Reiko Heckel
Stefan Milius (Eds.)

LNCS 8089

Algebra and Coalgebra in Computer Science

5th International Conference, CALCO 2013
Warsaw, Poland, September 2013
Proceedings

Springer

Lecture Notes in Computer Science 8089

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Reiko Heckel Stefan Milius (Eds.)

Algebra and Coalgebra in Computer Science

5th International Conference, CALCO 2013
Warsaw, Poland, September 3-6, 2013
Proceedings

Springer

Volume Editors

Reiko Heckel
University of Leicester, Department of Computer Science
University Road
Leicester, LE1 7RH, UK
E-mail: reiko@mcs.le.ac.uk

Stefan Milius
Friedrich-Alexander Universität Erlangen-Nürnberg
Lehrstuhl für Theoretische Informatik
Martenstr. 3
91058 Erlangen-Nürnberg, Germany
E-mail: stefan.milius@fau.de

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-40205-0 e-ISBN 978-3-642-40206-7
DOI 10.1007/978-3-642-40206-7
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013944581

CR Subject Classification (1998): F.4, F.3, D.2, F.1, F.2, I.1, G.2, G.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

CALCO, the International Conference on Algebra and Coalgebra in Computer Science, is a high-level, bi-annual event formed by joining CMCS (the International Workshop on Coalgebraic Methods in Computer Science) and WADT (the Workshop on Algebraic Development Techniques). CALCO aims to bring together researchers and practitioners with interests in foundational aspects, and both traditional and emerging uses of algebras and coalgebras in computer science. The study of algebra and coalgebra relates to the data, process, and structural aspects of software systems.

Previous CALCO editions took place in Swansea (UK, 2005), Bergen (Norway, 2007), Udine (Italy, 2009) and Winchester (UK, 2011). CALCO 2013, the fifth conference in the series, took place in Warsaw (Poland), September 3–6, 2013.

CALCO 2013 received 33 submissions, out of which 18 were selected for presentation at the conference. The standard of submissions was generally very high. The selection process was carried out by the Program Committee, taking into account the originality, quality, and relevance of the material presented in each submission, based on the opinions of three or four expert reviewers for each submission. The selected and revised papers are included in this volume, together with contributions by the invited speakers Andrej Bauer, Mikołaj Bojańczyk, Neil Ghani, and Damien Pous.

CALCO 2013 was co-located with two workshops. The CALCO Early Ideas Workshop, CALCO EI, was dedicated to presentation of work in progress and original research proposals. PhD students and young researchers were particularly encouraged to contribute. CALCO EI was organized by Monika Seisenberger. The CALCO Tools Workshop, organized by Lutz Schröder, is dedicated to tools based on algebraic and/or coalgebraic principles. These tool papers also appear in this volume.

We wish to thank all the authors for submitting their papers to CALCO 2013, the Program Committee for its diligent work in the selection process, and the external reviewers for their support in evaluating papers.

We are grateful to the University of Warsaw and the Polish Mathematical Society for hosting CALCO 2013 and to the Organizing Committee, chaired by Bartek Klin and Andrzej Tarlecki, for all the local arrangements. We also thank the Warsaw Centre of Mathematics and Computer Science for their financial support. At Springer, Alfred Hofmann and his team supported the publishing process. We gratefully acknowledge the use of EasyChair, the conference management system by Andrei Voronkov.

June 2013

Reiko Heckel
Stefan Milius

Organization

CALCO Steering Committee

Jiří Adámek	Technical University of Braunschweig, Germany
Michel Bidoit	CNRS and ENS de Cachan, France
Corina Cîrstea	University of Southampton, UK
Andrea Corradini	University of Pisa, Italy
José Luiz Fiadeiro	Royal Holloway, University of London, UK
H. Peter Gumm (Co-chair)	Philipps University Marburg, Germany
Rolf Hennicker	Ludwig-Maximilians-University Munich, Germany
Bart Jacobs	Radboud University Nijmegen, Germany
Bartek Klin	University of Warsaw, Italy
Hans-Jörg Kreowski	University of Bremen, Germany
Alexander Kurz	University of Leicester, UK
Marina Lenisa	University of Udine, Italy
Ugo Montanari	University of Pisa, Italy
Larry Moss	Indiana University, Bloomington, USA
Till Mossakowski (Co-chair)	DFKI Lab Bremen and University of Bremen, Germany
Fernando Orejas	Politechnical University Catalunia, Barcelona, Spain
Francesco Parisi-Presicce	University of Rome La Sapienza, Italy
Dirk Pattinson	Australian National University, Australia
John Power	University of Bath, UK
Horst Reichel	Technical University of Dresden, Germany
Jan Rutten	CWI Amsterdam and Radboud University Nijmegen, The Netherlands
Lutz Schröder	Friedrich-Alexander University Erlangen-Nürnberg, Germany
Andrzej Tarlecki	University of Warsaw, Poland

Program Committee

Luca Aceto	Reykjavik University, Iceland
Jiří Adámek	Technical University of Braunschweig, Germany
Lars Birkedal	IT University of Copenhagen, Denmark
Filippo Bonchi	CNRS, ENS-Lyon, France

