state.

Note that, better diagnostic information can be given when using the latter approach. If an error occurs, it can be stated exactly which control requirement is violated by which sequence and where in that particular sequence the control requirement is violated.

6.6.3 Safety specification and model checking

Sometimes, control requirements are written with the particular plant to be controlled in mind. In this case there might be some general safety requirements that has to be satisfied on top of the existing control requirements. Of course, one way of ensuring satisfiability of these general safety requirements would be simply to add them as control requirements. This, however, might constrain execution rapidly. Furthermore, the state space of the control agent might be enlarged substantially. Instead, by adding them as a specification to be satisfied, it can automatically be checked whether the restrictions implied by the stated control requirements together with the information given about the plant will imply satisfaction of the specification.

This possibility of adding a specification and having it checked can be seen as way of doing model checking. Since everything is trace-based, the model checking is done in a linear time setting.

In model checking one has a system or a model, M, which is to be checked against a specification ϕ written in some kind of logic, say linear time temporal logic (LTL) [101]. In our case the model is the (possibly stated) behavior of the plant restricted by the declared control restrictions. If the model satisfies the formula, normally written as $M \models \phi$, then when using M, one is guaranteed that it obeys the behavior described by ϕ. If, on the other hand, $M \not\models \phi$, then it is considered good practice for the model checker to provide a counter example, that is, the trace of a possible execution in M which does not satisfy ϕ. Note, however, that since only partial information about the plant is given, it might be that the counter example is not a possible run of the real plant. On the other hand if the checking goes through, one can be sure that the specification is satisfied.

Checking safety specifications Given a specification ϕ_s it is straightforward to check whether it is satisfied by the system. Be system we mean the described behavior of the plant restricted by the declared control requirements. One just checks in Mona whether $\phi_{\text{all}} \Rightarrow \phi_s$. Note, that ϕ_{all} as defined earlier is the conjunction of the control requirements and the formula describing the plant. If the implication is not accepted by Mona, a counter example will be given, that is, a sequence of events allowed by the plant and the control restrictions which does not satisfy ϕ_s. Again, the message sequence chart diagrams might prove to be a useful formalism for displaying counterexamples.

Checking of specification formulae, however, as already hinted upon earlier, suffers from not being able to use the symbolic representation presented earlier in the discussion of modular synthesis. In order to check a specification, ϕ_s, one has to check the implication $(\phi_1 \wedge \cdots \wedge \phi_n \wedge \phi_p) \Rightarrow \phi_s$. Thus there is no way around computing the conjunction $(\phi_1 \wedge \cdots \wedge \phi_n \wedge \phi_p)$.

6.7 Example Applications

The framework presented here has been used in connection with both concurrency control in interactive Web services [116] and the control of embedded systems in terms of LEGO Mindstorms robots [50]. We give in this section a short description of these two example applications.

6.7.1 Interactive Web services

As mentioned our method has been implemented as part of <bigwig>, a tool for specifying and generating interactive web services. Our method for control synthesis is used as an integral part of <bigwig> to define safety constraints. In fact, via use of a powerful macro mechanism [14] the method has been used to extend the Web programming language in <bigwig> with concepts and primitives for concurrency control, such as, semaphores and monitors.

Examples have proven that the method is not only easy to use by virtue of the powerful logic-based definition language but also practical because of the efficiency of the generated code for the control agents. The latter due to the modular synthesis technique. In the domain of web services where all processes normally run on the same machine (the web server) we have not yet seen the need for decentralized control. But as web services gradually move towards distributed implementations, the need for decentralized control will become evident.

The <bigwig> implementation provides an even more expressive specification language than what is presented here. To allow formulation of requirements that go beyond regularity, a notion of triggers is used in the specification language. The triggers are compiled into counters in the corresponding control agents. For a more thorough description see Chapter 7 [116].

6.7.2 LEGO Mindstorms

In the world of LEGO RCX robots one is restricted to use a crudely simple almost assembly-like language for writing control programs. This makes it almost impossible to take difficult synchronization aspects into account. Using the approach presented in this chapter plus tools for wrapping the existing RCX robot control program and automatic transformation of Mona automata into RCX code one can concentrate on the high-level programming of the robot and simply specify the concurrency control separately in M2L-Str.

The specific RCX construction that we tested the framework on was a LEGO crane with three engines. The associated constraints were dealing with mutual exclusion and priority. For more details we refer to Chapter 8 [50].

6.8 Related Work

The idea of using temporal logic as a language from which control processes can be generated is not new. For example in [76] the whole system, that is, not only the controller, is synthesized from a temporal logic specification.

In [14, 116] similar results are obtained for the restricted case where all events are controllable. The results are being used extensively in the <bigwig> project [33] where the controlled plant is an interactive web service.

David Sands and Nils Klarlund pointed out a resemblance between our approach and the notion of path expressions [7]. Path expressions are used to specify allowed executions of a concurrent program. That is, using regular expressions one can restrict execution of the concurrent program to traces that conform to the stated expression. While being very elegant, this approach is tailored toward control of concurrent programs and as such, it does not have a mechanism for distinguishing between controllable and uncontrollable events. When implementing path expressions or something similar, a tool such as AMoRe [78] would be needed to transform regular expressions into automata.

Recent work of P. Madhusudan and P. S. Thiagarajan on controllers for discrete event systems via morphisms [75] considers the problem of synthesizing a correct controller from a branching time perspective. Via use of behavior preserving morphisms they present polynomial upper bounds of both the problem of deciding the existence of a controller and if it exists of the size of it. In a "true" concurrency model, however as it is also shown in [75], even the problem of checking for the existence of a controller is undecidable.

6.9 Conclusions and Future Work

We have described a method for synthesis of a control process from a high-level logic-based specification. Techniques for coping with the state space explosion problem have been presented. The method has been implemented using the Mona tool and has successfully been applied in the domains of interactive web services and LEGO Mindstorms robots.

The that fact that no liveness properties can be checked is rather unsatisfactory. Simple checks can easily be added and implemented by inspection of the automaton. For instance, one should be able to check that from any accepting state it is possible to get to a state where a certain event, e, is enabled. To check whether an event is always enabled, one can just make it an uncontrollable event, and the property will automatically be checked.

Decentralized control and model checking cannot readily be combined, that is, one has to compute the product state space of the controller in order to check it against a specification. If, however, the specification has the form of a conjunction and the conjuncts are independent in a way which is compatible with the independence found in the control requirements, the specification can be checked without explicitly computing the full state space. It would be interesting to examine whether the described scenario is in fact frequently occurring.

The techniques presented in this paper along with Mona as the leveraging engine thus comprise a promising approach to logic-based control synthesis.

Chapter 7

Controllers for Web Services

Contents

This chapter is on distributed safety controllers for web services. The work has been published in the proceedings of Fundamental Approaches to Software Engineering, FASE'98 [116]. It is joint work with Michael I. Schwartzbach.

We show how to use high-level synchronization constraints, written in a version of monadic second-order logic on finite strings, to synthesize safety controllers for interactive web services. We improve on the naive runtime model to avoid state-space explosions and to increase the flow capacities of services.

7.1 Introduction

An Interactive Web Service consists of a global state (typically a database) and a number of distinct sessions that each contain some local state and a sequential, imperative action. A web client may invoke an individual thread of one of the given session kinds. The execution of this thread may interact with the client and inspect or modify the global state.

To alleviate laborious low-level encodings of such services, the Mawl language [67, 28] has been suggested as a high-level notation that is compiled into low-level CGI-scripts. It directly provides programming constructs corresponding to global state, dynamic document, sessions, local state, imperative actions, and client interactions. This system shows great promise to facilitate the efficient production of reliable web services.

While Mawl thus offers automatic synthesis of many advanced concepts, it still relies on standard low-level semaphore programming for concurrency control. We have designed a variation of Mawl, called <`bigwig`>, on which we are currently performing a number of experiments. One of these is to synthesize the concurrency control from a high-level notation that is designed to be simple and intuitive. Our notation is based on second-order monadic logic on finite strings, M2L-Str.

As an example of a <`bigwig`> service, consider the example in Fig. 7.1, which provides a counter for a page. The intended behavior should be clear. By default <`bigwig`> provides exclusive write-access to components of the global state, but this is clearly not enough even for this simple example, where the updates of the counter variable must be atomic, which requires some sort of critical region.

Larger web services often require quite complicated concurrency control, which is hard to implement and maintain (and not the kind of issue on which most web programmers want to spend their time).

The web programming environment, with rapidly changing code, fast machines, and slow networks seems an ideal niche for a radical approach of synthesizing finite-state controllers from high-level specifications without suffering an unacceptable performance loss compared to hand-written code. This paper provides the foundations for these ideas.

```
service {
    global counter: int = 0;
    document ThePage { You are visitor number < [num] >. };
    session ReadMe {
        counter:=counter+1;
        show ThePage[num ← counter]
    }
}
```

Figure 7.1: A simple <bigwig> service.

7.2 Labeled Services

Before presenting the actual high-level notation for concurrency control we need to make one important extension to the basic language for writing service code. The high-level notation for concurrency control needs a way of referring to points in the service code. For this purpose we add the possibility of having labels in the code. As an example it might very well later turn out to be advantageous to be able to refer to, say, the beginning and the end of a critical region.

With labels in the service code, a run of a service gives rise to the sequence of labels that are passed in turn during the run. We have the basic assumption that no two labels are passed at exactly the same time. In the absence of this assumption, we should replace the word sequence by pomset, partially ordered multi-set. Later, though, we do consider independence models in order to avoid the state explosion problem and to increase parallelism.

In addition to the labels added by the programmer, the following labels are generated automatically because standard safety requirements almost always involve these labels.

- For each global variable X we generate the labels take-X and give-X. These labels are put in just before and just after each assignment to the global variable X. They will make us able to ensure that global variables can only be updated by one session thread at a time.
- For each session definition A we generate labels start-A and end-A. They are put at the beginning and at the end of session A, respectively.

The kind of sequences that a run of a service S gives rise to will thus be strings over the alphabet Σ_S given by

$$\begin{aligned}\Sigma_S = \mathit{labels}(S) \;\cup\; \{\,\textsf{take-X} \mid X \in \mathit{globals}(S)\,\} \;\cup\; \{\,\textsf{give-X} \mid X \in \mathit{globals}(S)\,\} \\ \cup\, \{\,\textsf{start-A} \mid A \in \mathit{sessions}(S)\,\} \;\cup\; \{\,\textsf{end-A} \mid A \in \mathit{sessions}(S)\,\},\end{aligned}$$

where *labels*, *globals*, and *sessions* are the functions that given a service S evaluates to the names of the labels, global variables, and sessions of S, respectively. Where obvious from the context we will drop the subscript S.

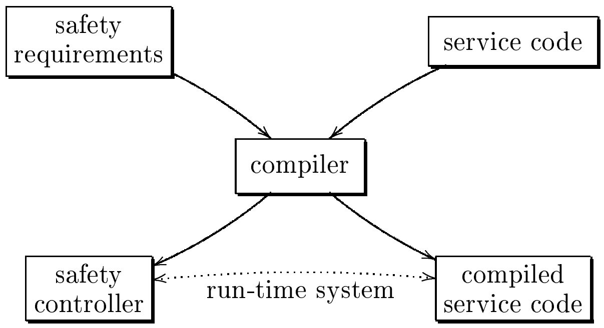


Figure 7.2: Overview of the setup.

7.3 Safety Requirements

While programming a service, one often needs to make sure certain properties hold. We offer a way of synthesizing runtime controllers from static *safety requirements*. These requirements — written in a dialect of M2L-Str — can together with the service code be compiled into executable code containing a runtime system that automatically ensures that the safety requirements are met, namely by compiling the safety requirements into a runtime safety controller. For a diagram of the overall compilation process see Fig. 7.2.

M2L-Str is a very expressive logic in which several other logics can be encoded, such as, interval logic and all sorts of linear time temporal logics. For an introduction to and a discussion of M2L-Str see Chapter 5 [44]. The specific high-level notation built on top of M2L-Str for writing <bigwig> safety requirements is called the *<bigwig> service logic*. The specifics of <bigwig> service logic are dealt with later in this chapter.

One might argue in favor of other specification formalisms, such as, Colored Petri Nets or Message Sequence Chart Diagrams. One of the reasons why we have chosen M2L-Str is that we have a very well functioning tool available for doing computations in this logic [44]. Also, since we do not aim for traditional model checking, but rather for synthesizing controllers to be run on fast machines in slow networks, we are in the fortunate position to choose whatever logic provides the most succinct and intuitive syntax. All in all, M2L-Str is very powerful and yet just simple enough to actually allow calculations.

A formula ϕ in M2L-Str over the alphabet Σ will as explained in an earlier chapter — when interpreted over a finite string w — either evaluate to true or to false and we shall write $w \models \phi$ or $w \not\models \phi$, respectively. The *language* associated with ϕ is

$$L(\phi) = \{ w \in \Sigma^* \mid w \models \phi \}.$$

We shall denote by $pre\phi$ the prefix closure of $L(\phi)$.

We will use M2L-Str formulae as safety requirements as follows. Given a safety requirement ϕ we want to restrict the execution of the service S to allow only runs $\sigma = \sigma_0\sigma_1\sigma_2 \cdots \in \Sigma^\omega$ for which

$$\{\, \sigma_0 \ldots \sigma_{n-1} \in \Sigma^* \mid n \geq 0 \,\} \subseteq \mathit{pre}\phi.$$

That is, we only allow a run σ if all its finite prefixes, $\epsilon, \sigma_0, \sigma_0\sigma_1, \sigma_0\sigma_1\sigma_2, \ldots$, are in the prefix closure of the language associated with ϕ.

Example 7.3.1 A safety requirement formula might look as follows.

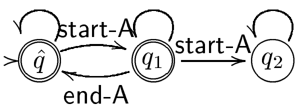
$$\begin{array}{l} \forall \textbf{time}\ \mathsf{t,t''}\!:\ (\mathsf{t<t''} \wedge \mathsf{start\text{-}A(t)} \wedge \mathsf{start\text{-}A(t'')}) \\ \quad \Rightarrow \exists \textbf{time}\ \mathsf{t'}\!:\ \mathsf{t<t'<t''} \wedge \mathsf{end\text{-}A(t')}. \end{array}$$

The formula will ensure that at most one session A thread will be allowed to execute at a time.

The above formula will occur automatically once per global variable X with start-A and end-A replaced by take-X and give-X respectively. This will ensure that only one session thread can update a given global variable at a time. One might argue that this could be implemented by simply using a semaphore for each global variable. True, but what we are dealing with here is just one specific safety requirement. Our technique can uniformly handle all safety requirements expressible in M2L-Str, that is, errors are less likely to occur. Furthermore we will argue later that the technique handles specific requirements like critical regions as efficiently as if implemented directly for instance by means of a semaphore.

It has been known since the late sixties that M2L-Str characterizes regularity [102]. The Mona system provides an algorithm for translating M2L-Str formulae into minimal deterministic finite state automata (MDFA). Furthermore, regularity is preserved under prefix closure. Thus we have a method for producing from the safety requirements an MDFA that will function as our safety controller.

Example 7.3.2 The minimal safety controller corresponding to the requirement of Example 7.3.1 will thus be



with the convention that non-labeled transitions are implicitly labeled by Σ minus the labels that occur on other outgoing transitions from that state.

As can be seen, it is not possible to start a new session A thread if an A thread is already running.

7.4 Labeled Services with Safety Requirements

Given a service S with induced automaton A_S and a safety controller A_c we can quite precisely define the restricted behavior that we expect from the composite system. First a definition.

We define the *product automaton* $A_1 \times A_2$ of two automata A_1 and A_2, $A_i = (Q_i, q_i^{\text{in}}, \Sigma, \rightarrow_i, F_i)$ to be

$$A_1 \times A_2 = (Q_1 \times Q_2, (q_1^{\text{in}}, q_2^{\text{in}}), \rightarrow, F_1 \times F_2),$$

where

$$(q_1, q_2) \xrightarrow{\sigma} (q_1', q_2') \text{ iff } q_i \xrightarrow{\sigma}_i q_i' \text{ for } i = 1, 2.$$

Thus the restricted behavior that we want our composite system to have is that of $A_S \times A_c$. For a product $A_1 \times A_2$ of two systems A_1 and A_2 we have $L(A) = L(A_1) \cap L(A_2)$. Thus the product of the service and the controller will allow — among the possible runs of the service — exactly those that also meet the safety requirements.

7.4.1 Implementing the Naive Runtime System

The service does not constitute a finite state automaton. Therefore we cannot produce the full runtime system (the combined system consisting of both the service A_S and the controller A_c) by simply computing the product automaton at compile time. What we will do instead is to implement the implicit synchronization of the product automaton directly as part of the runtime system. The runtime system will then consist of three parts:

- a safety controller A_c,
- for each label $\sigma \in \Sigma$ a run-time queue $\text{rtq}(\sigma)$, and
- the current session threads of the service.

All session threads and the controller run in parallel having the queues as shared resources.

- The code for the sessions will then be compiled such that each time a session thread wants to pass a label σ it pushes its session thread *id* onto $\text{rtq}(\sigma)$ and then waits for permission to continue. When permission later is granted by the controller it will pass the label σ and continue its execution.
- The safety controller will be looping while doing the following. Check if any of the queues corresponding to the enabled transitions are non-empty. In case it finds a non-empty queue, say $\text{rtq}(\sigma)$, it
  1. removes a session thread *id* from $\text{rtq}(\sigma)$,
  2. changes its state corresponding to making the enabled σ-transition, and

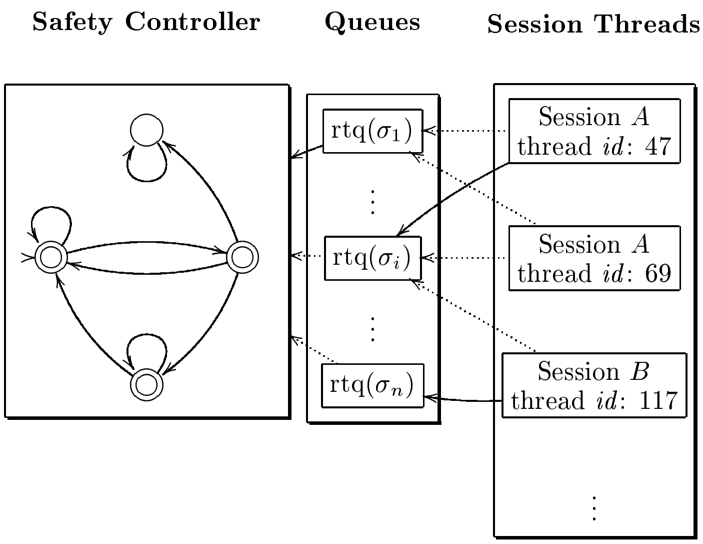


Figure 7.3: Sketch of the runtime system.

3. wakes up the session thread corresponding to the removed id.

This way the runtime system will behave as the product of the service and the controller. A diagram of a simple runtime system can be found in Fig. 7.3.

7.5 Improvements on the Runtime System

In the following section we will present two major improvements on the runtime system. Since the service part of the runtime system is very hard to reason about at compile time, our improvements will concentrate on the safety controller (which is just a finite automaton) and the shared queues. Both improvements are achieved using the notion of distributed automata as presented in Chapter 6.

- The first improvement concerns the avoidance of the state explosion problem.
- The second improvement increases parallelism of the system by inferring independence information.

7.5.1 The State Explosion Problem

We attack the state space problem using the modular synthesis techniques described in Chapter 6.

Example 7.5.1 Consider the example of having n global variables $X_1, \ldots, X_n$. In order to obey the requirement of mutual exclusion on assignment to globals this will automatically generate the following list of safety constraints.

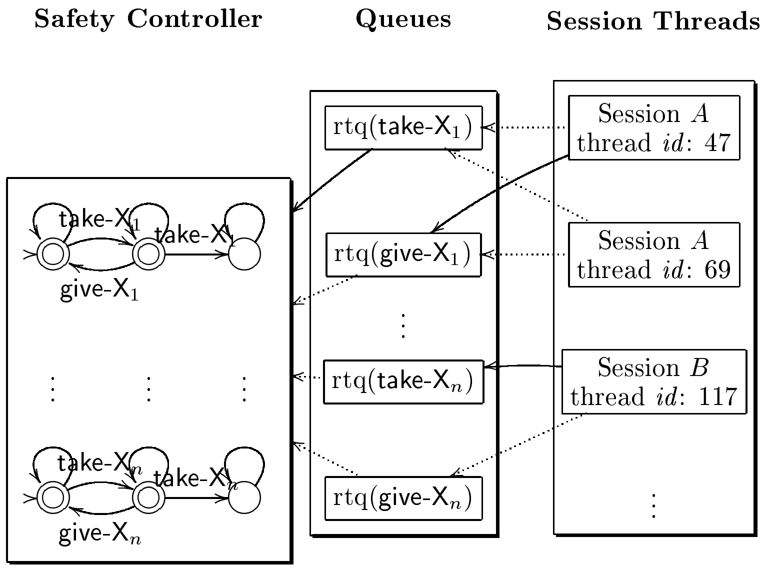
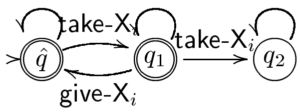


Figure 7.4: Runtime system with distributed automaton.

$$
\begin{array}{l}
\forall \textbf{time}\ t,t'':\ (t<t''\ \wedge\ \text{take-X}_1(t)\ \wedge\ \text{take-X}_1(t''))\\
\quad \Rightarrow \exists \textbf{time}\ t':\ t<t'<t''\ \wedge\ \text{give-X}_1(t');\\
\qquad\qquad \vdots\\
\forall \textbf{time}\ t,t'':\ (t<t''\ \wedge\ \text{take-X}_n(t)\ \wedge\ \text{take-X}_n(t''))\\
\quad \Rightarrow \exists \textbf{time}\ t':\ t<t'<t''\ \wedge\ \text{give-X}_n(t');
\end{array}
$$

The corresponding safety controller will have $2^n + 1$ states — it will look like the nth dimensional cube plus the "error state". If we use the distributed automaton approach and partition the safety requirements according to the semicolons then the controller will have only a linear ($3n$) number of states. We will have n copies of the three state automaton



and the runtime system will then be as illustrated in Fig. 7.4.

7.5.2 Inference of Independence Information

Again to obtain decentralized control, we make use of the technique for independence analysis described in Chapter 6.

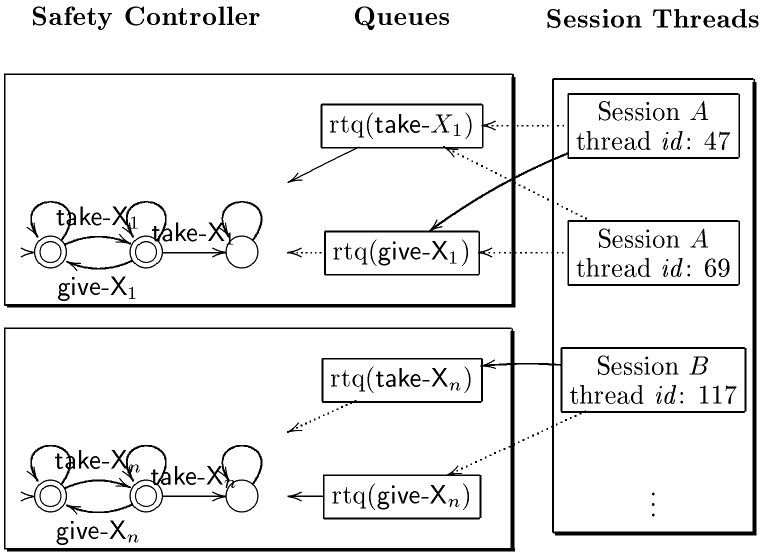


Figure 7.5: Runtime system with independent distributed automata.

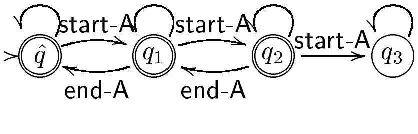
Example 7.5.2 A diagram of the runtime system with n global variables, $X_1, \ldots, X_n$, where we have both reduced the state space and exploited independence information can be found in Fig. 7.5.

As can be seen, we handle the case of critical regions just as efficiently as if we had implemented it using a semaphore, both with respect to space requirements and with respect to the degree of parallelism. Thus we do not lose anything by formulating simple safety requirements in this way. What we gain, however, is a uniform framework in which we can formulate all safety requirements.

Inference of independence information via search for locally dead labels and transitions will thus in general lead to a substantial increase of parallelism in the controller. This increase of parallelism will improve on the overall performance of the runtime system, that is, on the flow capacities of the provided service.

7.6 Beyond Regularity

In general one can have any number of session A threads. This can of course be constrained to any fixed maximal number by use of safety requirements. For instance, the controller



will guarantee that at any time there will be at most two session A threads.

But very often one has requirements like: I only want to enter this part of the code, such as, pass label foo, if there are no session C threads — without putting a bound on the maximal number of session C threads. This cannot be expressed in M2L-Str. The positions at which there are no session C threads are those where the number of start-C and end-C labels occurring before that position are the same but the standard example of a *non*-regular language is $\{ a^n b^n \mid n \geq 0 \}$ thus the property is not regular and therefore cannot be expressed in M2L-Str.

In order to accommodate the need for these kind of requirements we will introduce the notion of a *counter*. We could declare the counter last-A as follows.

counter last-A : #start-A − #end-A;

This would have the effect that an extra label, last-A, would be added to the alphabet. The extra label would be passed implicitly every time the right-hand side of the counter declaration reaches zero, that is, in this case every time the last session A thread has passed its end-A label. Thus, we produce a controller that only allows runs that both pass the safety requirements and furthermore have the property that last-A occurs exactly at positions following occurrences of end-A that results in a prefix that has the same number of start-A and end-A occurrences.

So in principle we want to make an intersection of a regular and a context free language which in general of course is non-regular. We still want to implement the controller as a finite state (distributed) automaton but this is no longer possible. However, if we equip the automaton with integer variables — one for each counter — then we will have enough machinery to recognize the intersection of the two languages. More specifically, consider the three step loop in the description of the controller in the naive runtime system:

> The safety controller will be looping while doing the following. Check if any of the queues corresponding to the enabled transitions are non-empty. In case it finds a non-empty queue, say rtq(σ), it
>
> 1. removes a session thread *id* from rtq(σ),
> 2. changes its state corresponding to making the enabled σ-transition, and
> 3. wakes up the session thread corresponding to the removed *id*.

Apart from the fact that there are now several of these controllers each having a subset of labels to take care of, we must add the following conditional as a fourth step.

> If σ occurs on the right-hand side of a counter declaration, cnt_i, then increment or decrement the variable associated with that counter. If it reaches zero then change the state corresponding to taking the cnt_i-transition.

By using the counter last-A we thus

- add the label last-A and
- in the controller we intersect with the language where last-A occurs after the termination of the last session A thread.

Therefore, one can now write a safety requirement ensuring that there are no active A sessions. For instance, the predicate

$$\begin{aligned}\text{zero-A}(t'') \equiv \exists\textbf{time}\ t:\ & t \leq t'' \wedge (t=0 \vee \text{last-A}(t)) \wedge \\ & \forall\textbf{time}\ t':\ t \leq t' \leq t'' \Rightarrow \neg\text{start-A}(t')\end{aligned}$$

will only evaluate to true at positions where there are no active A sessions.

Example 7.6.1 We can now formulate the (non-regular) requirement from the beginning of this section, "I only want to pass label foo if there are no active C sessions", as a safety requirement:

$$\forall\textbf{time}\ t:\ \text{foo}(t) \Rightarrow \text{zero-C}(t).$$

7.7 <bigwig> service logic

The <bigwig> service logic is a high-level notation built on top of M2L-Str suitable for writing safety requirements for <bigwig> service code implementing an interactive web service. It inherits from M2L-Str the usual universal and existential quantifications over both first order variables (ranging over instances of discrete time) and monadic second order variables (ranging over sets of instances of time). Also, it has standard boolean connectives like negation, conjunction, disjunction, implication, and so on, as well as operations on first order terms, such as, given an instance of time t one can point out its successor (t+1), operations on second order terms, such as, taking the union of two sets, plus of course the basic formula that tests membership of a position in a position set. Furthermore <bigwig> service logic provides basic formulae to test whether a label LL is passed at time t: LL(t).

Example 7.7.1 Consider some <bigwig> code with a critical region that needs exclusive access to, say, a global resource. The way one makes the region critical is by first adding labels around it.

```
...code ...
label begin-crt-region;
...critical region ...
label end-crt-region;
...more code ...
```

Then, in order to make the code between these labels act as a critical region the following safety constraint is added to the set of requirements.

$$\begin{aligned}\forall\textbf{time}\ \ & \text{beg}_1,\ \text{beg}_2: \\ & ((\text{beg}_1 < \text{beg}_2) \wedge \text{begin-crt-region}(\text{beg}_1) \wedge \text{begin-crt-region}(\text{beg}_2)) \\ & \Rightarrow \exists\textbf{time}\ \text{end}_1:\ (\text{beg}_1 < \text{end}_1 < \text{beg}_2) \wedge \text{end-crt-region}(\text{end}_1).\end{aligned}$$

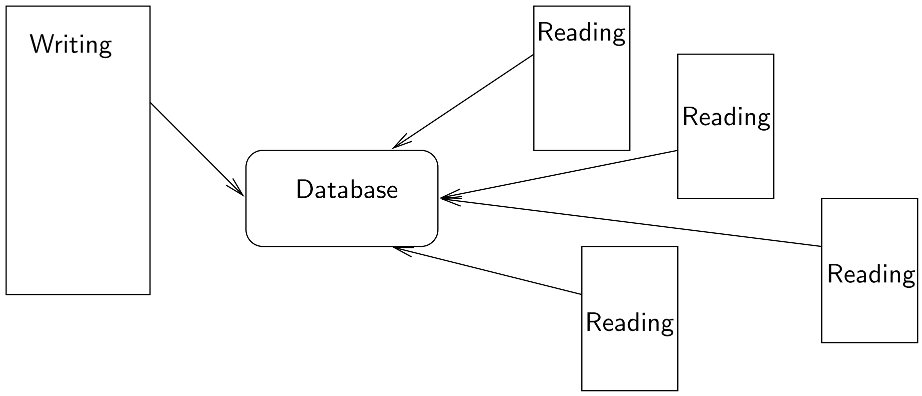


Figure 7.6: A Writing thread and several Reading threads accessing a Database.

Example 7.7.2 Another thing that <bigwig> service logic is suitable for is formulating requirements regarding priority. We are given some service S with sessions Reading and Writing (See Fig. 7.6). Reading threads read data from a Database, display the data in a proper way, read some more data, display it, and so forth. Writing threads can only be started by the service administrator. Furthermore, at most one Writing session thread is allowed at a time. This last condition can be satisfied by the constraint

$$\forall \textbf{time}\ t\colon \text{start-Writing}(t) \Rightarrow (t{=}0 \vee \text{zero-Writing}(t\text{-}1)),$$

where zero-Writing and later zero-Reading are defined as the zero-A predicate from the previous section.

Of course Writing must not be started unless there are no active Reading threads and vice versa. This can also be formulated in a straightforward way using the constraint

$$\forall \textbf{time}\ t\colon \text{zero-Writing}(t) \vee \text{zero-Reading}(t).$$

But what if it is of great importance that the administrator gets access to the database as soon as possible? For instance if the database contains prices of products that a company sells and corrections have to be made to these prices. Thus, we want to give Writing priority over Reading. This can be managed in the following way (excluding the last formula above). We assume that the critical regions of Writing, that is, those that access the database, are surrounded by labels start-Crt and end-Crt.

- Then, in order to make sure that as soon as the Writing thread has started, no more Reading threads will start, we add the constraint

$$\forall \textbf{time}\ t\colon \text{start-Reading}(t) \Rightarrow \text{zero-Writing}(t).$$

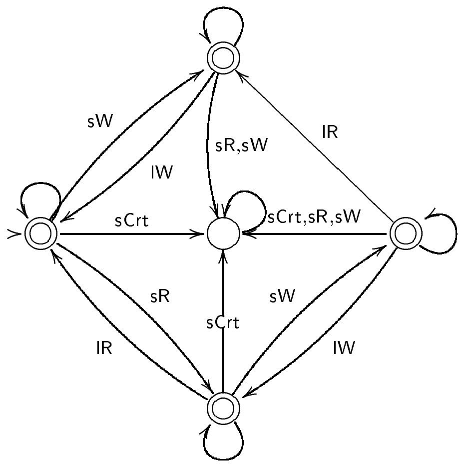


Figure 7.7: Automaton corresponding to the Writing/Reading setup.

