Bidirectional Interplay between Mathematics and Computer Science: Safety and Extensibility in Computer Algebra and Haskell

著者（英）	Hiromi ISHII
year	2019
その他のタイトル	数学と計算機科学の相互作用：計算代数とHaskellにおける安全性と拡張性
学位授与大学	筑波大学 (University of Tsukuba)
学位授与年度	2018
報告番号	12102甲第8930号
URL	http://doi.org/10.15068/00156548

Bidirectional Interplay between Mathematics and Computer Science: Safety and Extensibility in Computer Algebra and Haskell

Hiromi ISHII

February 2019

Bidirectional Interplay between Mathematics and Computer Science: Safety and Extensibility in Computer Algebra and Haskell

Hiromi ISHII
Doctoral Program in Mathematics

Submitted to the Graduate School of
Pure and Applied Sciences
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy in
Science

at the
University of Tsukuba

PREFACE

In this thesis, we will see a bidirectional interplays between mathematics and computer science. The main research contributions, which we will see in Chapters 3 and 4, are based on two individual papers [41, 32].

The contents of this thesis can be divided into two parts. In Part I, we will review preliminaries which can help understanding the main research contributions of the thesis. Main research contributions are given in Part II.

Part I consists of two chapters. Chapter 1 is devoted to computer algebra. There, we will briefly review the basic theory and facts about *Gröbner basis* and introduce state-of-the-art algorithms for computing Gröbner basis, including Hilbert-driven, F_4, and F_5 algorithms. Then, in Chapter 2, we will skim through the overview of the *Haskell* programming language, which we will use as the main programming language in the research contributions. The first three sections covers basic concepts in functional programming and Haskell. Then, in the last section 2.4, we will introduce advanced language extensions that we need. Since these language features are not standard part of Haskell, readers are advised to read this section to get better understanding of the research contributions.

The main research contributions are stated in Part II, consisting of two independent chapters. Chapter 3 is devoted to "Freer monads and More extensible effects", which is based on the contents of Kiselyov–Ishii [41]. There, we will see how constructions in mathematics can be applied to functional programming to achieve composability. What we see in Chapter 4 is, in a sense, opposite direction of application: we see how methods developed in functional programming can be applied to mathematics, especially to computer algebra. The contents is based on Ishii [32] and we apply progressive type-systems and formal methods to computer algebra to achieve safe and extensible system.

ACKNOWLEDGEMENTS

First of all, the author would like to express deep gratitude to the supervisor Professor Akira Terui. He gracefully accepted me as a Ph.D. student even though it is at the end of the second year of my doctoral course that I discussed with him about changing a supervisor and asked him to accept me as a student. Since then, he have spared me a tremendous amount of time to discuss the plan of my graduation and given me a lot of helpful advices. It is no doubt that, without his help, I couldn't even complete my doctoral course.

The author is also grateful to the ex-supervisor Professor Masahiro Shioya. He instructed me for four years since my master's course, and gave helpful advices. Even though I've already left from his supervision, but my mathematical sense owes a lot to his instruction.

Great gratitude goes to my supervisors during undergraduate, Professor Katsuya Eda and Professor Hajime Kaji. Professor Eda gives me generous and kind advice and taught me an element of mathematical logic. It is Professor Kaji that gave me the first contact with Gröbner basis; without his lecture, I couldn't even come up with the topic of this thesis. I really appreciate these two supervisors, since experience in seminars with them constitute the backbone of my mathematical skill.

The author is indebted to Professor Daisuke Ikegami and Professor Toshimichi Usuba, for they having hosted seminars on set-theory and spared me a lot of time for discussions. Even though the topic is not directly related to the contents of this thesis, their graceful guidance helps me a lot and time I have experimented with them was so joyful.

The author would also thank Professor Oleg Kiselyov for discussions about extensible effects and permitting me to reprint the co-authored paper in this thesis. Writing a paper with him was so stimulating activity for me.

Professor Kazuhiro Yokoyama helped me a lot and spared me time for helpful discussions during the trip to Lille for Computer Algebra in Scientific Conference 2018.