Corina Cîrstea	University of Southhampton, UK
Bob Coecke	University of Oxford, UK
Andrea Corradini	University of Pisa, Italy
Mai Gehrke	Université Paris Diderot – Paris 7, France
H. Peter Gumm	Philipps University Marburg, Germany
Gopal Gupta	University of Texas at Dallas, USA
Ichiro Hasuo	Tokyo University, Japan
Reiko Heckel (Co-chair)	University of Leicester, UK
Bart Jacobs	Radboud University Nijmegen, The Netherlands
Ekaterina Komendantskaya	University of Dundee, Scotland, UK
Barbara König	University of Duisburg-Essen, Germany
José Meseguer	University of Illinois, Urbana-Champaign, USA
Marino Miculan	University of Udine, Italy
Stefan Milius (Co-chair)	Friedrich-Alexander University Erlangen-Nürnberg, Germany
Larry Moss	Indiana University, Bloomington, USA
Till Mossakowski	DFKI Lab Bremen and University of Bremen, Germany
Prakash Panangaden	McGill University, Montreal, Canada
Dirk Pattinson	Australian National University, Australia
Dusko Pavlovic	Royal Holloway, University of London, UK
Daniela Petrişan	University of Leicester, UK
John Power	University of Bath, UK
Jan Rutten	CWI Amsterdam and Radboud University Nijmegen, The Netherlands
Lutz Schröder	Friedrich-Alexander University Erlangen-Nürnberg, Germany
Monika Seisenberger	Swansea University, UK
Alexandra Silva	Radboud University Nijmegen and CWI Amsterdam, The Netherlands
Paweł Sobociński	University of Southhampton, UK
Sam Staton	University of Cambridge, UK
Yde Venema	University of Amsterdam, The Netherlands
Uwe Wolter	University of Bergen, Norway

Additional Reviewers

Giorgio Bacci	Mihai Codescu	Giuseppe Greco
Ulrich Berger	Josée Desharnais	Helle Hvid Hansen
Arnar Birgisson	Brian Devries	Chris Heunen
Aleš Bizjak	Ross Duncan	Tom Hirschowitz
Roberto Bruni	Murdoch Gabbay	Jean-Baptiste Jeannin
Martin Churchill	Rajeev Goré	Henning Kerstan

Aleks Kissinger
Robbert Krebbers
Sebastian Küpper
Clemens Kupke
Alberto Lluch Lafuente
Guy McCusker
Robert Myers
Marco Peressotti
Vaughan Pratt
Joshua Sack
Mehrnoosh Sadrzadeh
Isar Stubbe
Viktor Winschel
Joost Winter
Fabio Zanasi

Table of Contents

Invited Talks

An Effect System for Algebraic Effects and Handlers 1
Andrej Bauer and Matija Pretnar

Automata and Algebras for Infinite Words and Trees 17
Mikołaj Bojańczyk

Positive Inductive-Recursive Definitions 19
Neil Ghani, Lorenzo Malatesta, and Fredrik Nordvall Forsberg

Coalgebraic Up-to Techniques 34
Damien Pous

Contributed Papers

Exploiting Algebraic Laws to Improve Mechanized Axiomatizations 36
Luca Aceto, Eugen-Ioan Goriac, Anna Ingolfsdottir, Mohammad Reza Mousavi, and Michel A. Reniers

Positive Fragments of Coalgebraic Logics 51
Adriana Balan, Alexander Kurz, and Jiří Velebil

Many-Valued Relation Lifting and Moss' Coalgebraic Logic 66
Marta Bílková and Matěj Dostál

Saturated Semantics for Coalgebraic Logic Programming 80
Filippo Bonchi and Fabio Zanasi

Presenting Distributive Laws 95
Marcello M. Bonsangue, Helle Hvid Hansen, Alexander Kurz, and Jurriaan Rot

Interaction and Observation: Categorical Semantics of Reactive Systems Trough Dialgebras 110
Vincenzo Ciancia

Homomorphisms of Coalgebras from Predicate Liftings 126
Sebastian Enqvist

From Kleisli Categories to Commutative C^*-Algebras: Probabilistic Gelfand Duality 141
Robert Furber and Bart Jacobs

Trace Semantics via Generic Observations 158
Sergey Goncharov

Full Abstraction for Fair Testing in CCS 175
Tom Hirschowitz

A Simple Case of Rationality of Escalation 191
Pierre Lescanne

Coalgebras with Symmetries and Modelling Quantum Systems 205
Daniel Marsden

From Operational Chu Duality to Coalgebraic Quantum Symmetry 220
Yoshihiro Maruyama

Noninterfering Schedulers: When Possibilistic Noninterference Implies Probabilistic Noninterference 236
Andrei Popescu, Johannes Hölzl, and Tobias Nipkow

Simulations and Bisimulations for Coalgebraic Modal Logics 253
Daniel Gorín and Lutz Schröder

A Coalgebraic View of ε-Transitions 267
Alexandra Silva and Bram Westerbaan

Nets, Relations and Linking Diagrams 282
Paweł Sobociński

A Logic-Programming Semantics of Services 299
Ionuţ Ţuţu and José Luiz Fiadeiro

CALCO-Tools Workshop

Preface to CALCO-Tools 314
Lutz Schröder

Checking Conservativity with Hets 315
Mihai Codescu, Till Mossakowski, and Christian Maeder