- Of course, we should still make sure that we have mutual exclusion. This is done by adding the constraint

  $$\forall \textbf{time}\ \mathsf{t}\text{: } \mathsf{start\text{-}Crt(t)} \Rightarrow \mathsf{zero\text{-}Reading(t)}.$$

  Thus, we make sure that we do not enter any critical region in Writing unless the last Reading thread has ended its execution.

Using the above approach we satisfy the crucial property of mutual exclusion. Furthermore, we impose restrictions that will make sure the administrator gets priority over ordinary users. The corresponding automaton, however, is rather complex, see Fig. 7.7. Thus, producing it by hand is cumbersome and error prone.

Furthermore, our approach is modular in the sense that requirements can be added and deleted at will without changing the existing requirements. In the automata world small changes can result in extensive changes of the automaton. Solutions using semaphores lack in a similar fashion the property of being modular.

7.8 Related Work

Mawl has already been mentioned as an example of a domain-specific language for describing sequential transaction-oriented Web applications. The techniques available for doing synchronization when dealing with concurrency in Mawl is limited to working with critical regions, though. Our work on service logic extends this to working with arbitrary safety constraints.

In general, there are increasingly many systems for doing web-programming, such as, [67, 48, 37], but so far none of seem of them seem to support proper handling of safety requirements.

The area of control theory is of course huge. We are only dealing with control of discrete systems, though. Ramadge and Wonham give a good survey on "The Control of Discrete Event Systems" [103]. Many of the notions presented here are similar to those of Ramadge and Wonham.

Distributed automata are simply a special case of the product automata of [127]. They again are a special case of the non-cellular asynchronous automata of [134].

7.9 Discussion

When dealing with inference of independence information, the important part is that of choosing the right partition of the safety requirement. We plan to do static analyses on the safety requirements and combine the achieved information with appropriate heuristics to obtain hopefully good results. Intuitively, the formulae of the safety requirements implicitly say something about which requirements are closely related and which are not.

The idea of having a central controller is proving useful for other aspects of web services. We plan to include support for various kinds of event handling and database locking. Also, the controller may drive an automatically generated service monitor, allow maintenance and performance statistics.

Chapter 8

Controllers for Embedded Systems

Contents

The work presented in this chapter is on the use of partial control synthesis for embedded systems, more specifically for the LEGO Mindstorms robots. It has been submitted for publication and is joint work with Thomas Hune.

We propose a method based on formal logic and automata for partial control synthesis in the restricted environment of embedded LEGO Mindstorms systems. The method merges an existing control programs with logic-based specifications. This yields a new control program restricting the behaviour of the old control program such that the specifications are satisfied. The method is presented through a concrete example.

8.1 Introduction

Writing control programs can often be an error prone task, especially if a number of special cases must be taken into account. Time is often spent on handling special case or failure situations rather than solving the actual problem at hand. A number of different methods and tools have been developed to ease this task. One well known method is based on a control automaton running in parallel with the actual program [103, 116]. The automaton controls the input and output events of the program. This is a way of restricting the sequences of I/O actions occurring.

Specifying the right control automaton will often be as error prone as writing the full control program. The use of logical formulae for specifying the properties is well understood [76]. There are various logics which could be used as specification language. We propose using monadic second order logic over strings (M2L-Str) [21] because this logic has a number of nice properties such as being expressive and succinct. Given an M2L-Str formula, the tool Mona [60] minimal deterministic automaton (MDFA) accepting the language specified by the formula.

The described method for *partial* synthesis differs from traditional synthesis as described in e.g. [76], in that the method is partly based on an existing control program. The aim of the partial synthesis is to restrict the behaviour of the existing control program. The allowed executions are described using formulas over a set of I/O events. These events define the interface between the existing control program and the specification.

We will look at a control program for a moving crane. For this example we have implemented our method in the LEGO RCX system [72]. Using the LEGO RCX system is interesting for at least two reasons. First of all the environment of the RCX system and especially the programming language is quite restricted, so it is not obvious that it is feasible to implement our method at all. Secondly, using the LEGO RCX system one can build actual physical systems for testing the control programs. We have built the crane and used it with different control programs.

The language running on the LEGO RCX brick (RCX language) is an assembly-like language with a few high level features, like a notion of task or process. Programs are written on a PC and downloaded to the RCX brick where they are interpreted.

The following section describes the LEGO system. In Section 8.3 the crane example is presented. The logic-based specification language is presented in Section 8.4, and the merge of automata with the RCX code in Section 8.5. Finally, Section 8.6 rounds off with conclusions, related work, and directions for future work.

8.2 The LEGO system

The experiments we have made are based on the LEGO RCX system and the associated RCX language. The language is an assembly like language with some high level concepts like tasks. The language is restricted in a number of ways, e.g. it is able to address only 32 integer variables and allows only ten tasks in a program. Furthermore, one cannot use symbolic names in programs.

A small operating system is running on the RCX with processes for handling I/O and one process running an interpreter for the RCX language. The RCX brick has three output ports (for motors and lights) and three input ports. Four kinds of sensors for the input ports are supplied by LEGO: touch, temperature, rotation, and light.

Though the limitations described above are specific to RCX language, other low-level machine dependent have similar kinds of restrictions. Developing a framework for working with concurrency constraints in this setting is thus appropriate.

8.2.1 The RCX language

A program consists of a collection of at most ten tasks. There is no special way to communicate between tasks but all variables are shared, providing a way of communicating. A task can start another task with the command `StartTask(i)` and stop it with the command `StopTask(i)`. Starting a task means restarting it from the beginning. That is, there is no way of resuming the execution of a task nor spawning an extra "instance" of a task.

The language has some commands for controlling the actuators, the main ones being `On(li)` and `Off(li)` where `li` is a list of ports. There are also a number of instructions for manipulating variables. All of these take three integer arguments. The first argument specifies the target variable, the second the type of the source, and the third the source. The most important types of sources are: variables (the third argument is then the number of the variable), constants (the third argument is then the value), and sensor readings (the third argument is then the number of the sensor). These types of sources can be used in `SetVar(i,j,k)` for assigning a value to a variable. In the instructions for calculating like `SumVar(i,j,k)`, `SubVar(i,j,k)`, and `MulVar(i,j,k)` sensor readings are not allowed.

There are two ways of defining loops: the `Loop(j,k)` and the `While(j,k,l,m,n)` instruction. The arguments of the `Loop` indicates how many times the body should be iterated in the same way as the source of the instructions for calculating. The `While` loop is iterated as long as the condition specified by the arguments is satisfied. The first two and last two arguments are the sources as in the assignment and `l` specifies a relation from the set $\{=, <, >, \neq\}$.

There is also a conditional, `If(j,k,l,m,n)`, with the condition specified as in the `While` construct. An optional `Else` branch can be specified as well.

One can block a task for a given time using the `Wait(j,k)` statement. When the specified time has passed, execution of the task is resumed.

The statements presented above constitute the part of the RCX language which we have used for implementing control automata.

8.2.2 Scheduling

During execution a task can either be enabled or blocked. A task can be blocked by a `StopTask(i)` instruction, by a `Wait(j,k)` instruction, or by finishing its execution. Initially only task zero is enabled. The enabled tasks are executed in a round robin fashion, where each task executes one instruction and then leaves control for the next task.

Knowing how concurrent processes are executed is important when reasoning about the overall behavior of a program. Since the language does not offer a way of resuming execution of a task this has to be programmed explicitly.

8.3 Example

As an example we will look at a LEGO crane which we will program in the RCX language. We have built the crane and tested it with different control programs. The crane is run by three motors connected to the RCX. One motor is driving the wheels, one is turning the turret around, and one is moving the hook up and down. The input for the three motors are three touch sensors, which is all the RCX brick has room for. This means we can only turn motors on and off. Therefore the crane alternates between moving forward and backward each time the motor is turned on. The direction of turret and the hook is controlled in a similar way.

A very basic control program for the crane could consist of four tasks. One task for setting up the sensors and motors, and starting the other tasks. For each of the three inputs, one task for monitoring input and controlling the motor correspondingly. Task 1 for monitoring sensor 0 would then be

```
BeginOfTask 1
Loop 2, 0               'An infinite loop
  SetFwd "0"            'Set direction of motor 0 to forward
  SetVar 1, 9, 0        'Read sensor 0
  While 0, 1, 3, 2, 1   'While sensor 0 != 1
    SetVar 1, 9, 0
  EndWhile
  On "0"                'Start motor 0
```

```
  Wait 2, 100          'Wait
  SetVar 1, 9, 0       'Read sensor 0
  While 0, 1, 3, 2, 1  'While sensor 0 != 1
    SetVar 1, 9, 0
  EndWhile
  Off "0"              'Stop motor 0
  Wait 2, 100          'Wait
  SetRwd "0"           'Set direction of motor to reverse
  SetVar 1, 9, 0       'Read sensor 0
  While 0, 1, 3, 2, 1  'While sensor 0 != 1
    SetVar 1, 9, 0
  EndWhile
  On "0"               'Start motor 0
  Wait 2, 100          'Wait
  SetVar 1, 9, 0       'Read sensor 0
  While 0, 1, 3, 2, 1  'While sensor 0 != 1
    SetVar 1, 9, 0
  EndWhile
  Off "0"              'Stop the motor
  Wait 2, 100          'Wait
EndLoop
EndOfTask
```

The *Wait* statements ensures that one touch of the sensor is not read as two touches. We could of course have tested for this but for our example this will do. The two other tasks for controlling the remaining two motors look similar, only the numbers of variables, sensors and motors are different.

For the purpose of illustrating the presented method we choose to place the following constraints on the behaviour of the crane. First of all we only want one thing happening at a time, so we will not allow for two motors to be turned on at the same time. Pressing the touch sensor could now be seen as a request which the control program may grant (and start the motor) when all the motors are stopped. Moreover, we want that moving the hook has higher priority than the wheels and the turret. Requests from the other two are handled in order of arrival.

The first constraint on the behaviour is basically mutual exclusion which is nontrivial to implement in the RCX language (this is an integrated part of the implementation of the automata-based approach described in Section 8.5). On top of this we have a mixed priority and queue scheme. Implementing and debugging this by hand would indeed be tedious.

8.4 Logic-based specifications

Basically we could keep the initial simple control program if we could prune out some unwanted executions. If we just had the constraint that only one motor was allowed to be turned on at a time, we could base this pruning strictly on the I/O actions. To meet the constraints regarding priorities we introduce a request from the tasks, when they want to start a motor. We can change our initial control program slightly such that a request is issued when a sensor is touched, and the motor is turned on when an answer to the request arrives. This is still a simple program to write. Execution of the program gives rise to a sequence of events. In our case we will consider input (requests), and two kinds of output (start and stop motor) as events. We then implement the automaton accepting the language of events satisfying the introduced constraints. With this approach we can thus keep the control program simple.

Traditionally, control languages are described by automata. Certainly, automata are in some cases a good formalism to work with. However, as properties grow in complexity so will the automata. In this section we will describe the use of a logic formalism from which we can automatically generate automata.

8.4.1 Terminology

In order for a set of sequences, L, to be a legal *control language*, it must be prefix closed, that is, if the sequence w is in L then all prefixes of w must be in L as well. When using deterministic finite state automata to specify sets of sequences, checking for prefix closedness is easy. One just has to make sure that all transitions from non-accepting states go to non-accepting states.

Since our deterministic control automata will represent only prefix closed languages, we can depict them without drawing the non-accepting state (if minimized, the automaton will have only one non-accepting state). Transitions, $q \xrightarrow{\sigma} q'$, where q' is non-accepting will thus be left out as well.

Example 8.4.1 The automaton for our example with mutual exclusion and priority given to engine one can be found in Fig. 8.1. As can be seen, the automaton is inherently complex indicating that one would rather have it combined automatically with existing control code rather than merging these requirements into the code by hand. Furthermore, as considered later, just the mere task of coming up with such an automaton is cumbersome and non-trivial.

8.4.2 Specification logic

It would be nice if instead of converting the informal requirement in Section 8.3 into a complex automaton, one could write it up formally in a specification formalism closer to natural language. That is, we would like to be able to write something like the following.

- Only one motor can be turned on at a time;
- If the wheels get turned on, then the hook must not be requesting and the wheels must have been the first to make a request; and
- If the turret gets turned on, then the hook must not be requesting and the wheels must have been the first to make a request.

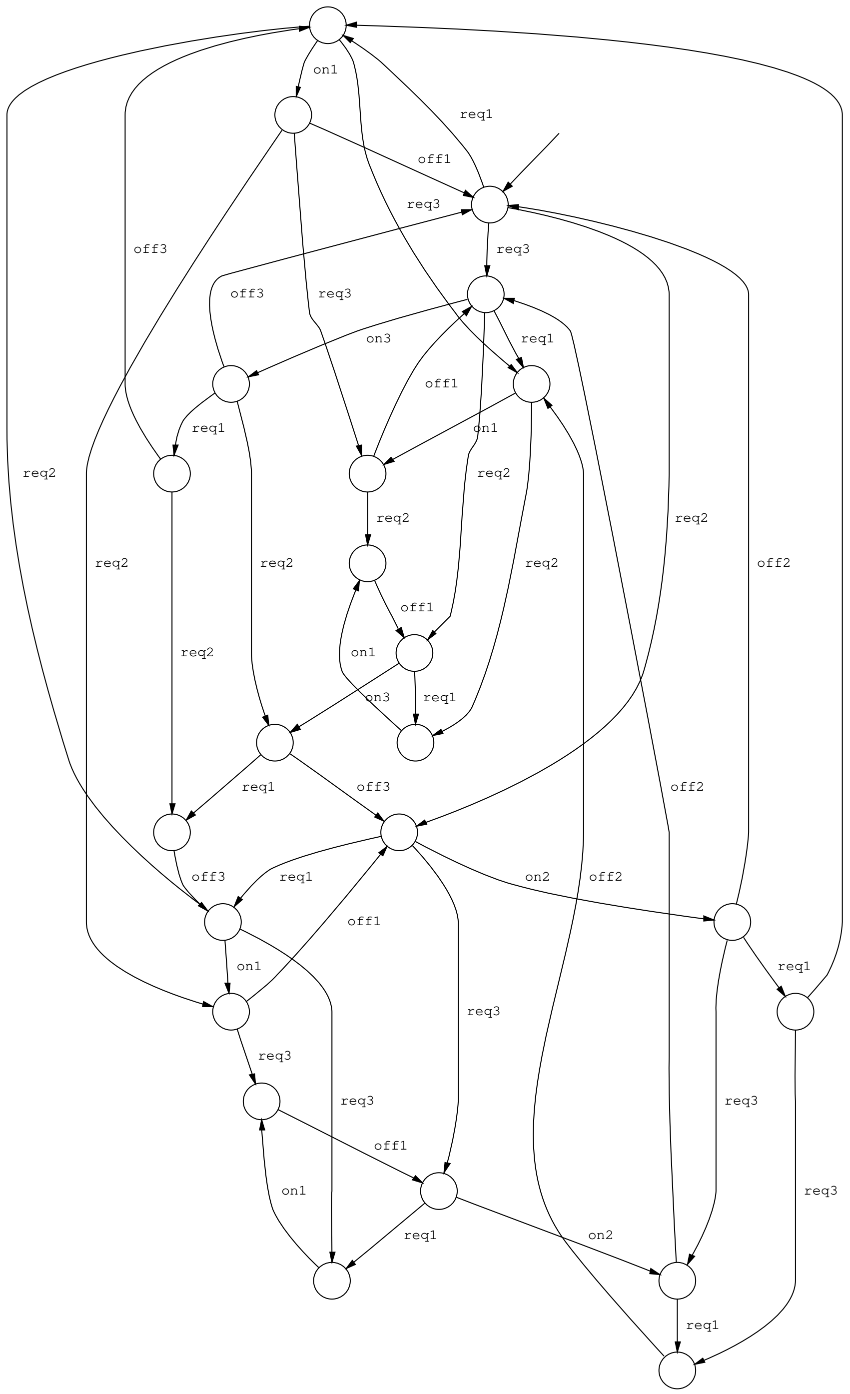


Figure 8.1: The inherently non-comprehensible automaton giving priority to engine one.

We therefore turn to a formalism that is as expressive as finite state automata and yet still allows for separation of the declaratively specified requirements (previously our control automaton) and the operational part of the control program (the existing RCX program).

Experience has shown that logic is a suitable specification formalism for control languages. In fact, we have previously quite successfully been using the Mona tool for similar tasks [116] and therefore propose using this to define our automata. The use of logic has several advantages over traditional automata-based approaches:

- logic is a lot closer to natural language than automata and thus much easier to work with;
- our formalism, based on monadic second-order logic on strings (M2L-Str), is *succinct* and *expressive* allowing complicated requirements to be formulated easily and compactly; and
- specifications are easy to *maintain*, that is it is easier to add, remove, and modify parts of a logic-based specification.

Moreover, using logic does not add any additional runtime overhead, since requirements written in the logic-based specification language can be compiled *automatically* into control automata by use of the Mona tool.

M2L-Str is the input language of Mona and the basis of our specification formalism. One might argue in favor of other specification formalisms such as high-level Petri Nets [55] or Message Sequence Charts [79]. Being a logic formalism, however, M2L-Str has the advantage that specifications can be developed iteratively, that is, one can easily add, delete, and modify parts of a specification. It also has a readable textual format. Moreover, the formalism in use should be simple enough that a runtime checker, such as an automaton, can actually be calculated and downloaded to the RCX brick. Thus, M2L-Str is powerful and yet just simple enough to actually subject it to calculations using Mona.

Experience in using M2L-Str as a language for defining control requirements has shown that only the first-order fraction of the logic is used in practice [14, 116]. We shall thus consider only first order quantifications, though second-order quantifications could be added at no extra cost.

The abstract syntax of the logic is given by the following grammar:

$$\begin{aligned} \phi &::= \exists p : \phi' \mid \neg\phi' \mid \phi' \wedge \phi'' \mid \phi' \vee \phi'' \mid \phi' \Rightarrow \phi'' \mid \sigma(t) \mid \ldots \\ t &::= t + 1 \mid \ldots \end{aligned}$$

That is, M2L-Str has the constructs: universal and existential quantifications over first order variables (ranging over positions in the sequence of events), standard boolean connectives such as negation, conjunction, disjunction, and implication, and basic formulae, $\sigma(t)$, to test whether an event σ can be found at position t. It also has operations on terms, such as, given a position t one can point out its successor $(t + 1)$.

A formula ϕ in M2L-Str over the event set Σ will – when interpreted over a finite sequence of events w – either evaluate to true or to false and we shall write this as $w \models \phi$ or $w \not\models \phi$, respectively. The *language* associated with ϕ is $L(\phi) = \{ w \in \Sigma^* \mid w \models \phi \}$.

The language associated with an M2L-Str formula is guaranteed to be regular. In fact, it has been known since the sixties that M2L-Str characterizes regularity [21, 32]. The Mona tool implements the constructive proof of the fact that for each M2L-Str formula there is a minimal deterministic finite state automaton accepting the language of the formula. That is, Mona translates M2L-Str formulae into minimal deterministic finite state automata (MDFA).

Example 8.4.2 In fact, with this logic-based specification language in place, we can write a description closely related to the informal requirements that gives rise to the exact same automaton as the one presented in Fig. 8.1.

$$\begin{aligned}
&\forall t : (\texttt{on1}(t) \vee \texttt{on2}(t) \vee \texttt{on3}(t)) \Rightarrow \mathit{alloff}(t) \wedge \\
&\forall t : \texttt{on2}(t) \Rightarrow (\neg \mathit{requesting1}(t) \wedge \mathit{req2first}(t)) \wedge \\
&\forall t : \texttt{on3}(t) \Rightarrow (\neg \mathit{requesting1}(t) \wedge \mathit{req3first}(t)),
\end{aligned}$$

where the basic formulae `on1`(t), `on2`(t), and `on3`(t) are true if the event at position t is `on1`, `on2`, or `on3`, respectively. The formula *alloff*(t) has been defined to be true if and only if at time t none of the three engines are on. The formula *requesting1*(t) is true if and only if at some time t' before t there has been a request to start engine for the hook and in between t' and t engine one has not yet been started. Finally, *req2first*(t) is true at time t if among the requests that have been made and yet not been accounted for, the request for engine two, that is, the engine for the wheels was the first. Similarly for the formula *req3first*(t).

That is, the requirement stated above ensures that

- in order for any of the three engines to be turned on, all three engines must be off first;
- in order for engine two to get turned on, engine one must not be requesting and engine two must have been the first to make a request; and
- similarly for engine three. (That is, in order for engine three to get turned on, engine one must not be requesting and engine three must have been the first to make a request.)

For instance, say that we would like to change the control language of our example in such a way that all three tasks have equal priority. This would imply changing the overall structure of the control automaton — a tedious and error prone task. As the following example will show, modifying the logical formula is indeed quite comprehensible in the case of the LEGO crane requirements.

Example 8.4.3 Say that we would like to change the requirements in such a way that all engines were given equal priority, that is, they would be turned on

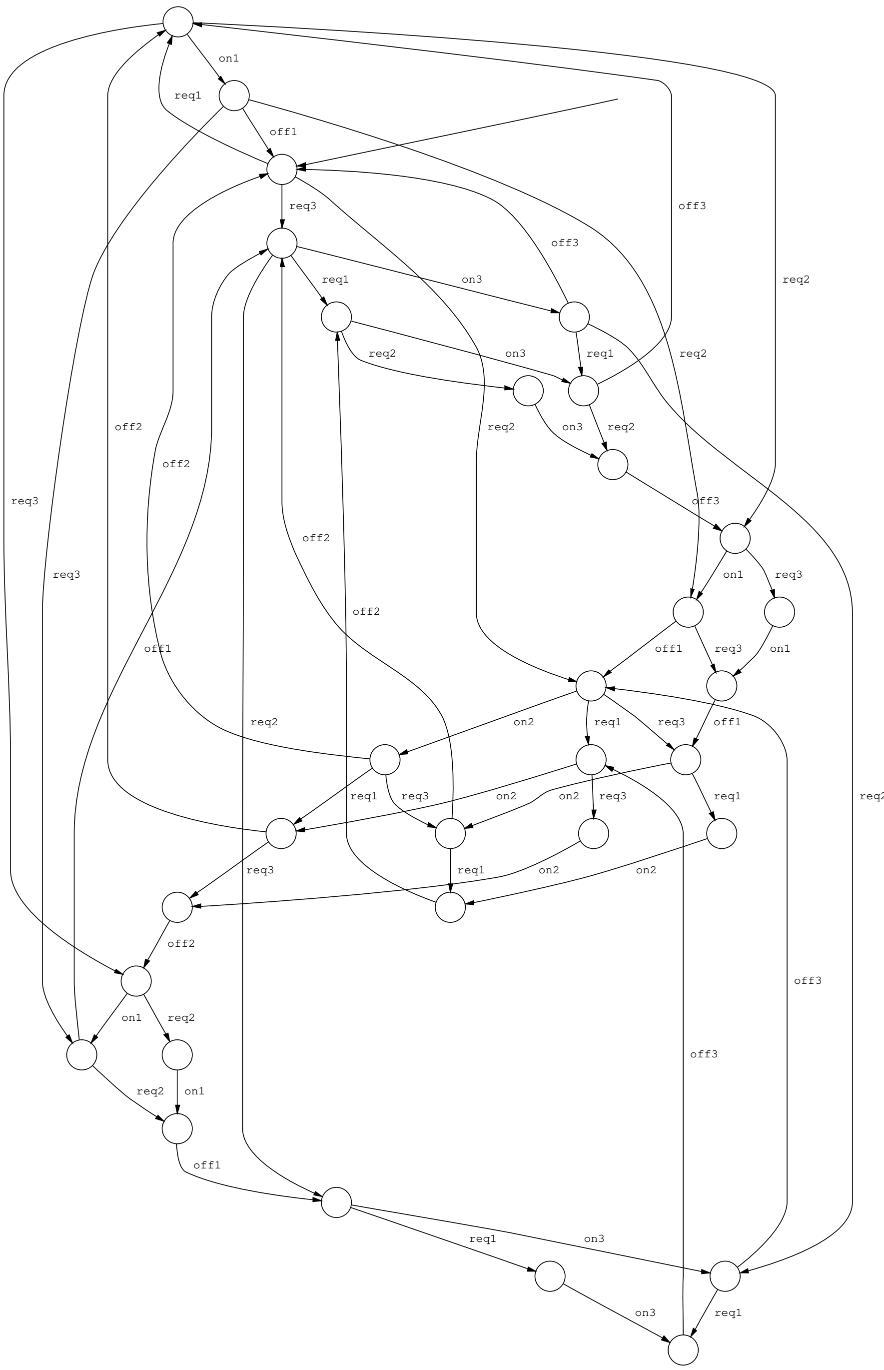


Figure 8.2: Control automaton giving equal priority to engine one, two, and three. Note the lack of resemblance with the automaton in Fig. 8.1.

in a first come first served manner. Then, the new (Fig. 8.2) and old (Fig. 8.1) control automata would differ quite substantially. Thus producing the new automaton from the existing one by hand would be a rather cumbersome task. In fact, throwing the carefully constructed automaton giving priority to engine one away and producing the new automaton from scratch would probably be easier.

Using our logic-based specification, however, all we have to do is change to last two lines of our specification slightly resulting in the following fifo requirement.

$$\begin{aligned}
&\forall t : (\mathtt{on1}(t) \vee \mathtt{on2}(t) \vee \mathtt{on3}(t)) \Rightarrow \mathit{alloff}(t)\wedge \\
&\forall t : \mathtt{on1}(t) \Rightarrow \mathit{req1first}(t)\wedge \\
&\forall t : \mathtt{on2}(t) \Rightarrow \mathit{req2first}(t)\wedge \\
&\forall t : \mathtt{on3}(t) \Rightarrow \mathit{req3first}(t).
\end{aligned}$$

Note that the sub-formulae, such as, *alloff()* and *req2first()* are not changed at all. Thus, once the specification is in place it is relatively easy to change it in various ways.

8.5 Merging automata and RCX code

Given a control automaton and the basic control program, one can synthesize the complete control program. In this section we describe how to translate an automaton into RCX code running in conjunction the existing RCX control program.

8.5.1 Wrapping the RCX code

The execution of RCX code is restricted to sequences allowed by a control language as follows. Having the control language defined by means of an automaton, all we have to do is merge the existing RCX code with the RCX code generated from the automaton. Merging RCX code with an automaton can in some sense be considered a program transformation. Each statement involving a request or writing to an output port is replaced by a block of code that tests whether the operation is legal according to the control automaton. For our example, transforming the code for turning motor 0 on, will lead to the following piece of code.

```
While 0, 4, 3, 2, 1 'While automaton has not accepted the command
  SetVar 31, 2, 1   'Arg := on0, the argument for the automaton
  GoSub 0           'Run the automaton
  SetVar 4, 0, 22   'Local success:= global success
EndWhile
On "0"              'Execution of the actual command
```

We have chosen to implement the automaton as a subroutine. Since arguments for subroutines are not allowed in the language, passing an argument to the automaton has to be done via a global variable. Similarly, since a subroutine cannot return a value, return values are also placed in global variables for the process to read. The while loop delays the action until the automaton accepts execution of it.

8.5.2 Implementing mutual exclusion

However, the idea described above is not sufficient since cannot allow more tasks to use the automaton simultaneously. Therefore we need to change the loop. In the RCX language the problem is obvious since we are using global variables for passing arguments and results. In general, this is also a problem since the outcome of the automaton depends on its state when execution begins. If a process accesses the automaton while it is used by another process, the state variable might be corrupted. Therefore we must have exclusive access to the automaton.

In our implementation we have used Dijkstra's algorithm to implement mutual exclusion between several processes [31]. But any correct mutual exclusion algorithm would of course do. The algorithm uses some global variables but this is no problem in the RCX language since all variables are shared. There are no gotos in the RCX language. Therefore, we have used an extra while loop and a success variable for each task. Except from these details, the algorithm is followed directly. Entering the critical region for process one is done as follows.

```
SetVar 27, 2, 0            'success(1):=0
While 0, 27, 2, 2, 0       'while success(1) == 0
  SetVar 30, 2, 1          '  flag(1):=1
  While 0, 24, 3, 2, 1     '  while turn <> 1 do
    If 0, 24, 2, 2, 2      '    if turn == 2 then
      If 0, 29, 2, 2, 0    '      if flag(2) == 0 then
        SetVar 24, 2, 1    '        turn:=1
      EndIf                '      endif
    Else                   '    else      #turn is 3
      If 0, 28, 2, 2, 0    '      if flag(3) == 0 then
        SetVar 24, 2, 1    '        turn:=1
      EndIf                '      endif
    EndIf                  '    endif
  EndWhile                 '  endwhile
  SetVar 30, 2, 2          '  flag(1):=2
  SetVar 27, 2, 1          '  success(1):=1
  If 0, 29, 2, 2, 2        '  if flag(2) == 2 then
    SetVar 27, 2, 0        '    success(1):=0
  EndIf                    '  endif
  If 0, 28, 2, 2, 2        '  if flag(3) == 2 then
    SetVar 27, 2, 0        '    success(1):=0
```

```
  EndIf                    '  endif
EndWhile                   'endwhile
```

The last two if statements is an ‘unfolding’ of the for loop

```
for j<>1 do
  if flag(j)==2 then success(1):=0
```

Leaving the critical region is simple, just setting ones own flag to zero.

```
  SetVar 30, 2, 0          'flag(1):=0
```

Thus, for our example the transformed code for `On "0"` will then be

```
While 0, 4, 3, 2, 1 'While automaton has not accepted the command
  'Enter critical region
  ...
  SetVar 31, 2, 1   'Arg := on1, the argument for the automaton
  GoSub 0           'Execute the automaton
  SetVar 4, 0, 22   'Local success:= global success
  'Exit critical region
  SetVar 30, 2, 0   'flag(1):=0
EndWhile
On "0"              'Execution of the actual command
```

8.5.3 Implementing automata

In general, an automaton, $A = (Q, \mathit{init}(,)\Sigma, \rightarrow, F)$, gets implemented as follows. During the run of an automaton a variable, `state`, holds the number of the current state. The transition relation is implemented as nested switch statements of depth two, where the variable `label` holds the next event.

```
switch (state) {
case q:
  switch (label) {
  case σ:
    state = q'; /* if q -σ-> q' */
    break;
  case σ':  ...
  }
case q':  ...
}
```

That is, we use the standard implementation strategy for implementing deterministic finite state automata. In the RCX language, however, there is no switch statement so we have used nested if statements instead. For the following example we have written the code for the automaton by hand. Generating the code automatically from the transition table of the automaton would be straightforward.

```
BeginOfSub 0
SetVar 22, 2, 1         'Global success :=1
If 0, 23, 2, 2, 0       'Initial state
  If 0, 31, 2, 2, 1     'Label==on1
    SetVar 23, 2, 1     'New state := state 1
  Else
    If 0, 31, 2, 2, 2   'Label == on2
      SetVar 23, 2, 2   'New state :=state 2
    Else
      If 0, 31, 2, 2, 3 'Label==on3
        SetVar 23, 2, 3 'New state := state 3
      Else
        SetVar 22, 2, 0 'Global success :=0
      EndIf
    EndIf
  EndIf
Else
  If 0, 23, 2, 2, 1     'State 1
  ...
  EndIf
EndIf
EndOfSub
```

From the example presented above it is clear that the method easily generalizes to any problem that can be formulated in terms of finite state automata. That is, the fact that we can nicely combine the run of an automaton with the execution of standard RCX code with wrapped input/output statements is not just an artifact of the crane example.

