Coalgebraic effects and their cohandlers in programming languages

(Efekty koalgebraiczne oraz ich kohandlery
w językach programowania)

Mateusz Urbańczyk

Praca inżynierska

Promotor: dr Maciej Piróg

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

1 września 2020

Abstract

. . .

. . .

Contents

Chapter 1

Introduction

My algebraic methods are really methods of working and thinking; this is why they have crept in everywhere anonymously. ~ Emmy Noether

In this thesis we discuss issues with excessive generality of algebraic effects, and propose a solution without coalgebraic means along with presentation of experimental programming language Freak, which is a language with (co) algebraic effects and (co) handlers, where implementation is based on Continuation Passing Style translation.

1.1 Problem Statement

Algebraic effects, while not being a new concept, in the recent years have received a lot of attention [6], both from the theoretical and practical side.

They give developer necessary tools for declarative programming and writing programs in a way to strive from exposing implementation details, by defining computational effects as an API in our code. In the same time, they preserve a lot of flexibility and have a strong theoretical background.

In fact, that very flexibility can be troublesome. Allowing developer for too much freedom may lead to undesired behaviour and incorrect programs, for this reason, we develop type systems, to detect errors before they occur.

Multi-shot resumptions in effect handlers and possibility to drop resumption are powerful tools to describe fairly complex logic in a concise way. That being said, they give rise to issues that would not occur in standard control flow, especially when composing various effects together. Unwanted interaction with external resources multiple times or not invoking finalisation code are cases that may occur while using algebraic effects and handlers. Let's consider the following example:

```
handle
```

```
    let fh <- do Open "praise.txt" in
    let c <- do Choice () in
    if c then do Write (fh, "Guy Fieri") else do Write (fh, "is cool") ;
    close fh
with {
    return x -> [x] |
    choose () k -> return (append (k true) (k false)) |
}
```

In the above code we open file, and based on nondeterministic choice, we write to the file, and then close it. This piece of code looks harmless, however, as we invoke resumption for the second time, we are attempting to write to the closed file, and then close it once again. In fact, dropping resumption is also considered harmful, as we would not attempt to close the file.

This captures the excessive generality of effects, and is the issue that we would like to address with coalgebraic effects.

1.2 Thesis Outline

In the following chapter we provide a background about effects, define algebraic effects in categorical setting as well as point out dual coalgebraic effects and cohandlers, finishing with showing related work in this area. Chapter 3 describes possible ways to approach issue of excessive generality, Chapter 4 shows Freak language by examples along with usage guide, and then Chapter 5 describes the language's syntax, operational semantics and CPS translation. In next one, Chapter 6, implementation details are revealed. We conclude in Chapter 7 by stating what are the possible augmentations, that are intended to be made in the future.

Chapter 2

Background

2.1 Computational Effects

Since the rapid development of computational theory in 1930s by A. Turing, K. Godel and A. Church, we have a well-established notion of what can and what cannot be done through algorithmic means, which we can almost directly translate to being computable by our machines. Through next years we have developed mainstream languages that are used almost everywhere, with great success.

Under these circumstances one may pose a question, why do we still bother with development of languages theory, since so much has been done already. Is there anything that drives us towards further research? Indeed, one active branch revolves around equational theory to assess equality of two programs, which we know that in the general setting is undecidable. Proof methods may include extensional, contextual or logical equality. However, there is no doubt that these formal ways of reasoning about programs, while being crucial for assessing correctness, do not bring direct benefits for everyday use cases, as they are rarely accessible by a common developer.

Other branch of languages theory, that we shall investigate more in this thesis, is about taming complexity of programs. Various methods of static analysis has been developed for various use cases, most notably, type systems. Thanks to strong and static type systems along with their implementations, we have solid tools to work efficiently on functions that are pure. That being said, we claim that the core complexity of programs comes from side effects, or more generally, computational effects, which we cannot avoid in writing anything useful.

We need to have a good way for handling computational effects. One of the ways to model them, can be done through monads [25, 20]. However, they were found to be, to say the least, cumbersome to work with when the number of different effects increase. It is perhaps not a coincidence that many functional programming languages do not have them, because of the non-composable nature of them, or at

least not composable in their implementations.