The HI-Maude Tool 322
Muhammad Fadlisyah and Peter Csaba Ölveczky

Constructor-Based Inductive Theorem Prover 328
Daniel Găină, Min Zhang, Yuki Chiba, and Yasuhito Arimoto

A Timed CTL Model Checker for Real-Time Maude 334
Daniela Lepri, Erika Ábrahám, and Peter Csaba Ölveczky

Hybridisation at Work 340
Renato Neves, Alexandre Madeira, Manuel A. Martins, and Luís S. Barbosa

`Penrose`: Putting Compositionality to Work for Petri Net Reachability 346
Paweł Sobociński and Owen Stephens

QStream: A Suite of Streams 353
Joost Winter

Author Index 359

An Effect System for Algebraic Effects and Handlers

Andrej Bauer and Matija Pretnar

Faculty of Mathematics and Physics
University of Ljubljana

Abstract. We present an effect system for algebraic effects and handlers. Because handlers may transform an effectful computation into a pure one, the effect system is non-monotone in the sense that effects do not just accumulate, but may also be deleted from types or generally transformed. We also provide denotational semantics for the effect system, based on a domain-theoretic model with partial equivalence relations. The semantics validates equational reasoning about effectful computations.

1 Introduction

An *effect system* supplements a traditional type system for a programming language with information about which computational effects may, will, or will not happen when a piece of code is executed. A well designed and solidly implemented effect system helps the programmer understand the code and find mistakes, but it can also be used to safely rearrange, optimize, and parallelize code [1,2]. As many before us [1,3,4,5] we take on the task of striking just the right balance between simplicity and expressivity by devising an effect system for the programming language Eff [6]. The novelty here is that Eff has not only *first-class algebraic effects* [7], but also *effect handlers* [8]. Therefore, an effect system for Eff is by its nature *non-monotone* — an effectful computation may become pure when enclosed by a handler — so effects do not just accumulate in the types, but also get deleted and generally transformed. Another feature of our effect system is that its denotational semantics validates *equational* reasoning, which is traditionally thought to be in the dominion of pure languages, and can be tricky once effects are included [9].

The paper is organized as follows. In §2 we describe the *core Eff* and an effect system for it. In §3 we give a denotational semantics for core Eff, and use it to validate program transformation rules that can be used for equational reasoning about effectful computations.

2 Core Eff

Eff is a ML-style [10,11] programming language. Effects in ML are not visible in the types. For example, inserting a print statement in the middle of code does

R. Heckel and S. Milius (Eds.): CALCO 2013, LNCS 8089, pp. 1–16, 2013.

not create any changes in the typing information. In contrast, in the monadic style such a change would taint all enclosing types with the I/O monad [12]. It is our intention to augment the types with information about computational effects which is unobtrusive for programmers, i.e., in the implementation we envision *effect inference* which serves primarily as a program analysis tool.

In ML-style languages effects are provided through built-in functions and special-purpose constructs such as exceptions and references. Eff is based on the *algebraic* approach in which effects are accessed uniformly and exclusively through *operations*, which are a primitive concept. Examples of operations are reading and writing on a communication channel, updating and looking up the contents of a reference, and raising an exception. Thus, in Eff each terminating computation results either in an (effect-free) value, or it triggers an operation. Each operation has an associated (delimited) *continuation*, which is a suspended computation expecting the result of the operation and doing whatever is to be done after the operation is performed.

Operations by themselves do not actually perform effects. Instead their behavior is controlled by a second primitive notion, the *effect handlers*. These are a direct generalization of exception handlers to other operations. The most significant difference between exception handlers and effect handlers is that the latter have access to the continuation of the handled operation. With handlers we may implement a great variety of computational effects, such as transactional memory, various non-deterministic execution strategies, stream redirection, cooperative multi-threading, delimited continuations, and others.

The current implementation of Eff includes a number of features, such as syntactic sugar, products, records, inductive types, type definitions, effect definitions, etc., which are inessential for a conceptual analysis. We therefore restrict attention to *core Eff* whose syntax is shown below.

Types

$$\begin{aligned}
\text{Effect type } E &::= \texttt{effect}\ (\texttt{operation}\ \mathtt{op}_i : A_i \to B_i)_i\ \texttt{end} \\
\text{Expression type } A, B &::= \texttt{nat} \mid \texttt{bool} \mid \texttt{unit} \mid \texttt{empty} \mid E^{\{\iota_1, \ldots, \iota_n\}} \mid \\
&\qquad A \to \underline{C} \mid \underline{C} \Rightarrow \underline{D} \\
\text{Computation type } \underline{C}, \underline{D} &::= A ! \{\iota_1 \,\#\, \mathtt{op}_1, \ldots, \iota_n \,\#\, \mathtt{op}_n\}
\end{aligned}$$