8.5.4 Variations of the method

The action taken by the wrapped output statements when immediate permission is not granted from the control automaton can vary. That is, there are various ways of handling this temporary lack of controller acknowledgment:

- in the above example the task would simply *busy wait* asking the controller over and over whether its label had been enabled; but
- one could also simply *cancel* or ignore the output statement requesting permission and continue execution. That is, the busy waiting while loop would be replaced by an if statement.

The former would often be the preferred approach in cases where the internal state of the code for the embedded system is important, such as, in a train gate controller. The latter would be a good choice in cases where the code is written in a reactive style, constantly changing output action based on newly read input.

There is also a question of where to put the automaton.

- In the example above the automaton was put *alongside the wrapped RCX code*. Placing the code implementing the automaton at this level seems a natural choice when dealing with properties about the behavior of a specific RCX program solving a particular problem.

- However, in cases where one would like to protect, say hardware, one should rather place the automaton *at the level of the operating system* in the driver for the particular piece of hardware.

In connection with the latter one might add as colloquial remark that, when not using the interpreted RCX language described in this chapter but rather writing in the native machine language of the processor, it is possible on the RCX to destroy the infra-red communication mechanism.

8.6 Conclusion

We have suggested using control automata for synthesizing parts of control programs for embedded systems. Using this method one can add to a basic control program a control automaton which will ensure certain properties are satisfied. Since defining automata by hand can be as difficult as writing control programs, we suggest using logical formulae for specifying the properties and translate these to automata using the Mona tool.

The approach has been implemented in the restricted setting of the LEGO RCX system. This opens for the possibility of testing the implementations on real systems.

8.6.1 Related work

Control automata The use of finite state automata for the control of robots is not novel. Ramadge and Wonham [103] give a survey of classic results.

Layered control Our method can in some sense be considered a special case of the layered approach to robot control proposed by Brooks [17]. He describes a layered architecture for controlling mobile robots. Each module or layer is a fairly simple computational machine and higher-level layers can subsume the roles of lower levels. In our case the existing control program might be considered the high-level layer implementing, say, a neural network for adaptive behavior. The control automaton in turn guarantees that whatever behavior is implemented by the control program, the stated control requirements will be obeyed.

Control synthesis for Web services The method presented here has been used successfully in <bigwig> [116] as described in Chapter 7. Our method for control synthesis is used as an integral part of <bigwig> to define safety constraints. In fact, via use of a powerful macro mechanism [14] the method has been used to extend the Web programming language in <`bigwig`> with concepts and primitives for concurrency control, such as, semaphores and monitors.

8.6.2 Future work

Measuring overhead There is an overhead connected with gaining exclusive access to the automaton and running it. How much time is spent on gaining access to the automaton of course depends on the arrival of input events. We would like to calculate some specific times for this given some input sequences. A tool for translating RCX programs to timed models supported by Uppaal [69] exists [49]. Using Uppaal one can "measure" the time spent by a program from an input is read until the response arrives. This gives a way of calculating the overhead involved in using the approach.

Distributed robots The example presented in this chapter only has one robot and the control restrictions are consequently imposed on that particular robot only. One could easily imagine having several robots in a distributed environment. Writing control requirements for the whole system could again be done in the logic-based formalism. By use of modular synthesis and distributed control [103] via independence analysis [116] one can statically infer information about which constraints to put locally on the robots and which to put on the (sometimes necessary) central controller.

Part III

Full Abstraction for FPC

Chapter 9

A Relational Account of Call-by-Value Sequentiality

Contents

A preliminary version of this work appeared in *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science* [110]. A full version has been accepted for publication in *Information and Computation* [111]. It is joint work with Jon G. Riecke.

We construct a model for FPC, a purely functional, sequential, call-by-value language. The model is built from partial continuous functions, in the style of Plotkin, further constrained to be uniform with respect to a class of logical relations. We prove that the model is fully abstract.

9.1 Introduction

The problem of finding an abstract description of sequential functional computation has been one of the most enduring problems of semantics. The problem dates from a seminal paper of Plotkin [98], who pointed out that certain elements in Scott models are not definable. In Plotkin's example, the function

$$por(x, y) = \begin{cases} \mathit{true} & \text{if } x \text{ or } y = \mathit{true} \\ \mathit{false} & \text{if } x \text{ and } y = \mathit{false} \\ \perp & \text{otherwise} \end{cases}$$

where $\perp$ denotes divergence, cannot be programmed in the language PCF, a purely functional, sequential, call-by-name language with booleans and numbers as base types. The problem is called the "sequentiality problem" because, intuitively, the only way to program *por* involves evaluating boolean expressions in parallel. Of course, there are other elements in the Scott model of PCF that cannot be programmed: the domains have uncountably many elements. Nevertheless, the *por* function causes problems for reasoning about programs: it causes two terms in the language PCF to be distinct denotationally even though the terms cannot be distinguished by any program.

The problem of modelling sequentiality is enduring because it is robust. For instance, changing the reduction strategy of PCF from call-by-name to call-by-value makes no difference: versions of the *por* function reappear in the standard Scott model (albeit at higher type). Even for languages that lack an explicit base type, e.g., the polymorphic λ-calculus with recursion, the known Scott models contain parallel elements that cause a failure of full abstraction.

When examples like *por* do not exist, the denotational model is said to be **fully abstract**. More precisely, full abstraction requires an operational definition of equivalence (interchangeability in all programs) to match operational equivalence. Shortly after Plotkin's paper, Milner proved that there was exactly *one* fully abstract model of PCF meeting certain conditions [85]. Until recently, all descriptions of this fully abstract model have used operational semantics (see, for instance, [89, 126]). New constructions using logical relations [94, 121] have yielded a more abstract understanding of Milner's model. Game semantics [4, 52, 90] has also been used to give other semantic constructions of fully abstract models of PCF, even though it is still open whether these models are isomorphic to Milner's model.

This paper adapts and extends the logical-relations model for PCF to a call-by-value setting. More precisely, it constructs a model for a call-by-value, purely functional, sequential language called FPC. FPC includes a base type with one convergent value, strict products, strong (categorical) sums, functions, and recursive types. Full abstraction for FPC is interesting for at least two reasons. First, FPC can be regarded as the purely functional, non-polymorphic sublanguage of Standard ML [84]: recursive types and sums are the basis of `datatype` declarations, and both Standard ML and FPC are call-by-value. By studying FPC, we learn more about programming languages like Standard ML. Second, FPC can serve as an expressive metalanguage for denotational semantics [41, 100]. FPC, for instance, has enough expressive power to encode a call-by-value version of PCF (the base type of numbers can be encoded via a recursive type). Given a fully abstract translation [109] from a language into FPC, the model of FPC yields a fully abstract model of the language.

There is another, purely technical reason to be interested in FPC: since it contains only one trivial base type, we can learn more about the structure of Sieber's logical relations by studying FPC. Sieber's model of PCF, and the fully abstract model of PCF using Kripke relations [94], begin from a set of n-ary relations (for all n) at a **flat** base type—i.e., where the convergent elements of the domain are unordered. PCF thus builds in a limited form of sum type, available only at the base type. When we decompose that base type into a sum of the primitive base type, we may learn more about the nature of sequential computation.

As a preview to seeing how our relations decompose Sieber's, it is helpful to recall Sieber's definition. Suppose $A \subseteq B \subseteq \{1, \ldots, n\}$, and let $S^n_{A,B}$ be

$$\{(d_1, \ldots, d_n) \mid (\forall i \in A.\ d_i \neq \bot) \Rightarrow (\forall i, j \in B.\ d_i = d_j)\}.$$

Then R is an n-ary **sequentiality relation** if R is the intersection of relations of the form $S^n_{A,B}$. Sieber's sequentiality relations have an elegant semantic definition: nothing in the definition refers to the terms or operations of PCF. The relations can be used in simple proofs of the non-definability of elements, since all PCF terms preserve the sequentiality relations. For instance, it is easy to show that *por* does not preserve the sequentiality relation $S^3_{\{1,2\},\{1,2,3\}}$, and hence it must not be definable. The relations also seem to say something about sequential computation: if certain elements of a tuple must converge, then other elements must converge.

On a technical level, Sieber's definition of "sequentiality relation" seems limited to flat base types. It is difficult to see how, for instance, to extend the definition to complex sums such as $\mathbf{int} \oplus (\mathbf{int} \Rightarrow \mathbf{int})$, since it does not make sense to check for equality at functional type. Instead of directly extending Sieber's relations, our relations break down sequentiality relations into two components. The first captures the sequential behavior of *termination*: if certain elements in a tuple in the relation terminate, then certain other elements in the tuple must terminate. The second captures the behavior of *sums*: if certain elements in a tuple lie in one side of a sum type, other components must lie in the same side of the sum type. This is much like Sieber's definition in asking for *equality* of all B-indexed elements of a tuple in $S^n_{A,B}$. The interesting case comes when this second component of relations is lifted to types other than sums: when, for example, the tuple is a tuple of elements in a function type. In essence, the second component of relations encodes a form of "computation tree," stating which subtuples of a tuple form consistent traces of the computation so far.

We begin by introducing the syntax and operational semantics of FPC. We then describe the form of the relations, showing the decomposition into the two portions described above. We follow the O'Hearn-Riecke construction for PCF [94] and lift the relations to *Kripke relations of varying arity* [58]. We then define a category, in which the objects have partial-order as well as relational structure, and in which the morphisms preserve the partial-order and relational structure of objects. A model of FPC lives inside this category. The Kripke relations are then used to establish the full abstraction of the model.

9.2 The Language FPC

FPC is a call-by-value, purely functional language with single base type `unit`, sums, products, functions, and recursive types [41, 100]. Our version of FPC has types given by the grammar

$$s, t ::= \mathtt{void} \mid \mathtt{unit} \mid (s \oplus t) \mid (s \otimes t) \mid (s \Rightarrow t) \mid \alpha \mid (\mathtt{rec}\ \alpha.\ t)$$

where α ranges over a collection of **type variables**. This version of FPC has one more base type, namely `void`, than the standard version of FPC. The type `void` stands for the type with no convergent elements, and can be represented by the type $(\mathtt{rec}\ \alpha.\ \alpha)$, but having an explicit name makes some of the notation simpler. Types are identified up to renaming of type variables bound by `rec`. We assume that all types appearing in terms and the typing rules are closed unless otherwise noted.

The raw terms of FPC are given by the grammar

$$\begin{array}{rcl} M, N, P & ::= & \Omega \mid x \mid (\lambda x : t.\ M) \mid (M\ N) \mid \\ & & \langle\rangle \mid \langle M, N\rangle \mid (\mathtt{proj}_i\ M) \mid (\mathtt{inj}_i\ M) \mid \\ & & (\mathtt{case}\ M\ \mathtt{of}\ \mathtt{inj}_1(x).N\ \mathtt{or}\ \mathtt{inj}_2(x).P) \mid \\ & & (\mathtt{intro}_{\mathtt{rec}\ \alpha.\ s}\ M) \mid (\mathtt{elim}_{\mathtt{rec}\ \alpha.\ s}\ M) \end{array}$$

Ω denotes a divergent term, and only appears in our version of FPC in order to make the proofs simpler. A typing judgement is a formula of the form $\Gamma \vdash M : t$ where M is a term, t a type, and Γ is a **typing context**, i.e., a finite function from variables to types. Rules for deriving typing judgements appear in Table 9.1.

Evaluation rules, written in natural style, for FPC appear in Table 9.2. In the rules, we use the notation $M[N/x]$ to denote capture-free substitution of N for x in M. Notice that function application in FPC is call-by-value: arguments

Table 9.1: Type Rules for FPC.

$$\Gamma, x : t \vdash x : t$$

$$\Gamma \vdash \Omega : s$$

$$\Gamma \vdash \langle\rangle : \mathtt{unit}$$

$$\frac{\Gamma, x : s \vdash M : t}{\Gamma \vdash (\lambda x : s.\, M) : (s \Rightarrow t)}$$

$$\frac{\Gamma \vdash M : (s \Rightarrow t) \qquad \Gamma \vdash N : s}{\Gamma \vdash (M\ N) : t}$$

$$\frac{\Gamma \vdash M : s \qquad \Gamma \vdash N : t}{\Gamma \vdash \langle M, N\rangle : (s \otimes t)}$$

$$\frac{\Gamma \vdash M : (s_1 \otimes s_2)}{\Gamma \vdash (\mathtt{proj}_i\ M) : s_i}$$

$$\frac{\Gamma \vdash M : s_i}{\Gamma \vdash (\mathtt{inj}_i\ M) : (s_1 \oplus s_2)}$$

$$\frac{\Gamma \vdash M : (s_1 \oplus s_2) \qquad \Gamma, x : s_i \vdash N_i : t}{\Gamma \vdash (\mathtt{case}\ M\ \mathtt{of}\ \mathtt{inj}_1(x).N_1\ \mathtt{or}\ \mathtt{inj}_2(x).N_2) : t}$$

$$\frac{\Gamma \vdash M : s[\mathtt{rec}\ \alpha.\ s/\alpha]}{\Gamma \vdash (\mathtt{intro}_{\mathtt{rec}\ \alpha.\ s}\ M) : (\mathtt{rec}\ \alpha.\ s)}$$

$$\frac{\Gamma \vdash M : (\mathtt{rec}\ \alpha.\ s)}{\Gamma \vdash (\mathtt{elim}_{\mathtt{rec}\ \alpha.\ s}\ M) : s[\mathtt{rec}\ \alpha.\ s/\alpha]}$$

Table 9.2: Evaluation Rules for FPC.

$$\langle\rangle \Downarrow \langle\rangle$$

$$(\lambda x : s.\, M) \Downarrow (\lambda x : s.\, M)$$

$$\frac{M \Downarrow (\lambda x : s.\, M') \qquad N \Downarrow V' \qquad M'[V'/x] \Downarrow V}{(M\ N) \Downarrow V}$$

$$\frac{M_1 \Downarrow V_1 \qquad M_2 \Downarrow V_2}{\langle M_1, M_2\rangle \Downarrow \langle V_1, V_2\rangle}$$

$$\frac{M \Downarrow \langle V_1, V_2\rangle}{(\mathtt{proj}_i\ M) \Downarrow V_i}$$

$$\frac{M \Downarrow V}{(\mathtt{inj}_i\ M) \Downarrow (\mathtt{inj}_i\ V)}$$

$$\frac{M \Downarrow (\mathtt{inj}_i\ V) \qquad N_i[V/x] \Downarrow R}{(\mathtt{case}\ M\ \mathtt{of}\ \mathtt{inj}_1(x).N_1\ \mathtt{or}\ \mathtt{inj}_2(x).N_2) \Downarrow R}$$

$$\frac{M \Downarrow V}{(\mathtt{intro}_{\mathtt{rec}\ \alpha.\ s}\ M) \Downarrow (\mathtt{intro}_{\mathtt{rec}\ \alpha.\ s}\ V)}$$

$$\frac{M \Downarrow (\mathtt{intro}_{\mathtt{rec}\ \alpha.\ s}\ V)}{(\mathtt{elim}_{\mathtt{rec}\ \alpha.\ s}\ M) \Downarrow V}$$

to functions must be values before they are substituted into bodies of functions. We write $M \Downarrow$ if there is a term V such that $M \Downarrow V$, and $M \Uparrow$ if there is no such V. The operational approximation relation can then be defined as follows:

Definition 9.2.1 $M \sqsubseteq_{FPC} N$ if for any context $C[\cdot]$ such that $C[M]$ and $C[N]$ are closed, well-typed terms, $C[M] \Downarrow$ implies $C[N] \Downarrow$.

FPC is a sparse language, but it still has enough computing power for many applications. For instance, [100] and [41] show how to build recursion operators using recursive types. One can encode a sequencing operation in FPC: $(M; N)$ stands for the term $((\lambda x : s.\, N)\ M)$, where x does not occur free in N. Indeed, the semantics of many programming languages—including non-functional languages—can be given by translation to FPC. FPC's main deficiency as a metalanguage for denotational semantics is a lack of parametric polymorphism (as in the Girard-Reynolds calculus [38, 106]), which precludes a good representation of abstract data types.

9.3 Category of Meanings

This section defines a category suitable for interpreting FPC, and gives various constructions for defining the meaning of types and terms. Objects in the category will have both partial-order structure and relational structure. More precisely, the objects are composed of dcpo's, i.e., **directed-complete posets** not necessarily possessing a least element, and relations on those dcpo's. The morphisms of the category preserve the dcpo structure (i.e., are partial, continuous functions), and preserve the relational structure (a property we call **uniformity**). The formal definitions of dcpo and continuity may be found elsewhere [42, 100].

The construction consists of five main parts:

1. Definition of the relational structures (Sections 9.3.2-9.3.3).

2. Definition of the category, and the verification that it is a dcpo-enriched category (Section 9.3.4).

3. Definitions of meanings for `void`, `unit` as objects in the category, and of tensor product $\otimes$, coproduct $\oplus$, and function space $\Rightarrow$ as functors, and verification that they satisfy certain conditions (Sections 9.3.5-9.3.6).

4. Verification that the category is "partial cartesian closed" (defined below), and hence a model of FPC without recursive types (Section 9.3.7).

5. Construction of colimits, and verification that the colimits produce objects in the category (Section 9.3.8).

Most of the proofs are straightforward and follow well-established patterns from category theory.

9.3.1 Preliminaries

Given two sets X, Y, the set $(X \oplus Y)$ is the disjoint union, $(X \times Y)$ is the cartesian product, $[X \to_t Y]$ is the set of total functions from X to Y, and $[X \to_p Y]$ is the set of partial functions from X to Y. The identity function on a set X is denoted id_X, and $(h; f)$ denotes the diagrammatic composition of two functions h and f. The functions $inj_i : X_i \to (X_1 \oplus X_2)$ are the injection functions into the disjoint sum, and $proj_i : (X_1 \times X_2) \to X_i$ are the projection functions from the cartesian product. For products, we abuse notation and use $\langle x, y\rangle$ for elements of $X \times Y$ and $\langle f, g\rangle$ for functions $W \to (X \times Y)$ when $f : W \to X$ and $g : W \to Y$.

Partial functions require some special notation. We write $f(d)\downarrow$ when a partial function f is defined on argument d, and $f(d)\uparrow$ when $f(d)$ is undefined. Kleene equality takes definedness between mathematical expressions into account: we write $exp_1 \simeq exp_2$ if, whenever one side is defined, both sides are defined and equal. Similarly, we extend an ordering relation $\sqsubseteq$ to $\precsim$ as follows: $exp_1 \precsim exp_2$ if, when exp_1 is defined and equal to e_1, then exp_2 is defined and equal to some e_2 and $e_1 \sqsubseteq e_2$.

Following Jung and Tiuryn, we often represent the elements of relations as *finite, total functions* from indices to values instead of tuples of values; this simplifies some of the definitions and proofs [58, 94].

9.3.2 Termination and path theories

There are two kinds of relations in the model, **termination relations** and **computational relations**.

The first kind of relation uses subsets of indices generated from simple implicational theories. If w is a finite set of indices, then a w**-termination theory** is a set of implications of the form $(w' \vdash d)$ where $d \in w$ and w' is a subset of w. Intuitively, each implication states a property similar to the Sieber's sequentiality relations: if a function halts on the indices in the premise, it must halt on the index in the conclusion. Indeed, the termination part of Sieber's relations $S^n_{A,B}$ can be encoded using implications:

$$\text{if } A \subseteq B \subseteq w = \{1, \ldots, n\} \text{ and } B \setminus A = \{d_1, \ldots, d_k\},$$
$$\text{then } S^w_{A,B} \text{ corresponds to the implications } (A \vdash d_1), \ldots, (A \vdash d_k).$$

The subsets of w that validate the implications in the theory are the building blocks of termination relations. Suppose $w' \subseteq w$; then we say that w' **validates** $(w'' \vdash d)$, written $w' \models (w'' \vdash d)$, if $w'' \subseteq w'$ implies $d \in w'$. If T is a w-termination theory, we say that $w' \subseteq w$ is a T**-model** if $w' \models \psi$ for all $\psi \in T$.

There is an alternative characterization of T-models that can be helpful in proving facts about termination relations. Suppose w is a finite set and $X \subseteq \mathcal{P}(w)$. Then X is a **closure system** if $w \in X$ and X is closed under intersection. A closure system determines a w-termination theory:

Proposition 9.3.1 *Suppose w is a finite set. Then $X \subseteq \mathcal{P}(w)$ is a closure system iff there is a w-termination theory T such that X is the set of T-models.*

Proof: ($\Leftarrow$) Suppose T is a w-theory. Then obviously w is a T-model. To see closure under intersection, suppose w_1, w_2 are T-models. Suppose $(w' \vdash d) \in T$, and $w' \subseteq w_1 \cap w_2$. Then $d \in w_1$ and $d \in w_2$, hence $d \in w_1 \cap w_2$. Thus, $w_1 \cap w_2$ is also a T-model.

($\Rightarrow$) Suppose X is a closure system. Define the theory T by

$$(w'' \vdash d) \in T \text{ iff for all } w' \in X,\ w' \models (w'' \vdash d).$$

Let $Y = \{w' \mid w' \text{ is a } T\text{-model}\}$; we want to show that $X = Y$. It is obvious that $X \subseteq Y$: if $w' \in X$, then by construction it validates all the formulas in T and hence is a T-model. Conversely, suppose $w' \notin X$. Let

$$w_0 = \bigcap \{w_1 \in X \mid w' \subseteq w_1\}.$$

Since X is closed under finite intersections, and the set above is finite because w is, w_0 must be in X. But since $w' \notin X$ and $w' \subseteq w_0$, it must be the case that there is a $d \in w_0$ such that $d \notin w'$. Now consider the formula $(w' \vdash d)$. Note that $(w' \vdash d) \in T$, and $w' \not\models (w' \vdash d)$. Thus, $w' \notin Y$, completing the proof.

(See [132], page 22 for the same proof.) The proof is similar to a part of Sieber's proof that the sequentiality relations are precisely those that are closed under the operations of PCF [121].

The second kind of relation is built from sets of sets of T-models. If T is a w-termination theory, define a **path set** to be a set $\{w_1, \ldots, w_n\}$ of nonempty, disjoint T-models. (The reason for the name "path set" will become clear shortly.) By convention, when we write a path set we assume there are no duplicates. For example, when we write the path set $\{w_1, \ldots, w_n\}$, each w_i differs from the others. A T**-path theory** S is a set of path sets that satisfies the following conditions:

- $\{\} \in S$ (using the alternative notation for the empty set);
- If w' is a nonempty T-model, then $\{w'\} \in S$;
- If $X = \{w_1, \ldots, w_k\}$ and $X' = \{w'_1, \ldots, w'_l\}$ are in S, and $w_{i,j} = (w_i \cap w'_j)$, then $X \sqcap X' = \{w_{i,j} \mid w_{i,j} \text{ is nonempty}\} \in S$; and
- If $\{w_1, \ldots, w_k\}$ and $\{w'_1, \ldots, w'_l\}$ are in S, and for some j and all i, $w'_i \subseteq w_j$, then
$$\{w_1, \ldots, w_{j-1}, w'_1, \ldots, w'_l, w_{j+1}, \ldots, w_k\} \in S.$$

This definition appears to be related to the notion of a Grothendieck topology [35].

9.3.3 Relations

Termination theories are the building blocks of the first kind of relations, the termination relations. Let T be a w-termination theory, and D be a dcpo. A

T-termination relation on D is a set of the form

$$R \subseteq \bigcup_{w' \text{ is a } T\text{-model}} [w' \to_t D],$$

such that the following properties hold:

1. **Directed completeness:** R is directed complete, where $f \sqsubseteq g$ iff f and g have the same domain w', and for all $d \in w'$, $f(d) \sqsubseteq g(d)$ in D.

2. **Downward closure:** For any $f \in R$ with domain w' and T-model $w'' \subseteq w'$, $(\underline{\lambda} d \in w''.\, f(d)) \in R$.

Here, $(\underline{\lambda} d \in w''.\, f(d))$ stands for the function with domain w'', whose return value is $f(d)$; thus, an element of a termination relation R is a function from a subset of indices to elements of the dcpo. To get some intuition, it is again helpful to think of indices as possible arguments to a function, and the elements as the return values of the function. An element of R then represents a "related" set of values returned by a function.

There is a subtle point in the definition of the $\sqsubseteq$ relation: only elements of R that have the *same domain* can be related. One might imagine a different definition, with "tuples" in the relations being partial functions whose domains are T-models. Under this alternative definition, two tuples might be related by $\sqsubseteq$ even if they had different domains. The definition of $\otimes$ on relations does not produce a directed complete relation, however, using this alternative definition.

The second form of relations, computational relations, is built from path theories. These relations lift a termination relation to a partial function on indices. Suppose S is a T-path theory and R is a T-termination relation on D. The **computational relation** R_S is defined by

$$\begin{aligned} R_S = \{\, f \in [w \to_p D] \mid{} & \text{there exists } \{w_1, \ldots, w_k\} \in S \text{ such that } f(d)\!\downarrow \text{ iff} \\ & d \in w_i \text{ for some } i, \text{ and for all } i,\ (\underline{\lambda} d \in w_i.\, f(d)) \in R \,\} \end{aligned}$$

In this case, the order relation on elements of R_S is defined in the usual way:

$$f \sqsubseteq g \text{ if whenever } f(d)\!\downarrow, \text{ then } g(d)\!\downarrow \text{ and } f(d) \sqsubseteq g(d).$$

The computational relations are reminiscent of Moggi's analysis of call-by-value via monads [87], and they will play a similar role in the semantics below.

These definitions have a common intuition of "tests" that one can apply to a function, with the tests designed so that only sequential functions will pass all tests. In this case, a test consists of a number of simultaneous applications of the function to arguments, where the arguments must also pass the test. Termination theories are used to test the termination behavior of functions; path theories are used to test the consistency of the branching structure of functions.

For example, let `bool` denote the FPC type ($\texttt{unit} \oplus \texttt{unit}$) and `true`,`false` denote the terms ($\texttt{inj}_1\langle\rangle$), ($\texttt{inj}_2\langle\rangle$), and consider the term

$$f = (\lambda x : \texttt{bool}.\ \texttt{if}\ x\ \texttt{then}\ \texttt{false}\ \texttt{else}\ \texttt{true}).$$

Suppose we pick the termination theory $\{1, 2 \vdash 3\}$. For the function f to pass a test determined by this termination theory, we must apply it to three arguments themselves satisfy the test. Such a test of three arguments might be, for instance, the tuple $\mathtt{true}, \mathtt{true}, \mathtt{true}$. When the function is applied to three arguments, the result is $\mathtt{false}, \mathtt{false}, \mathtt{false}$. All three answers terminate, so this test of the function succeeds. The test $\mathtt{true}, \mathtt{true}, \Omega$ does not yield a successful test: the function returns $\mathtt{false}$, $\mathtt{false}$, and diverges respectively. Fortunately, however, the arguments themselves do not pass the test themselves, so the function need not satisfy this test.

Tests of the branching behavior of functions using the path theories are similar. A computation branches based on its inputs and returns some final results at the end. Each set in $\{w_1, \ldots, w_n\}$ represents a path in that computational tree; the answers returned at the end of a path must be consistent, hence the restriction of the function to $(\underline{\lambda} d \in w_i.\, f(d))$ must be in R. For example, suppose the termination theory is as above, and that the path theory contains the set $\{\{1\}, \{2\}, \{3\}\}$. Then the function need not return consistent results given three arguments, but the function must halt on all three arguments.

We can now see from where the four conditions on path theories come. The first condition says that the empty set is a valid test; the second says that a potential path set that does no branching is a valid path set; the third says path sets can be combined; the fourth says that an element of a path set may be replaced by finer-grain path set, which amounts to adding a set of branches to a non-branching part of the computation.

9.3.4 Definition of the category

One could build a category from termination and path theories, but the result would probably not contain a fully abstract model of FPC (see the discussion in Section 7). Instead, we extend the relations to Kripke relations of varying arity [58]. Kripke relations of this kind begin from an **index category** whose objects are finite sets and whose morphisms are total functions (not necessarily all of them). Suppose $\mathcal{C}$ is an index category. A **$\mathcal{C}$-termination theory** T is an $\mathrm{Ob}(\mathcal{C})$-indexed family of w-termination theories T^w such that, for any $\varphi : v \xrightarrow{\mathcal{C}} w$, if w' is a T^w-model, then $\varphi^{-1}(w')$ is a T^v-model, where $\varphi^{-1}(w') = \{d \in v \mid \varphi(d) \in w'\}$. A Kripke relation is a set of termination relations that must fit together. Let $\mathcal{C}$ be an index category, T be a $\mathcal{C}$-termination theory, and D be a dcpo. A family R is a **$\mathcal{C}, T$-termination relation on** D if R is an $\mathrm{Ob}(\mathcal{C})$-indexed family of T^w-termination relations R^w on D satisfying the

> **Kripke monotonicity condition:** For any $f \in R^w$ with domain w' and $\varphi : v \xrightarrow{\mathcal{C}} w$, then $(\underline{\lambda} d \in v'.\, f(\varphi(d))) \in R^v$, where $v' = \varphi^{-1}(w')$.

Path theories also extend straightforwardly to the Kripke case. Let $\mathcal{C}$ be an index category and T be a $\mathcal{C}$-termination theory. Then S is a **$\mathcal{C}, T$-path theory** if S is a family, indexed by objects w of $\mathcal{C}$, of T^w-path theories. If S is a $\mathcal{C}, T$-path theory and R is a $\mathcal{C}, T$-termination relation on D, we let R^w_S denote the computational relation built from R^w and S^w.

We now have enough machinery to build the category $\mathcal{RCPO}$ (for dcpo's with relational structure).

- Objects. An object A consists of a dcpo $|A|$ and a $\mathcal{C}, T$-termination relation $A(T,S)$ on $|A|$ for each index category $\mathcal{C}$, $\mathcal{C}$-termination theory T, and $\mathcal{C}, T$-path theory S. Objects must also satisfy the

  Concreteness Condition: For any $a \in |A|$, $(\underline{\lambda} d \in w.\, a) \in A(T,S)^w$.

- Morphisms. A morphism $f : A \to B$ is a partial continuous function $f : |A| \to_p |B|$ satisfying the

  Uniformity Condition: For all $\mathcal{C}$, $\mathcal{C}$-termination theories T, $\mathcal{C}, T$-path theories S, and $h \in A(T,S)^w$, $(h; f) \in B(T,S)^w_S$.

The definitions of composition and identities are the same as on partial continuous functions. It is straightforward to check that $\mathcal{RCPO}$ is indeed a category; the only non-obvious step is checking that composition produces uniform functions, which follows from

Lemma 9.3.2 *If $f \in A(T,S)^w_S$ and $g : A \to B$, then $(f; g) \in B(T,S)^w_S$.*

Proof: By definition, there exists a path set $\{w_1, \ldots, w_n\}$ such that $f(d)\downarrow$ iff $d \in w_i$ for some i and $(\underline{\lambda} d \in w_i.\, f(d)) \in A(T,S)^w$ for all i. By the uniformity of g, for all i, $h_i = ((\underline{\lambda} d \in w_i.\, f(d)); g) \in B(T,S)^w_S$. Thus, for all i, there exists a path set $\{w_{i,1}, \ldots, w_{i,k_i}\}$ such that

- $h_i(d)\downarrow$ iff $d \in w_{i,j}$ for some j; and
- $(\underline{\lambda} d \in w_{i,j}.\, h_i(d)) \in B(T,S)^w$ for all j.

Note that each $w_{i,j} \subseteq w_i$, since the domain of h_i is a subset of w_i. Thus, by the properties of path sets, $\{w_{1,1}, \ldots, w_{n,k_n}\}$ is a path set. Note also that $g(f(d))\downarrow$ iff $d \in w_{i,j}$ for some i, j, and $(\underline{\lambda} d \in w_{i,j}.\, g(f(d))) \in B(T,S)^w$ for all i, j. Thus, $(f; g) \in B(T,S)^w_S$.

