

Undergraduate Topics in Computer Science

Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

For other volumes:

<http://www.springer.com/series/7592>

Gilles Dowek · Jean-Jacques Lévy

Introduction to the Theory of Programming Languages

 Springer

Gilles Dowek
Labo. d'Informatique
École polytechnique
route de Saclay
91128 Palaiseau
France
gilles.dowek@polytechnique.edu

Jean-Jacques Lévy
Centre de Recherche Commun
INRIA-Microsoft Research
Parc Orsay Université
28 rue Jean Rostand
91893 Orsay Cedex
France
jean-jacques.levy@inria.fr

Series editor
Ian Mackie

Advisory board
Samson Abramsky, University of Oxford, Oxford, UK
Chris Hankin, Imperial College London, London, UK
Dexter Kozen, Cornell University, Ithaca, USA
Andrew Pitts, University of Cambridge, Cambridge, UK
Hanne Riis Nielson, Technical University of Denmark, Lyngby, Denmark
Steven Skiena, Stony Brook University, Stony Brooks, USA
Iain Stewart, University of Durham, Durham, UK

The work was first published in 2006 by Les éditions de l'École polytechnique with the following title: 'Introduction à la théorie des langages de programmation'. The translator of the work is Maribel Fernandez.

ISSN 1863-7310
ISBN 978-0-85729-075-5 e-ISBN 978-0-85729-076-2
DOI 10.1007/978-0-85729-076-2
Springer London Dordrecht Heidelberg New York

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

© Springer-Verlag London Limited 2011

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licenses issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.

The use of registered names, trademarks, etc., in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

Printed on acid-free paper

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

What Is the Theory of Programming Languages?

The ultimate, definitive programming language has not been created yet, far from it. Almost every day a new language is created, and new functionalities are added to existing languages. Improvements in programming languages contribute to making programs more reliable, shorten the development time, and make programs easier to maintain. Improvements are also needed to satisfy new requirements, such as the development of parallel, distributed or mobile programs.

The first thing that we need to describe, when defining a programming language, is its *syntax*. Should we write $x := 1$ or $x = 1$? Should we put brackets after an `if` or not? More generally, what are the strings of symbols that can be used as a program? There is a useful tool for this: the notion of a *formal grammar*. Using a grammar, we can describe the syntax of the language in a precise way, and this makes it possible to build programs to check the syntactical correctness of programs.

But it is not sufficient to know what a syntactically correct program is in order to know what is going to happen when we run the program. When defining a programming language, it is also necessary to describe its *semantics*, that is, the expected behaviour of the program when it is executed. Two languages may have the same syntax but different semantics.

The following is an example of what is meant (informally) by semantics. Function evaluation is often explained as follows. “*The result \forall of the evaluation of an expression of the form $f e_1 \dots e_n$, where the symbol f is a function defined by the expression $f x_1 \dots x_n = e'$, is obtained in the following way. First, the arguments e_1, \dots, e_n are evaluated, returning values w_1, \dots, w_n . Then, these values are associated to the variables x_1, \dots, x_n , and finally the expression e' is evaluated. The value \forall is the result of this evaluation.*”

This explanation of the semantics of the language, expressed in a natural language (English), allows us to understand what happens when a program is executed, but is it precise? Consider, for example, the program

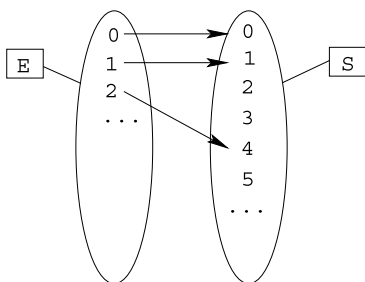
```
f x y = x
g z = (n = n + z; n)
n = 0; print(f (g 2) (g 7))
```

Depending on the way we interpret the explanation given above, we can deduce that the program will result in the value 2 or in the value 9. This is because the natural language explanation does not indicate whether we have to evaluate $g\ 2$ before or after $g\ 7$, and the order in which we evaluate these expressions is important in this case. Instead, the explanation should have said: “the arguments e_1, \dots, e_n are evaluated *starting from* e_1 ” or else “*starting from* e_n ”.

If two different programmers read an ambiguous explanation, they might understand different things. Even worse, the designers of the compilers for the language might choose different conventions. Then the same program will give different results depending on the compiler used.

It is well known that natural languages are too imprecise to express the syntax of a programming language, a formal language should be used instead. Similarly, natural languages are too imprecise to express the semantics of a programming language, and we need to use a formal language for this.

What is the semantics of a program? Let us take for instance a program p that requests an integer, computes its square, and displays the result of this operation. To describe the behaviour of this program, we need to describe a relation R between the input value and the associated output.



The semantics of this program is, thus, a relation R between elements of the set E of input values and elements of the set S of output values, that is, a subset of $E \times S$.

The semantics of a program is then a binary relation. The semantics of a programming language is, in turn, a ternary relation: “the program p with input value e returns the output value s ”. We denote this relation by p , $e \mapsto s$. The program p and the input e are available before the execution of the program starts. Often, these two elements are paired in a *term* $p\ e$, and the semantics of the language assigns a value to this term. The semantics of the language is then a binary relation $t \mapsto s$.