Terms

$$\begin{aligned}
\text{Expression } e &::= x \mid 0 \mid \texttt{succ}\ e \mid \texttt{true} \mid \texttt{false} \mid \\
&\qquad () \mid \iota \mid \texttt{fun}\ x \mapsto c \mid e \,\#\, \mathtt{op} \mid h \\
\text{Handler } h &::= \texttt{handler}\ \texttt{val}\ x \mapsto c_v \mid (e_i \,\#\, \mathtt{op}_i\ x_i\ k_i \mapsto c_i)_i \\
\text{Computation } c &::= \texttt{val}\ e \mid \texttt{let}\ x = c_1\ \texttt{in}\ c_2 \mid \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2 \mid \\
&\qquad \texttt{iszero}\ e \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{absurd}\ e \mid e_1\ e_2 \mid \\
&\qquad \texttt{with}\ e\ \texttt{handle}\ c
\end{aligned}$$

We describe informally what the various parts mean, and refer the readers to [6] for a more thorough introduction. Eff uses a fine grain call-by-value evaluation strategy [13], which means that it distinguishes effect-free *expressions* and possibly effectful *computations*. There are corresponding *expression* and *computation types*.

2.1 Expressions

An expression is either a variable x, zero 0, a successor $\texttt{succ}\ e$, a boolean value `true` or `false`, the unit `()`, an effect instance ι, a function abstraction of a computation, an operation $e\,\texttt{\#}\,\texttt{op}$, or a handler transforming computations to computations. We briefly comment on the ones that are peculiar to Eff.

An *effect type*

$$\texttt{effect}\ (\texttt{operation}\ \texttt{op}_i : A_i \to B_i)_i\ \texttt{end}$$

declares operations $\texttt{op}_i$ with given types of parameters A_i and results B_i. (Here and elsewhere, $(\cdots)_i$ indicates that $\cdots$ may be repeated finitely many times.) For example, the effect type of exceptions is

```
effect
    operation abort : unit → empty
end
```

and the effect type `ref` for a reference holding a natural number is

```
effect
    operation lookup : unit → nat
    operation update : nat → unit
end
```
(1)

Effect instances ι are a way of making several copies of the same computational effect. For example, there may be several communication channels, several mutable references, etc. In this respect Eff differs from [5], where bare operations are considered; in terms of Eff that is like having a single instance of each effect.

The expression type $E^{\{\iota_1,\ldots,\iota_n\}}$ is inhabited by expressions which evaluate to one of the instances $\iota_1, \ldots, \iota_n$, whose effect type is E. We call $\{\iota_1, \ldots, \iota_n\}$ a *region* and abbreviate it as ρ. A smaller region is more informative, so ideally we would like all of them to be just singletons. But this is not possible because instances are first-class values and so we can write

$$\texttt{let}\ x = (\texttt{if}\ b\ \texttt{then}\ \iota_1\ \texttt{else}\ \iota_2)\ \texttt{in}\ \cdots$$

The best we can say about x is that its type is $E^{\{\iota_1,\iota_2\}}$.

2.2 Computations

A *computation type* $A!\{\iota_1 \,\#\, \mathtt{op}_1, \ldots, \iota_n \,\#\, \mathtt{op}_n\}$ means that the computation either produces a (pure) value of type A, or triggers one of the listed operations $\iota_1 \,\#\, \mathtt{op}_1, \ldots, \iota_n \,\#\, \mathtt{op}_n$. We abbreviate such finite sets of operations with the letter δ and call them *dirt*. A computation is either a pure value $\mathtt{val}\ e$, a $\mathtt{let}$ binding, a recursive function definition, a zero-test $\mathtt{iszero}\ e$, a conditional statement, a destructor for the $\mathtt{empty}$ type, an application, or a $\mathtt{handle}$ construct.

The computation $\mathtt{val}\ e$ is pure and indicates a "final" result e, while an operation applied to a parameter $\iota \,\#\, \mathtt{op}\, e$ is the principal way of triggering an effect. By itself $\iota \,\#\, \mathtt{op}\, e$ is just a suspended computation with an associated continuation. For an actual effect to take place it has to be handled by a handler, as described below. The continuation associated with $\iota \,\#\, \mathtt{op}\, e$ is $\mathtt{fun}\ x \mapsto \mathtt{val}\ x$. Such operations are known as *generic effects* [7].

A binding $\mathtt{let}\ x = c_1\ \mathtt{in}\ c_2$ is evaluated as follows:

1. if c_1 evaluates to $\mathtt{val}\ e$ then the binding evaluates to c_2 with x bound to e,
2. if c_1 evaluates to an operation $\iota \,\#\, \mathtt{op}\, e$ with continuation κ, then the binding evaluates to $\iota \,\#\, \mathtt{op}\, e$ with continuation $\mathtt{fun}\ y \mapsto (\mathtt{let}\ x = \kappa\, y\ \mathtt{in}\ c_2)$.

It may help to think of $\mathtt{val}$ and $\mathtt{let}$ as being similar to Haskell $\mathtt{return}$ and $\mathtt{do}$, respectively. In ML $\mathtt{val}$ is invisible, while $\mathtt{let}$ is essentially the same as ours.