Moreover, the category is dcpo-enriched, i.e., under the usual pointwise ordering of morphisms with $f \sqsubseteq g$ iff for any $a \in |A|$, $f(a) \precsim g(a)$, the set of morphisms from A to B $Hom_{\mathcal{RCPO}}(A,B)$ is a dcpo. The proof of this fact, stated precisely as Proposition 9.3.5 below, requires two lemmas:

Lemma 9.3.3 *Suppose $f \in A(T,S)^w_S$ with path set $W = \{w_1, \ldots, w_k\}$. Suppse $W' = \{w'_1, \ldots, w'_l\} \in S$, and $\bigcup W = \bigcup W'$. Then $W \sqcap W' = W'' = \{w''_1, \ldots, w''_m\}$ is also a path set for f. That is, $d \in w''_i$ iff $f(d)\downarrow$ and for all i, $(\underline{\lambda} d \in w''_i.\, f(d)) \in A(T,S)^w$.*

Proof: It is easy to see that $\bigcup W'' = \bigcup W$, and hence it follows that $d \in w''_i \in W''$ iff $f(d)\downarrow$.

Lemma 9.3.4 *If $A(T,S)$ is a $\mathcal{C}, T$-relation on A, then $A(T,S)^w_S$ is directed complete.*

Proof: Suppose $X = \{f_i \mid i \in I\}$ is directed in $A(T,S)^w_S$, and $f = (\bigsqcup X)$. Then for all i, there exists a path set $\sigma_i = \{w_{i,1}, \ldots, w_{i,k_i}\}$ such that $f_i(d)\downarrow$ iff $d \in w_{i,j}$ for some j, and $(\underline{\lambda} d \in w_{i,j}.\, f_i(d)) \in A(T,S)^w$ for all j. Note that if $f_i, f_j \in X$, then there is an f_l such that $f_i, f_j \sqsubseteq f_l$ (by directedness), and hence $\bigcup \sigma_i, \bigcup \sigma_j \subseteq \bigcup \sigma_l$. Since each $\bigcup \sigma_i$ is a subset of the finite set w, there is a k such that for all $f_i \sqsupseteq f_k$, $\bigcup \sigma_i = \bigcup \sigma_k$.

Let $w' = \bigcup \sigma_k$. It is easy to see that $d \in w'$ iff $f(d)\downarrow$. What we need to show is that w' can be subdivided into a set of disjoint sets that forms a path set, and that this path set "witnesses" the membership of f in $A(T,S)^w_S$. There are only finitely many distinct path sets $W_1, \ldots, W_k$ for the f_i's above f_k such that for all i, $\bigcup W_i = w'$. Define

$$W = W_1 \sqcap W_2 \sqcap \ldots \sqcap W_k.$$

By Lemma 9.3.3, W is a path set for f. Now, note that

$$f = \bigcup \{ \bigsqcup \{ (\underline{\lambda} d \in w_j.\, f_i(d)) \mid f_i \sqsupseteq f_k \} \mid w_j \in W \}$$

By the directed completeness of $A(T,S)^w$, for any $w_j \in W$,

$$\bigsqcup \{ (\underline{\lambda} d \in w_j.\, f_i(d)) \mid f_i \sqsupseteq f_k \} \in A(T,S)^w.$$

Thus, $f \in A(T,S)^w_S$.

Proposition 9.3.5 *$Hom_{\mathcal{RCPO}}(A,B)$ is a dcpo.*

Proof: Suppose $X = \{f_i \mid i \in I\} \subseteq Hom_{\mathcal{RCPO}}(A,B)$ is directed. We need to show that $(\bigsqcup X)$ is uniform; it is routine to show that it is continuous. Suppose $h \in A(T,S)^w$. By the uniformity of each f_i, $(h; f_i) \in B(T,S)^w_S$. It is easy to see that $\{\, (h; f_i) \mid i \in I \,\}$ is directed. Thus, by Lemma 9.3.4, $(h; (\bigsqcup X)) \in B(T,S)^w_S$.

9.3.5 Tensor products and coproducts

The category admits certain standard constructions. For instance, it is easy to show that *void* and *unit*, defined by

$$\begin{array}{rclrcl} |void| & = & \emptyset & |unit| & = & \{\top\} \\ void(T,S)^w & = & \emptyset & unit(T,S)^w & = & \{\, (\underline{\lambda} d \in w'.\, \top) \mid w' \text{ is a } T^w\text{-model} \,\} \end{array}$$

are objects in the category. The relational component of *unit* captures much of the intuition embedded in termination relations: elements in the termination relation are functions which terminate precisely on elements in a T^w-model.

The category has a notion of tensor product. Suppose A, B are objects, and $f : A \to B$ and $g : C \to D$ are morphisms. Define

$$\begin{array}{rcl} |A \otimes B| & = & |A| \times |B| \quad \text{(cartesian product)} \\ (A \otimes B)(T,S)^w & = & \{ \langle g, h \rangle \mid g \in A(T,S)^w,\ h \in B(T,S)^w, \\ & & \qquad \text{and } g, h \text{ have the same domain} \} \\ (f \otimes g)\langle x, y \rangle & = & \langle f(x), g(y) \rangle \end{array}$$

Proposition 9.3.6 $(- \otimes -)$ *is a bifunctor on* $\mathcal{RCPO}$*, covariant in both arguments.*

Proof: The only non-obvious part is checking that the relations generated by $\otimes$ is directed complete. Suppose $X = \{f_i \mid i \in I\} \subseteq (A \otimes B)(T,S)^w$ is directed. Then all of the elements of X have the same domain, say w' (which is a T^w-model). Note that there exist $g_i \in A(T,S)^w$ and $h_i \in B(T,S)^w$ such that $f_i = \langle g_i, h_i \rangle$. Therefore, it must be the case that the sets

$$X_1 = \{g_i \mid i \in I\} \qquad\qquad X_2 = \{h_i \mid i \in I\}$$

are directed. By hypothesis, $\bigsqcup X_1 \in A(T,S)^w$ and $\bigsqcup X_2 \in B(T,S)^w$. Since $(\bigsqcup X) = \langle \bigsqcup X_1, \bigsqcup X_2 \rangle$, $(\bigsqcup X) \in (A \otimes B)(T,S)^w$.

The proof of Proposition 9.3.6 requires that directed sets in $(A \otimes B)(T,S)^w$ have the same domain, which explains the definition earlier of the order on elements of termination relations.

The category also has coproducts in the usual sense. Suppose A, B are objects, and $f : A \to B$ and $g : C \to D$ are morphisms. Define

$$\begin{array}{rcl} |A \oplus B| & = & |A| \oplus |B| \quad \text{(disjoint union)} \\ (A \oplus B)(T,S)^w & = & \{\, (g; \mathit{inj}_1) \mid g \in A(T,S)^w \,\} \cup \{\, (g; \mathit{inj}_2) \mid g \in B(T,S)^w \,\} \\ (f \oplus g)(x : A \oplus C) & = & \begin{cases} f(y) & \text{if } x = \mathit{inj}_1(y) \\ g(y) & \text{if } x = \mathit{inj}_2(y) \end{cases} \end{array}$$

This definition yields the coproduct in the category, as the following proposition shows.

Proposition 9.3.7 $\mathcal{RCPO}$ *has coproducts.*

Proof: It is not hard to prove that $\oplus$ is a bifunctor on $\mathcal{RCPO}$. To finish the proof that $\oplus$ yields the coproduct in the category, suppose $f : A \to C$ and $g : B \to C$ are morphisms. Define

$$[f,g](d) = \begin{cases} f(a) & \text{if } d = \mathit{inj}_1(a) \text{ and } f(a)\downarrow \\ g(b) & \text{if } d = \mathit{inj}_2(b) \text{ and } g(b)\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

We need to show that $\mathit{inj}_1 : A \to (A \oplus B)$, $\mathit{inj}_2 : B \to (A \oplus B)$, and $[f,g] : (A \oplus B) \to C$ are morphisms in the category, and that $[f,g]$ is the unique morphism making the diagram

$$\begin{tikzcd} A \arrow[r, "\mathit{inj}_1"] \arrow[dr, "f"'] & (A \oplus B) \arrow[d, "{[f,g]}"] & B \arrow[l, "\mathit{inj}_2"'] \arrow[dl, "g"] \\ & C & \end{tikzcd}$$

commute.

To see that the injections are morphisms, consider the case of inj_1. It is obvious that inj_1 is a partial continuous function, so we only need to verify that inj_1 satisfies the uniformity condition. Suppose $h \in A(T,S)^w$ with domain $w' \subseteq w$. We need to show $(h; inj_1) \in (A \oplus B)(T,S)^w_S$. Since w' is a T^w-model, $\{w'\}$ is itself a path set. It is then easy to see that $(h; inj_1)(d)\downarrow$ iff $d \in w'$, and $(\underline{\lambda} d \in w'.\,(h; inj_1)(d)) = (h; inj_1) \in (A \oplus B)(T,S)^w$ (by definition of $(A \oplus B)(T,S)^w$). Thus $(h; inj_1) \in (A \oplus B)(T,S)^w_S$.

To see that $[f,g]$ is a morphism, it is easy to check that $[f,g]$ is a partial continuous function. We need only verify the uniformity condition. Suppose $h \in (A \oplus B)(T,S)^w$; we want to show that $(h; [f,g]) \in C(T,S)^w_S$. By definition, either $h = (h_1; inj_1)$ for $h_1 \in A(T,S)^w$ or $h = (h_2; inj_2)$ for $h_2 \in B(T,S)^w$. Consider the first case (the second case is analogous). Then $(h; [f,g]) = (h_1; f)$. By uniformity, $(h_1; f) \in C(T,S)^w_S$. Thus, $(h; [f,g]) \in C(T,S)^w_S$.

What remains to be shown is that $[f,g]$ makes the diagram commute, and that $[f,g]$ is the unique such morphism. We consider the left triangle (the right triangle is analogous). So suppose $a \in |A|$. Observe that $f(a)\downarrow$ iff $(inj_1; [f,g])(a)\downarrow$, so if $f(a)\downarrow$, then

$$(inj_1; [f,g])(a) \quad = \quad f(a)$$

and hence the triangle commutes. To see the uniqueness of $[f,g]$, suppose that $h : (A \oplus B) \to C$ is such that the diagram commutes, i.e., $f = (inj_1; h)$ and $g = (inj_2; h)$. Since the injection functions are total functions, the domain of h must be

$$\{d \in |A \oplus B| \mid (d = inj_1(a) \text{ and } f(a)\downarrow) \text{ or } (d = inj_2(b) \text{ and } f(b)\downarrow)\}.$$

Thus $[f,g]$ and h have the same domain.

Now suppose $h(d)\downarrow$. Suppose $d = inj_1(a)$ (the other case is analogous). Then

$$h(d) \quad = \quad (inj_1; h)(a) = f(a) = [f,g](d).$$

That is, $h = [f,g]$.

9.3.6 Function space

The category also has an operation associated with a space of functions. Suppose A, B are objects, and $f : A \to B$ and $g : C \to D$ are morphisms. Define

$$\begin{array}{rcl}
|A \Rightarrow B| & = & Hom_{\mathcal{RCPO}}(A,B) \\
(A \Rightarrow B)(T,S)^w & = & \{\, f \mid \forall \varphi : v \xrightarrow{\mathrm{e}} w, g \in A(T,S)^v. \\
& & \quad (\underline{\lambda} d \in v.\,(f(\varphi(d))\,(g(d)))) \in B(T,S)^v_S\} \\
(f \Rightarrow g)(h : B \to C) & = & (f; h; g) : A \to D
\end{array}$$

The definition of function space on relations is the same as the one for Kripke relations [58, 94], except that the result of applying related arguments to related results must be in the computational relation, not the termination relation. Again, this is reminiscent of the monad style of semantics [87].

We must verify that $\Rightarrow$ is a bifunctor, contravariant in the first argument and covariant in the second. The proof is divided into two parts:

Proposition 9.3.8 *$(A \Rightarrow B)$ is an object.*

Proof: By Proposition 9.3.5, we know that $|A \Rightarrow B|$ is directed complete. Suppose $\mathcal{C}$ is an index category, T is a $\mathcal{C}$-termination theory, and S is a $\mathcal{C}, T$-path theory. We need to show that $(A \Rightarrow B)(T, S)$ is a $\mathcal{C}, T$-termination relation and the concreteness condition holds.

First, we must show that $(A \Rightarrow B)(T, S)$ is a $\mathcal{C}, T$-termination relation. To see directed completeness, suppose $X = \{f_i \mid i \in I\} \subseteq (A \Rightarrow B)(T, S)^w$ is directed. Then all of the elements of X have the same domain, say w' (which is a T^w-model). To see $(\bigsqcup X) \in (A \Rightarrow B)(T, S)^w$, suppose $\varphi : v \xrightarrow{\mathcal{C}} w$ and $h \in A(T, S)^v$. Then for all $i \in I$, $g_i = (\underline{\lambda} j \in v.\, f_i\ (\varphi(j))\ (h(j))) \in B(T, S)^v_S$. Since $\{g_i \mid i \in I\}$ is directed, by Lemma 9.3.4 $\bigsqcup g_i \in B(T, S)^v_S$. Thus,

$$(\underline{\lambda} j \in v.\, (\bigsqcup X)\ (\varphi(j))\ (h(j))) \in B(T, S)^v_S$$

as needed.

The difficulty lies in showing downward closure and Kripke monotonicity. Suppose $f \in (A \Rightarrow B)(T, S)^w$ with domain w', and suppose w'' is a T^w-model with $w'' \subseteq w'$. Let $f' = (\underline{\lambda} d \in w''.\, f(d))$. To see that $f' \in (A \Rightarrow B)(T, S)^w$, suppose $\varphi : v \xrightarrow{\mathcal{C}} w$ and $g \in A(T, S)^v$. Then we know that

$$h = (\underline{\lambda} d \in v.\, f\ (\varphi(d))\ (g(d))) \in B(T, S)^v_S.$$

Thus, there is some path set $\{v_1, \ldots, v_n\}$ such that $h(d)\downarrow$ iff $d \in v_i$ for some i, and for all i,

$$(\underline{\lambda} d \in v_i.\, f\ (\varphi(d))\ (g(d))) \in B(T, S)^v.$$

Let $h' = (\underline{\lambda} d \in v.\, f'\ (\varphi(d))\ (g(d)))$. We need to show that $h' \in B(T, S)^v_S$. First, we need to build up the right path set. Notice that $v'' = \varphi^{-1}(w'')$ is a T^v-model by the definition of $\mathcal{C}$-termination theories. Since $X = \{v_1, \ldots, v_n\}$ and $X'' = \{v''\}$ are path sets, $X \sqcap X'' = \{v'_1, \ldots, v'_k\}$ is a path set (recall that $X \sqcap X''$ is the set of all $(v \cap v'')$, with $v \in X$ and $v'' \in X''$, that are non-empty). It is clear that $h'(d)\downarrow$ iff $d \in v'_i$ for some i. Also, since each $v'_i \subseteq v_j$ for some j, it follows by downward closure that $(\underline{\lambda} d \in v'_i.\, f\ (\varphi(d))\ (g(d))) \in B(T, S)^v$ for any i. Because $(\underline{\lambda} d \in v'_i.\, f\ (\varphi(d))\ (g(d))) = (\underline{\lambda} d \in v'_i.\, f'\ (\varphi(d))\ (g(d)))$, it follows that

$$(\underline{\lambda} d \in v'_i.\, f'\ (\varphi(d))\ (g(d))) \in B(T, S)^v.$$

Thus, $h' \in B(T, S)^v_S$ as desired.

To see Kripke monotonicity, suppose $\varphi : v \xrightarrow{\mathcal{C}} w$ and $f \in (A \Rightarrow B)(T, S)^w$ with domain w'. Let $v' = \varphi^{-1}(w')$; we know that v' is a T^v-model since T is a $\mathcal{C}$-termination theory. We want

$$(\underline{\lambda} d \in v'.\, f(\varphi(d))) \in (A \Rightarrow B)(T, S)^v.$$

So suppose $\psi : u \xrightarrow{\mathrm{e}} v$ and $g \in A(T,S)^u$. Then

$$\begin{array}{rcl}(\underline{\lambda}d \in u.\,(\underline{\lambda}e \in v'.\,f(\varphi(e)))\ (\psi\ d)\ (g(d))) & = & (\underline{\lambda}d \in u.\,f(\varphi(\psi(d)))\ (g(d))) \\ & \in & B(T,S)^u_S\end{array}$$

by the fact that $(\psi;\varphi) : u \xrightarrow{\mathrm{e}} w$. Thus, $(\underline{\lambda}d \in v'.\,f(\varphi(d))) \in (A \Rightarrow B)(T,S)^v$.

Finally, to show concreteness, suppose $f \in |A \Rightarrow B|$; we need to show that $(\underline{\lambda}i \in w.\,f) \in (A \Rightarrow B)(T,S)^w$. From the definition, we must show that, for $\varphi : v \xrightarrow{\mathrm{e}} w$ and $h \in A(T,S)^v$, $(\underline{\lambda}j \in v.\,((\underline{\lambda}i \in w.\,f)\ (\varphi(j)))\ (h(j))) \in B(T,S)^v_S$. But this reduces to $(\underline{\lambda}j \in v.\,f\ (h(j))) \in B(T,S)^v_S$, which is the uniformity condition on $\mathcal{RCPO}$-morphisms.

Proposition 9.3.9 $(- \Rightarrow -)$ *is a bifunctor on* $\mathcal{RCPO}$ *that is contravariant in the first argument and covariant in the second.*

Proof: By Proposition 9.3.8, if A, B are objects, then $|A \Rightarrow B|$ is too. To check the functor part, suppose $f : A' \to A$ and $g : B \to B'$ in $\mathcal{RCPO}$. The uniformity condition is preserved by composition (see Lemma 9.3.2), as is relevant domain-theoretic structure, so we may conclude that for any h, $(f;h;g) \in |A' \Rightarrow B'|$. To see that $(f \Rightarrow g)$ *as a function* satisfies the uniformity condition, consider $m \in (A \Rightarrow B)(T,S)^w$: we need to show that

$$(\underline{\lambda}d \in w.\,f;m(d);g) \in (A' \Rightarrow B')(T,S)^w_S.$$

If we can show that $h' = (\underline{\lambda}d \in w'.\,\underline{\lambda}a.\,g(m\ d\ f(a))) \in (A' \Rightarrow B')(T,S)^w$, where w' is the domain of m, then we will be done—the required path set will be $\{w'\}$. So suppose $\varphi : v \xrightarrow{\mathrm{e}} w$ and $h \in A'(T,S)^v$ and let

$$\begin{array}{rcl}h'' & = & (\underline{\lambda}d \in v.\,h'(\varphi(d))(h(d))) \\ & = & (\underline{\lambda}d \in v.\,g(m(\varphi(d))(f(h(d))))).\end{array}$$

We then need to show that $h'' \in B'(T,S)^v_S$. By the uniformity of f, we get $(h;f) \in A(T,S)^v_S$. Thus, there exists a path set $\{v_1,\ldots,v_n\}$ such that $f(h(d))\downarrow$ iff $d \in v_i$ for some i, and $f_i = (\underline{\lambda}d \in v_i.\,f(h(d))) \in A(T,S)^v$ for all i. Thus, for each f_i,

$$h_i = (\underline{\lambda}d \in v_i.\,m\ (\varphi(d))\ (f_i(d))) \in B(T,S)^v_S$$

Therefore, for each i, there exists a path set $\{v_{i,1},\ldots,v_{i,k_i}\}$ such that

- $h_i(d)\downarrow$ iff $d \in v_{i,j}$ for some j; and
- $h_{i,j} = (\underline{\lambda}d \in v_{i,j}.\,m\ (\varphi(d))\ (f_i(d))) \in B(T,S)^v$ for all j.

Note as well that each $v_{i,j} \subseteq v_i$. By the uniformity of g, $(h_{i,j};g) \in B'(T,S)^v_S$. Thus, for each i,j, there exists a path set $\{v_{i,j,1},\ldots,v_{i,j,l_{i,j}}\}$ such that

- $h_{i,j}(d)\downarrow$ iff $d \in v_{i,j,k}$ for some k; and
- $h_{i,j,k} = (\underline{\lambda}d \in v_{i,j,k}.\,g(m\ (\varphi(d))\ (f_i(d)))) \in B'(T,S)^v$ for all k.

Again, all of the $v_{i,j,k} \subseteq v_{i,j}$. Thus $\{v_{1,1,1}, \ldots, v_{n,k_n,l_{n,k_n}}\}$ is a path set and

- $h''(d)\downarrow$ iff $d \in v_{i,j,k}$ for some i, j and k; and
- $(\underline{\lambda} d \in v_{i,j,k}.\, h''(d)) = h_{i,j,k} \in B'(T,S)^v$ for all i, j and k.

It follows that $h'' \in B'(T,S)^v_S$ and thus $h' \in (A' \Rightarrow B')(T,S)^w$ as desired.

9.3.7 Partial CCC

$\mathcal{RCPO}$ is a **partial cartesian closed category** [30]. Since there are a number of conditions to check, we break the proof up into three parts: $\mathcal{RCPO}$ is a category of partial morphisms, it is partial cartesian, and it is partial cartesian closed.

A **category of partial morphisms** is a category with a notion of "total" morphisms and a "restriction relation" $\leqslant$ on morphisms. Moreover, the identities must be total, and composition must preserve totality and be monotonic with respect to $\leqslant$. In $\mathcal{RCPO}$, the total morphisms are simply the total functions. If f and f' are morphisms from A to B, then $f \leqslant f'$ iff

$$f(a)\downarrow \text{ implies } f(a) = f'(a), \text{ for all } a \in |A|.$$

It is thus easy to see that

Proposition 9.3.10 $\mathcal{RCPO}$ *is a category of partial morphisms.*

A category of partial morphisms is **cartesian** if it has an object U—called a **domain classifier**—with a map $\bigcirc_A : A \to U$ for any object A. There must also be a morphism $f \cap g : A \to U$ for every $f, g : A \to U$, and the following properties must hold:

- If $f : A \to U$, then $f \leqslant \bigcirc_A$.
- The total arrows are exactly those arrows $f : A \to B$ such that $f; \bigcirc_B = \bigcirc_A$.
- If $f, f', g : A \to B$ are such that $f, f' \leqslant g$ and $f; \bigcirc_B \leqslant f'; \bigcirc_B$, then $f \leqslant f'$.
- If $g : B \to U$ and $h, h' : A \to B$ are such that $h \leqslant h'$, then $(h; g) = (h'; g) \cap (h; \bigcirc_B)$.

Furthermore, for a category of partial morphisms to be cartesian it must have, for any pair of objects A, B, a partial product given by an object $A \otimes B$, projection arrows $\mathit{proj}_1 : A \otimes B \to A$ and $\mathit{proj}_2 : A \otimes B \to B$, and a pairing operation, associating $\langle f, g\rangle : A \to B \otimes C$ with $f : A \to B, g : A \to C$ such that the following properties hold:

- proj_1 and proj_2 are total.
- Pairing is monotonic with respect to $\leqslant$ in both arguments.

- For any $h : A \to (B \otimes C)$, $\langle h; \mathit{proj}_1, h; \mathit{proj}_2 \rangle = h$
- If $f : A \to B$ and $g : A \to C$, then $(\langle f, g\rangle; \mathit{proj}_1) \leqslant f$ and $(\langle f, g\rangle; \mathit{proj}_2) \leqslant g$.
- If $f : A \to B$ and $g : A \to C$, then for all $h : D \to A$ we have $(h; \langle f, g\rangle) = \langle (h; f), (h; g)\rangle$.
- If $f : A \to B$ and $g : A \to C$, then $(\langle f, g\rangle; \bigcirc_{B\otimes C}) = (f; \bigcirc_B) \cap (g; \bigcirc_C)$.

Proposition 9.3.11 *$\mathcal{RCPO}$ is a partial cartesian category.*

Proof: The domain classifier is the object *unit*; for every object A, we pick the morphism $\bigcirc_A : A \to \mathit{unit}$ to be the unique morphism mapping all elements of A to $\top$. Also, define

$$(f \cap f')(a) = \begin{cases} \top & \text{if } f(a)\downarrow \text{ and } f'(a)\downarrow \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The partial product is the tensor product defined previously, and the projection arrows and pairing operation the obvious ones. It is easy to see that these maps exist, and that the above properties hold.

Let $(A \to_T B)$ denote the set of total morphisms between objects A, B in $\mathcal{RCPO}$. A partial cartesian category is a **partial cartesian closed category** if there exists a binary operation $\Rightarrow$ of partial exponentiation on objects such that for any objects A, B, C there exist natural transformations *curry*(−) from $(A \otimes B) \to C$ onto $A \to_T (B \Rightarrow C)$, and *uncurry*(−) from $A \to_T (B \Rightarrow C)$ onto $(A \otimes B) \to C$, $\mathit{uncurry}(\mathit{curry}(f)) = f$ and $\mathit{curry}(\mathit{uncurry}(g)) = g$, and for any $f' : D \to A$,

$$(f'; \mathit{curry}(f)) \leqslant \mathit{curry}(\langle \mathit{proj}_1; f', \mathit{proj}_2\rangle; f).$$

Proposition 9.3.12 *$\mathcal{RCPO}$ is a partial cartesian closed category.*

Proof: The partial exponentiation is the function space $\Rightarrow$ defined previously. For $f : (A \otimes B) \to C$ and $g : A \to_T (B \Rightarrow C)$, define

$$\begin{aligned} \mathit{curry}(f) &= (\underline{\lambda} a \in |A|.\ \underline{\lambda} b \in |B|.\ f\ \langle a, b\rangle) \\ \mathit{uncurry}(g) &= (\underline{\lambda}\langle a, b\rangle \in |A \otimes B|.\ g(a)(b)). \end{aligned}$$

We first need to show that the maps are well-defined, and that they satisfy the uniformity conditions. To see that $\mathit{curry}(f)$ is a well-defined total function from $|A|$ to $|B \Rightarrow C|$, suppose $a \in |A|$ and $h \in B(T, S)^w$; we need to show $(h; \mathit{curry}(f)(a)) \in C(T, S)^w_S$.

By the concreteness condition $h' = (\underline{\lambda} d \in w'.\ a) \in A(T, S)^w$, where w' is the domain of h. Then $\langle h', h\rangle \in (A \otimes B)(T, S)^w$ and by uniformity of f we get $(\langle h', h\rangle; f) \in C(T, S)^w_S$. Since $(\langle h', h\rangle; f) = (h; \mathit{curry}(f)(a))$, we are done.

To see the uniformity of $\mathit{curry}(f)$, suppose $h_A \in A(T, S)^w$; we need to show $(h_A; \mathit{curry}(f)) \in (B \Rightarrow C)(T, S)^w_S$. Let w' be the domain of h_A. Since

$(\underline{\lambda} d \in w'.\, \mathit{curry}(f)(h_A(d))) = (h_A; \mathit{curry}(f))$ and $\{w'\}$ is a path set, it suffices to prove $(h_A; \mathit{curry}(f)) \in (B \Rightarrow C)(T,S)^w$. Now, given $\varphi : v \to w$ and $h_B \in B(T,S)^v$ we must prove $(\underline{\lambda} d \in v.\, (h_A; \mathit{curry}(f))(\varphi(d))(h_B(d))) \in C(T,S)^v_S$. By Kripke Monotonicity of $A(T,S)^w$, $h'_A = (\varphi; h_A) \in A(T,S)^v$. Let v_1 be the domain of h'_A and v_2 be the domain of h_B, and $v' = (v_1 \cap v_2)$. By downward closure,

$$\begin{aligned} h''_A &= (\underline{\lambda} d \in v'.\, h'_A(d)) \in A(T,S)^v \\ h''_B &= (\underline{\lambda} d \in v'.\, h_B(d)) \in B(T,S)^v \end{aligned}$$

Therefore $\langle h''_A, h''_B \rangle \in (A \otimes B)(T,S)^v$. By the uniformity of f,

$$\begin{aligned} (\underline{\lambda} d \in v.\, (h_A; \mathit{curry}(f))(\varphi(d))(h_B(d))) &= (\underline{\lambda} d \in v.\, \mathit{curry}(f)(h''_A(d))(h''_B(d))) \\ &= (\underline{\lambda} d \in v.\, (\langle h''_A, h''_B \rangle; f)(d)) \\ &\in C(T,S)^v_S \end{aligned}$$

as desired.

Similarly, we need to show that $\mathit{uncurry}(g)$ is a well-defined partial function from $|A \otimes B|$ to $|C|$ and that it satisfies the uniformity condition. Well-definedness follows immediately from definition. Now for uniformity, suppose $h \in (A \otimes B)(T,S)^w$; we need to show $(h; \mathit{uncurry}(g)) \in C(T,S)^w_S$. Notice that h has the form $h = \langle h_A, h_B \rangle$, where $h_A \in A(T,S)^w$ and $h_B \in B(T,S)^w$. By uniformity of g, $(h_A; g) \in (B \Rightarrow C)(T,S)^w_S$. This means that there exists a path set $\{w_1, \ldots, w_n\}$ such that

- $(h_A; g)(d)\!\downarrow$ iff $d \in w_i$ for some i; and
- $(\underline{\lambda} d \in w_i.\, (h_A; g)(d)) \in (B \Rightarrow C)(T,S)^w$ for all i.

Now, since $h_B \in B(T,S)^w$ and id_w (the identity) is a morphism in $\mathcal{C}$, we have

$$g_i = (\underline{\lambda} d \in w.\, (\underline{\lambda} e \in w_i.\, (h_A; g)(e))(d)(h_B(d))) \in C(T,S)^w_S.$$

Thus there exist path sets $\{w_{i,1}, \ldots, w_{i,k_i}\}$ such that

- $g_i(d)\!\downarrow$ iff $d \in w_{i,j}$ for some i, j; and
- $(\underline{\lambda} d \in w_{i,j}.\, g_i(d)) \in C(T,S)^w$ for all i, j.

From the observation that $w_i \supseteq w_{i,j}$, it follows that $\{w_{1,1}, \ldots, w_{n,k_n}\}$ is a path set. Furthermore, for

$$g' = (\underline{\lambda} d \in w.\, g(h_A(d))(h_B(d))),$$

we have that

- $g'(d)\!\downarrow$ iff $d \in w_{i,j}$ for some i, j; and
- $(\underline{\lambda} d \in w_{i,j}.\, g'(d)) \in C(T,S)^w$ for all i, j.

Since $g' = (h; \mathit{uncurry}(g))$, we are done.

It is not hard to prove that $\mathit{curry}(-)$ and $\mathit{uncurry}(-)$ are natural transformations, and that they form an isomorphism pair. Thus, suppose $f' : D \to A$; we need to show that

$$(f'; \mathit{curry}(f)) \leqslant \mathit{curry}(\langle \mathit{proj}_1; f', \mathit{proj}_2\rangle; f).$$

Given $d \in |D|$, if $f'(d)\!\uparrow$ then $(f'; \mathit{curry}(f))(d)$ is not defined either and there is nothing to show. So suppose $f'(d)\!\downarrow$. Then it suffices to show

$$(f'; \mathit{curry}(f))(d) = \mathit{curry}(\langle \mathit{proj}_1; f', \mathit{proj}_2\rangle; f)(d).$$

Now, if $f\langle f'(d), b\rangle\!\uparrow$, then both the left and right hand side functions above are undefined at b as well. If on the other hand $f\langle f'(d), b\rangle\!\downarrow$ then both of the above functions will equal that value on b.

9.3.8 Colimits

Coproducts and the partial cartesian closed structure give most of the structure necessary to interpret FPC types. The only part left is the interpretation of recursive types. To this end, we rework the standard colimit construction from domain theory [3, 41, 100].

The basis of the colimit construction is embedding-projection pairs. Given objects A, B, then f is an **embedding-projection pair** (ep-pair for short) if $f = \langle f^e : A \to B, f^p : B \to A\rangle$, $(f^e; f^p) = id_A$, and $(f^p; f^e) \sqsubseteq id_B$. We abuse notation and write $f : A \to B$ when f is an ep-pair from A to B, and $id : A \to A$ for the ep-pair $\langle id, id\rangle$. The composition of ep-pairs is pointwise: if $f : A \to B$ and $g : B \to C$, then $(f; g) = \langle f^e; g^e, g^p; f^p\rangle : A \to C$.