The staff of office of mathematical department at University of Tsukuba have helped me a lot for paperworks. I would also thank graduate and undergraduate colleagues. Having seminars with them has always been a fun and exciting experience.

The author also wants to express great gratitude to an anonymous foreign visitor at Tsukuba Station. He rescued me from a firmly sealed W.C. on the day of the submission of this thesis. Without him, I could not submit this thesis, or, even worse, I must end my life in WC.

Aside from academic activity, having a joyful time with dearest friends make me feel alive. Thank you all for being my side.

The author acknowledges that the content of this paper is partially supported by Grant-in-Aid for JSPS Research Fellow Number 17J00479.

Finally, I would like to express the deepest appreciation to my parents, Toshiyuki and Miyuki Ishii. Without their mental and financial support, I could not pursue my goal in graduate school – of course, I don't even exist at all without them :-)

CONTENTS

Part I

INTRODUCTION AND PRELIMINARIES

1

A BRIEF INTRODUCTION TO COMPUTER ALGEBRA

In this chapter, we will briefly skim through basic concepts in computer algebra on which the contents of Chapter 4 are based. For more detail of this chapter, we refer readers to standard textbooks such as Cox–Little–O'Shea [14]. Even though the main contents of Chapter 4 doesn't require theoretical details, we include them here for completion.

1.1 BASIC DEFINITIONS AND FACTS ABOUT GRÖBNER BASIS

Notation. *In what follows in this chapter, k denotes a field and $k[X_1, \dots, X_n]$ the n-variate polynomial ring over k. For simplicity, we write it as $k[\mathbf{X}]$ if n is obvious from the context.*

We identify the set M^n of n-variate monomials *with the set $\mathbb{N}^n$ of n-tuples of natural numbers (including 0). In particular, we identify $\gamma = (k_1, \dots, k_n) \in \mathbb{N}^n$ with the monomial $\mathbf{X}^\gamma = X_1^{k_1} \dots X_n^{k_n}$.*

Definition 1.1. 1. A binary relation $\prec$ on $\mathbb{N}^n$ is a *monomial ordering* if it is a well-ordering compatible with the multiplication on M^n.

Let $\prec$ be an n-variate monomial ordering and $f = \sum_{\gamma \in \Gamma} a_\gamma \mathbf{X}^\gamma$, where $\Gamma \subset \mathbb{N}^n$ is finite and $a_\gamma \neq 0$ for all $\gamma \in \Gamma$.

2. The *leading monomial* of f with respect to $\prec$, denoted by $\mathrm{LM}_\prec(f)$ is the $\prec$-largest element of Γ. In this situation, we call $a_{\mathrm{LM}(f)}$ as the *leading coefficient* of f with respect to $\prec$ and denote it by $\mathrm{LC}_\prec(f)$. Then, the *leading term* $\mathrm{LT}_\prec(f)$ is simply $\mathrm{LT}_\prec(f) := \mathrm{LC}_\prec(f)\,\mathrm{LM}_\prec(f)$.

   We omit $\prec$ if it is clear from the context.

3. Let $I \subseteq k[\mathbf{X}]$ be an ideal. $G = \{g_1, \dots, g_\ell\} \subset I$ is a *Gröbner basis* of I if $\langle \mathrm{LM}(I) \rangle = \langle \mathrm{LM}(g_1), \dots, \mathrm{LM}(g_\ell) \rangle$.

4. Let $f, g, h \in k[\mathbf{X}]$. We say that h is a *reduction* of f modulo g, denoted by $f \underset{g}{\rightarrow} h$, if there is some term $c\mathbf{X}^\gamma \in k[\mathbf{X}]$, $c \neq 0$, such that $h = f - g \cdot c\mathbf{X}^\gamma$ and $\mathrm{LM}(c\mathbf{X}^\gamma g) \preccurlyeq \mathrm{LM}(f)$. For a set $F \subseteq k[\mathbf{X}]$, we write $f \underset{F}{\rightarrow} h$ if there is some $g \in F$ with $f \underset{g}{\rightarrow} h$.