Let's put the following functions

$$f : a \to b,\ g : b \to c$$
$$f' : a \to m\ b,\ g' : b \to m\ c$$
$$f'' : a \to m'\ m\ b,\ g'' : b \to m'\ m\ c$$

where m, m' are monads. Functions f, g can be composed using standard composition, for f', g' we can use Kleisli composition operator from monad m. What in case of adding one more effect, f'' and g''?

It turns out, that it becomes complex and unpleasant to combine two functors together to form a new monad, and for this purpose, monad transformers [17] arose in Haskell. Most of the common languages avoid this by not expressing computational effects in the type system, and instead one may think about functions as being implicitly embed in a Kleisli Category over a functor T [13], where T is a hidden signature over all possible side effects that occur in our program.

TODO Reconsider reference to ncatlab on Kleisli Cat, often it's not the best resource

Not only we would like to bring back effects to our type system [24], but also do it in a way that is composable. This is where algebraic effects comes to the rescue. From theoretical point of view, we need to develop equational theory about our effects to assert correctness of our language as well as to have the right hammer to reason about our programs. From practical side, we need to have the way of taming computational effects in our programs. That is the point where we would like to introduce to unfamiliar readers a notion of algebraic effects.

2.2 Algebraic Effects

Algebraic effects can be thought of as an public interface for computational effects. Declarative approach allow us to write programs in which the actual semantic of source code is dependent on handler that defines the meaning of a subset of effects.

This is really an incredible feature from practical point of view, as we may substitute logic depending on the execution environment. As an example, fetching for resources can behave differently as we run tests, debug our code, or run it on production. In the same manner, they neatly allow us to abstract over implementation details.

Patterns like these are well known in programming, for which other alternatives arose. One can mention interfaces from object-oriented programming, which are also a way to describe the API of a certain component. In order to abstract from implementation details, we pass an object around which represents a certain

interface. While this is certainly better than having no abstractions at all, we need to pass this interface into every function that is going to use it. This practice is called dependency injection, and while it sounds like it solves some of the issues, we end up in functions that need to carry representants of the interfaces, and pass them in every subcall that they make.

TODO maybe example with comparison for these two approaches could be useful here?

That being said, it's only one particular issue that algebraic effects address. Their handlers may also drop the resumption or invoke it more than once. There are many other great sources for getting familiarized with effects from practical side [21, 4, 14], so we will omit further explanations. More examples can be found in Chapter 4.

2.3 Categorical Setting of Universal Algebra

Algebraic effects can be described via operational means, however, for the purpose of presenting the duality between algebra and coalgebra, we allow ourselves to wander a bit deeper into category theory and describe effects from denotational point of view [2].

2.3.1 Algebraic Theories

Definition 2.1. A *signature* Σ is given by a collection of operation symbols op_i with associated parameters P_i and arities A_i, where P_i and A_i are objects in the category of our interest. We will write an operation as $op_i : P_i \rightsquigarrow A_i$

Definition 2.2. Collection of *Σ-terms* is a free algebra with a generator X for a functor μH_Σ that maps objects into trees over a given signature Σ and morphisms into folds over trees.

Definition 2.3. A *Σ-Equation* is an object X and a pair of *Σ-terms* $l, r \in Tree_\Sigma(X)$, written as

$$X \mid l = r$$

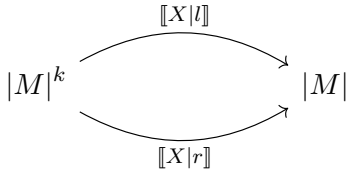
Definition 2.4. An *algebraic theory* $T = (\Sigma_T, \mathcal{E}_T)$, is given by a signature Σ_T and a collection $\mathcal{E}_T$ of *Σ_T-equations*.

Definition 2.5. An *interpretation I over a given signature* Σ is given by a carrier object $|I|$ and for each $op_i : P_i \rightsquigarrow A_i$ in Σ a map

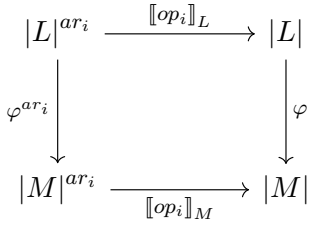
$$[\![op_i]\!]_I : P_i \times |I|^{A_i} \to |I|$$

Interpretation may be naturally extended to *Σ-terms*, such that a given *Σ-term* $X \mid t$ is interpreted by a map which sends variables into projections from environment and terms into map composition over each subterm.