Colimits are formed from chains of objects connected by ep-pairs. Formally, an **expanding sequence** is a tuple

$$(\{D_n \mid n \geq 0\}, \{f_n \mid f_n : D_n \to D_{n+1} \text{ is an ep-pair}\}).$$

When $n \leq m$, we write $f_{mn} : D_n \to D_m$ for the ep-pair defined by induction as follows:

$$\begin{array}{rcl} f_{nn} & = & id \\ f_{(m+1)n} & = & f_{mn}; f_m. \end{array}$$

Given an expanding sequence, define D by

$$\begin{array}{rcl} |D| & = & \{g \in [\mathbf{N} \to_p \bigcup D_i] \mid g(i)\!\downarrow \text{ implies } g(i) \in D_i, \text{ and} \\ & & \qquad \text{for all } n \leq m,\, g(n) \simeq f^p_{mn}(g(m))\} \\ D(T,S)^w & = & \{h \in [w' \to_t D] \mid w' \text{ is a } T^w\text{-model, and} \\ & & \qquad \text{for all } \varphi : v \xrightarrow{\mathrm{e}} w \text{ and } n \in \mathbf{N}, \\ & & \qquad (\underline{\lambda} d \in v.\, h\,(\varphi(d))\,(n)) \in D_n(T,S)^v_S\} \end{array}$$

where ordering on $|D|$ is pointwise ordering on the functions. The ep-pairs $\langle \mu_m^e : D_m \to D, \mu_m^p : D \to D_m \rangle$, defined by

$$\begin{array}{rcl} \mu_m^p(x) & \simeq & x(m) \\ \mu_m^e(x)(l) & \simeq & \bigsqcup_{k \geq m,l} (f_{kl}^p(f_{km}^e(x))) \end{array}$$

make D into a colimit of the expanding sequence, as the next two lemmas show.

Lemma 9.3.13 *D is an object in $\mathcal{RCPO}$.*

Proof: It is easy to show that $|D|$ is directed complete. To see that $D(T,S)^w$ is a w-termination relation, we check directed completeness and leave the straightforward checks of downward closure, Kripke monotonicity, and concreteness to the reader. Suppose that $\{h_i \mid i \in I\} \subseteq D(T,S)^w$ is directed, where each h_i has domain w'. Let h be the least upper bound of the h_i's; we need to show that $h \in D(T,S)^w$. To show this, consider any $\varphi : v \to w$ and $n \in \mathbf{N}$. Then

$$(\underline{\lambda} d \in v.\, h\,(\varphi(d))\,(n)) = \bigsqcup_{i \in I} (\underline{\lambda} d \in v.\, h_i\,(\varphi(d))\,(n)) \in D_n(T,S)_S^w$$

by the directed completeness of the $D_n(T,S)_S^w$'s. Thus,

$$(\underline{\lambda} d \in v.\, h\,(\varphi(d))\,(n)) \in D_n(T,S)_S^w$$

as required.

Lemma 9.3.14 *For all m, $\mu_m^e : D_m \to D$ and $\mu_m^p : D \to D_m$ are morphisms. Moreover, $(\mu_m^p; \mu_m^e) \sqsubseteq id_D$, $(\mu_m^e; \mu_m^p) = id_{D_m}$, and*

$$\bigsqcup_{m \geq 0} \mu_m^p; \mu_m^e \quad = \quad id_D$$

Proof: The only difficult part of the proof lies in showing that μ_m^e and μ_m^p satisfy the uniformity property. To see that μ_m^p is uniform, suppose $h \in D(T,S)^w$. Then $(h; \mu_m^p) = (\underline{\lambda} d \in w.\, h\,(d)\,(m)) \in D_m(T,S)_S^w$, which is what we needed to show.

The proof of the uniformity of μ_m^e is more involved. Suppose $h \in D_m(T,S)^w$, and consider $g = (h; \mu_m^e)$. We show that $g \in D(T,S)^w$, which will show that $g \in D(T,S)_S^w$. Note that g has the same domain as h, so consider any given $\varphi : v \xrightarrow{e} w$ and $n \in \mathbf{N}$. Note that

$$(\underline{\lambda} d \in v.\, g\,(\varphi(d))\,(n)) = (\underline{\lambda} d \in v.\, \mu_m^e\,(h\,(\varphi(d)))\, n) = \bigsqcup_{k \geq m,n} (\varphi; h; f_{km}^e; f_{kn}^p)$$

By the uniformity of f_{kn}^p and f_{km}^e, we know that $(\varphi; h; f_{kn}^e; f_{km}^p) \in D_n(T,S)_S^v$. Thus, by the directed completeness of $D_n(T,S)_S^w$,

$$(\underline{\lambda} d \in v.\, g\,(\varphi(d))\,(n)) \in D_n(T,S)_S^v$$

as required.

9.4 Interpretation of FPC

The constructions in the category $\mathcal{RCPO}$ in Section 9.3 can now be used to build a model of FPC [41, 112]. A model has both a meaning for types and for terms. Types denote objects in the category, and terms denote morphisms.

The meaning of a closed type is clear except for the meanings of recursive types, which necessitates describing the meaning of an open type expression. For this reason, the meaning of a type is a *functor* from its free type variables to the category $\mathcal{RCPO}$. In making this precise, most of the difficulty lies in finding the right category of free type variables. To this end, define a **pre-ep-pair** to be a pair $f = \langle f^e, f^p \rangle$ of morphisms $f^e : D \to E$ and $f^p : E \to D$ in $\mathcal{RCPO}$; we write $f : D \hookrightarrow E$ as shorthand for saying that f is such a pre-ep-pair. Note that a pre-ep-pair is just like an ep-pair, but without the requirements that $(f^e; f^p) = id$ and $(f^p; f^e) \sqsubseteq id$. The use of pre-ep-pairs instead of ep-pairs makes certain theorems (particularly those in Section 9.9) easier to prove.

A **type environment** η is a function from type variables to objects of $\mathcal{RCPO}$, and a **type environment map** $\pi : \eta \to \eta'$ is a function from type variables to pre-ep-pairs such that $\pi(\alpha) : \eta(\alpha) \hookrightarrow \eta'(\alpha)$ for all α. The category $\mathcal{E}$ has type environments as objects, and type environment maps as morphisms. It is simple to check that $\mathcal{E}$ is a category.

The meaning of a type s, then, is a functor $[\![s]\!] : \mathcal{E} \to \mathcal{RCPO}$. The definition on objects of $\mathcal{E}$—except in the case of recursive types—is

$$
\begin{array}{rcl}
[\![\alpha]\!]\eta & = & \eta(\alpha) \\
[\![\mathtt{unit}]\!]\eta & = & unit \\
[\![s \oplus t]\!]\eta & = & ([\![s]\!]\eta \oplus [\![t]\!]\eta) \\
[\![s \otimes t]\!]\eta & = & ([\![s]\!]\eta \otimes [\![t]\!]\eta) \\
[\![s \Rightarrow t]\!]\eta & = & ([\![s]\!]\eta \Rightarrow [\![t]\!]\eta)
\end{array}
$$

The operation on morphisms of $[\![s]\!]\pi$—except for recursive types—is

$$
\begin{array}{rcl}
[\![\alpha]\!]\pi & = & \pi(\alpha) \\
[\![\mathtt{unit}]\!]\pi & = & \langle id_{unit}, id_{unit} \rangle \\
[\![s \oplus t]\!]\pi & = & ([\![s]\!]\pi \oplus [\![t]\!]\pi) \\
[\![s \otimes t]\!]\pi & = & ([\![s]\!]\pi \otimes [\![t]\!]\pi) \\
[\![s \Rightarrow t]\!]\pi & = & ([\![s]\!]\pi \Rightarrow [\![t]\!]\pi)
\end{array}
$$

where, abusing notation, $\oplus$, $\otimes$, and $\Rightarrow$ work on pre-ep-pairs as follows:

$$
\begin{array}{rcl}
([\![s]\!]\pi \oplus [\![t]\!]\pi) & = & \langle ([\![s]\!]\pi)^e \oplus ([\![t]\!]\pi)^e, ([\![s]\!]\pi)^p \oplus ([\![t]\!]\pi)^p \rangle \\
([\![s]\!]\pi \otimes [\![t]\!]\pi) & = & \langle ([\![s]\!]\pi)^e \otimes ([\![t]\!]\pi)^e, ([\![s]\!]\pi)^p \otimes ([\![t]\!]\pi)^p \rangle \\
([\![s]\!]\pi \Rightarrow [\![t]\!]\pi) & = & \langle ([\![s]\!]\pi)^p \Rightarrow ([\![t]\!]\pi)^e, ([\![s]\!]\pi)^e \Rightarrow ([\![t]\!]\pi)^p \rangle
\end{array}
$$

(Recall the actions of $\oplus$, $\otimes$, and $\Rightarrow$ on morphisms from Section 9.3.)

The case of recursive types requires more work. Suppose $s = (\texttt{rec}\ \alpha.\ t)$. Let $\langle T^i_{s,\eta}, f^i_{s,\eta} \mid i \geq 0\rangle$ be the expanding sequence given by

$$\begin{array}{rclrcl} T^0_{s,\eta} & = & \mathit{void} & f^0_{s,\eta} & = & !_{T^1_{s,\eta}} \\ T^{n+1}_{s,\eta} & = & [\![t]\!](\eta[\alpha \mapsto T^n_{s,\eta}]) & f^{n+1}_{s,\eta} & = & [\![t]\!](\mathit{id}[\alpha \mapsto f^n_{s,\eta}]) \end{array}$$

where $!_D$ is the unique ep-pair (not just pre-ep-pair) from *void* to D. This is an expanding sequence because the maps $f^n_{s,\eta}$'s are ep-pairs (by a simple induction on the definition). If D is the colimit of the expanding sequence, then define $[\![s]\!]\eta = D$. Moreover, define the morphisms $\mathit{intro}_{s,\eta} : [\![t[s/\alpha]]\!]\eta \to [\![s]\!]\eta$ and $\mathit{elim}_{s,\eta} : [\![t[s/\alpha]]\!]\eta \to [\![s]\!]\eta$ and

$$\begin{array}{rcl} \mathit{intro}_{s,\eta} & = & \bigsqcup_{n\geq 0} [\![t]\!]\mathit{id}[\alpha \mapsto \mu^n_{s,\eta}]^p ; \mu^{n+1,e}_{s,\eta} \\ \mathit{elim}_{s,\eta} & = & \bigsqcup_{n\geq 0} \mu^{n+1,p}_{s,\eta} ; [\![t]\!]\mathit{id}[\alpha \mapsto \mu^n_{s,\eta}]^e \end{array}$$

These maps will be used to give meaning to `intro` and `elim`.

For the morphism part of the functor in the case of recursive types, consider the diagram in Figure 9.1. Given $\pi : \eta_1 \to \eta_2$, consider the expanding sequences $\langle T^i_{s,\eta_1}, f^i_{s,\eta_1} \mid i \geq 0\rangle$, and $\langle T^i_{s,\eta_2}, f^i_{s,\eta_2} \mid i \geq 0\rangle$, with colimiting cocones $\langle [\![s]\!]\eta_1, \mu^i_{s,\eta_1} \mid i \geq 0\rangle$ and $\langle [\![s]\!]\eta_2, \mu^i_{s,\eta_2} \mid i \geq 0\rangle$, respectively. Define the family of pre-ep-pairs $p^n_{s,\pi}$, where $p^n_{s,\pi}$ is a pre-ep-pair from T^n_{s,η_1} to T^n_{s,η_2} by

$$\begin{array}{rcl} p^0_{s,\pi} & = & !_{\mathit{void}} \\ p^{k+1}_{s,\pi} & = & [\![t]\!](\pi[\alpha \mapsto p^k_{s,\pi}]) \end{array}$$

A simple induction argument shows that the diagram in Figure 9.1 commutes (assuming that $[\![\cdot]\!]$ is a functor, which holds by construction). From this fact, it is clear that

$$\langle [\![s]\!]\eta_2, \mu^n_{s,\eta_2} \circ p^n_{s,\pi} \mid n \geq 0\rangle$$

is a cocone for the expanding sequence $\langle T^i_{s,\eta_1}, f^i_{s,\eta_1} \mid i \geq 0\rangle$. Define $[\![s]\!]\pi$ to be the unique mediating map from $[\![s]\!]\eta_1$ to $[\![s]\!]\eta_2$, which must exist by the colimiting properties.

The meaning of terms can now be given using the combinators of the category. If $\Gamma = x_1 : t_1, ..., x_n : t_n$, define $[\![\Gamma]\!] = [\![t_1]\!] \otimes \ldots \otimes [\![t_n]\!]$. (If Γ is empty, $[\![\Gamma]\!]$ is the object *unit*). Elements of $|[\![\Gamma]\!]|$ are called **environments**. For an environment $\rho \in |[\![\Gamma]\!]|$, we write $\rho(x)$ for projection to the component corresponding to variable x. The meaning of a judgement $\Gamma \vdash M : t$ is an $\mathcal{RCPO}$-morphism

$$
\begin{tikzcd}
T^0_{s,\eta_1} \arrow[r, "f^0_{s,\eta_1}"] \arrow[d, "p^0_{s,\pi}"'] & T^1_{s,\eta_1} \arrow[r, "f^1_{s,\eta_1}"] \arrow[d, "p^1_{s,\pi}"'] & T^2_{s,\eta_1} \arrow[r, "f^2_{s,\eta_1}"] \arrow[d, "p^2_{s,\pi}"'] & \cdots & & \llbracket s \rrbracket \eta_1 \arrow[d, "\llbracket s \rrbracket \pi"] \\
T^0_{s,\eta_2} \arrow[r, "f^0_{s,\eta_2}"] & T^1_{s,\eta_2} \arrow[r, "f^1_{s,\eta_2}"] & T^2_{s,\eta_2} \arrow[r, "f^2_{s,\eta_2}"] & \cdots & & \llbracket s \rrbracket \eta_2
\end{tikzcd}
$$

Figure 9.1: Colimits and definition of $\llbracket s \rrbracket(\pi)$, where $s = (\mathtt{rec}\ \alpha.\ t)$.

$\llbracket \Gamma \vdash M : t \rrbracket : \llbracket \Gamma \rrbracket \to \llbracket t \rrbracket$. The definition is by induction on the structure of terms:

$$\llbracket \Gamma, x : t \vdash x : t \rrbracket \rho = \rho(x) \qquad\qquad \llbracket \Gamma \vdash \langle\rangle : \mathtt{unit} \rrbracket \rho = \top$$

$$\llbracket \Gamma \vdash (\lambda x : s.\ M) : (s \Rightarrow t) \rrbracket \rho = f, \quad \text{where } f(d) \simeq \llbracket \Gamma, x : s \vdash M : t \rrbracket \rho[x \mapsto d]$$

$$\llbracket \Gamma \vdash (M\ N) : t \rrbracket \rho \simeq (\llbracket \Gamma \vdash M : (s \Rightarrow t) \rrbracket \rho)\ (\llbracket \Gamma \vdash N : s \rrbracket \rho)$$

$$\llbracket \Gamma \vdash (\mathtt{intro}_{\mathtt{rec}\ \alpha.\ s}\ M) : (\mathtt{rec}\ \alpha.\ s) \rrbracket \rho \simeq \mathit{intro}(\llbracket \Gamma \vdash M : s[\mathtt{rec}\ \alpha.\ s/\alpha] \rrbracket \rho)$$

$$\llbracket \Gamma \vdash (\mathtt{elim}_{\mathtt{rec}\ \alpha.\ s}\ M) : s[\mathtt{rec}\ \alpha.\ s/\alpha] \rrbracket \rho \simeq \mathit{elim}(\llbracket \Gamma \vdash M : (\mathtt{rec}\ \alpha.\ s) \rrbracket \rho)$$

$$\llbracket \Gamma \vdash \langle M, N \rangle : (t_1 \otimes t_2) \rrbracket \rho \simeq \langle \llbracket \Gamma \vdash M : t_1 \rrbracket \rho, \llbracket \Gamma \vdash N : t_2 \rrbracket \rho \rangle$$

$$\llbracket \Gamma \vdash (\mathtt{proj}_i\ M) : t_i \rrbracket \rho \simeq \mathit{proj}_i(\llbracket \Gamma \vdash M : (t_1 \otimes t_2) \rrbracket \rho)$$

$$\llbracket \Gamma \vdash (\mathtt{inj}_i\ M) : (t_1 \oplus t_2) \rrbracket \rho \simeq \mathit{inj}_i(\llbracket \Gamma \vdash M : t_i \rrbracket \rho)$$

$$\llbracket \Gamma \vdash (\mathtt{case}\ M\ \mathtt{of}\ \mathtt{inj}_1(x).N_1\ \mathtt{or}\ \mathtt{inj}_2(x).N_2) : t \rrbracket \rho \simeq \\ \begin{cases} \llbracket \Gamma, x : s_i \vdash N_i : t \rrbracket \rho[x \mapsto d] & \text{if } \llbracket \Gamma \vdash M : (s_1 \oplus s_2) \rrbracket \rho = \mathit{inj}_i(d) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Here, the notation $\rho[x \mapsto d]$ denotes the environment in which the x component is extended (or overwritten) to d. For notational convenience, if $\emptyset \vdash M : s$, we write $\llbracket M \rrbracket$ for the corresponding element $\llbracket \emptyset \vdash M : s \rrbracket \top \in \llbracket s \rrbracket$.

The model is adequate:

Theorem 9.4.1 (Adequacy) *Suppose M is a closed FPC term of type s. Then $M \Downarrow V$ iff $\llbracket M \rrbracket \downarrow$.*

The proof appears in Section 9.8.

Corollary 9.4.2 *For closed terms M and N of type s,*

$$\textit{if } \llbracket M \rrbracket \precsim \llbracket N \rrbracket, \textit{ then } M \sqsubseteq_{FPC} N.$$

Proof: Suppose $[\![M]\!]\lesssim[\![N]\!]$. To show $M \sqsubseteq_{FPC} N$, suppose $C[\cdot]$ is a context such that $C[M] \Downarrow$. By the Adequacy Theorem 9.4.1, $[\![C[M]]\!]\downarrow$. By the hypothesis, it follows that $[\![C[N]]\!]\downarrow$ and thus $C[N] \Downarrow$.

9.5 Examples

Even though the semantic category $\mathcal{RCPO}$ is arguably complicated, it does support simple reasoning about definability of FPC terms. This section gives two examples of non-definability proofs, with non-sequential functions drawn from the literature.

9.5.1 Parallel convergence testing is not definable

Consider the partial continuous function

$$f : (\mathit{unit} \Rightarrow \mathit{unit}) \otimes (\mathit{unit} \Rightarrow \mathit{unit}) \to \mathit{unit}$$

defined

$$f\langle g, h\rangle = \begin{cases} \top & \text{if } g(\top)\downarrow \text{ or } h(\top)\downarrow \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The function f appears to need to do its calculation "in parallel." We would like to prove that f is *not* a morphism in the category, which will immediately imply that f is not definable.

The argument follows the proof of Sieber that *por* is not definable (also due to Plotkin, see [8]). In this case, we need to exhibit an index category $\mathcal{C}$, a choice of $\mathcal{C}$-termination theory T, and a $\mathcal{C}, T$-path theory S. Pick $\mathcal{C}$ to be the category with just one index set $w = \{1, 2, 3\}$ and the identity map, T^w to be the theory with just one implication $(1, 2 \vdash 3)$, and S^w to be the set $\{\,\{\,\}, \{w'\} \mid w' \text{ is a } T^w\text{-model}\,\}$. Let $h : \mathit{unit} \to \mathit{unit}$ be the identity function, and $h' : \mathit{unit} \to \mathit{unit}$ be the empty partial function. Also, let

$$g_1 = \langle h, h'\rangle, \qquad g_2 = \langle h', h\rangle, \qquad g_3 = \langle h', h'\rangle.$$

Returning to standard tuple notation for relations, we claim

$$(g_1, g_2, g_3) \in ((\mathit{unit} \Rightarrow \mathit{unit}) \otimes (\mathit{unit} \Rightarrow \mathit{unit}))(T, S)^w.$$

To prove the claim, we must show that $h_1 = (h, h', h')$ and $h_2 = (h', h, h')$ are both in $(\mathit{unit} \Rightarrow \mathit{unit})(T, S)^w$; this will show that $(g_1, g_2, g_3) = \langle h_1, h_2\rangle$ is in $((\mathit{unit} \Rightarrow \mathit{unit}) \otimes (\mathit{unit} \Rightarrow \mathit{unit}))(T, S)^w$.

We show the claim for the case of h_1 and leave the analogous case of h_2 to the reader. Pick any $u \in \mathit{unit}(T, S)^w$, and let $u' = (\underline{\lambda} d \in w.\, h_1(d)(u(d)))$. We must show $u' \in \mathit{unit}(T, S)^w_S$. By choice of T^w, the domain of u is either $\emptyset, \{1\}, \{2\}, \{3\}, \{1, 3\}, \{2, 3\}$, or $\{1, 2, 3\}$. We verify the fact for the cases $\{\}$ and $\{1, 2, 3\}$. The remaining cases are very similar and omitted here. First, consider the case of $\emptyset$. By definition, the domain of u' is $\emptyset$ as well and the existence of the path set $\{\}$ justifies $u' \in \mathit{unit}(T,S)^w_S$. For the case of $\mathit{dom}(u) = \{1,2,3\}$ we get $\mathit{dom}(u') = \{1\}$ since $h_1 = (h, h', h')$. Thus, via the path set $\{\{1\}\}$, we conclude that $u' \in \mathit{unit}(T,S)^w_S$.

However, since $f(g_1) = \top$, $f(g_2) = \top$, and $f(g_3)$ is undefined,

$$(f(g_1), f(g_2), f(g_3)) \notin \mathit{unit}(T,S)^w_S,$$

so f is not uniform. Thus, f cannot be a morphism, and hence it cannot be definable.

9.5.2 Sieber's example is not definable

The second example is also due to Sieber [121], a modified example due to Curien [29]. Let *bool* be the object $(\mathit{unit} \oplus \mathit{unit})$, and

$$A = (\mathit{unit} \Rightarrow \mathit{bool}) \otimes (\mathit{unit} \Rightarrow \mathit{bool}).$$

Let *true* denote $\mathit{inj}_1(\top) \in \mathit{bool}$ and *false* denote $\mathit{inj}_2(\top) \in \mathit{bool}$. Consider the morphisms $g_1, g_2, g_3, g_4 : A \to \mathit{unit}$ defined by

$$\begin{array}{rcl}
g_1\langle h_1, h_2\rangle & \simeq & \begin{cases} \top & \text{if } h_2(\top) = \mathit{true} \\ \top & \text{if } h_1(\top) = \mathit{true} \text{ and} \\ & \quad h_2(\top) = \mathit{false} \\ \text{undefined} & \text{otherwise} \end{cases} \\
g_2\langle h_1, h_2\rangle & \simeq & \begin{cases} \top & \text{if } h_1(\top) = \mathit{false} \\ \text{undefined} & \text{otherwise} \end{cases} \\
g_3\langle h_1, h_2\rangle & \simeq & \begin{cases} \top & \text{if } h_2(\top) = \mathit{false} \\ \text{undefined} & \text{otherwise} \end{cases} \\
g_4\langle h_1, h_2\rangle & \simeq & \text{undefined}
\end{array}$$

We claim that any $f : (A \Rightarrow \mathit{unit}) \to \mathit{unit}$ satisfying

$$f(g_1) = \top, \qquad f(g_2) = \top, \qquad f(g_3) = \top, \qquad f(g_4)\uparrow$$

is not definable in FPC.

The proof of nondefinability requires a nontrivial use of path sets. Pick $\mathcal{C}$ to be the trivial index category $\mathcal{C}$ with object $w = \{1,2,3,4\}$. Let T^w be the termination theory T^w with just one implication $(1,2,3 \vdash 4)$. Let S be the set of path sets $\{w_1, \ldots, w_n\}$ such that

- If $1 \in w_i$ and $2 \in w_j$, then $i = j$; and
- If $1 \in w_i$ and $3 \in w_j$, then $i = j$.

It is not hard to prove that this is a $\mathcal{C}, T$-path theory, and that

$$(g_1, g_2, g_3, g_4) \in (A \Rightarrow \mathit{unit})(T,S)^w.$$

However, it is evident that

$$(f(g_1), f(g_2), f(g_3), f(g_4)) \notin \mathit{unit}(T,S)^w_S$$

because of the conditions on the path sets. Thus, there is no such definable f.

9.6 Full Abstraction

We prove full abstraction for FPC in two steps. Let Finite FPC be the sublanguage of FPC with no recursive types. First, we reduce the full abstraction problem to the problem of showing that all elements in Finite FPC types are definable. The reduction uses techniques from [112], and we give the basic outline of the proof. Second, we prove that all elements in Finite FPC types are definable by terms. The proof follows the structure of the proof of full abstraction for PCF [94].

9.6.1 Reduction to definability for Finite FPC

Define the unwinding of all type recursions to depth n by

$$\begin{array}{rclcrcl} \alpha^n & = & \alpha & \quad & (s \oplus t)^n & = & (s^n \oplus t^n) \\ \mathtt{void}^n & = & \mathtt{void} & & (s \otimes t)^n & = & (s^n \otimes t^n) \\ \mathtt{unit}^n & = & \mathtt{unit} & & (s \to t)^n & = & (s^n \to t^n) \end{array}$$

$$(\mathtt{rec}\ \alpha.\ t)^n = \underbrace{t^n[t^n[\ldots[t^n}_{n}[\mathtt{void}/\alpha]/\alpha]\ldots]/\alpha]/\alpha$$

In Section 9.9, we construct terms $\mathtt{clm}^p_{s,n} : s \to s^n$ and $\mathtt{clm}^e_{s,n} : s^n \to s$ that, intuitively, denote a sequence of "colimiting" maps. The terms satisfy the following property:

Lemma 9.6.1 $\bigsqcup_{n \geq 0}([\![\mathtt{clm}^p_{s,n}]\!]; [\![\mathtt{clm}^e_{s,n}]\!]) = id_{[\![s]\!]}$.

The construction uses the techniques in [112], although we repeat the salient details in Section 9.9 for completeness. These terms allow us to reduce the problem to definability for Finite FPC.

Theorem 9.6.2 *Suppose for all Finite FPC types s, all elements of $[\![s]\!]$ are definable by closed terms. Let M, N be closed FPC terms of type s. If $[\![M]\!] \precnapprox [\![N]\!]$, then $M \not\sqsubseteq_{FPC} N$.*

Proof: We claim that it is enough to prove the theorem for all Finite FPC types s. To see this, suppose s is a general FPC type. We know by Lemma 9.6.1 that $[\![\mathtt{clm}^p_{s,n}\ M]\!] \precnapprox [\![\mathtt{clm}^p_{s,n}\ N]\!]$ for some n. Both of the terms $M' = (\mathtt{clm}^p_{s,n}\ M)$ and $N' = (\mathtt{clm}^p_{s,n}\ N)$ have a Finite FPC type s. Thus, if we can find a context $C[\cdot]$ distinguishing M' from N', then the context $C[\mathtt{clm}^p_{s,n}\ [\cdot]]$ distinguishes M and N.

We now prove the theorem for all Finite FPC types s by induction on s, giving two of the five cases and leaving the others to the reader. In the first case, suppose $s = (t \oplus t')$. There are three subcases:

- $[\![M]\!]\downarrow$ and $[\![N]\!]\uparrow$. Then, by Adequacy, the context $[\cdot]$ distinguishes M and N.

- $[\![M]\!] = inj_i(d)$ and $[\![N]\!] = inj_j(e)$, where $i \neq j$. By Adequacy, we get that $M \Downarrow (\mathtt{inj}_i\ V)$ and $N \Downarrow (\mathtt{inj}_j\ V')$. Suppose $i = 1$ (the other case is analogous); then the context $(\mathtt{case}\ [\cdot]\ \mathtt{of}\ \mathtt{inj}_1(x).x\ \mathtt{or}\ \mathtt{inj}_2(x).\Omega)$ distinguishes M and N.

- $[\![M]\!] = inj_i(d)$ and $[\![N]\!] = inj_i(e)$. By Adequacy, $M \Downarrow (\mathtt{inj}_i\ V)$ and $N \Downarrow (\mathtt{inj}_i\ V')$. Note that $V \not\sqsubseteq_{FPC} V'$. Suppose $i = 1$ (the other case is analogous), and the context distinguishing V, V' is $C[\cdot]$. Then the context $(\mathtt{case}\ [\cdot]\ \mathtt{of}\ \mathtt{inj}_1(x).C[x]\ \mathtt{or}\ \mathtt{inj}_2(x).\Omega)$ distinguishes M and N.

In the second case, suppose $s = (t \to t')$. If $[\![M]\!]\downarrow$ and $[\![N]\!]\uparrow$, then the context $[\cdot]$ distinguishes M and N. If both are defined, on the other hand, by Adequacy, $M \Downarrow (\lambda x : t.\, M')$ and $N \Downarrow (\lambda x : t.\, N')$. Since $[\![M]\!] \not\sqsubseteq [\![N]\!]$, there must be an argument $d \in [\![t]\!]$ such that $[\![M]\!](d) \lnsim [\![N]\!](d)$. By hypothesis, there is a closed Finite FPC term P such that $d = [\![P]\!]$. Thus, $[\![M\ P]\!] \lnsim [\![N\ P]\!]$. By induction, there is a context $C[\cdot]$ distinguishing $(M\ P)$ and $(N\ P)$. Thus, the context $C[[\cdot]\ P]$ distinguishes M and N.

9.6.2 Definability of elements of Finite FPC types

To prove that all elements of Finite FPC type are representable by terms, we will consider a particular index category $\mathcal{C}$, particular $\mathcal{C}$-termination theory T, and particular $\mathcal{C}, T$-path theory S. Define the index category to be the set of sets of the form

$$[s_1, \ldots, s_n] = |[\![s_1]\!] \otimes \ldots \otimes [\![s_n]\!]|.$$

where $[\,] = unit$. Morphisms are the projections $[s_1, \ldots, s_{n+k}] \to [s_1, \ldots, s_n]$. For any object $w = [s_1, \ldots, s_n]$ of $\mathcal{C}$, let $s = (s_1 \otimes \ldots \otimes s_n)$ and

$$X^w = \{\, w' \subseteq w \mid \text{there is a closed } M : (s \Rightarrow \mathtt{unit})\ .\ [\![M]\!](d)\downarrow \text{ iff } d \in w' \,\}.$$

The path sets in S^w are defined as follows. $\{w_1, \ldots, w_k\}$ is a path set if:

- $w_1, \ldots, w_k \in X^w$; and
- there is an $M : (s \Rightarrow \bar{k})$ such that $[\![M]\!](d) = \mathbf{i}$ iff $d \in w_i$, where

  $$\bar{n} = \underbrace{(\mathtt{unit} \oplus \ldots \oplus \mathtt{unit})}_{n}$$

  and $\mathbf{1} = inj_1(\top)$, $\mathbf{2} = inj_2(inj_1(\top))$, and so on.

In the following, we use if-then-else and other notation in terms when working with the types $\bar{n}$; these are just shorthand for Finite FPC terms.

Using the alternative characterization of T-models given by Proposition 9.3.1, it is not hard to establish the following

Lemma 9.6.3 *The set $X = \{\, X^w \mid w \in Obj(\mathcal{C}) \,\}$ defines a $\mathcal{C}$-termination theory T by taking X^w to be a set of models.*

Proof: Let $\varphi : v \xrightarrow{\mathcal{C}} w$ and

$$\begin{array}{rclcrcl} w & = & [s_1, \ldots, s_n] & \quad & s & = & (s_1 \otimes \ldots \otimes s_n) \\ v & = & [s_1, \ldots, s_n, \ldots, s_{n+k}] & & s' & = & (s_1 \otimes \ldots \otimes s_n \otimes \ldots \otimes s_{n+k}) \end{array}$$

First, we establish that each $X^w \in X$ is a closure system; by Proposition 9.3.1, it will then be clear that each T^w is a w-theory. Note that $w \in X^w$ (the requisite term is $(\lambda x : s.\, \langle\rangle)$ of type $(s \Rightarrow \mathtt{unit})$). So suppose $w_1, w_2 \in X^w$. Then there exist M_1, M_2 with $[\![M_i]\!](d)\!\downarrow$ iff $d \in w_i$. Consider the term

$$M = (\lambda x : s.\, (M_1\ x); (M_2\ x)).$$

Then it is clear that $[\![M]\!](d)\!\downarrow$ iff $d \in w_1 \cap w_2$. Thus, $(w_1 \cap w_2) \in X^w$.