The handling construct applies a handler to a computation. If h is the *handler*

$$\mathtt{handler}\ \mathtt{val}\ x \mapsto c_v \mid (e_i \,\#\, \mathtt{op}_i\, x_i\, k_i \mapsto c_i)_i$$

and c is a computation then $\mathtt{with}\ h\ \mathtt{handle}\ c$ first evaluates c and then evaluates the clause of the handler that matches the result given by c. If no clause matches, c evaluated to an operation which is propagated outwards. In all cases the handler wraps itself around the continuation so that subsequent operations are handled as well. More precisely:

1. if c evaluates to $\mathtt{val}\ e$, then the handling construct evaluates to c_v with x bound to e,
2. if c evaluates to $e_i \,\#\, \mathtt{op}_i\, e'$ with continuation κ, then the handling construct evaluates to c_i with x_i and k_i bound to e' and $\mathtt{fun}\ y \mapsto \mathtt{with}\ h\ \mathtt{handle}\ \kappa\, y$, respectively.
3. if c evaluates to any other operation $\iota \,\#\, \mathtt{op}\, e'$ with continuation κ, then the handling construct acts as if h contained the clause

$$\iota \,\#\, \mathtt{op}\, x\, k \mapsto (\mathtt{let}\ y = \iota \,\#\, \mathtt{op}\, x\ \mathtt{in}\ k\, y).$$

Thus it evaluates to $\iota \,\#\, \mathtt{op}\, e'$ with continuation $\mathtt{fun}\ y \mapsto \mathtt{with}\ h\ \mathtt{handle}\ (\kappa\, y)$.

We may wrap several handling constructs around c, in which case the inner handler takes precedence. Note that $\mathtt{let}\ x = c_1\ \mathtt{in}\ c_2$ is equivalent to

$$\mathtt{with}\ (\mathtt{handler}\ \mathtt{val}\ x \mapsto c_2)\ \mathtt{handle}\ c_1$$

so we could theoretically omit $\mathtt{let}$.

2.3 Typing Rules

The typing system of core Eff has two typing judgments,

$$\Gamma \vdash_{\mathsf{e}} e : A \qquad \text{and} \qquad \Gamma \vdash_{\mathsf{c}} c : \underline{C}$$

stating that an expression e has expression type A, and that a computation c has computation type $\underline{C}$, respectively. As usual, Γ is a typing context

$$x_1 : A_1, \ldots, x_n : A_n$$

which assigns expression types A_i to distinct variables x_i. We also have subtyping of expression types $A \leq A'$ and computation types $\underline{C} \leq \underline{C'}$. We fix an assignment Ξ of effect types to instances, and assume implicitly that all dirts and regions appearing in the rules are well formed, i.e., if $\iota \mathbin{\#} \mathtt{op}$ appears then in fact $(\iota : E) \in \Xi$ and $\mathtt{op} \in E$, and that a region ρ in E^ρ mentions only instances whose effect type is E.

The typing rules for variables, primitive constants and functions are standard:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash_{\mathsf{e}} x : A} \qquad \Gamma \vdash_{\mathsf{e}} 0 : \mathtt{nat} \qquad \frac{\Gamma \vdash_{\mathsf{e}} e : \mathtt{nat}}{\Gamma \vdash_{\mathsf{e}} \mathtt{succ}\ e : \mathtt{nat}} \qquad \Gamma \vdash_{\mathsf{e}} () : \mathtt{unit}$$

$$\Gamma \vdash_{\mathsf{e}} \mathtt{true} : \mathtt{bool} \qquad \Gamma \vdash_{\mathsf{e}} \mathtt{false} : \mathtt{bool} \qquad \frac{\Gamma, x : A \vdash_{\mathsf{c}} c : \underline{C}}{\Gamma \vdash_{\mathsf{e}} (\mathtt{fun}\ x \mapsto c) : A \to \underline{C}}$$

The typing rule for instances

$$\frac{\iota \in \rho}{\Gamma \vdash_{\mathsf{e}} \iota : E^\rho}$$

verifies that the instance ι is in the region ρ and consults Ξ to check that ρ contains only instances whose effect type is E. An operation $e \mathbin{\#} \mathtt{op}$ has a function type, where the dirt in the codomain must contain operations $\iota \mathbin{\#} \mathtt{op}$ for ι ranging over the region associated with e:

$$\frac{\Gamma \vdash_{\mathsf{e}} e : E^\rho \qquad (\mathtt{op} : A \to B) \in E \qquad \forall \iota \in \rho \,.\, (\iota \mathbin{\#} \mathtt{op}) \in \delta}{\Gamma \vdash_{\mathsf{e}} e \mathbin{\#} \mathtt{op} : A \to B ! \delta}$$

The handler type $A ! \delta \Rightarrow \underline{C}$ expresses the fact that a handler transforms computations of type $A ! \delta$ to computations of type $\underline{C}$. The typing rule for handlers

$$\frac{\begin{array}{c}\Gamma, x : A \vdash_{\mathsf{c}} c_v : \underline{C} \\ \Gamma \vdash_{\mathsf{e}} e_i : E_i^{\rho_i} \qquad (\mathtt{op}_i : A_i \to B_i) \in E_i \qquad \Gamma, x_i : A_i, k_i : B_i \to \underline{C} \vdash_{\mathsf{c}} c_i : \underline{C} \\ \forall (\iota \mathbin{\#} \mathtt{op}) \in \delta \,.\, \exists i \,.\, \rho_i = \{\iota\} \wedge \mathtt{op}_i = \mathtt{op}\end{array}}{\Gamma \vdash_{\mathsf{e}} (\mathtt{handler}\ \mathtt{val}\ x \mapsto c_v \mid (e_i \mathbin{\#} \mathtt{op}_i\ x_i\ k_i \mapsto c_i)_i) : A ! \delta \Rightarrow \underline{C}}$$