   We denote the transitive and reflexive closure of $\underset{g}{\rightarrow}$ by $\underset{g}{\overset{*}{\rightarrow}}$; similar for $\underset{F}{\overset{*}{\rightarrow}}$.

5. f is of *F-normal form* if $f = 0$ or there is no f^* such that $f \underset{F}{\overset{*}{\rightarrow}} f^*$.

By the well-foundedness of monomial orderings, the relations $\underset{g}{\rightarrow}$ and $\underset{F}{\rightarrow}$ are strongly normalising; i.e. F-reduction of any polynomial f terminates after finite steps and hence f has at least one F-normal form. Also, by the definition of monomial orderings, f-reduction is compatible with polynomial addition and multiplication.

Although normal forms might not be unique for general *set* of polynomials, one can compute *one* normal form for each polynomial:

Algorithm 1.2 (Multivariate polynomial division)
Let $f, f_1, \dots, f_n$ be polynomials. A polynomial *remainder* $\overline{f}^{(f_1,\dots,f_n)}$ of f with respect to a *tuple* $(f_1, \dots, f_n)$ is computed as follows:

1. $r \longleftarrow 0$.
2. If $f = 0$, then return r.
3. If there is no i such that $\mathrm{LM}(f_i)$ divides $\mathrm{LM}(f)$, then $f \longleftarrow f - \mathrm{LT}(f)$ and go to (2).
4. If there is such i, pick the *least* such i.
5. $r \longleftarrow r + \frac{\mathrm{LT}(f)}{\mathrm{LT}(f_i)} f_i, \quad f \longleftarrow f - \mathrm{LT}(f)$.
6. Go to (2).

The division algorithm above terminates thanks to the well-foundedness of a monomial order. It is obvious that $\overline{f}^{(f_1,\dots,f_n)}$ gives *one of* normal forms of f modulo $\{f_1, \dots, f_n\}$, but the result depends on the ordering of f_i's and might result in different value after reordering.

The prominent feature of Gröbner basis G is that every polynomial has the *unique* normal form with respect to it, and hence every polynomial has the unique remainder w.r.t. it regardless of particular ordering of elements of G. In other words, $\overline{f}^G$ becomes well-defined for a *set* G, not a tuple, and it coincides with the unique G-normal form of f:

Fact 1.3 (Buchberger). *Let I be an ideal and $G = \{g_1, \dots, g_k\} \subset I$ be a finite subset. The followings are equivalent:*

1. *G is a Gröbner basis for I,*
2. *Every non-zero $f \in I$ is reducible by G,*
3. *Every $f \in I$ has the unique G-normal form 0,*
4. *G generates I and $\xrightarrow[G]{}$ has the Church–Rosser property; i.e. every $f \in k[\mathbf{X}]$ has the unique G-normal form.*

Proof. (1) $\Longrightarrow$ (2): Pick any non-zero $f \in I$. By the assumption, we have $\langle \mathrm{LT}(I)\rangle = \langle \mathrm{LT}(g_1), \dots, \mathrm{LT}(g_k)\rangle$. Clearly we have $\mathrm{LT}(f) \in \langle \mathrm{LT}(I)\rangle$; hence the is some i such that $\mathrm{LT}(g_i) \mid \mathrm{LT}(f)$. Hence we have $f \xrightarrow[G]{} f - \frac{\mathrm{LT}(f)}{\mathrm{LT}(g_i)} g_i$.

(2) $\Longrightarrow$ (3): pick $f \in I$. By the well-foundedness of a monomial ordering, f must have at least one G-normal form f^*. But by (2), such f^* must be zero because otherwise there must be h such that $f^* \xrightarrow[G]{} h$, which contradicts that f^* is G-normal form.