Second, suppose w' is a T^w-model. Then by definition, there is a term $N : (s \Rightarrow \mathtt{unit})$ such that $[\![N]\!](d)\!\downarrow$ iff $d \in w'$. Let $v' = \varphi^{-1}(w')$, and

$$M = (\lambda x : s'.\, N\ \langle x_1, \ldots, x_n \rangle)$$

where x_i is shorthand for the i^{th} projection of the term x. Then $[\![M]\!](d)\downarrow$ iff $d \in v'$, so v' is a T^v-model as required. We conclude that T is indeed a $\mathcal{C}$-termination theory.

Lemma 9.6.4 *S is a $\mathcal{C}, T$-path theory.*

Proof: We need to show S^w satisfies four properties. Suppose w' is a T^w-model and $X = \{w_1, \ldots, w_k\}$ and $X' = \{w'_1, \ldots, w'_l\}$ are path sets in S^w. Then there exists a term M defining w', and terms N and P defining these path sets.

- The term $(\lambda x : s.\, \Omega)$ defines the path set $\{\}$.
- The term $(\lambda x : s.\, (M\ x); \mathbf{1})$ defines the path set $\{w'\}$.
- If $X \sqcap X' = \{w_{i_1} \cap w_{j_1}, \ldots, w_{i_n} \cap w_{j_n}\}$, then the term

  $$\begin{array}{l} (\lambda x : s.\, \mathtt{if}\ ((N\ x) = i_1) \wedge ((P\ x) = j_1)\ \mathtt{then}\ \mathbf{1}\ \mathtt{else}\ \ldots \\ \qquad\qquad \mathtt{if}\ ((N\ x) = i_n) \wedge ((P\ x) = j_n)\ \mathtt{then}\ (i_n * j_n)\ \mathtt{else}\ \Omega) \end{array}$$

  defines the path set $X \sqcap X'$.
- Suppose there is a j such that for all i, $w'_i \subseteq w_j$. Then the term

  $$\begin{array}{l} (\lambda x : s.\, \mathtt{if}\ (N\ x) < j\ \mathtt{then}\ (N\ x)\ \mathtt{else} \\ \qquad\qquad \mathtt{if}\ (N\ x) > j\ \mathtt{then} (N\ x) + l - 1\ \mathtt{else}\ (P\ x) + j - 1) \end{array}$$

  defines the path set $\{w_1, \ldots, w_{j-1}, w'_1, \ldots, w'_l, w_{j+1}, \ldots, w_k\}$.

This completes the proof.

The next lemma is the main one needed for full abstraction. It shows that every element of the computational relations is represented by a term in Finite FPC.

Lemma 9.6.5 *Suppose $w = [s_1, \ldots, s_n]$ and $s = (s_1 \otimes \ldots \otimes s_n)$. Then $g \in [\![t]\!](T,S)^w_S$ iff there exists a closed, Finite FPC term $M : (s \Rightarrow t)$ such that $g = [\![M]\!]$.*

Proof: By induction on t.

- $t = \texttt{void}$. If $g \in [\![t]\!](T,S)^w_S$, then the term $M = (\lambda x : s.\, \Omega)$ defines g. The converse is also trivial.

- $t = \texttt{unit}$. For ($\Leftarrow$), suppose $M : (s \Rightarrow \texttt{unit})$. By definition, M defines a T^w-model w' via

$$[\![M]\!](d)\!\downarrow \text{ iff } d \in w'.$$

  Thus, $[\![M]\!] \in [\![\texttt{unit}]\!](T,S)^w_S$. For ($\Rightarrow$), suppose $g \in [\![\texttt{unit}]\!](T,S)^w_S$. Thus, there exists a path set $\{w_1, \ldots, w_k\}$ such that

  - $g(d)\!\downarrow$ iff $d \in w_i$ for some i; and
  - $(\underline{\lambda} d \in w_i.\, g(d)) \in [\![\texttt{unit}]\!](T,S)^w$.

  (Of course, in this particular case of `unit`, the second clause is redundant.) By the definition of path sets, there is a $P : (s \Rightarrow \bar{k})$ such that $[\![P]\!](d) = \mathbf{i}$ iff $d \in w_i$, and undefined otherwise. Let

$$M = (\lambda x : s.\, (P\ x); \langle\rangle).$$

  It is not hard to see that $[\![M]\!] = g$ as needed.

- $t = (t_1 \oplus t_2)$. For ($\Leftarrow$), suppose $M : (s \Rightarrow (t_1 \oplus t_2))$. Consider the following two terms:

$$\begin{array}{rcl} N_1 & = & (\lambda x : s.\, \texttt{case}\ (M\ x)\ \texttt{of}\ \texttt{inj}_1(y).y\ \texttt{or}\ \texttt{inj}_2(y).\Omega) : (s \Rightarrow t_1) \\ N_2 & = & (\lambda x : s.\, \texttt{case}\ (M\ x)\ \texttt{of}\ \texttt{inj}_1(y).\Omega\ \texttt{or}\ \texttt{inj}_2(y).y) : (s \Rightarrow t_2) \end{array}$$

  By induction ($\Leftarrow$), $[\![N_i]\!] \in [\![t_i]\!](T,S)^w_S$. Thus, we have for some path sets $X = \{w_1, \ldots, w_k\}$ and $X' = \{w'_1, \ldots, w'_l\}$,

  - $[\![N_1]\!](d)\!\downarrow$ iff $d \in w_i$ for some i;
  - $(\underline{\lambda} d \in w_i.\, [\![N_1]\!](d)) \in [\![t_1]\!](T,S)^w$ for all i;
  - $[\![N_2]\!](d)\!\downarrow$ iff $d \in w'_i$ for some i; and
  - $(\underline{\lambda} d \in w'_i.\, [\![N_2]\!](d)) \in [\![t_2]\!](T,S)^w$ for all i.

  Note that for any i, j, w_i and w'_j are disjoint: if $d \in w_i$, then $[\![N_1]\!](d)\!\downarrow$, so $[\![M]\!](d) = inj_1(a)$, so $[\![N_2]\!](d)\!\uparrow$, so $d \notin w'_j$. Note also that the set $\{w_1, \ldots, w_k, w'_1, \ldots, w'_l\}$ is a legal path set via the term

$$(\lambda x : s.\, \texttt{case}\ (M\ x)\ \texttt{of}\ \texttt{inj}_1(y).(P_1\ x)\ \texttt{or}\ \texttt{inj}_2(y).(P_2\ x) + k)$$

  if P_1 is the term for X and P_2 is the term for X'. Finally, note that

– $[\![M]\!](d)\downarrow$ iff $d \in w_i$ or w'_i for some i; and
– For the first set of T^w-models,

$$(\underline{\lambda} d \in w_i.\, [\![M]\!](d)) = ((\underline{\lambda} d \in w_i.\, [\![N_1]\!](d)); inj_1) \in [\![t_1 \oplus t_2]\!](T,S)^w$$

and for the second set,

$$(\underline{\lambda} d \in w'_i.\, [\![M]\!](d)) = ((\underline{\lambda} d \in w'_i.\, [\![N_2]\!](d)); inj_2) \in [\![t_1 \oplus t_2]\!](T,S)^w.$$

Therefore, $[\![M]\!] \in [\![t]\!](T,S)^w_S$.

For ($\Rightarrow$), suppose $g \in [\![t_1 \oplus t_2]\!](T,S)^w_S$. By definition, there exists a path set $\{w_1, \ldots, w_k\}$ such that

– $g(d)\downarrow$ iff $d \in w_i$ for some i; and
– $g_i = (\underline{\lambda} d \in w_i.\, g(d)) \in [\![t_1 \oplus t_2]\!](T,S)^w$ for all i.

Then by definition, $g_i = (h_i; inj_{j_i})$ for some $h_i \in [\![t_{j_i}]\!](T,S)^w$. Since $\{w_i\}$ itself is a path set we get that $h_i \in [\![t_{j_i}]\!](T,S)^w_S$, and thus by induction ($\Rightarrow$), there exists an $N_i : (s \Rightarrow t_{j_i})$ such that

$$d \in w_i \text{ iff } [\![N_i]\!](d)\downarrow, \text{ and } h_i = (\underline{\lambda} d \in w_i.\, [\![N_i]\!](d)).$$

Since $\{w_1, \ldots, w_k\}$ is a path set, there is a term $P : s \Rightarrow \bar{k}$ defining that path set. Let $M_i = (\lambda x : s.\, \mathtt{inj}_{j_i}\ (N_i\ x))$ and

$$M = (\lambda x : s.\, \mathtt{if}\ (P\ x) = 1\ \mathtt{then}\ (M_1\ x)\ \mathtt{else}\ \ldots$$
$$\mathtt{if}\ (P\ x) = k\ \mathtt{then}\ (M_k\ x)\ \mathtt{else}\ \Omega).$$

Then $[\![M]\!] = g$.

- $t = (t_1 \otimes t_2)$. For ($\Leftarrow$), suppose $M : (s \Rightarrow (t_1 \otimes t_2))$. Consider the following two terms:

$$N_i \quad = \quad (\lambda x : s.\, \mathtt{proj}_i\ (M\ x)) : (s \Rightarrow t_i).$$

By induction ($\Leftarrow$), $[\![N_i]\!] \in [\![t_i]\!](T,S)^w_S$. That is, there exists some path sets $X = \{w_1, \ldots, w_k\}$ and $X' = \{w'_1, \ldots, w'_l\}$, such that

– $[\![N_1]\!](d)\downarrow$ iff $d \in w_i$ for some i;
– $(\underline{\lambda} d \in w_i.\, [\![N_1]\!](d)) \in [\![t_1]\!](T,S)^w$ for all i;
– $[\![N_2]\!](d)\downarrow$ iff $d \in w'_i$ for some i; and
– $(\underline{\lambda} d \in w'_i.\, [\![N_2]\!](d)) \in [\![t_2]\!](T,S)^w$ for all i.

Note that $X \sqcap X' = \{w''_1, \ldots, w''_m\}$ is a path set, and note that

$$[\![M]\!](d)\downarrow \text{ iff } d \in w''_i \text{ for some } i; \text{ and}$$

$$(\underline{\lambda} d \in w''_i.\, [\![M]\!](d)) = \langle (\underline{\lambda} d \in w''_i.\, [\![N_1]\!](d)), (\underline{\lambda} d \in w''_i.\, [\![N_2]\!](d)) \rangle$$
$$\in [\![t_1 \otimes t_2]\!](T,S)^w.$$

Therefore, $[\![M]\!] \in [\![t]\!](T,S)^w_S$.

For the converse, suppose $g \in [\![t_1 \otimes t_2]\!](T,S)^w_S$. Then there exists a path set $\{w_1,\ldots,w_k\}$ such that

- $g(d)\downarrow$ iff $d \in w_i$ for some i; and
- $g_i = (\underline{\lambda} d \in w_i.\, g(d)) \in [\![t_1 \otimes t_2]\!](T,S)^w$.

By definition, $g_i = \langle h_{i,1}, h_{i,2}\rangle$ for some $h_{i,j} \in [\![t_j]\!](T,S)^w$. By induction ($\Rightarrow$), there exist $N_{i,j} : (s \Rightarrow t_j)$ such that

$$d \in w_i \text{ iff } [\![N_{i,j}]\!](d)\downarrow, \text{ and } h_{i,j} = (\underline{\lambda} d \in w_i.\, [\![N_{i,j}]\!](d)).$$

Since we have a path set, there is a term $P : s \Rightarrow \bar{k}$ defining that path set. Let $M_i = (\lambda x : s.\, \langle N_{i,1}\; x, N_{i,2}\; x\rangle)$ and

$$M = (\lambda x : s.\, \texttt{if}\ (P\ x) = 1\ \texttt{then}\ (M_1\ x)\ \texttt{else}\ \ldots \\ \texttt{if}\ (P\ x) = k\ \texttt{then}\ (M_k\ x)\ \texttt{else}\ \Omega).$$

Then $[\![M]\!] = g$.

- $t = (t_1 \Rightarrow t_2)$. For ($\Leftarrow$), suppose $M : (s \Rightarrow (t_1 \Rightarrow t_2))$. There are two cases: either $[\![M]\!]$ is undefined on every argument, or $[\![M]\!]$ is defined on at least one argument. In the first case, since $\{\}$ is a path set in S^w, it is not hard to see that $[\![M]\!] \in [\![t_1 \Rightarrow t_2]\!](T,S)^w_S$. In the second case, let

$$M' = (\lambda x : s.\, (M\ x); \mathbf{1}).$$

By the definition of S^w, M' defines a path set consisting of one element; call it $\{w'\}$. We use this as our path set.

1. Obviously, $[\![M]\!](d)\downarrow$ iff $d \in w'$.
2. To see $(\underline{\lambda} d \in w'.\, [\![M]\!](d)) \in [\![t_1 \Rightarrow t_2]\!](T,S)^w$, suppose $\varphi : v \xrightarrow{\mathrm{e}} w$ and $g \in [\![t_1]\!](T,S)^v$, where $v = [s_1,\ldots,s_{n+k}]$. Furthermore, we let $s' = (s_1 \otimes \ldots \otimes s_{n+k})$. Then, by induction ($\Rightarrow$), there is a closed $N : (s' \Rightarrow t_1)$ such that

$$d \in v' \text{ iff } [\![N]\!](d)\downarrow, \text{ and } g = (\underline{\lambda} d \in v'.\, [\![N]\!](d))$$

where v' is the domain of g. Consider the term

$$P = (\lambda x : s'.\, (M\ \langle x_1,\ldots,x_n\rangle)\ (N\ x)) : (s' \Rightarrow t_2)$$

By induction ($\Leftarrow$), $[\![P]\!] \in [\![t_2]\!](T,S)^v_S$. But note that

$$[\![P]\!](d) \simeq (\underline{\lambda} d \in w'.\, [\![M]\!](d))\ (\varphi(d))\ (g(d)).$$

Thus,

$$(\underline{\lambda} d \in v.\, (\underline{\lambda} d \in w'.\, [\![M]\!](d))\ (\varphi(d))\ (g(d))) \in [\![t_2]\!](T,S)^v_S$$

as required.

For the converse, suppose $g \in [\![t_1 \Rightarrow t_2]\!](T,S)^w_S$. Then there exists a path set $\{w_1, \ldots, w_k\}$ such that

- $g(d)\downarrow$ iff $d \in w_i$ for some i; and
- $g_i = (\underline{\lambda} d \in w_i.\, g(d)) \in [\![t_1 \Rightarrow t_2]\!](T,S)^w$.

Let $v = [s_1, \ldots, s_n, t_1]$, $\varphi : v \to w$, and $s' = (s_1 \otimes \ldots \otimes s_n \otimes t_1)$, and $N = (\lambda x : s'.\, x_{n+1})$. By induction ($\Leftarrow$), $[\![N]\!] \in [\![t_1]\!](T,S)^v_S$. But note that the T^v-model that makes $[\![N]\!]$ converge is all of v. Thus, there is a path set $\{v_1, \ldots, v_l\} \in S^v$ with $(v_1 \cup \ldots \cup v_l) = v$ and

$$h_j = (\underline{\lambda} d \in v_j.\, [\![N]\!](d)) \in [\![t_1]\!](T,S)^v$$

for all $j \in \{1, \ldots, l\}$. Therefore,

$$h_{i,j} = (\underline{\lambda} d \in v.\, g_i\ (\varphi(d))\ (h_j(d))) \in [\![t_2]\!](T,S)^v_S.$$

By induction ($\Rightarrow$), there exist terms $P_{i,j} : (s' \Rightarrow t_2)$ such that $h_{i,j} = [\![P_{i,j}]\!]$. But note that

$$[\![P_{i,j}]\!]\langle d_1, \ldots, d_n, d\rangle \quad \simeq \quad (g_i\ \langle d_1, \ldots, d_n\rangle\ d).$$

Let Q be the term that defines the path set $\{w_1, \ldots, w_k\}$ and Q_j be the term that defines each v_j. Define

$$\begin{aligned} Q'_i = (\lambda x : s.\, \lambda y : t_1.\, &\texttt{if}\ Q_1\ \langle x,y\rangle\ \texttt{then}\ P_{i,1}\ \langle x,y\rangle\ \texttt{else}\ \ldots \\ &\texttt{if}\ Q_l\ \langle x,y\rangle\ \texttt{then}\ P_{i,l}\ \langle x,y\rangle\ \texttt{else}\ \Omega) \end{aligned}$$

and let

$$\begin{aligned} M = (\lambda x : s.\, &\texttt{if}\ (Q\ x) = 1\ \texttt{then}\ (Q'_1\ x)\ \texttt{else}\ \ldots \\ &\texttt{if}\ (Q\ x) = k\ \texttt{then}\ (Q'_k\ x)\ \texttt{else}\ \Omega) \end{aligned}$$

Then it is not hard to see that $[\![M]\!] = g$.

This completes the induction and hence the proof.

Corollary 9.6.6 *All elements of closed Finite FPC types are definable.*

Proof: Suppose s is a closed Finite FPC type, and $g \in [\![s]\!]$. By concreteness,

$$h = (\underline{\lambda} d \in [\,].\, g) \in [\![s]\!](T,S)^{[\,]}_S.$$

By Lemma 9.6.5, $h = [\![N]\!]$ for some closed term N of type ($\texttt{unit} \to s$). Then $g = [\![N\ \langle\rangle]\!]$.

9.6.3 Putting it together

We may now prove the main theorem of the paper.

Theorem 9.6.7 (Full Abstraction) *Suppose M, N are closed terms of type s. Then $[\![M]\!] \lessapprox [\![N]\!]$ iff $M \sqsubseteq_{FPC} N$.*

Proof: ($\Leftarrow$) is immediate from Theorem 9.6.2 and Corollary 9.6.6, and ($\Rightarrow$) is immediate from Corollary 9.4.2.

9.7 Discussion

We have shown in this paper how to construct a fully abstract model for call-by-value FPC using logical relations, generalizing the approach taken for PCF [94, 121]. The model supports a simple form of reasoning for showing that certain values are not definable.

There are other ways to build fully abstract models for FPC. For instance, Riecke and Viswanathan give a dcpo-based model for call-by-value FPC [113, 114], using the syntactic methods of Milner [85]. This construction sheds little light into the structure of FPC, except that the model validates least fixpoint reasoning. Games semantics has also been applied to full abstraction for FPC by McCusker, who builds a model of *call-by-name* FPC [82, 81]. The sums in this model are *separated*: applying the injection operations to the meaning of a divergent computation returns a convergent value (on which a case expression can branch). Game semantics has been recently adapted to the call-by-value setting [2, 46]. These papers devise models by loosening the restrictions of the original games semantics [4, 52, 90] to include strategies that start with the opponent's *answer* rather than a *question.* Intuitively, this means that the value supplied to a call-by-value function is immediately available without interrogation by the player. The basic definitions are quite different from our logical-relations-based model.

Much of the complexity of our model of FPC lies in the use of Kripke relations. On the one hand, since all examples of reasoning in the model seem to require only the "base" relations, it would be interesting to determine when base relations were sufficient. This kind of result might be analogous to Sieber's result that sequentiality relations suffice for proving facts about PCF up to third-order types [121]. On the other hand, recent results of Ralph Loader suggest that one must go beyond base relations to achieve full abstraction. We conjecture that the following problem is undecidable: given a type in Finite FPC, can one decide how many elements there are in the model of that type? If we remained only with the "base" relations, the problem would be *decidable.* The related decision problem for PCF was first pointed out by Jung and Stoughton [57]; see [59, 94] for a further discussion. Loader shows that the decision problem for PCF over the single boolean base type is undecidable [74]. We expect that the proof will carry over to Finite FPC.

The model presented here has some ad hoc features that should be examined more closely. The construction of path sets, for instance, might be better expressed in terms of Grothendieck topologies; recent work by Fiore and Simpson [35] may be useful. Other recent work by Marz has defined a more general category of "sequential domains" that is suitable for call-by-name [77]. Call-by-value can be simulated in Marz's framework through a lifting operation. Marz's conditions on path sets are perhaps easier to understand than our definitions, and also do not rely on the intermediate step of termination theories. Such a simplification here would make the model easier to construct. It would be useful, among all of these definitions, to settle on a single, well-motivated definition.

We have some hope that the relational account can be adapted to extensions of FPC with other kinds of effects other than simple functional branching, such as continuation-based control operations. We also believe that there is a relationship between "single-threading" of state [93, 95] and sequentiality; it would be interesting to see if our model can be adapted to model a single-threaded global state. One strength of the current model is its clean separation of values and computations. We conjecture that only the definition of "path theories" must change to reflect the new settings.

Other extensions seem more difficult. For instance, we began by trying to find a similar relation-based model for a *linear* type system, but ran into technical difficulties. The extension of FPC with a notion of *local* state, as in Idealized Algol [107] or Standard ML [84], also seems to be difficult. One interesting, though non-trivial, direction would be to extend the language with parametric polymorphism. A different kind of relations would be needed in this instance to model parametricity.

9.8 Proof of Adequacy

To establish the adequacy of the model, we follow the proof techniques of [83, 100, 112]. There are two main steps. First, we define a notion of an "adequacy relation" between elements of the model and terms, and show that the meanings of terms and the terms themselves are related by any adequacy relation. Second, we construct a particular adequacy relation. The properties of the adequacy relation allow us to conclude that the model is adequate. Since the proof is not novel, we omit many of the details that are easy to check.

9.8.1 Adequacy relations

Definition 9.8.1 An **adequacy relation** is a family of relations $\leq_s$, indexed by closed types, between elements of $[\![s]\!]$ and closed terms of type s. The following conditions must also hold.

1. $d \leq_{\texttt{void}} M$ holds vacuously (since there is no $d \in \mathit{void}$).
2. $\top \leq_{\texttt{unit}} M$ iff $M \Downarrow \langle\rangle$.
3. $(\mathit{inj}_i\ e) \leq_{s_1 \oplus s_2} M$ iff $M \Downarrow (\texttt{inj}_i\ V)$ and $(e \leq_{s_i} V)$.
4. $\langle e_1, e_2\rangle \leq_{s_1 \otimes s_2} M$ iff $M \Downarrow \langle V_1, V_2\rangle$ and $(e_i \leq_{s_i} V_i)$.

5. $f \leq_{s_1 \Rightarrow s_2} M$ iff $M \Downarrow (\lambda x : s_1 .\, N)$, and $e \leq_{s_1} P$ implies $f(e) \lesssim_{s_2} (M\ P)$.

6. $e \leq_{\texttt{rec}\ \alpha.\ s} M$ iff $M \Downarrow (\texttt{intro}\ V)$ and $\mathit{elim}(e) \leq_{s[\texttt{rec}\ \alpha.\ s/\alpha]} V$.

In the clause for function types, $\mathit{exp} \lesssim_s M$ means either exp is undefined, or it is defined and $\mathit{exp} \leq_s M$.

Lemma 9.8.2 *Suppose $\leq$ is an adequacy relation, $d \leq_s M$, $M \Downarrow V$, and $N \Downarrow V$. Then $d \leq_s N$.*

Proof: By a simple case analysis on s.

Lemma 9.8.3 *Suppose $\leq$ is an adequacy relation, $e_i \leq_{s_i} M_i$, $\rho = \langle e_1, \ldots, e_n \rangle$, $\Gamma = (x_1 : s_1, \ldots, x_n : s_n)$, and $\Gamma \vdash M : s$. Then*

$$[\![\Gamma \vdash M : s]\!]\rho \lesssim_s M[M_1, \ldots, M_n/x_1, \ldots, x_n].$$

Proof: By induction on the derivation of $\Gamma \vdash M : s$. We give a few of the cases and leave the others to the reader.

1. $\Gamma \vdash x_i : s_i$. Then $[\![\Gamma \vdash x_i : s_i]\!]\rho = \rho(x_i) \leq_{s_i} M_i$ by the hypothesis.

2. $\Gamma \vdash (N\ P) : t$ where $\Gamma \vdash N : (s \to t)$ and $P : s$. Follows easily from the induction hypothesis.

3. $\Gamma \vdash (\lambda x : s .\, N) : (s \to t)$ where $\Gamma, x : s \vdash N : t$. Suppose $e \leq_s P$, and let

$$M' = ((\lambda x : s .\, N)\ P)[M_1, \ldots, M_n/x_1, \ldots, x_n]$$

   By the induction hypothesis,

$$d' = [\![\Gamma, x : s \vdash N : t]\!]\rho[x \mapsto e] \lesssim_t N[M_1, \ldots, M_n, P/x_1, \ldots, x_n, x] = N'.$$

   Thus, either d' is not defined, or it is defined and $d' \leq_t N'$. In the first case, $d' \lesssim_t M'$. In the second case, $N' \Downarrow V$ for some V. Note then that $M' \Downarrow V$. By Lemma 9.8.2, $d' \leq_t M'$. Thus, by the property of adequacy relations,

$$[\![\Gamma \vdash (\lambda x : s .\, N) : (s \to t)]\!]\rho \lesssim_{s \to t} (\lambda x : s .\, N)[M_1, \ldots, M_n/x_1, \ldots, x_n].$$

   as desired.

4. $\Gamma \vdash (\texttt{elim}\ N) : t[\texttt{rec}\ \alpha.\ t/\alpha]$, where $\Gamma \vdash N : (\texttt{rec}\ \alpha.\ t)$. Follows directly from the induction hypothesis and the definition of adequacy relations.

5. $\Gamma \vdash (\texttt{intro}\ N) : (\texttt{rec}\ \alpha.\ t)$, where $\Gamma \vdash N : t[\texttt{rec}\ \alpha.\ t/\alpha]$. By the induction hypothesis,

$$[\![\Gamma \vdash N : t[\texttt{rec}\ \alpha.\ t/\alpha]]\!]\rho \lesssim_{t[\texttt{rec}\ \alpha.\ t/\alpha]} N[M_1, \ldots, M_n/x_1, \ldots, x_n].$$

   Suppose the left side is defined (the other case holds easily). Then

$$N[M_1, \ldots, M_n/x_1, \ldots, x_n] \Downarrow V \text{ for some } V,$$

and hence

$$(\texttt{intro}\ N)[M_1,\ldots,M_n/x_1,\ldots,x_n] \Downarrow (\texttt{intro}\ V).$$

Recall that $\mathit{elim} \circ \mathit{intro} = \mathit{id}$. Thus, since $N[M_1,\ldots,M_n/x_1,\ldots,x_n] \Downarrow V$ and $(\texttt{elim}\,(\texttt{intro}\ V)) \Downarrow V$, it follows by Lemma 9.8.2 that

$$\mathit{elim}([\![\Gamma \vdash (\texttt{intro}\ N) : \texttt{rec}\ \alpha.\ t]\!]\rho) \leq_{t[\texttt{rec}\ \alpha.\ t/\alpha]} V.$$

Therefore, by definition of adequacy relations,

$$[\![\Gamma \vdash (\texttt{intro}\ N) : \texttt{rec}\ \alpha.\ t]\!]\rho \lesssim_{\texttt{rec}\ \alpha.\ t} (\texttt{intro}\ N)[M_1,\ldots,M_n/x_1,\ldots,x_n].$$

as desired.

This completes the induction and hence the proof.

Definition 9.8.4 Suppose s is a closed type. A relation R between elements of $[\![s]\!]$ and closed terms of type s is **directed complete** if for any directed set $\{d_i \mid i \in I\}$ such that for all $i \in I$, $d_i\ R\ M$, then $(\sqcup d_i)\ R\ M$.

Definition 9.8.5 Suppose θ is a substitution from type variables to closed types, and R is a map from type variables to directed-complete relations. We say that R is **compatible with** θ if for all α, $R(\alpha)$ is a binary relation between $[\![\theta(\alpha)]\!]$ and closed terms of type $\theta(\alpha)$.

Definition 9.8.6 For all open types s, define the family

$$(\leq_{s,R} \mid R \text{ compatible with } \theta),$$

where $\leq_{s,R}$ is a relation between elements of $[\![\theta(s)]\!]$ and closed terms of type $\theta(s)$, as follows:

1. $d \leq_{\alpha,R} M$ iff $d\ R(\alpha)\ M$.
2. $d \leq_{\texttt{void},R} M$ holds vacuously.
3. $\top \leq_{\texttt{unit},R} M$ iff $M \Downarrow \langle\rangle$.
4. $(\mathit{inj}_i\ e) \leq_{s_1 \oplus s_2,R} M$ iff $M \Downarrow (\texttt{inj}_i\ V)$ and $(e \leq_{s_i,R} V)$.
5. $\langle e_1, e_2\rangle \leq_{s_1 \otimes s_2,R} M$ iff $M \Downarrow \langle V_1, V_2\rangle$ and $(e_i \leq_{s_i,R} V_i)$.
6. $f \leq_{s_1 \Rightarrow s_2,R} M$ iff
   $M \Downarrow (\lambda x : s_1.\ N)$, and $e \leq_{s_1,R} P$ implies $f(e) \lesssim_{s_2,R} (M\ P)$.
7. If $s = (\texttt{rec}\ \alpha.\ t)$, then $e \leq_{s,R} M$ iff for all $n \geq 0$, $(\mu^{n,p}_{\theta(s)}\ e) \lesssim^s_{n,R} M$, where $\leq^s_{n,R}$ are the relations defined by

   $$\begin{aligned} e \leq^s_{0,R} N &\iff \text{true} \\ e \leq^s_{n+1,R} N &\iff N \Downarrow (\texttt{intro}\ V') \text{ and } e \leq_{t,R[\alpha \mapsto \leq^s_{n,R}]} V'. \end{aligned}$$

   (Recall that $\mu^n_{\theta(s)}$ are the maps in the colimiting cocone from Section 9.3.8.)

Lemma 9.8.7 *Suppose R is compatible with θ. Then for any type s, $\leq_{s,R}$ is a directed-complete relation between elements of $[\![\theta(s)]\!]$ and closed terms of type $\theta(s)$.*

Lemma 9.8.8 $\leq_{t,R[\alpha\mapsto\leq_{s,R}]}=\leq_{t[s/\alpha],R}$

Proof: By induction on t. The only difficult case is $t = (\texttt{rec}\ \beta.\ u)$. Let $R' = R[\alpha \mapsto \leq_{s,R}]$ and $t' = t[s/\alpha]$. Note that $\leq^{t'}_{0,R}=\leq^{t}_{0,R'}$, and $\leq^{t'}_{n,R}=\leq^{t}_{n,R'}$ implies that $\leq^{t'}_{n+1,R}=\leq^{t}_{n+1,R'}$ using the induction hypothesis. Thus, for all n, $\leq^{t'}_{n,R}=\leq^{t}_{n,R'}$, and hence

$$d \leq_{t,R'} M \Longleftrightarrow d \leq_{t',R} M$$

as desired.

Definition 9.8.9 If s is a closed type and R is a relation between elements of $[\![s]\!]$ and closed terms of type s, we write $exp\ \widetilde{R}\ M$ if either the expression exp is undefined, or it is defined and $exp\ R\ M$.

Lemma 9.8.10 *Suppose $\pi : \eta \to \eta'$ is a ep-pair environment (i.e., mapping type variables to ep-pairs, not just pre-ep-pairs) and for all α,*

$$\begin{array}{lll} d\ R(\alpha)\ M & \textit{implies} & \pi(\alpha)^e(d)\ \widetilde{R'(\alpha)}\ M \\ d\ R'(\alpha)\ M & \textit{implies} & \pi(\alpha)^p(d)\ \widetilde{R(\alpha)}\ M \end{array}$$

Then

1. *If $d \leq_{s,R} M$, then $([\![s]\!]\pi)^e(d) \leq_{s,R'} M$.*
2. *If $d \leq_{s,R'} M$, then $([\![s]\!]\pi)^p(d) \lesssim_{s,R} M$.*

Proof: By induction on s, proving both claims simultaneously. Most of the cases are straightforward; the only case that is not is when $s = (\texttt{rec}\ \alpha.\ t)$. For this case, we first prove a claim. Let $\leq^s_{n,R}$ and $\leq^s_{n,R'}$ be the relations defined above. We claim that for all n,

- If $e \leq^s_{n,R} N$, then $p^{n,e}_{s,\pi}(e) \leq^s_{n,R'} N$.
- If $e \leq^s_{n,R'} N$, then $p^{n,p}_{s,\pi}(e) \lesssim^s_{n,R} N$.