checks that the instances and operations are paired up correctly according to their effect types, all clauses have computation type $\underline{C}$ under suitable assumptions on bound variables, and every operation in δ is handled by a clause. Note

that an operation $\iota \,\#\, \mathtt{op}$ is taken to be handled by a clause $e_i \,\#\, \mathtt{op}_i \, x_i \, k_i \mapsto c_i$ only if $\mathtt{op} = \mathtt{op}_i$ and e_i is ascertained to have exactly the region $\{\iota\}$. If the region were $\{\iota, \iota'\}$ we could not tell whether the clause handles $\iota \,\#\, \mathtt{op}$ or $\iota' \,\#\, \mathtt{op}$.

The typing rules for computations hold no suprises at all:

$$\frac{\Gamma \vdash_{\mathsf{e}} e : A}{\Gamma \vdash_{\mathsf{c}} \mathtt{val}\ e : A!\delta} \qquad \frac{\Gamma \vdash_{\mathsf{c}} c_1 : A!\delta \qquad \Gamma, x : A \vdash_{\mathsf{c}} c_2 : B!\delta}{\Gamma \vdash_{\mathsf{c}} (\mathtt{let}\ x = c_1\ \mathtt{in}\ c_2) : B!\delta}$$

$$\frac{\Gamma, f : A \to \underline{C}, x : A \vdash_{\mathsf{c}} c_1 : \underline{C} \qquad \Gamma, f : A \to \underline{C} \vdash_{\mathsf{c}} c_2 : \underline{D}}{\Gamma \vdash_{\mathsf{c}} (\mathtt{let\ rec}\ f\, x = c_1\ \mathtt{in}\ c_2) : \underline{D}}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e : \mathtt{nat}}{\Gamma \vdash_{\mathsf{c}} \mathtt{iszero}\ e : \mathtt{bool}!\delta} \qquad \frac{\Gamma \vdash_{\mathsf{e}} e : \mathtt{bool} \qquad \Gamma \vdash_{\mathsf{c}} c_1 : \underline{C} \qquad \Gamma \vdash_{\mathsf{c}} c_2 : \underline{C}}{\Gamma \vdash_{\mathsf{c}} (\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2) : \underline{C}}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e : \mathtt{empty}}{\Gamma \vdash_{\mathsf{c}} (\mathtt{absurd}\ e) : \underline{C}} \qquad \frac{\Gamma \vdash_{\mathsf{e}} e_1 : A \to \underline{C} \qquad \Gamma \vdash_{\mathsf{e}} e_2 : A}{\Gamma \vdash_{\mathsf{c}} e_1\, e_2 : \underline{C}}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e : \underline{C} \Rightarrow \underline{D} \qquad \Gamma \vdash_{\mathsf{c}} c : \underline{C}}{\Gamma \vdash_{\mathsf{c}} (\mathtt{with}\ e\ \mathtt{handle}\ c) : \underline{D}}$$

The structural subtyping rules are expected as well [14]:

$$\frac{\Gamma \vdash_{\mathsf{e}} e : A \qquad A \le A'}{\Gamma \vdash_{\mathsf{e}} e : A'} \qquad \frac{\Gamma \vdash_{\mathsf{c}} c : \underline{C} \qquad \underline{C} \le \underline{C}'}{\Gamma \vdash_{\mathsf{c}} c : \underline{C}'} \qquad A \le A$$

$$\frac{A \le A' \qquad A' \le A''}{A \le A''} \qquad \frac{A' \le A \qquad \underline{C} \le \underline{C}'}{(A \to \underline{C}) \le (A' \to \underline{C}')} \qquad \frac{\underline{C}' \le \underline{C} \qquad \underline{D} \le \underline{D}'}{(\underline{C} \Rightarrow \underline{D}) \le (\underline{C}' \Rightarrow \underline{D}')}$$

$$\frac{\rho \subseteq \rho'}{E^{\rho} \le E^{\rho'}} \qquad \frac{A \le A' \qquad \delta \subseteq \delta'}{A!\delta \le A'!\delta'}$$

There are two subsumption rules, one for each typing judgment. The subtyping relation is reflexive and transitive. Function and handler types are contravariant in the domain and covariant in the codomain. Indeed, we may always pretend that a handler handles fewer operations, and triggers more operations than it actually does. An effect type is covariant in the region, while a computation type is covariant in both its parts.

Subtyping buys us additional expressivity. For instance, assuming ι_1 and ι_2 are instances of E,

$$\begin{aligned}
&\mathtt{let}\ a = \iota_1\ \mathtt{in}\\
&\mathtt{let}\ b = (\mathtt{if}\ p\ \mathtt{then}\ a\ \mathtt{else}\ \iota_2)\ \mathtt{in}\\
&\qquad \mathtt{val}\ (\mathtt{handler}\ \mathtt{val}\ x \mapsto \mathtt{val}\ ()\ \mid a \,\#\, \mathtt{op}\, x\, k \mapsto \mathtt{val}\ ())
\end{aligned}$$

we would like to give a the type $E^{\{\iota_1\}}$ so that the handler can have the more specific type $A!\{\iota_1 \,\#\, \mathtt{op}\} \Rightarrow \mathtt{unit}!\varnothing$, but then without subtyping $E^{\{\iota_1\}} \leq E^{\{\iota_1,\iota_2\}}$ we could not give b its type $E^{\{\iota_1,\iota_2\}}$.