(3) $\Longrightarrow$ (4): It is clear that G generates I. Pick any $f \in k[\mathbf{X}]$ and let $f_1, f_2 \in k[\mathbf{X}]$ be G-normal forms of f. Then, by the definition of the reduction relation, clearly we have $f_1 - f_2 \in I$. By (3), we have $f_1 - f_2 \xrightarrow[G]{*} 0$. But, since the relation $\xrightarrow[G]{*}$ is compatible with addition, we have $f_1 = f_1 - f_2 + f_2 \xrightarrow[G]{*} f_2$. Then, since both f_1 and f_2 are G-normal form, we must have $f_1 = f_2$.

(4) $\Longrightarrow$ (1): it suffices to show that for any $f \in I$ there is some $g \in G$ with $\mathrm{LM}(g) \mid \mathrm{LM}(f)$. Since G generates I and f has the unique G-normal form, we have $\overline{f}^G = 0$. In particular, $\mathrm{LM}(f)$ must be canceled at the first iteration of Division Algorithm 1.2 and the step (4) is executed. In particular, there is some $g \in G$ such that $\mathrm{LM}(g) \mid \mathrm{LM}(f)$. This situation is exactly what we wanted. □

Corollary 1.4 (Buchberger). *Every ideal over $k[\mathbf{X}]$ has a Gröbner basis. In particular, there is an algorithm that, given a monomial ordering $\prec$ and set $\{f_1, \dots, f_n\}$ of polynomials, computes a Gröbner basis of $\langle f_1, \dots, f_n\rangle$ with respect to $\prec$.*

We can solve various problems about polynomial rings and modules with Gröbner basis. For example, the original motivation behind Gröbner basis is to solve the *Ideal Membership Problem*:

Definition 1.5 (***Ideal Membership Problem***)**.** The *Ideal Membership Problem* is the following problem: given any polynomials $f, f_1, \dots, f_n \in k[\mathbf{X}]$, decide whether $f \in \langle f_1, \dots, f_n \rangle$ or not.

Theorem 1.6 (**Buchberger**)**.** *Ideal Membership Problem is decidable.*

Proof. Let $G = \{g_1, \dots, g_k\}$ be a Gröbner basis of $I := \langle f_1, \dots, f_n \rangle$. It is clear that $f \in I$ iff $f \xrightarrow[G]{*} 0$.

But since G is a Gröbner basis and by Theorem 1.3, whether $f \xrightarrow[G]{*} 0$ or not can be easily determined just by checking if $\overline{f}^G = 0$. □

1.2 ALGORITHMS FOR COMPUTING GRÖBNER BASES

Buchberger gave an algorithm to compute Gröbner basis for a given ideal. Later, other, more efficient algorithms are proposed for computing Gröbner basis. In this section, we first review the basics of Buchberger's algorithm and then briefly skim through other Gröbner basis algorithms.

1.2.1 *Buchberger Algorithm*

The central concept in Buchberger Algorithm is the S-polynomial of a pair of polynomials:

Definition 1.7. The *S-polynomial* $S(f, g)$ of polynomials f and g is defined as follows:

$$S(f,g) := \frac{\mathrm{LCM}(\mathrm{LT}(f), \mathrm{LT}(g))}{\mathrm{LT}(f)} f - \frac{\mathrm{LCM}(\mathrm{LT}(f), \mathrm{LT}(g))}{\mathrm{LT}(g)} g.$$

Lemma 1.8 (**Buchberger Criterion**)**.** *$G = \{g_1, \dots, g_n\}$ is a Gröbner basis if and only if $S(g_i, g_j) \xrightarrow[G]{*} 0$ for all $i < j \leq n$.*

The intuition behind the Buchberger's algorithm is to add enough S-polynomials so that the Buchberger Criterion holds.

Algorithm 1.9 (**Buchberger Algorithm**)

1 **INPUT** $f_1, \dots, f_n \in k[\mathbf{X}]$
2 **OUTPUT** a Gröbner basis G for an ideal $\langle f_1, \dots, f_n \rangle$
3 $G \leftarrow (f_1, \dots, f_n)$
4 $R \leftarrow \emptyset$
5 **do**
6 $\quad G \leftarrow G \frown R$
7 $\quad R \leftarrow \left\{ \overline{S(p,q)}^G \;\middle|\; p \neq q \in G \right\} \setminus \{0\}$
8 **until** $R = \emptyset$
9 **return** G

The heaviest part of Buchberger's algorithm is polynomial-remaindering. Many of the improvements to Gröbner basis computation proposed so far concentrate on how to reduce the total count of polynomial remaindering.