We prove the claim by induction on n. The basis is easy, so consider the induction case. Let

$$\pi' = \pi[\alpha \mapsto p^n_{s,\pi}] : \eta[\alpha \mapsto T^n_{s,\eta}] \to \eta'[\alpha \mapsto T^n_{s,\eta'}]$$

Since $p^n_{s,\pi}$ is an ep-pair (by a simple induction on the definition of $p^n_{s,\pi}$), π' is an ep-pair environment. To see the second part of the claim, suppose $e \leq^s_{n+1,R'} N$. Then $N \Downarrow (\texttt{intro}\ V)$ and $e \leq_{t,R'[\alpha\mapsto\leq^s_{n,R'}]} V$. By the induction hypothesis,

$$p^{n+1,p}_{s,\pi}(e) = ([\![t]\!]\pi')^p(e) \lesssim_{t,R[\alpha\mapsto\leq^s_{n,R}]} V$$

and hence $p^{n+1,p}_{s,\pi}(e) \lesssim^s_{n+1,R} N$. The first part holds analogously, completing the proof of the claim.

Using the claim, we can now prove the lemma. We prove only the second part and leave the other to the reader. Suppose $d \leq_{s,R'} M$. Then for all n, $\mu^{n,p}_{s,\eta'}(d) \lesssim^s_{n,R'} M$. By the claim, $p^{n,p}_{s,\pi}(\mu^{n,p}_{s,\eta'}(d)) \lesssim^s_{n,R} M$. By general facts about the colimit, $p^{n,p}_{s,\pi} \circ \mu^{n,p}_{s,\eta'} = \mu^{n,p}_{s,\eta} \circ (\llbracket s\rrbracket\pi)^p$, so for all n,

$$\mu^{n,p}_{s,\eta}\ ((\llbracket s\rrbracket\pi)^p\ (d)) \lesssim^s_{n,R}\ M.$$

Thus, $(\llbracket s\rrbracket\pi)^p\ (d) \lesssim_{s,R} M$ as desired.

Lemma 9.8.11 *Suppose $s = (\texttt{rec}\ \alpha.\ t)$ and $\leq^s_{n,R}$ are the relations defined above. If $d \leq^s_{k,R} M$, then $\mu^{k,e}_{s,\eta}(d) \leq_{s,R} M$.*

Proof: First, we claim that the following two statements hold:

- If $d \leq^s_{n,R} M$, then $f^{n,e}_{s,\eta}(d) \leq^s_{n+1,R} M$.
- If $d \leq^s_{n+1,R} M$, then $f^{n,p}_{s,\eta}(d) \lesssim^s_{n,R} M$.

We prove the claim by induction on n. The base case is straightforward. For the induction case, we prove the first statement and leave the second to the reader. Now, assume that $d \leq^s_{n,R} M$ implies $f^{n,e}_{s,\eta}(d) \leq_{n+1,R} M$. Recall that $f^{n+1,e}_{s,\eta} = (\llbracket t\rrbracket(id[\alpha \mapsto f^n_{s,\eta}]))^e$. Therefore, for any β,

$$d\ R[\alpha \mapsto \leq^s_{n,R}](\beta)\ M \text{ implies } (id[\alpha \mapsto f^n_{s,\eta}](\beta))^e(d)\ R[\alpha \mapsto \leq^s_{n+1,R}](\beta)\ M.$$

So suppose $d \leq^s_{n+1,R} M$; then $M \Downarrow (\texttt{intro}\ V)$ and $d \leq_{t,R[\alpha\mapsto\leq_{n,R}]} V$. By Lemma 9.8.10 (1),

$$(\llbracket t\rrbracket(id[\alpha \mapsto f^n_{s,\eta}]))^e(d) \leq_{t,R[\alpha\mapsto\leq_{n+1,R}]} V$$

and so $f^{n+1,e}_{s,\eta}(d) \leq^s_{n+2,R} M$.

We can now prove the lemma. Suppose $d \leq^s_{k,R} M$. We show that for all $n \geq 0$, $\mu^{n,p}_{s,\eta}\ (\mu^{k,e}_{s,\eta}\ (d)) \lesssim^s_{n,R} M$. There are two cases:

- If $n < k$, then
$$\mu^{n,p}_{s,\eta}\ (\mu^{k,e}_{s,\eta}\ (d)) = f^{n,p}_{s,\eta}\ (f^{n+1,p}_{s,\eta}\ (\ldots f^{k,p}_{s,\eta}\ (d)\ldots)).$$
By repeated uses of the second part of the claim,
$$f^{n,p}_{s,\eta}\ (f^{n+1,p}_{s,\eta}\ (\ldots f^{k,p}_{s,\eta}\ (d)\ldots)) \lesssim^s_{n,R} M.$$
- If $n \geq k$, we proceed by induction on n. In the basis, $n = k$, and $\mu^{n,p}_{s,\eta}\ (\mu^{k,e}_{s,\eta}\ (d)) = d \leq^s_{n,R} M$ by hypothesis. For the induction case, note that
$$\mu^{n+1,p}_{s,\eta}\ (\mu^{k,e}_{s,\eta}\ (d)) = f^{n,e}_{s,\eta}\ (\mu^{n,p}_{s,\eta}\ (\mu^{k,e}_{s,\eta}\ (d))).$$
By the induction hypothesis, $\mu^{n,p}_{s,\eta}\ (\mu^{k,e}_{s,\eta}\ (d)) \leq^s_{n,R} M$. By the first part of the claim,
$$f^{n,e}_{s,\eta}\ (\mu^{n,p}_{s,\eta}\ (\mu^{k,e}_{s,\eta}\ (d))) \leq^s_{n+1,R} M$$
as desired.

This completes the proof.

Lemma 9.8.12 *Let R be compatible with θ. Then the family of relations $\leq_{s,R}$ for all closed types s forms an adequacy relation.*

Proof: The only tricky part then is to verify condition (6) of the definition of adequacy relations. Suppose $s = (\texttt{rec}\ \alpha.\ t)$ is a closed type, and pick any η. To see one direction of (6), suppose $d \ \leq_{s,R}\ M$. Then for all n,

$$d_n = \mu^{n,p}_{s,\eta}\ (d) \lesssim^s_{n,R} M.$$

In particular, $M \Downarrow (\texttt{intro}\ V)$ and for all n,

$$d_{n+1} \lesssim_{t,R[\alpha \mapsto \leq^s_{n,R}]} V.$$

By Lemmas 9.8.10 and 9.8.11,

$$([\![t]\!](id[\alpha \mapsto \mu^n_{s,\eta}]))^e(d_{n+1}) \lesssim_{t,R[\alpha \mapsto \leq_{s,R}]} V.$$

But note that $([\![t]\!](id[\alpha \mapsto \mu^n_{s,\eta}]))^e = elim \circ \mu^{n+1,e}_{s,\eta}$. Thus, for all n,

$$elim(\mu^{n+1,e}_{s,\eta}(d_{n+1})) \lesssim_{t,R[\alpha \mapsto \leq_{s,R}]} V$$

and hence by Lemma 9.8.7,

$$\bigsqcup_{n \geq 0} elim(\mu^{n+1,e}_{s,\eta}(d_{n+1})) = elim(d) \lesssim_{t,R[\alpha \mapsto \leq_{s,R}]} V$$

as desired. The converse holds by a similar argument.

The Adequacy Theorem 9.4.1 now follows directly from Lemmas 9.8.3 and 9.8.12.

9.9 Proof of Lemma 9.6.1

The proof of Lemma 9.6.1 uses techniques from [112]. In outline, we show how to construct, from the syntax of the language, pre-ep-pairs which mimic the semantic pre-ep-pairs in the construction of the model. We use these to construct the desired terms $\texttt{clm}^e_{s,n}$ and $\texttt{clm}^p_{s,n}$.

Given type expressions s and s', a pair of closed terms

$$M : s \Rightarrow s' \text{ and } M' : s' \Rightarrow s$$

is called a **syntactic pre-ep-pair** from s to s' (written $\langle M, M' \rangle : s \hookrightarrow s'$). Some notation on syntactic pre-ep-pairs will be helpful. If P is a syntactic pre-ep-pair, P^e and P^p represent its first and second components respectively, and P^{-1} denotes the pair $\langle P^p, P^e \rangle$. There are also several constructions for building complex syntactic pre-ep-pairs from simple ones. Let $M : s \hookrightarrow t$, $N : t \hookrightarrow u$ and $P : s' \hookrightarrow t'$. Define

$$\mathtt{id}_s : s \hookrightarrow s = \langle \lambda x : s.\, x, \lambda x : s.\, x\rangle \qquad !_t : \mathtt{void} \hookrightarrow t = \langle \Omega, \Omega\rangle$$

$$\mathtt{iso}_{\mathtt{rec}\ \alpha.\ t} : t[\mathtt{rec}\ \alpha.\ t/\alpha] \hookrightarrow \mathtt{rec}\ \alpha.\ t = \langle \mathtt{intro}_{\mathtt{rec}\ \alpha.\ t}, \mathtt{elim}_{\mathtt{rec}\ \alpha.\ t}\rangle$$

$$\mathtt{iso}^{-1}_{\mathtt{rec}\ \alpha.\ t} : (\mathtt{rec}\ \alpha.\ t) \hookrightarrow t[\mathtt{rec}\ \alpha.\ t/\alpha] = \langle \mathtt{elim}_{\mathtt{rec}\ \alpha.\ t}, \mathtt{intro}_{\mathtt{rec}\ \alpha.\ t}\rangle$$

$$(N \circ M) : s \hookrightarrow u = \langle N^e \circ M^e, M^p \circ N^p\rangle$$

$$(M \oplus P) : (s \oplus s') \hookrightarrow (t \oplus t') =$$

$$\langle \lambda x : s \oplus s'.\, \mathtt{case}\ x\ \mathtt{of}\ \mathtt{inj}_1(y).(M^e\ y)\ \mathtt{or}\ \mathtt{inj}_2(y).(P^e\ y),$$
$$\lambda x : t \oplus t'.\, \mathtt{case}\ x\ \mathtt{of}\ \mathtt{inj}_1(y).(M^p\ y)\ \mathtt{or}\ \mathtt{inj}_2(y).(P^p\ y)\rangle$$

$$(M \otimes P) : (s \otimes s') \hookrightarrow (t \otimes t') = \langle \lambda x : s \otimes s'.\, \langle M^e\ (\mathtt{proj}_1\ x), P^e\ (\mathtt{proj}_2\ x)\rangle,$$
$$\lambda x : t \otimes t'.\, \langle M^p\ (\mathtt{proj}_1\ x), P^p\ (\mathtt{proj}_2\ x)\rangle\rangle$$

$$(M \Rightarrow P) : (s \Rightarrow s') \hookrightarrow (t \Rightarrow t') =$$
$$\langle \lambda x : s \Rightarrow s'.\, P^e \circ x \circ M^p, \lambda x : t \Rightarrow t'.\, P^p \circ x \circ M^e\rangle$$

where, abusing notation, $(Q \circ Q')$ stands for the term $(\lambda x.\, Q\ (Q'\ x))$.

We use these constructions in defining terms $\langle \mathtt{F}^e_s, \mathtt{F}^p_s\rangle$ that syntactically represent the action, on *syntactic* pre-ep-pairs, of the functor $[\![s]\!]$ on *semantic* pre-ep-pairs. The terms are defined by induction on the structure of s. As with the functors, the syntactic terms take in a map, usually denoted $\mathtt{pi}$, from type variables to syntactic pre-ep-pairs. These maps go between **type substitutions** θ, i.e., maps from type variables to closed types. More precisely, we write $\mathtt{pi} : \theta \hookrightarrow \theta'$ if θ, θ' are type substitutions such that for all α, $\mathtt{pi}(\alpha) : \theta(\alpha) \hookrightarrow \theta'(\alpha)$. We write $\overline{\mathtt{id}} : \theta \hookrightarrow \theta$ to denote the map where for all type variables α, $\overline{\mathtt{id}}(\alpha) = \mathtt{id}_{\theta(\alpha)}$.

To define the syntactic representation of the functor, let s be a type, and $\mathtt{pi} : \theta \hookrightarrow \theta'$. Let $\theta(\tau)$ denote the type produced from applying the type substitution θ to τ. The term $\mathtt{F}_s(\mathtt{pi}) : \theta(s) \hookrightarrow \theta'(s)$ is a syntactic pre-ep-pair defined by induction on s:

$$\begin{array}{rcl}
\mathtt{F}_\alpha(\mathtt{pi}) & = & \mathtt{pi}(\alpha) \\
\mathtt{F}_{\mathtt{unit}}(\mathtt{pi}) & = & \mathtt{id}_{\mathtt{unit}} \\
\mathtt{F}_{s \Rightarrow t}(\mathtt{pi}) & = & \mathtt{F}_s(\mathtt{pi}) \Rightarrow \mathtt{F}_t(\mathtt{pi}) \\
\mathtt{F}_{\mathtt{rec}\ \alpha.\ t}(\mathtt{pi}) & = & \mathtt{fix}\ (\lambda f.\, \mathtt{iso}_{\theta'(\mathtt{rec}\ \alpha.\ t)} \circ \mathtt{F}_t(\mathtt{pi}[\alpha \mapsto f]) \circ \mathtt{iso}^{-1}_{\theta(\mathtt{rec}\ \alpha.\ t)})
\end{array}$$

A similar definition appears elsewhere [1]. In the definition, the term $\mathtt{fix}$ in the last line is shorthand for the term $\lambda g.\, \Delta\ (\mathtt{intro}\ \Delta)$, where

$$\Delta = \lambda x.\, \langle \lambda y.\, (\mathtt{proj}_1\ (g\ ((\mathtt{elim}\ x)\ x)))\ y, \lambda z.\, (\mathtt{proj}_2\ (g\ ((\mathtt{elim}\ x)\ x)))\ z\rangle.$$

The term $\mathtt{fix}$ is a simple modification of the call-by-value recursion combinator (see [41]); it defines a pair $\langle \mathtt{F}^e_{\mathtt{rec}\ \alpha.\ t}(\mathtt{pi}), \mathtt{F}^p_{\mathtt{rec}\ \alpha.\ t}(\mathtt{pi})\rangle$ by mutual recursion.

We will show that the semantic functor $[\![s]\!]$ coincides with the syntactic functor $\mathtt{F}_s$. In order to do this, we must have a way to connect semantic pre-ep-environments π with syntactic pre-ep-environments $\mathtt{pi}$. Suppose π is a morphism in $\mathcal{E}$ (see previous subsection) and η is a type environment. Then π **matches** $\mathtt{pi} : \theta \hookrightarrow \theta'$ if for all type variables α, $[\![\mathtt{pi}(\alpha)]\!] = \pi(\alpha)$.

Suppose $\mathtt{pi} : \theta \hookrightarrow \theta'$ and $s = (\mathtt{rec}\ \alpha.\ t)$. For any $\mathtt{g} : \theta(s) \hookrightarrow \theta'(s)$, define $\mathtt{J}_s(\mathtt{pi}, \mathtt{g}) : \theta(s) \hookrightarrow \theta'(s)$ by

$$\mathtt{J}_s(\mathtt{pi}, \mathtt{g}) = \mathtt{iso}_{\theta'(s)} \circ \mathtt{F}_t(\mathtt{pi}[\alpha \mapsto \mathtt{g}]) \circ \mathtt{iso}^{-1}_{\theta(s)}.$$

Define

$$\begin{array}{rcl} \mathtt{J}^0_s(\mathtt{pi}, \mathtt{g}) & = & \mathtt{g} \\ \mathtt{J}^{n+1}_s(\mathtt{pi}, \mathtt{g}) & = & \mathtt{J}_s(\mathtt{pi}, \mathtt{J}^n_s(\mathtt{pi}, \mathtt{g})) \end{array}$$

Lemma 9.9.1 $[\![\mathtt{F}_s(\mathtt{pi})]\!] = \bigsqcup_{n \geq 0} [\![\mathtt{J}^n_s(\mathtt{pi}, \Omega)]\!]$.

Theorem 9.9.2 (Correspondence) *Suppose π matches $\mathtt{pi} : \theta_1 \hookrightarrow \theta_2$. Then $[\![s]\!]\pi = [\![\mathtt{F}_s(\mathtt{pi})]\!]$.*

Proof: By induction on the structure of s. We consider four of the cases here.

1. $s = \alpha$. Then $[\![\mathtt{F}_\alpha(\mathtt{pi})]\!] = [\![\mathtt{pi}(\alpha)]\!] = \pi(\alpha) = [\![\alpha]\!](\pi)$ by the hypothesis that π matches $\mathtt{pi}$.

2. $s = \mathtt{unit}, \mathtt{void}$. Then $[\![\mathtt{F}_s(\mathtt{pi})]\!] = \mathtt{id}_s = [\![s]\!]\pi$.

3. $s = (s_1 \Rightarrow s_2)$. Then

$$\begin{array}{rcl} [\![\mathtt{F}_s(\mathtt{pi})]\!] & = & [\![(\mathtt{F}_{s_1}(\mathtt{pi}) \Rightarrow \mathtt{F}_{s_2}(\mathtt{pi}))]\!] \\ & = & ([\![\mathtt{F}_{s_1}(\mathtt{pi})]\!] \Rightarrow [\![\mathtt{F}_{s_2}(\mathtt{pi})]\!]) \\ & = & [\![s_1]\!]\pi \Rightarrow [\![s_2]\!]\pi \\ & = & [\![s]\!]\pi \end{array}$$

4. $s = (\mathtt{rec}\ \alpha.\ t)$. Define type environments η_1 and η_2 by $\eta_1(\alpha) = [\![\theta_1(\alpha)]\!]$ and $\eta_2(\alpha) = [\![\theta_2(\alpha)]\!]$. We claim that $[\![\mathtt{F}_s(\mathtt{pi})]\!]$ is the unique mediating map from $[\![s]\!]\eta_1 = T_{s,\eta_1}$ to $[\![s]\!]\eta_2 = T_{s,\eta_2}$. The proof of the claim is quite similar to the proof of the existence of Freyd's minimal invariants from [3, 97], but we present the argument in detail because of the slight difference in setting with type environments.

   Recall the definitions of $\mu^n_{s,\eta}$ and $p^n_{s,\pi}$ and the definition of $\mathtt{J}^n_s$ above. To prove the claim, we note three facts:

   - For any type environment η, $\mu^{n+1}_{s,\eta} = [\![\mathtt{iso}]\!] \circ [\![t]\!](id[\alpha \mapsto \mu^n_{s,\eta}])$.
   - $[\![\mathtt{J}^n_s(\mathtt{pi}, \Omega)]\!]$ is a pair of partial functions.
   - $[\![\mathtt{J}^n_s(\mathtt{pi}, \Omega)]\!] \circ \mu^n_{s,\eta_1} = \mu^n_{s,\eta_2} \circ p^n_{s,\pi}$.

   The first fact holds from general principles about the colimit. The second may be proved by induction on n, using the induction hypothesis of the theorem. The proof of the third proceeds by induction on n. The base case, when $n = 0$, is easy. For the induction step,

$$
\begin{aligned}
&[\![\mathtt{J}_s^{n+1}(\mathtt{pi}, \Omega) \circ \mu_{s,\eta_1}^{n+1}]\!] \\
&\quad = [\![\mathtt{iso}]\!] \circ [\![\mathtt{F}_t(\mathtt{pi}[\alpha \mapsto \mathtt{J}_s^n(\mathtt{pi}, \Omega)])]\!] \circ [\![\mathtt{iso}^{-1}]\!] \circ \mu_{s,\eta_1}^{n+1} \\
&\quad = [\![\mathtt{iso}]\!] \circ [\![t]\!](\pi[\alpha \mapsto [\![\mathtt{J}_s^n(\mathtt{pi}, \Omega)]\!]]) \circ [\![t]\!](id[\alpha \mapsto \mu_{s,\eta_1}^n]) \\
&\quad = [\![\mathtt{iso}]\!] \circ [\![t]\!](\pi[\alpha \mapsto [\![\mathtt{J}_s^n(\mathtt{pi}, \Omega)]\!] \circ \mu_{s,\eta_1}^n]) \\
&\quad = [\![\mathtt{iso}]\!] \circ [\![t]\!](\pi[\alpha \mapsto \mu_{s,\eta_2}^n \circ p_{s,\pi}^n]) \\
&\quad = [\![\mathtt{iso}]\!] \circ [\![t]\!](id[\alpha \mapsto \mu_{s,\eta_2}^n]) \circ [\![t]\!](\pi[\alpha \mapsto p_{s,\pi}^n]) \\
&\quad = \mu_{s,\eta_2}^{n+1} \circ p_{s,\pi}^{n+1}
\end{aligned}
$$

as desired, where the second line follows by the induction hypothesis of the theorem (the fact that the semantic functor $[\![s]\!]$ coincides with $\mathtt{F}_s$) , and the fourth line follows by the induction hypothesis.

These facts can be used to prove the claim. Any mediating map must be unique, since $\langle [\![s]\!]\eta_1, \mu_{s,\eta_1}^n \mid n \geq 0 \rangle$ is a colimiting cocone for the chain $\langle T_{s,\eta_1}^i, f_{s,\eta_1}^i \mid i \geq 0 \rangle$ and $\langle [\![s]\!]\eta_2, \mu_{s,\eta_2}^n \circ p_{s,\pi}^n \mid n \geq 0 \rangle$ is a cocone for the same chain. Thus, all we need to do is to show that $\mathtt{F}_s(\mathtt{pi})$ is a mediating map, i.e., for all $n \geq 0$,

$$[\![\mathtt{F}_s(\mathtt{pi})]\!] \circ \mu_{s,\eta_1}^n = \mu_{s,\eta_2}^n \circ p_{s,\pi}^n.$$

To see this, for any type s and type environment η, define $f_{s,\eta}^{n \Rightarrow m}$ to be the composition of the maps from $T_{s,\eta}^n$ to $T_{s,\eta}^m$; then for any $n \leq m$,

$$
\begin{aligned}
[\![\mathtt{J}_s^m(\mathtt{pi}, \Omega)]\!] \circ \mu_{s,\eta_1}^n &= [\![\mathtt{J}_s^m(\mathtt{pi}, \Omega)]\!] \circ \mu_{s,\eta_1}^m \circ f_{s,\eta_1}^{n \Rightarrow m} \\
&= \mu_{s,\eta_2}^m \circ p_{s,\pi}^m \circ f_{s,\eta_1}^{n \Rightarrow m} \\
&= \mu_{s,\eta_2}^m \circ f_{s,\eta_2}^{n \Rightarrow m} \circ p_{s,\pi}^n \\
&= \mu_{s,\eta_2}^n \circ p_{s,\pi}^n
\end{aligned}
$$

and hence

$$
\begin{aligned}
[\![\mathtt{F}_s(\mathtt{pi})]\!] \circ \mu_{s,\eta_1}^n &= (\bigsqcup_{m \geq 0} [\![\mathtt{J}_s^m(\mathtt{pi}, \Omega)]\!]) \circ \mu_{s,\eta_1}^n \\
&= \bigsqcup_{m \geq 0} \mu_{s,\eta_2}^n \circ p_{s,\pi}^n = \mu_{s,\eta_2}^n \circ p_{s,\pi}^n
\end{aligned}
$$

as desired.

This completes the induction and hence the proof.

Define two families of syntactic pre-ep-pairs

$$\mathtt{unwnd}_{s,i} : s^i \hookrightarrow s^{i+1} \text{ and } \mathtt{clm}_{s,i} : s^i \hookrightarrow s$$

by induction on s. Most of the definition is straightforward—the main difficulty lies when s is a recursive type.

1. For type variables, $\mathtt{unwnd}_{\alpha,n} = \mathtt{id}_\alpha$ and $\mathtt{clm}_{\alpha,n} = \mathtt{id}_\alpha$.

2. For the base types $s = \mathtt{void}, \mathtt{unit}$, let $\mathtt{unwnd}_{s,n} = \mathtt{id}_s$ and $\mathtt{clm}_{s,n} = \mathtt{id}_s$.

3. For sum, product, and function types, let

$$\begin{array}{rcl} \mathtt{unwnd}_{s_1 \Rightarrow s_2,n} & = & \mathtt{unwnd}_{s_1,n} \Rightarrow \mathtt{unwnd}_{s_2,n} \\ \mathtt{clm}_{s_1 \Rightarrow s_2,n} & = & \mathtt{clm}_{s_1,n} \Rightarrow \mathtt{clm}_{s_2,n} \end{array}$$

   and similarly for sums and products.

4. For recursive types, suppose $s = (\mathtt{rec}\ \alpha.\ t)$. Define the family of types $T^n_{s,k}$, for $n, k \geq 0$, by

$$\begin{array}{rcl} T^0_{s,k} & = & \mathtt{void} \\ T^{n+1}_{s,k} & = & t_k[T^n_{s,k}/\alpha]. \end{array}$$

   Notice that, for any n, $T^n_{s,n}$ is the same as s_n defined above. Define two families of syntactic pre-ep-pairs

$$\mathtt{f}^n_{s,k} : T^n_{s,k} \hookrightarrow T^n_{s,k+1} \text{ and } \mathtt{g}^n_{s,k} : T^n_{s,k} \hookrightarrow T^{n+1}_{s,k}$$

   by

$$\begin{array}{rclcrcl} \mathtt{f}^0_{s,k} & = & !_{\mathtt{void}} & \quad & \mathtt{f}^{n+1}_{s,k} & = & \mathtt{F}_{t_{k+1}}(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{f}^n_{s,k}]) \circ \mathtt{unwnd}_{t,k}[T^n_{s,k}/\alpha] \\ \mathtt{g}^0_{s,k} & = & !_{T^1_{s,k}} & & \mathtt{g}^{n+1}_{s,k} & = & \mathtt{F}_{t_k}(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{g}^n_{s,k}]) \end{array}$$

   A diagram clarifies these types and functions:

$$\begin{tikzcd}
\vdots & \vdots & \vdots & \\
T^2_{s,0} \arrow[u, "\mathtt{g}^0_{s,2}"] \arrow[r, "\mathtt{f}^2_{s,0}"'] & T^2_{s,1} \arrow[u, "\mathtt{g}^1_{s,2}"] \arrow[r, "\mathtt{f}^2_{s,1}"'] & T^2_{s,2} \arrow[u, "\mathtt{g}^2_{s,2}"] \arrow[r, "\mathtt{f}^2_{s,2}"'] & \cdots \\
T^1_{s,0} \arrow[u, "\mathtt{g}^0_{s,1}"] \arrow[r, "\mathtt{f}^1_{s,0}"'] & T^1_{s,1} \arrow[u, "\mathtt{g}^1_{s,1}"] \arrow[r, "\mathtt{f}^1_{s,1}"'] & T^1_{s,2} \arrow[u, "\mathtt{g}^2_{s,1}"] \arrow[r, "\mathtt{f}^1_{s,2}"'] & \cdots \\
\mathtt{void} \arrow[u, "\mathtt{g}^0_{s,0}"] \arrow[r, "\mathtt{f}^0_{s,0}"'] & \mathtt{void} \arrow[u, "\mathtt{g}^1_{s,0}"] \arrow[r, "\mathtt{f}^0_{s,1}"'] & \mathtt{void} \arrow[u, "\mathtt{g}^2_{s,0}"] \arrow[r, "\mathtt{f}^0_{s,2}"'] & \cdots
\end{tikzcd}$$

Then define the maps $\mathtt{unwnd}_{s,n}$ along the main diagonal:

$$\mathtt{unwnd}_{s,n} \quad = \quad \mathtt{f}^{n+1}_{s,n} \circ \mathtt{g}^{n}_{s,n}$$

For the colimiting maps, define $\mathtt{clm}^n_{s,k} : T^n_{s,k} \hookrightarrow s$ by

$$\begin{array}{rcl} \mathtt{clm}^0_{s,k} & = & !_s \\ \mathtt{clm}^{n+1}_{s,k} & = & \mathtt{iso}_s \circ \mathtt{F}_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{clm}^n_{s,k}]) \circ \mathtt{clm}_{t,k}[T^n_{s,k}/\alpha] \end{array}$$

and set $\mathtt{clm}_{s,n} = \mathtt{clm}^n_{s,n}$.

In order to prove Lemma 9.6.1, we need to break up the definition of the colimiting maps on recursive types into two pieces. Suppose $s = (\mathtt{rec}\ \alpha.\ t)$. Define the syntactic pre-ep-pairs $\mathtt{h}^n_{s,k} : T^n_{s,k} \hookrightarrow T^n_s$ and $\mathtt{mu}^n_s : T^n_s \hookrightarrow s$, where

$$\begin{array}{rclrcl} T^0_s & = & \mathtt{void} & T^{n+1}_s & = & t[T^n_s/\alpha] \\ \mathtt{h}^0_{s,k} & = & \mathtt{id}_{\mathtt{void}} & \mathtt{h}^{n+1}_{s,k} & = & \mathtt{F}_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{h}^n_{s,k}]) \circ \mathtt{clm}_{t,k}[T^n_{s,k}/\alpha] \\ \mathtt{mu}^0_s & = & !_s & \mathtt{mu}^{n+1}_s & = & \mathtt{iso}_s \circ \mathtt{F}_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{mu}^n_s]) \end{array}$$

Lemma 9.9.3 $[\![\theta(\mathtt{mu}^n_s)]\!] = \mu^n_{\theta(s)}$.

The proof follows from the Correspondence Theorem and the properties of colimits.

Lemma 9.9.4 $[\![\theta(\mathtt{clm}^n_{s,k})]\!] = [\![\mathtt{mu}^n_s \circ \mathtt{h}^n_{s,k}]\!]$.

Proof: By induction on n; to simplify notation, we omit meaning brackets below. The base case is trivial; the induction step is as follows:

$$\begin{aligned} &\theta(\mathtt{mu}^{n+1}_s \circ \mathtt{h}^{n+1}_{s,k}) \\ &\qquad = \theta(\mathtt{iso}_s \circ \mathtt{F}_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{mu}^n_s]) \circ \mathtt{F}_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{h}^n_{s,k}]) \circ \mathtt{clm}_{t,k}[T^n_{s,k}/\alpha]) \\ &\qquad = \theta(\mathtt{iso}_s \circ \mathtt{F}_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{mu}^n_s \circ \mathtt{h}^n_{s,k}]) \circ \mathtt{clm}_{t,k}[T^n_{s,k}/\alpha]) \\ &\qquad = \theta(\mathtt{iso}_s \circ \mathtt{F}_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{clm}^n_{s,k}]) \circ \mathtt{clm}_{t,k}[T^n_{s,k}/\alpha]) \\ &\qquad = \theta(\mathtt{clm}^{n+1}_{s,k}) \end{aligned}$$

where the first equality follows from the definition of $\mathtt{mu}^{n+1}_s, \mathtt{h}^{n+1}_{s,k}$, the second from syntactic functoriality, the third from the induction hypothesis, and the forth from the definition of $\mathtt{clm}^{n+1}_{s,k}$. This completes the induction step and hence the proof.

Lemma 9.9.5 *For any s and θ,* $\bigsqcup_{n\geq 0}[\![\theta(\mathtt{clm}^e_{s,n} \circ \mathtt{clm}^p_{s,n})]\!] = id_{[\![\theta(s)]\!]}$.