2.4 Example: State Handlers

We demonstrate the features of our type system by looking at a state handler for the reference effect `ref` shown in (1). Define `state` to be the function

$$\begin{aligned}\mathtt{fun}\ r \mapsto (\mathtt{handler}\ \mathtt{val}\ x \mapsto \mathtt{val}\ (\mathtt{fun}\ s \mapsto \mathtt{val}\ x)\\ \mid r \,\#\, \mathtt{lookup}\ x\ k \mapsto \mathtt{val}\ (\mathtt{fun}\ s \mapsto k\ s\ s)\\ \mid r \,\#\, \mathtt{update}\ s'\ k \mapsto \mathtt{val}\ (\mathtt{fun}\ s \mapsto k\ ()\ s'))\end{aligned}$$

In words, `state` accepts an instance r and returns a handler which handles lookups and updates on r by using the standard functional encoding of the state monad. For any instance ι, expression type A and dirt δ, the function `state` has the type

$$\mathtt{ref}^{\{\iota\}} \to (A!(\{\iota \,\#\, \mathtt{lookup}, \iota \,\#\, \mathtt{update}\} \cup \delta) \Rightarrow (\mathtt{nat} \to A!\delta)!\delta).$$

Thus, if c is a computation of type

$$A!(\{\iota \,\#\, \mathtt{lookup}, \iota \,\#\, \mathtt{update}\} \cup \delta)$$

then

$$\mathtt{let}\ h = \mathtt{state}\ \iota\ \mathtt{in}\ (\mathtt{with}\ h\ \mathtt{handle}\ c) \tag{2}$$

has the type $(\mathtt{nat} \to A!\delta)!\delta$. That is, we obtained a computation which possibly triggers effects δ and returns a function. Upon application of the function to an initial state effects δ may be triggered again, after which a final result of type A is obtained. In particular, if $\delta = \varnothing$ then (2) is a pure computation.

If we weakened the domain of `state` to $\mathtt{ref}^{\{\iota_1,\iota_2\}}$ then we would not be able to deduce that `state` handled anything, so we would only be able to assign to `state` the less useful type

$$\mathtt{ref}^{\{\iota_1,\iota_2\}} \to (A!\delta \Rightarrow (\mathtt{nat} \to A!\delta)!\delta).$$

We may handle several references by invoking several instantiations of `state`. For example, let c be the computation which swaps the contents of two references:

$$\begin{aligned}&\mathtt{let}\ x_1 = \iota_1 \,\#\, \mathtt{lookup}()\ \mathtt{in}\\ &\mathtt{let}\ x_2 = \iota_2 \,\#\, \mathtt{lookup}()\ \mathtt{in}\\ &\mathtt{let}\ u = \iota_1 \,\#\, \mathtt{update}\ x_2\ \mathtt{in}\\ &\mathtt{let}\ v = \iota_2 \,\#\, \mathtt{update}\ x_1\ \mathtt{in}\ ()\end{aligned}$$

By itself, c has the type

$$\mathtt{unit}!\{\iota_1 \,\#\, \mathtt{lookup}, \iota_1 \,\#\, \mathtt{update}, \iota_2 \,\#\, \mathtt{lookup}, \iota_2 \,\#\, \mathtt{update}\},$$

while

$$\texttt{let}\ h_1 = \texttt{state}\ \iota_1\ \texttt{in}\ (\texttt{with}\ h_1\ \texttt{handle}\ c)$$

has type $(\texttt{nat} \to \texttt{unit}!\{\iota_2\,\#\,\texttt{lookup}, \iota_2\,\#\,\texttt{update}\})!\{\iota_2\,\#\,\texttt{lookup}, \iota_2\,\#\,\texttt{update}\}$. If we handle both instances,

$$\begin{aligned}&\texttt{let}\ h_1 = \texttt{state}\ \iota_1\ \texttt{in}\\ &\texttt{let}\ h_2 = \texttt{state}\ \iota_2\ \texttt{in}\\ &\qquad \texttt{with}\ h_2\ \texttt{handle}\ (\texttt{with}\ h_1\ \texttt{handle}\ c),\end{aligned}$$

we get the pure type $\texttt{nat} \to (\texttt{nat} \to \texttt{unit}!\varnothing)!\varnothing$.

3 Denotational Semantics

An outline of a set-theoretic algebraic semantics of computational effects, as developed by [6,15,16,7,8], is shown in the following table.

Programming language	*Algebra*
effect type	algebraic signature
expression type	set
expression	element
computation type	free algebra
pure computation	generator
effectful computation	algebraic operation
handler	homomorphism of algebras

To properly account for recursion and non-termination we adapt it to a *domain-theoretic* semantics of algebraic effects under which expression and computation types are *domains*. We first give Curry-style semantics in which terms are interpreted without being typed, and then, following John Reynolds [17], we provide a Church-style semantics in which types receive meanings as well. It does not matter much what kind of domains we use, they could be ω-cpos, Scott domains, effective Scott domains, or any other kind of domains that model the basic type-theoretic operations (product, function space, coalesced sum).