Definition 2.6. A *model* M of an algebraic theory T is an interpretation of the signature Σ_T which validates all the equations $\mathcal{E}_\mathcal{T}$. That is, for every equation $X \mid l = r$ the following diagram commutes:



Definition 2.7. Let L, M be models of a theory T. *T-homomorphism* $\varphi : L \to M$ is a map such that, for every operation symbol op_i in T the following diagram commutes:

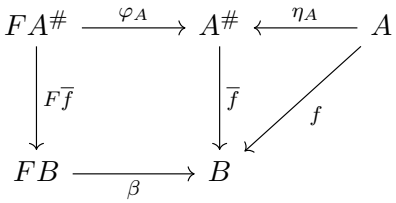


We denote here A^n as *n-ary* product of A.

Definition 2.8. Free F-algebra on an object A (of generators) in $\mathcal{C}$ is meant an algebra

$$\varphi_A : FA^\# \longrightarrow A$$

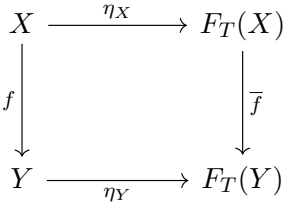
together with an universal arrow $\eta_A : A \longrightarrow A^\#$. Universality means that for every algebra $\beta : FB \longrightarrow B$ and every morphism $f : A \longrightarrow B$ in $\mathcal{C}$, there exists a unique homomorphism $\overline{f} : A^\# \longrightarrow B$ extending f, i.e. a unique morphism of $\mathcal{C}$ for which the diagram below commutes:



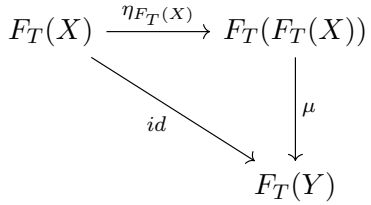
Definition 2.9. *Free model* is just a model that is free algebra.

Lemma 2.10. *Free models form monads*

Proof. Let M be a free model of an algebraic theory T in category $\mathcal{C}$. We have an endofunctor F_T, which takes objects into free model and maps to unique homomorphisms for which the following diagram commutes:



which immediately gives us η natural transformation for a monad. We can now define μ for a monad as a unique morphism for which the diagram commutes:



F_T is an endofunctor, therefore it sends X into an object in $\mathcal{C}$. From any object we have a unique map sending an object into model, which is $\eta_{F_T(X)}$. From free property of our model we get a unique map μ such that the diagram commutes, therefore we have a monad (F_T, η, μ).

□

Definition 2.11. *Handler* is a *T-homomorphism* between free models

$$H : |F_T(X)| \to |F_{T'}(X')|$$

that is, a map between carriers that preserves the structure of theory T.

Concluding from the theory that we have built, algebraic effects are just free models over computation trees, where the signature is the set of effects, and equational theory is defininig rules of rewriting expressions with effects. The latter is usually irrelevant from implementation perspective, but may be important for reasoning about programs.

Effect handlers on the other hand, are transformations from one computation tree over given signature of effects into another one, where handler may serve effects and also propagate new ones.

2.4 Duality

2.4.1 Comodels

Definition 2.12. Comodel in $\mathcal{C}$ is just a Model in $\mathcal{C}^{op}$.

We could end by stating this definition, and everything else would follow directly from duality. However, expanding definitions is going to give us better intuition, as well as give more solid ground for implementation. For this part, we are going to assume we operate in **Set** category.

2.4.2 Cooperations

We have defined interpretation of an operation as a map:

$$[\![op]\!]_M : |I|^A \to |I|^P$$

Let's now take a look on how it dualizes. In **Set**, we can think about A^B exponentials as B-*ary* coproducts over A, where for each argument we select one result. Every arrow in opposite category is reverted, thus coproducts becomes products, our morphism is reverted and model becomes comodel (*world*). Putting this together, we obtain the following map:

$$[\![op]\!]^W : |I| \times P \to |I| \times A$$

which is called a *cooperation*. Meaning of that morphism, is that based on coalgebra carrier and a parameter, we obtain new state of the coalgebra and a value generated by coalgebra.