To express the semantics of a programming language we need a language that can express relations.

When the semantics of a program is a functional relation, that is, for each input value there is at most one output value, we say that the program is *deterministic*. Video games are examples of non-deterministic programs, since some randomness is necessary to make the game enjoyable. A language is deterministic if all the programs that can be written in the language are deterministic, or equivalently, if the semantics is a functional relation. In this case, it is possible to define its semantics using a language to define functions instead of a language to define relations.

Acknowledgements

The authors would like to thank Gérard Assayag, Antonio Bucciarelli, Roberto Di Cosmo, Xavier Leroy, Dave MacQueen, Luc Maranget, Michel Mauny, François Pottier, Didier Rémy, Alan Schmitt, Élodie-Jane Sims and Véronique Viguié Donzeau-Gouge.

Contents

1	Terms and Relations	1
1.1	Inductive Definitions	1
1.1.1	The Fixed Point Theorem	1
1.1.2	Inductive Definitions	4
1.1.3	Structural Induction	6
1.1.4	The Reflexive-Transitive Closure of a Relation	6
1.2	Languages	7
1.2.1	Languages Without Variables	7
1.2.2	Variables	7
1.2.3	Many-Sorted Languages	9
1.2.4	Free and Bound Variables	10
1.2.5	Substitution	10
1.3	Three Ways to Define the Semantics of a Language	12
1.3.1	Denotational Semantics	12
1.3.2	Big-Step Operational Semantics	12
1.3.3	Small-Step Operational Semantics	12
1.3.4	Non-termination	13
2	The Language PCF	15
2.1	A Functional Language: PCF	15
2.1.1	Programs Are Functions	15
2.1.2	Functions Are First-Class Objects	15
2.1.3	Functions with Several Arguments	16
2.1.4	No Assignments	16
2.1.5	Recursive Definitions	16
2.1.6	Definitions	17
2.1.7	The Language PCF	17
2.2	Small-Step Operational Semantics for PCF	18
2.2.1	Rules	18
2.2.2	Numbers	19
2.2.3	A Congruence	20
2.2.4	An Example	21

2.2.5	Irreducible Closed Terms	22
2.2.6	Non-termination	23
2.2.7	Confluence	24
2.3	Reduction Strategies	24
2.3.1	The Notion of a Strategy	24
2.3.2	Weak Reduction	26
2.3.3	Call by Name	26
2.3.4	Call by Value	27
2.3.5	A Bit of Laziness Is Needed	27
2.4	Big-Step Operational Semantics for PCF	27
2.4.1	Call by Name	28
2.4.2	Call by Value	29
2.5	Evaluation of PCF Programs	31
3	From Evaluation to Interpretation	33
3.1	Call by Name	33
3.2	Call by Value	35
3.3	An Optimisation: de Bruijn Indices	36
3.4	Construction of Functions via Fixed Points	38
3.4.1	First Variation: Recursive Closures	38
3.4.2	Second Variation: Rational Values	40
4	Compilation	43
4.1	An Interpreter Written in a Language Without Functions	44
4.2	From Interpretation to Compilation	44
4.3	An Abstract Machine for PCF	45
4.3.1	The Environment	45
4.3.2	Closures	46
4.3.3	PCF Constructs	46
4.3.4	Using de Bruijn Indices	47
4.3.5	Small-Step Operational Semantics	48
4.4	Compilation of PCF	48
5	PCF with Types	51
5.1	Types	51
5.1.1	PCF with Types	52
5.1.2	The Typing Relation	53
5.2	No Errors at Run Time	54
5.2.1	Using Small-Step Operational Semantics	55
5.2.2	Using Big-Step Operational Semantics	55
5.3	Denotational Semantics for Typed PCF	56
5.3.1	A Trivial Semantics	56
5.3.2	Termination	57
5.3.3	Scott's Ordering Relation	58
5.3.4	Semantics of Fixed Points	59

- 6 Type Inference** 63
 - 6.1 Inferring Monomorphic Types 63
 - 6.1.1 Assigning Types to Untyped Terms 63
 - 6.1.2 Hindley’s Algorithm 64
 - 6.1.3 Hindley’s Algorithm with Immediate Resolution 66
 - 6.2 Polymorphism 68
 - 6.2.1 PCF with Polymorphic Types 68
 - 6.2.2 The Algorithm of Damas and Milner 70
- 7 References and Assignment** 73
 - 7.1 An Extension of PCF 74
 - 7.2 Semantics of PCF with References 75
- 8 Records and Objects** 81
 - 8.1 Records 81
 - 8.1.1 Labelled Fields 81
 - 8.1.2 An Extension of PCF with Records 82
 - 8.2 Objects 85
 - 8.2.1 Methods and Functional Fields 85
 - 8.2.2 What Is “Self”? 86
 - 8.2.3 Objects and References 88
- 9 Epilogue** 89
- References** 93
- Index** 95