3.1 Semantics of Expressions and Computations

Expressions could be modeled with *predomains*, because they are inert pieces of data, free from computational effects, including non-termination. However, the bottom element is useful for denotation of ill-typed expressions and runtime errors. In any case, the predomain nature of expressions will be captured later on by the partial equivalence relations. The domains for computation types are free in a suitable sense, i.e., they enjoy a recursion principle which acts as a substitute for the universality of free algebras. We assume a given set of all instances $\mathbb{I}$ and

a set of all operation symbols $\mathbb{O}$, write $+$ for a disjoint sum, and $\oplus$ for coalesced sum.

To interpret expressions and computations we need suitable domains of *values* V and *results* R, respectively. These have to be large enough to contain all possible denotations of expressions and computations, which is usually achieved by solving suitable recursive domain equations, such as

$$V = \mathbb{N}_\bot \oplus \{0,1\}_\bot \oplus \{\star\}_\bot \oplus \mathbb{I}_\bot \oplus R^V \oplus R^R,$$
$$R = (V + \mathbb{I} \times \mathbb{O} \times V \times R^V)_\bot.$$

The summands in the equation for V correspond to various expression types. The recursive domain equation for R says that a non-bottom element of R is either a value, or a quadruple $(\iota, \mathtt{op}, v, \kappa)$ corresponding to the operation $\iota \,\#\, \mathtt{op}$ applied to parameter v and with continuation κ. However, as in [17], we shall assume only that V and R are large enough to contain the various components needed for the semantics as *retracts*:

$$\mathbb{N}_\bot \underset{\mathsf{r}_{\mathtt{nat}}}{\overset{\mathsf{s}_{\mathtt{nat}}}{\rightleftarrows}} V \qquad \{0,1\}_\bot \underset{\mathsf{r}_{\mathtt{bool}}}{\overset{\mathsf{s}_{\mathtt{bool}}}{\rightleftarrows}} V \qquad \{\star\}_\bot \underset{\mathsf{r}_{\mathtt{unit}}}{\overset{\mathsf{s}_{\mathtt{unit}}}{\rightleftarrows}} V$$

$$\mathbb{I}_\bot \underset{\mathsf{r}_{\mathtt{effect}}}{\overset{\mathsf{s}_{\mathtt{effect}}}{\rightleftarrows}} V \qquad R^V \underset{\mathsf{r}_{\to}}{\overset{\mathsf{s}_{\to}}{\rightleftarrows}} V \qquad R^R \underset{\mathsf{r}_{\Rightarrow}}{\overset{\mathsf{s}_{\Rightarrow}}{\rightleftarrows}} V$$

and

$$(V + \mathbb{I} \times \mathbb{O} \times V \times R^V)_\bot \underset{\mathsf{r}_{\mathtt{res}}}{\overset{\mathsf{s}_{\mathtt{res}}}{\rightleftarrows}} R$$

Thus, R and V could be solutions to the above domain equations, but they could also both be a universal domain, or just a (non-trivial) reflexive domain.[1] Note that there are further canonical retractions

$$V \underset{\mathsf{r}_{\mathtt{val}}}{\overset{\mathsf{s}_{\mathtt{val}}}{\rightleftarrows}} (V + \mathbb{I} \times \mathbb{O} \times V \times R^V)_\bot$$

$$(\mathbb{I} \times \mathbb{O} \times V \times R^V)_\bot \underset{\mathsf{r}_{\mathtt{oper}}}{\overset{\mathsf{s}_{\mathtt{oper}}}{\rightleftarrows}} (V + \mathbb{I} \times \mathbb{O} \times V \times R^V)_\bot$$

which in combination with the section-retraction pair $(\mathsf{s}_{\mathtt{res}}, \mathsf{r}_{\mathtt{res}})$ convert elements of R to pure values and operations, respectively.

The domain $(V + \mathbb{I} \times \mathbb{O} \times V \times R^V)_\bot$ has the following *recursion principle*. Given a domain D and maps

$$h_{\mathtt{val}} : V \longrightarrow D \qquad \text{and} \qquad h_{\mathtt{oper}} : \mathbb{I} \times \mathbb{O} \times V \times R^V \longrightarrow D,$$

there is a unique strict map $h : (V + \mathbb{I} \times \mathbb{O} \times V \times R^V)_\bot \longrightarrow D$ such that, for all $v \in V$, $\iota \in \mathbb{I}$, $\mathtt{op} \in \mathbb{O}$, $\kappa : V \to R$,

$$h(\mathsf{s}_{\mathtt{val}}(v)) = h_{\mathtt{val}}\,(v),$$
$$h(\mathsf{s}_{\mathtt{oper}}(\iota, \mathtt{op}, v, \kappa)) = h_{\mathtt{oper}}(\iota, \mathtt{op}, v, h \circ \mathsf{r}_{\mathtt{res}} \circ \kappa).$$

[1] A domain D is *reflexive* if it contains its functions space D^D as a retract, and is therefore a model of the untyped λ-calculus.