2.4.3 Coalgebraic Effects

TODO
- coalg effects
- cohandlers

2.4.4 Coinductive Reasoning

Induction is a way of constructing new structures. Recursion is a way of folding inductively defined structure in a terminating way. Recursive functions (not to be confused with recursive computability class), should shrink the argument in each call, meaning that it eventually ends up terminating in a base case.

Coinduction is literally a dual notion to induction. We *observe* possibly infinite structures, by doing deconstruction. Corecursion, is a way of productively defining new, possibly enlarged structures. Due to infinity, the evaluation should be lazy, whereas in induction it may be eager.

Here is a table that summarizes difference between these two methods of reasoning:

feature	**induction**	**coinduction**
basic activity	construction	deconstruction
derived activity	deconstruction	construction
functions shape	inductive domain	coinductive codomain
(co) recursive calls	shrinks the argument	grows the result
functions feature	terminating	productive
evaluation	possibly eager	necessarily lazy

From categorical standpoint, coinduction is formed over a final coalgebra, where corecursion is the mediator between any coalgebra into a final one, and coinductive proof principle corresponds to the uniqueness of the mediator. Recall that coalgebraic effects arises from a particular type of a final coalgebra, namely, cofree coalgebra.

TODO Not sure which paper to cite here for a deeper dive into the upper topic. Book *Initial Algebras, Terminal Coalgebras, and the Theory of Fixed Points of Functors* by Adámek, Milius and S. Moss from which I have studied is no longer available on the internet, as it was a draft under construction.

Reader that focuses on pragmatism may pose a question, why do we even want to reason about infinite structures? They never appear in practice! In fact, it's very common to operate on never-ending transition systems or streams of data, where finitary means of reasoning are of no use, as we can't expect an end to stream.

Operating on coinductive structures may involve modification of the internal state of the machine that is generating the infinite streams, or in more concrete scenario, alternation of the external resource that is providing us the data.

For a better understanding and intuition, we shall illustrate now the difference based on inductive and coinductive relation. Recall that the set Λ of λ-*terms* is given by the following grammar:

$$e ::= x \mid \lambda x.e \mid e_1 e_2$$

where $\Lambda^0 \in \Lambda$ is a set of *closed* λ-*terms*, meaning, those terms have no free variables.

Let's now define an inductive, and coinductive predicate. Relation $\Downarrow \subseteq \Lambda^0 \times \Lambda^0$ (convergence) for call-by-value λ-*calculus* is defined as follows:

$$\frac{}{\lambda x.e \Downarrow \lambda x.e} \qquad \frac{e_1 \Downarrow \lambda x.e_0 \qquad e_0[e_2/x] \Downarrow e'}{e_1 e_2 \Downarrow e'}$$

which means that $\Downarrow$ is the *smallest* predicate *closed forward* under the above rules. Relation can be inductively built from empty set and then in a fixed-point manner we can add more terms. Let's now see a relation $\Uparrow \subseteq \Lambda^0$ of *divergent* terms, which is coinductively defined as follows:

$$\frac{e_1 \Uparrow}{e_1 e_2 \Uparrow} \qquad \frac{e_1 \Downarrow \lambda x.e_0 \qquad e_0[e_2/x] \Uparrow}{e_1 e_2 \Uparrow}$$

which means that $\Uparrow$ is the *greatest* predicate *closed backwards* under the above rules, relation can be coinductively obtained by starting with Λ_0, and then by removing elements that do not fit the rules.

For more detailed introduction into coinduction and other examples, see [11, 23].

TODO maybe also write coinductive proof based on two relations above?

2.5 Related Work

2.5.1 Algebraic Effects

Except from Links language [10], on which the implementation is based, there are currently many other alternatives available. One may take a look at Frank [18], which provides support for multihandlers, Koka [16], Helium [5] or Eff [3]. Except from separate languages, many libraries arose for existing ones like Haskell, Idris, Scala or Multicore OCaml.