Proof: We proceed by induction on s, omitting meaning brackets to simplify the notation. We give only the case of $s = (\mathtt{rec}\ \alpha.\ t)$ and leave the others to the reader. We claim

$$\bigsqcup_{k\geq 0} \theta(\mathtt{h}^{n,e}_{s,k} \circ \mathtt{h}^{n,p}_{s,k}) = id.$$

We prove the claim by induction on n. The basis, $n = 0$, is easy, so for the induction step,

$$
\begin{aligned}
&\bigsqcup_{k\geq 0} \theta(\mathtt{h}^{n+1,e}_{s,k} \circ \mathtt{h}^{n+1,p}_{s,k})\\
&= \bigsqcup_{k\geq 0} \theta(\mathtt{F}^e_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{h}^{n,e}_{s,k}]) \circ \mathtt{clm}^e_{t,k}[T^n_{s,k}/\alpha] \circ \mathtt{clm}^p_{t,k}[T^n_{s,k}/\alpha] \circ \mathtt{F}^p_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{h}^{n,e}_{s,k}]))\\
&= \bigsqcup_{k,m\geq 0} \theta(\mathtt{F}^e_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{h}^{n,e}_{s,k}]) \circ \mathtt{clm}^e_{t,m}[T^n_{s,k}/\alpha] \circ \mathtt{clm}^p_{t,m}[T^n_{s,k}/\alpha] \circ \mathtt{F}^p_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{h}^{n,e}_{s,k}]))\\
&= \bigsqcup_{k,m\geq 0} \theta(\mathtt{F}^e_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{h}^{n,e}_{s,k}]) \circ (\mathtt{clm}^e_{t,m} \circ \mathtt{clm}^p_{t,m})[T^n_{s,k}/\alpha] \circ \mathtt{F}^p_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{h}^{n,e}_{s,k}]))\\
&= \bigsqcup_{k\geq 0} \theta(\mathtt{F}^e_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{h}^{n,e}_{s,k}]) \circ (\bigsqcup_{m\geq 0} \mathtt{clm}^e_{t,m} \circ \mathtt{clm}^p_{t,m})[T^n_{s,k}/\alpha] \circ \mathtt{F}^p_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{h}^{n,e}_{s,k}]))\\
&= \bigsqcup_{k\geq 0} \theta(\mathtt{F}^e_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{h}^{n,e}_{s,k}]) \circ \mathtt{F}^p_t(\overline{\mathtt{id}}[\alpha \mapsto \mathtt{h}^{n,e}_{s,k}]))\\
&= id
\end{aligned}
$$

where the second line follows from Bekič's Lemma, the fifth from the global induction hypothesis, and the last by a simple induction on t, using the local induction hypothesis. Taking advantage of this fact,

$$
\begin{aligned}
\bigsqcup_{n\geq 0} \theta(\mathtt{clm}^e_{s,n} \circ \mathtt{clm}^p_{s,n}) &= \bigsqcup_{n\geq 0} \theta(\mathtt{clm}^{n,e}_{s,n} \circ \mathtt{clm}^{n,p}_{s,n})\\
&= \bigsqcup_{k,n\geq 0} \theta(\mathtt{clm}^{n,e}_{s,k} \circ \mathtt{clm}^{n,p}_{s,k})\\
&= \bigsqcup_{k,n\geq 0} \theta(\mathtt{mu}^{n,e}_{s} \circ \mathtt{h}^{n,e}_{s,k} \circ \mathtt{h}^{n,p}_{s,k} \circ \mathtt{mu}^{n,p}_{s})\\
&= \bigsqcup_{n\geq 0} \theta(\mathtt{mu}^{n,e}_{s} \circ (\bigsqcup_{k\geq 0} \mathtt{h}^{n,e}_{s,k} \circ \mathtt{h}^{n,p}_{s,k}) \circ \mathtt{mu}^{n,p}_{s})\\
&= \bigsqcup_{n\geq 0} \theta(\mathtt{mu}^{n,e}_{s} \circ \mathtt{mu}^{n,p}_{s})\\
&= \mathtt{id}_{\theta(s)}
\end{aligned}
$$

where the third line follows from Lemma 9.9.4, the fifth from the claim, and the last by Lemma 9.9.3 and the properties of colimits. This completes the induction and hence the proof.

Lemma 9.6.1 now follows directly from Lemma 9.9.5.

Appendix A

WebBoard source code

webboard.wig

```
#include <std.bigmac>
#include <sql.bigmac>

service {
  #include "webboard.doc"

  const int VIEW = 0;
  const int READ = 1;
  const int REPLY = 2;
  const int PREFS = 3;
  const int QUIT = 4;

  protected int no_logins;

  schema Message {
    int id, reply_to;
    string name, email, subject, content;
    time time_stamp;
  }
  protected relation Message board;
  protected int next_id;

  schema User {
    string login, name, email;
    int days;
    bool filter, ampm;
  }
  protected relation User users;

  format Re = concat("Re: ", anything);
  format Digit = range('0', '9');
```

```
format Number = plus(Digit);
format Alpha = union(range('a', 'z'), range('A', 'Z'));
format Word = plus(union(Digit, Alpha));
format Email = concat(Word, "@", Word, star(concat(".", Word)));

session Use() {
  int todo = VIEW;
  tuple User user;
  int id, no_read, no_write, no_login;

  tuple Message getMessage(int id, relation Message B) {
    tuple Message m;
    factor (B) { if (#.id==id) m = #; };
    return m;
  }

  string prettyMonth(time t) {
    return vector { "January", "February", "March", "April", "May", "June", "July",
      "August", "September", "October", "November", "December" }[getmonth(t)-1];
  }

  string twodigits(int n) {
    if (n<10) return "0" + n;
    else return (string) n;
  }

  string prettyTime(time t) {
    string date;

    date = prettyMonth(t) + " " + getday(t) + ", ";
    if (user.ampm) {
      date += ((gethour(t)+11)%12)+1 + ":" + twodigits(getminute(t)) + " ";
      if (gethour(t)<12) date += "am";
      else date += "pm";
    } else date += twodigits(gethour(t)) + ":" + twodigits(getminute(t));
    return date;
  }

  /* ------------------------ LOGIN ---------------------------- */

  tuple User Login() {
    relation User U;
    vector User V;
    string login;
    tuple User user;

    show LoginDoc receive [login = login];
    reader (users) U = users;
    V = (vector User) select * from U where #.login==login;
    if (|V|==0) {
```

```
    /* Add user */
    user = Preferences(tuple {login=login, name="", email="", ampm = false,
                              filter = false, days = 0});
    writer (users) users = Union(users, relation {user});
  } else user = V[0];
  return user;
}

/* ---------------------- VIEW BOARD ------------------------- */

html makeBoard(int id, relation Message B, time expire) {
  int i;
  html H = MainDoc;
  tuple Message message;
  vector Message children;

  if (id!=0) {
    message = getMessage(id,B);
    if (message.time_stamp > expire)
      H = MessageDoc <[id = message.id,
                       subject = message.subject,
                       name = message.name,
                       mailto = "mailto:" + message.email,
                       pretty_time = prettyTime(message.time_stamp)];
  }
  /* retrieve children */
  children = sort((vector Message) select * from B where (#.reply_to==id); id);
  for (i=0; i<|children|; i++) {
    html T = MoreMessagesDoc <[reply = makeBoard(children[i].id,B,expire)];
    H = H <[replies = T];
  }
  return H <[replies = ""]; /* Close */
}

void ViewBoard() {
  time expire;
  relation Message B;

  reader (board) B = board;
  if (user.filter) {
    expire = now();
    setday(expire, getday(expire)-user.days);
  }
  show ShowBoardDoc <[no_msgs = |B|, msgs = makeBoard(0,B,expire)] receive [id = id];
  switch(id) {
  case 0: todo = REPLY; break;
  case -1: todo = VIEW; break;
  case -2: todo = PREFS; break;
  case -3: todo = QUIT; break;
  default: todo = READ; break;
```

```
  }
}

/* -------------------- READ MESSAGE ------------------------ */

void ReadMessage() {
  bool reply;
  tuple Message message;
  relation Message B;

  reader (board) B = board;
  message = getMessage(id,B);
  no_read++;
  show ReadMessageDoc <[name = message.name,
                        subject = message.subject,
                        content = message.content,
                        mailto = "mailto:" + message.email,
                        pretty_time = prettyTime(message.time_stamp)]
    receive [reply = reply];
  if (reply) todo = REPLY;
  else todo = VIEW;
}

/* ------------------- REPLY MESSAGE ------------------------ */

string addReply(string s) {
  if (Re(s)) return s;
  else return "Re: "+s;
}

void ReplyMessage(int reply_to) {
  bool send;
  relation Message B;
  tuple Message New;
  string subject, content;

  if (reply_to!=0) {
    reader (board) B = board;
    subject = addReply(getMessage(reply_to,B).subject);
  }
  show ReplyDoc <[subject = subject] receive [subject = subject,
                                              content = content,
                                              send = send];
  if (send) {
    if (subject=="") subject = "no subject";
    writer (next_id) id = ++next_id;
    New = tuple { id = id,
                  reply_to = reply_to,
                  name = user.name,
                  email = user.email,
```

```
                    subject = subject,
                    content = content,
                    time_stamp = now() };
    no_write++;
    writer (board) board = Union(board, relation { New } );
  }
  todo = VIEW;
}

/* -------------------- PREFERENCES ------------------------- */

tuple User Preferences(tuple User user) {
  html P;
  P = PreferenceDoc <[login = user.login,
                      name = user.name,
                      email = user.email,
                      days = user.days];
  P = P <[filter = (user.filter ? "checked" : "")];
  P = P <[ampm = (user.ampm ? "checked" : "")];
  show P receive [user.name = name,
                  user.email = email,
                  user.ampm = ampm,
                  user.filter = filter,
                  user.days = days];
  todo = VIEW;
  return user;
}

void updateUser() {
  writer (users) {
    users = select * from users where #.login!=user.login;
    users = Union(users, relation{user});
  }
}

/* --------------------- SESSION CODE ----------------------- */

user = Login();
writer(no_logins) no_logins++;
while (todo!=QUIT) {
  switch (todo) {
  case VIEW: ViewBoard(); break;
  case READ: ReadMessage(); break;
  case REPLY: ReplyMessage(id); break;
  case PREFS: user = Preferences(user); updateUser(); break;
  default: exit; break;
  }
}
reader(no_logins) no_login = no_logins;
exit GoodbyeDoc <[name = user.name,
```

```
                                     no_read = no_read,
                                     no_write = no_write,
                                     no_login = no_login];
  }
}
```

webboard.doc

```
const html LoginDoc = <html>
  <h1>WebBoard Login</h1>
  <hr>
  Login: <input type="text" format="Word" name="login">
</html>;

/* ---------------------- PREFERENCES ------------------------ */

const html PreferenceDoc = <html>
  <h1>Preferences</h1>
  <hr>
  <h3>Identity</h3>
  Login: <b><[login]></b><br>
  Name: <input type=text name="name" value=["name"]><br>
  Email: <input type=text name="email" value=["email"] format=Email>
  <hr>
  <h3>Filter</h3>
  Use filter <input type="checkbox" name="filter" value="true" [filter]>
  <br>
  Filter messages older than than
  <input type=text name=days format=Number size=5 value=[days]> days.
  <hr>
  <h3>Appearance</h3>
  Display time using am/pm: <input type=checkbox name=ampm value=true [ampm]><br>
</html>;

/* ---------------------- VIEW BOARD ------------------------- */

const html ShowBoardDoc = <html>
  <h1>WebBoard</h1>
  <hr>
  <continue type=button name=id value=0>New Message</continue>
  <[msgs]>
  <p>
  <continue type=button name=id value="-1">Refresh</continue>
  <continue type=button name=id value="-2">Preferences</continue>
  <continue type=button name=id value="-3">Quit</continue>
  <hr>
  Total number of messages: <[no_msgs]>
</html>;

const html MainDoc = <html>
  <ul><[replies]></ul>
</html>;

const html MessageDoc = <html>
  <li><continue name=id value=[id]><b><[subject]></b></continue> -
    <i><a href=["mailto"]><[name]></a></i> (<[pretty_time]>)
```

```
  <ul><[replies]></ul>
</html>;

const html MoreMessagesDoc = <html>
  <[reply]><[replies]>
</html>;

/* --------------------- READ MESSAGE ------------------------- */

const html ReadMessageDoc = <html>
  <h1><[subject]></h1>
  <hr>
  <pre><[content]></pre>
  <hr><p>
  <a href=[mailto]><[name]></a> (<[pretty_time]>)
  <p>
  <continue type=button name=reply value=true>Reply</continue>
  <continue type=button name=reply value=false>Back</continue>
</html>;

/* -------------------- REPLY MESSAGE ------------------------- */

const html ReplyDoc = <html>
  <h1>Enter Reply</h1>
  Subject:<br>
  <input type="text" name="subject" value=["subject"]>
  <hr>
  Message:<br>
  <textarea name="content" cols=50 rows=20></textarea><br>
  <hr>
  <continue type=button name=send value=true>Submit</continue>
  <continue type=button name=send value=false>Cancel</continue>
</html>;

/* ----------------------- GOODBYE ---------------------------- */

const html GoodbyeDoc = <html>
  <h1>Goodbye <[name]>.</h1>
  Thank you for using the WebBoard.<p>
  You <i>read</i> <[no_read]> message(s) and <i>wrote</i> <[no_write]>.
  <br>
  There has been <[no_login]> <i>login(s)</i> to the WebBoard.
</html>;
```

Bibliography

[1] M. Abadi and M. P Fiore. Syntactic considerations on recursive types. In *Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 242–252, 1996.

[2] S. Abramsky and G McCusker. Call-by-value games. In *Computer Science Logic*, 1997. Submitted for publication.

[3] Samson Abramsky and Achim Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[4] Samson Abramsky, Pascale Malacaria, and Radha Jagadeesan. Full abstraction for PCF (extended abstract). In M. Hagiya and J.C. Mitchell, editors, *Theoretical Aspects of Computer Software*, number 789 in Lect. Notes in Computer Sci., pages 1–15. Springer-Verlag, 1994.

[5] A. Aho, J. Hopcroft, and J. Ullman. *Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[6] Henrik Reif Andersen. Partial model checking. In *Proceeding of the Tenth Annual Symposium on Logic in Computer Science, LICS'95*, pages 398–407. IEEE, 1995.

[7] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *Computing Surveys*, 15(1):3–43, March 1983.

[8] Egidio Astesiano and Gerardo Costa. Nondeterminism and fully abstract models. *RAIRO*, 14(4):323–347, 1980.

[9] David Atkins, Thomas Ball, Michael Benedikt, Glenn Bruns, Kenneth Cox, Peter Mataga, and Kenneth Rehor. Experience with a domain specific language for form-based services. In *Usenix Conference on Domain Specific Languages*, Santa Barbara, CA, October 1997.

[10] Leon Atkinson. *Core PHP Programming.* Prentice Hall Computer Books, May 1999.

[11] F. Balarin and A.L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer Aided Verification, CAV '93, LNCS 697*, pages 29–40, 1993.

[12] D. Basin and N. Klarlund. Automata based symbolic reasoning in hardware verification. To appear in *Formal Methods In System Design*. Extended version of: "Hardware verification using monadic second-order logic," *CAV '95*, LNCS 939, 1998.

[13] Morten Biehl, Nils Klarlund, and Theis Rauhe. Mona: decidable arithmetic in practice (demo). In *Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium, Uppsala, LNCS 1135*, 1996.

[14] Claus Brabrand. Synthesizing safety controllers for interactive Web services. Master's thesis, Department of Computer Science, University of Aarhus, December 1998. Available from `http://www.brics.dk/~brabrand/thesis/`.

[15] Claus Brabrand, Anders Møller, Anders Sandholm, and Michael I. Schwartzbach. <bigwig> — a language for developing interactive Web services. Submitted for publication.

[16] Claus Brabrand, Anders Møller, Anders Sandholm, and Michael I. Schwartzbach. A runtime system for interactive Web services. *Computer Networks*, 31:1391–1401, 1999. Also in Proceedings of the Eighth International World Wide Web Conference.

[17] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[18] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[19] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing surveys*, 24(3):293–318, September 1992.

[20] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[21] J.R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.

[22] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[23] J. Cai and R. Paige. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science*, 145(1-2):189–228, July 1995.

[24] A. Church. Logic, arithmetic, and automata. In Almquist and Wiksells, editors, *International Symposium of Mathematicians*, pages 23–35, 1962.

[25] Edmund M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[26] Rance Cleveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. Technical Report ECS-LFCS-89-83, Department of Computer Science, University of Edinburgh, 1989.

[27] M-M Corsini and A. Rauzy. Symbolic model checking and constraint logic programming: a cross-fertilisation. In *5th. Europ. Symp. on Programming, LNCS 788*, pages 180–194, 1994.

[28] K. Cox, T. Ball, and J. C. Ramming. Lunchbot: A tale of two ways to program web services. Technical Report BL0112650-960216-06TM, AT&T Bell Laboratories, 1996.

[29] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. John Wiley & Sons, 1986.

[30] P.-L. Curien and A. Obtułowicz. Partiality, cartesian closedness, and toposes. *Information and Computation*, 80:50–95, 1989.

[31] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[32] C.C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98:21–52, 1961.

[33] Michael I. Schwartzbach *et al.* `<bigwig>` project homepage. `http://www.brics.dk/bigwig/`.

[34] Alex Fedorov, Richard Harrison, Dave Sussman, Brian Francis, and Stephen Wood. *Professional Active Server Pages 2.0*. Wrox Press Inc., 2nd edition, March 1998.

[35] M. Fiore and A. Simpson. Lambda definability with sums via grothendieck logical relations. In *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 1999. To appear.

[36] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[37] Brian J. Fox. Meta-html: A dynamic programming language for www applications. www.metahtml.com/documentation/manifesto.html.

[38] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92. North-Holland, 1971.

[39] Patrice Godefroid. *Partial-order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1996.

[40] Shishir Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly & Associates, Inc., 1996.

[41] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[42] Carl A. Gunter and Dana S. Scott. Semantic domains. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 633–674. Elsevier, 1990.

[43] A. Gupta and A.L. Fisher. Parametric circuit representation using inductive boolean functions. In *Computer Aided Verification, CAV '93, LNCS 697*, pages 15–28, 1993.

[44] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1996.

[45] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[46] Kohei Honda and Nobuko Yoshida. Game theoretic analysis of call-by-value computation. In *Automata, Languages and Programming: 24^{th} International Colloquium*, Lect. Notes in Computer Sci. Springer-Verlag, 1997.

[47] J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and Paz A., editors, *Theory of machines and computations*, pages 189–196. Academic Press, 1971.

[48] The document *is* the application. Web Site. www.htmlscript.com.

[49] Thomas Hune. Modelling a real-time language. In *Proceedings of Fourth International Workshop on Formal Methods for Industrial Critical Systems*, 1999.

[50] Thomas Hune and Anders Sandholm. A case study on using automata in control synthesis. In *Fundamental Approaches to Software Engineering*, number LNCS 1783 in Lecture Notes in Computer Science, pages 349–362. Springer-Verlag, March/April 2000.

[51] Jason Hunter, William Crawford, and Paula Ferguson. *Java Servlet Programming*. O'Reilly & Associates, December 1998.

[52] J. M. E. Hyland and C.-H. Luke Ong. Pi-calculus, dialogue games and PCF. In *7th Annual ACM Conference on Functional Programming Languages and Computer Architecture, La Jolla, California*, 1995.

[53] J. Jensen, M. Jørgensen, and N. Klarlund. Monadic second-order logic for parameterized verification. Technical report, BRICS Report Series RS-94-10, Department of Computer Science, University of Aarhus, 1994.

[54] Jacob L. Jensen, Michael E. Joergensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *PLDI '97*, 1997.

[55] K. Jensen and G. Rozenberg, editors. *High-level Petri Nets – Theory and Application*. Springer-Verlag, 1991.

[56] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International series in computer science. Prentice-Hall, 1987.

[57] A. Jung and A. Stoughton. Studying the fully abstract model of PCF within its continuous function model. In *Typed Lambda Calculi and Applications*, volume 664 of *Lect. Notes in Computer Sci.*, pages 230–244. Springer-Verlag, 1993.

[58] A. Jung and J. Tiuryn. A new characterization of lambda definability. In *Typed Lambda Calculi and Applications*, volume 664 of *Lect. Notes in Computer Sci.*, pages 245–257. Springer-Verlag, 1993.

[59] Achim Jung, Marcelo Fiore, Eugenio Moggi, Peter W. O'Hearn, Jon G. Riecke, Giuseppe Rosolini, and Ian Stark. Domains and denotational semantics: History, accomplishments, and open problems. *Bulletin of the European Association for Theoretical Computer Science*, pages 227–256, June 1996.

[60] N. Klarlund and A. Møller. *MONA Version 1.3 User Manual*. BRICS Notes Series NS-98-3 (2.revision), Department of Computer Science, University of Aarhus, October 1998.

[61] N. Klarlund, M. Nielsen, and K. Sunesen. A case study in automated verification based on trace abstractions. In M. Broy, S. Merz, and K. Spies, editors, *Formal System Specification, The RPC-Memory Specification Case Study*, volume 1169 of *LNCS*, pages 341–374. Springer Verlag, 1996.

[62] Nils Klarlund. Mona & Fido: the logic-automaton connection in practice. In *CSL '97 Proceedings*. LNCS 1414, Springer-Verlag, 1998.

[63] Nils Klarlund, Jari Koistinen, and Michael I. Schwartzbach. Formal design constraints. In *Proceedings of OOPSLA '96, The Eleventh Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 1996.

[64] Nils Klarlund and Michael I. Schwartzbach. A domain-specific language for regular sets of strings and trees. In *Conference on Domain-Specific Languages*, pages 145–156. USENIX, 1997.

[65] Orna Kupferman. *Model Checking for Branching-Time Temporal Logics*. PhD thesis, Technion, Israil Institute of Technology, 1995 1995.

[66] B. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proc. Eigth Symp. Princ. of Distributed Computing*, pages 239–247, 1989.

[67] D. A. Ladd and J. C. Ramming. Programming the Web: An application-oriented language for hypermedia services. In *4th Intl. World Wide Web Conference*, 1995.

[68] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6, 1964.

[69] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *In Springer International Journal of Software Tools for Technology Transfer*, 1(1+2), 1997.

[70] Kim G. Larsen. Proof systems for hennessy-milner logic with recursion. In *Proceedings of CAAP*, number 299 in LNCS, pages 215–230, Nancy, France, March 1988. Springer-Verlag.

[71] Kim S. Larsen, Michael I. Schwartzbach, and Erik M. Schmidt. A new formalism for relational algebra. *Information Processing Letters*, 41:163–168, 1992.

[72] LEGO. *Software developers kit*, November 1998. See `http://www.legomindstorms.com/`.

[73] Michael R. Levy. Web programming in Guide. *Software—Practice and Experience*, 28(15):1581–1603, December 1998.

[74] R. Loader. Finitary PCF is not decidable. Unpublished manuscript available from `http://www.dcs.ed.ac.uk/~loader`, 1997.

[75] P. Madhusudan and P. S. Thiagarajan. Controllers for discrete event systems via morphisms. In *CONCUR'98*, Lecture Notes in Computer Science, LNCS. Springer-Verlag, August 1998.

[76] Z. Manna and A. Pnueli. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, January 1984.

[77] M Marz. A fully abstract model for sequential computation. Technical Report Technical Report CSR-98-06, Department of Computer Science, University of Birmingham, 1998.

[78] O. Matz, A. Miller, A. Potthoff, W. Thomas, and E. Valkema. Report on the program AMoRe. Technical Report 9507, Institut für Informatik und Praktische Mathematik der Christian-Albrechts-Universität zu Kiel, October 1995.

[79] S. Mauw and M. A. Reniers. An algebraic semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.

[80] Open Market, Inc. FastCGI: A high-performance Web server interface. Technical White Paper, www.fastcgi.com, April 1996.

[81] G. McCusker. Games and definability for FPC. *Bulletin of Symbolic Logic*, 3(3):347–362, 1997.

[82] Guy McCusker. Games and full abstraction for FPC. In *Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 174–183, 1996.

[83] Albert R. Meyer. Semantical paradigms: Notes for an invited lecture, with two appendices by Stavros S. Cosmadakis. In *Proceedings, Third Annual Symposium on Logic in Computer Science*, pages 236–255. IEEE, 1988.

[84] R. Milner, M. Tofte, R. Harper, and D MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[85] Robin Milner. Fully abstract models of the typed lambda calculus. *Theoretical Computer Sci.*, 4:1–22, 1977.

[86] Robin Milner. *Communication and Concurrency.* Prentice Hall International Series in Computer Science. Prentice Hall International, 1989.

[87] Eugenio Moggi. Notions of computation and monads. *Information and Control*, 93:55–92, 1991.

[88] Anders Møller. MONA project homepage. `http://www.brics.dk/mona/`.

[89] Ketan Mulmuley. *Full Abstraction and Semantic Equivalence.* ACM Doctoral Dissertation Award 1986. MIT Press, 1987.

[90] Hanno Nickau. Hereditarily sequential functionals. In *Proceedings of the Symposium on Logical Foundations of Computer Science: Logic at St. Petersburg*, Lect. Notes in Computer Sci. Springer-Verlag, 1994.

[91] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis: Flows and Effects*. Springer, 1999.

[92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, 1992.

[93] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda calculus. Unpublished manuscript available from `http://www.dcs.qmw.ac.uk/~ohearn`, 1997.

[94] Peter W. O'Hearn and Jon G. Riecke. Kripke logical relations and PCF. *Information and Computation*, 120(1):107–116, 1995.

[95] Peter W. O'Hearn and Robert D. Tennent. Parametricity and local variables. *J. ACM*, 42:658–709, 1995.

[96] R. Paige and R. Tarjan. Three efficient algorithms based on partition refinement. *SIAM Journal of Computing*, 16(6), 1987.

[97] Andrew M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.

[98] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Sci.*, 5:223–257, 1977.

[99] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus Univ., Computer Science Dept., Denmark, 1981.

[100] Gordon D. Plotkin. (Towards a) logic for computable functions. Unpublished manuscript, CSLI Summer School Notes, 1985.

[101] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[102] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.

[103] Peter J. G. Ramadge and W. Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[104] D. Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992.

[105] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM National Meeting (Boston, 1972)*, pages 717–740, 1972.

[106] John C. Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

[107] John C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981.

[108] J-K. Rho and F. Somenzi. Automatic generation of network invariants for the verification of iterative sequential systems. In *Computer Aided Verification, CAV '93, LNCS 697*, pages 123–137, 1993.

[109] Jon G. Riecke. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science*, 3:387–415, 1993. Preliminary version appears in *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 245–254, ACM, 1991.

[110] Jon G. Riecke and Anders Sandholm. A relational account of call-by-value sequentiality. In *Proceedings of the Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 258–267. IEEE Computer Society Press, 1997. Also available as BRICS Technical Report RS-97-41.

[111] Jon G. Riecke and Anders Sandholm. A relational account of call-by-value sequentiality. *To appear in Information and Computation*, 2000. Preliminary version appeared in Proceedings of the Twelfth Annual IEEE Symposium on Logic in Computer Science.

[112] Jon G. Riecke and R Subrahmanyam. Semantic orthogonality of type disciplines. Unpublished manuscript available from `http://www.cs.bell-labs.com/who/riecke/`, 1997.

[113] Jon G. Riecke and Ramesh Viswanathan. Full abstraction for call-by-value sequential languages. Unpublished manuscript, 1993.

[114] Jon G. Riecke and Ramesh Viswanathan. Isolating side effects in sequential languages. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 1–12. ACM, 1995.

[115] Anders Sandholm. Mona-based control synthesis. BRICS, University of Aarhus, 1999.

[116] Anders Sandholm and Michael I. Schwartzbach. Distributed safety controllers for Web services. In Egidio Astesiano, editor, *Fundamental Approaches to Software Engineering, FASE'98*, Lecture Notes in Computer Science, LNCS 1382, pages 270–284. Springer-Verlag, March/April 1998. Also available as BRICS Technical Report RS-97-47.

[117] Anders Sandholm and Michael I. Schwartzbach. A type system for dynamic Web documents. In *Principles of Programming Languages*, January 2000.

[118] Dana Scott. A type theoretical alternative to CUCH, ISWIM, OWHY. Unpublished manuscript, Oxford University, 1969.

[119] Dana Scott. A type theoretical alternative to CUCH, ISWIM, OWHY. *Theoretical Computer Sci.*, 121:411–440, 1993. Published version of unpublished manuscript, Oxford University, 1969.

[120] Dana S. Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. In J. Fox, editor, *Proceedings of the Symposium on Computers and Automata*, pages 19–46, New York, 1971. Polytechnic Institute of Brooklyn Press.

[121] Kurt Sieber. Reasoning about sequential functions via logical relations. In *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1992.

[122] D. Sieling and I. Wegener. Reduction of OBDDs in linear time. *IPL*, 48:139–144, 1993.

[123] A. P. Sistla and E. M. Clarke. The complexity of Propositional Linear Temporal Logic. *Journal of the ACM*, 32(3):733–749, 1985.

[124] M. Steinmann. Übersetzung von logischen Ausdrücken in Baumautomaten: Entwicklung eines Verfahrens und seine Implementierung. Unpublished, 1993.

[125] Colin P. Stirling and David Walker. Local model checking in the modal mu-calculus. Technical Report ECS-LFCS-89-78, LFCS, Department of Computer Science, University of Edinburgh, May 1989.

[126] Allen Stoughton. *Fully Abstract Models of Programming Languages*. Research Notes in Theoretical Computer Science. Pitman/Wiley, 1988. Revision of Ph.D thesis, Dept. of Computer Science, Univ. Edinburgh, Report No. CST-40-86, 1986.

[127] P. S. Thiagarajan. PTL over product state spaces. Technical Report TCS-95-4, School of Mathematics, SPIC Science Foundation, 1995.

[128] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.

[129] Antti Valmari. Stubborn sets for reduced state space generation. In G. Rozenberg, editor, *Advances in Petri Nets*, number 483 in LNCS, pages 491–515. Springer-Verlag, 1990.

[130] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings, Symposium on Logic in Computer Science*, pages 332–344, June 1986.

[131] Larry Wall, Tom Christensen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., second edition, September 1996.

[132] Wolgang Wechler. *Universal Algebra for Computer Scientists*. Number 25 in EATCS Monographs in Theoretical Computer Science. Springer-Verlag, 1992.

[133] Glynn Winskel. *The Formal Sematics of Programming Languages An Introduction.* Foundations of Computing. The MIT Press, 1993.

[134] W. Zielonka. *The Book of Traces*, chapter Asynchronous Automata, pages 205–248. World Scientific Publishing, 1995.

Recent BRICS Dissertation Series Publications

DS-00-2 Anders B. Sandholm. *Programming Languages: Design, Analysis, and Semantics*. February 2000. PhD thesis. xiv+233 pp.

DS-00-1 Thomas Troels Hildebrandt. *Categorical Models for Concurrency: Independence, Fairness and Dataflow*. February 2000. PhD thesis. x+141 pp.

DS-99-1 Gian Luca Cattani. *Presheaf Models for Concurrency*. April 1999. PhD thesis. xiv+255 pp.

DS-98-3 Kim Sunesen. *Reasoning about Reactive Systems*. December 1998. PhD thesis. xvi+204 pp.

DS-98-2 Søren B. Lassen. *Relational Reasoning about Functions and Nondeterminism*. December 1998. PhD thesis. x+126 pp.

DS-98-1 Ole I. Hougaard. *The CLP(OIH) Language*. February 1998. PhD thesis. xii+187 pp.

DS-97-3 Thore Husfeldt. *Dynamic Computation*. December 1997. PhD thesis. 90 pp.

DS-97-2 Peter Ørbæk. *Trust and Dependence Analysis*. July 1997. PhD thesis. x+175 pp.

DS-97-1 Gerth Stølting Brodal. *Worst Case Efficient Data Structures*. January 1997. PhD thesis. x+121 pp.

DS-96-4 Torben Braüner. *An Axiomatic Approach to Adequacy*. November 1996. Ph.D. thesis. 168 pp.

DS-96-3 Lars Arge. *Efficient External-Memory Data Structures and Applications*. August 1996. Ph.D. thesis. xii+169 pp.

DS-96-2 Allan Cheng. *Reasoning About Concurrent Computational Systems*. August 1996. Ph.D. thesis. xiv+229 pp.

DS-96-1 Urban Engberg. *Reasoning in the Temporal Logic of Actions — The design and implementation of an interactive computer system*. August 1996. Ph.D. thesis. xvi+222 pp.