One typical example of such heuristics is given by the following:

Lemma 1.10 (**Coprime Criterion**)**.** *If leading terms of f and g are coprime, i.e. if $\mathrm{LCM}(\mathrm{LT}(f), \mathrm{LT}(g)) = \mathrm{LT}(f)\,\mathrm{LT}(g)$, then $S(f,g) \xrightarrow[\{f,g\}]{} 0$.*

Hence, the algorithm obtained by replacing Line 7 in Buchberger Algorithm with the following will terminate and outputs a Gröbner basis of an input:

$$R \leftarrow \left\{ \overline{S(p,q)}^G \;\middle|\; p, q \in G, \mathrm{LT}(p) \text{ and } \mathrm{LT}(q) : \textit{coprime} \right\} \setminus \{0\}.$$

1.2.2 *Gröbner Basis via Homogenisation*

In homogeneous cases, i.e. when an input ideal is generated by homogeneous polynomials, Gröbner basis computation can be made much easier. Furthermore, for any monomial ordering $\prec$ and an ideal I, there is $\prec^{\mathrm{h}}$ such that dehomogenising a $\prec^{\mathrm{h}}$-Gröbner basis of I^{h} gives a Gröbner basis for I, where I^{h} is an ideal in $k[\mathbf{X}, Y]$ given by homogenising generators of I. Precise definitions and facts are as follows:

Definition 1.11. Let $f = \sum_{\gamma \in \Gamma} c_\gamma \mathbf{X}^\gamma \in k[\mathbf{X}]$ be a polynomial with $c_\alpha \neq 0$ of total degree d (i.e. $d = \max_{\gamma \in \Gamma} \deg \gamma$), $I = \langle f_1, \ldots, f_\ell \rangle \subseteq k[\mathbf{X}]$ an ideal and Y a variable distinct from X_i's.

1. The *homogenisation* f^{h} of f by a variable Y is given as follows:
$$f^{\mathrm{h}}(\mathbf{X}, Y) := Y^d f(X_1 Y^{-1}, \ldots, X_n Y^{-1}) = \sum_{\gamma \in \Gamma} c_\gamma \mathbf{X}^\gamma Y^{d-\deg(\gamma)}.$$
2. For any homogeneous polynomial $f \in k[\mathbf{X}, Y]$, the *dehomogenisation* $f^d \in k[\mathbf{X}]$ is defined by $f^{\mathrm{d}}(\mathbf{X}) := f(\mathbf{X}, 1)$.
3. For a set (or tuple) $F \subseteq k[\mathbf{X}]$ of polynomials, put $G^{\mathrm{h}} := \{ g^{\mathrm{h}} \mid g \in F \}$; similar for G^{d} for $G \subseteq k[\mathbf{X}, Y]$.
4. A monomial ordering $\prec^{\mathrm{h}}$ on $\mathbf{X}Y$ is defined as follows:
$$\mathbf{X}^\alpha Y^d \prec^{\mathrm{h}} \mathbf{X}^\beta Y^e \overset{\text{def}}{\iff} \begin{cases} \mathbf{X}^\alpha \prec \mathbf{X}^\beta, \text{or} \\ \mathbf{X}^\alpha = \mathbf{X}^\beta \text{ and } d < e. \end{cases}$$

Fact 1.12. *Let $\prec$ be a monomial ordering on $\mathbf{X}$, $I = \langle f_1, \ldots, f_\ell \rangle \subseteq k[\mathbf{X}]$ an ideal and G be a Gröbner basis for $\langle f_1^{\mathrm{h}}, \ldots, f_\ell^{\mathrm{h}} \rangle$ with respect to $\prec^{\mathrm{h}}$. Then G^d is a Gröbner basis for I with respect to $\prec$.*

Proof. See [13, Theorem 5]. □

Thanks to Theorem 1.3, we can convert any Gröbner basis algorithm for homogeneous ideals into general Gröbner basis computation.