In fact, libraries for algebraic effects also arose in mainstream languages, such as C [15] or Python [8]. In Python, effects are implemented through generators [12], using built-in feature of sending value when doing *yield* operation. Resumptions in handlers that are sending value are one-shot and tail-recursive, therefore we do not need to handle the coalgebraic part, at the cost of flexibility.

As can be seen in the J. Yallop repository [6], algebraic effects and handlers are now trending branch in the programming languages theory.

2.5.2 Coalgebraic Effects

Work that is the closest related to topic of the thesis, is work done by Danel Ahman and Andrej Bauer [1], as well as their implementation of a λ_{coop} language [7], which is

a calculus that ensures linear usage of resources as well as execution of code handling finalisation.

TODO Consider those
- From comodels to coalgebras: State and arrays.
- Tensors of comodels and models for operational semantics.
- Stateful runners of effectful computations.

Chapter 3

Potential Solutions

It can be seen, that issues lie with the fact, that interacting with coalgebras may lead to change of their internal state system. Examples include change of state in NFA, taking next element in an infinite stream or a closing file. Finalization must occur after initialization, thus we conclude that resumption should be called at least once. However, modifying state twice the same way can also lead to unwanted behaviour, as it was seen in Section 1.1, therefore it should be invoked exactly once.

Lemma 3.1. *Problem of detection whether resumption is called only once is undecidable.*

Proof. Follows directly from Rice's Theorem, as checking whether given function is going to be called is a nontrivial semantic property of a program. □

One of the ways to approach this problem, caused by extensive generality of effects, is the introduction of dual Coalgebraic effects, also named coeffects, for which we shall discuss various methods of implementation.

3.1 Dynamic Constraints Checking

One approach is to dynamically check against contract that continuation may only be called once. In this setting we are sure that our program will not accidentally go into wrong state, however by definition we lack static analysis to prevent errors before they occur.

3.2 Linear Types

Imposition of a single call of resumption, can be thought in a way that continuation is 'consumable', such that it's not available after one takes use of it. Putting it

in that way, it immediately reminds of linear logic [9], where premises of linear implications are no longer of use in conclusion, and in case of linear type theory, binding variable from expression A, disallows use of variables in A. Implementation of coalgebraic effects might be simplified in languages that already support linear type system, such as Rust [22].

In fact, there is a bijective translation between language with algebraic effects and calculus with linear type theory, and *every monad embeds in a linear state monad* [19].

TODO should this section be further extended?

3.3 Data-Flow Analysis

Another static analysis of the program.

- TODO Describe what is data-flow analysis
- TODO Give example for constant folding
- TODO Propose analysis for coalgebraic effects. Not sure if that's not too much, as it's not the goal of this thesis. Definitealy leave this section for later.

3.4 Cohandlers as Separate Constructs

Simplify semantics by separating coalgebraic effects into a new, restricted construct in programming language

TODO muuuch longer explanation of this topic

Chapter 4

(Co) Effectful Programming

Chapter 5

Calculus of Freak language

Chapter 6

Implementation

Chapter 7

Conclusion

Bibliography

[1] Danel Ahman and Andrej Bauer. "Runners in Action". In: *Lecture Notes in Computer Science* (2020), 29–55. ISSN: 1611-3349. DOI: 10.1007/978-3-030-44914-8_2. URL: http://dx.doi.org/10.1007/978-3-030-44914-8_2.

[2] Andrej Bauer. "What is algebraic about algebraic effects and handlers?" In: *CoRR* abs/1807.05923 (2018). arXiv: 1807.05923. URL: http://arxiv.org/abs/1807.05923.

[3] Andrej Bauer and Matija Pretnar. "An Effect System for Algebraic Effects and Handlers". In: *Logical Methods in Computer Science* 10.4 (2014). DOI: 10.2168/LMCS-10(4:9)2014. URL: https://doi.org/10.2168/LMCS-10(4:9)2014.

[4] Andrej Bauer and Matija Pretnar. "Programming with Algebraic Effects and Handlers". In: *CoRR* abs/1203.1539 (2012). arXiv: 1203.1539. URL: http://arxiv.org/abs/1203.1539.

[5] Dariusz Biernacki et al. "Handle with Care: Relational Interpretation of Algebraic Effects and Handlers". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 8:1–8:30. ISSN: 2475-1421. DOI: 10.1145/3158096. URL: http://doi.acm.org/10.1145/3158096.

[6] *Collaborative bibliography of work related to the theory and practice of computational effects.* URL: https://github.com/yallop/effects-bibliography.

[7] *Coop, langauge with coalgebraic effects.* URL: https://github.com/andrejbauer/coop.

[8] *Effect: library in Python implementing algebraic effects.* URL: https://github.com/python-effect/effect.

[9] Jean-Yves Girard. "Linear logic". In: *Theoretical Computer Science* 50.1 (1987), pp. 1 –101. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(87)90045-4. URL: http://www.sciencedirect.com/science/article/pii/0304397587900454.

[10] Daniel Hillerström et al. "Continuation Passing Style for Effect Handlers". In: *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*. Ed. by Dale Miller. Vol. 84. Leibniz International

Proceedings in Informatics (LIPIcs). 2017. DOI: 10.4230/LIPIcs.FSCD.2017.18.

[11] Bart Jacobs and Jan Rutten. "A Tutorial on (Co)Algebras and (Co)Induction". In: *EATCS Bulletin* 62 (1997), pp. 62–222.

[12] Satoru Kawahara and Yukiyoshi Kameyama. "One-shot Algebraic Effects as Coroutines". In: (2020).

[13] *Kleisli category definition on ncatlab*. URL: https://ncatlab.org/nlab/show/Kleisli+category.

[14] Daan Leijen. *Algebraic Effects for Functional Programming*. Tech. rep. MSR-TR-2016-29. 2016, p. 15. URL: https://www.microsoft.com/en-us/research/publication/algebraic-effects-for-functional-programming/.

[15] Daan Leijen. *Implementing Algebraic Effects in C*. Tech. rep. MSR-TR-2017-23. 2017. URL: https://www.microsoft.com/en-us/research/publication/implementing-algebraic-effects-c/.

[16] Daan Leijen. "Type Directed Compilation of Row-Typed Algebraic Effects". In: *Proceedings of Principles of Programming Languages (POPL'17), Paris, France*. 2017. URL: https://www.microsoft.com/en-us/research/publication/type-directed-compilation-row-typed-algebraic-effects/.

[17] Sheng Liang, Paul Hudak, and Mark Jones. "Monad Transformers and Modular Interpreters". In: POPL '95. San Francisco, California, USA: Association for Computing Machinery, 1995, 333–343. ISBN: 0897916921. DOI: 10.1145/199448.199528. URL: https://doi.org/10.1145/199448.199528.

[18] Sam Lindley, Conor McBride, and Craig McLaughlin. "Do be do be do". In: *CoRR* abs/1611.09259 (2016). arXiv: 1611.09259. URL: http://arxiv.org/abs/1611.09259.

[19] Rasmus Mogelberg and Sam Staton. "Linear usage of state". In: *Logical Methods in Computer Science [electronic only]* 10 (Mar. 2014). DOI: 10.2168/LMCS-10(1:17)2014.

[20] Eugenio Moggi. "Notions of Computation and Monads". In: *Inf. Comput.* 93.1 (July 1991), 55–92. ISSN: 0890-5401. DOI: 10.1016/0890-5401(91)90052-4. URL: https://doi.org/10.1016/0890-5401(91)90052-4.

[21] Matija Pretnar. "An Introduction to Algebraic Effects and Handlers. Invited Tutorial Paper". In: *Electron. Notes Theor. Comput. Sci.* 319.C (Dec. 2015), 19–35. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2015.12.003. URL: https://doi.org/10.1016/j.entcs.2015.12.003.

[22] *Rust language webpage*. URL: https://www.rust-lang.org/.

[23] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011. DOI: 10.1017/CBO9780511777110.

[24] J.P. Talpin and P. Jouvelot. "The Type and Effect Discipline". In: *Information and Computation* 111.2 (1994), pp. 245 –296. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1994.1046. URL: http://www.sciencedirect.com/science/article/pii/S0890540184710467.

[25] Philip Wadler. "The Essence of Functional Programming". In: POPL '92. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1992, 1–14. ISBN: 0897914538. DOI: 10.1145/143165.143169. URL: https://doi.org/10.1145/143165.143169.