1.2.3 *Hilbert-driven Algorithm for Homogeneous Ideal*

In the previous section, we have seen that Gröbner bases of general ideals can be reduced to those of *homogeneous* ones. We introduce one powerful homogeneous Gröbner basis computation algorithm: the Hilbert-driven algorithm.

Definition 1.13 (Hilbert–Poincaré series). Let m be a natural and I homogeneous ideal. For any set R of polynomials, we write $R_m := \{ h \in R \mid \deg(h) = m \}$.

1. The *Hilbert function* of $k[\mathbf{X}]/I$ at m is defined by:
$$\mathrm{HF}_I(m) := \dim(k[\mathbf{X}]_m / I_m).$$
2. The *Hilbert–Poincaré series* $P_I(t)$ of I is the generating function of its Hilbert function; i.e:
$$P_I(t) := \sum_{m=0}^{\infty} \mathrm{HF}_I(m) t^m$$

The following theorem illustrates why we define Hilbert–Poincaré series here:

Theorem 1.14. *For any homogeneous ideal I and finite $G \subset I$, the following are equivalent:*

1. *G is a Gröbner basis for I;*
2. $P_I(t) = P_{\langle \mathrm{LM}(G) \rangle}(t)$.

So, if one has an access to the Hilbert–Poincaré series for a given ideal, then one can use it to determine whether a procedure can stop or not.

We can easily compute the Hilbert–Poincaré series of given monomial ideal:

Lemma 1.15. *For a monomial ideal $I \subseteq k[X_1, \dots, X_n]$, one can compute Hilbert–Poincaré series $\mathrm{HPS}(I) = P_I(t)$ as follows.*

First let T be a set of minimal generators of I; i.e. the subset of monomials in I such that for any monomial $\mathbf{X}^\gamma \in I$ there is $\mathbf{X}^\delta \in T$ dividing $\mathbf{X}^\gamma$ and $\mathbf{X}^\alpha \perp \mathbf{X}^\beta$ for any $\mathbf{X}^\alpha, \mathbf{X}^\beta \in T$. Then $\mathrm{HPS}(I) = \mathrm{HPS}(T)$ can be computed by the following recurrence equation:

$$\mathrm{HPS}(T) = \begin{cases} (1-t)^{-n} & (\textit{if } T = \emptyset) \\ 0 & (\textit{if } T = \{ 1 \}) \\ (1-t)^{|T|-n} & (\textit{if } \forall m \in T \ \deg(m) = 1) \\ \mathrm{HPS}(T \cup \{ X_i \}) + t \cdot \mathrm{HPS}(T : X_i) & (\textit{if } \exists m \in T \ \deg(m) > 1 \land X_i \mid m). \end{cases}$$

Here, $(T : X_i) = \{ \mathbf{X}^\gamma \mid \mathbf{X}^\gamma X_i \in T \}$.

Hence, one can compute the Hilbert–Poincaré series of a given ideal provided that if one knows its Gröbner basis. Since we want to use Hilbert–Poincaré series to compute Gröbner basis, this situation might seem nonsense. But, by its definition, Hilbert–Poincaré series is defined for a polynomial and doesn't depend on particular monomial ordering. So, if one wants to compute a Gröbner basis for homogeneous ideal with respect to "heavy" monomial ordering, say $\prec$, one can calculate the Hilbert–Poincaré series from a Gröbner basis with respect to lighter ordering, e.g. $\prec_{\mathrm{grevlex}}$, and then use it to compute a Gröbner basis with respect to $\prec$. Here, one can regard this process as a *conversion of monomial ordering*.

1.2.4 *Faugère's F_4 Algorithm*

Faugère [17] proposed a matrix-based Gröbner basis computation algorithm called F_4. The basic idea behind F_4 is that polynomial remaindering can be seen as a matrix triangulation and one can use techniques in linear algebra to perform polynomial reduction simultaneously and efficiently.

Definition 1.16. Let $F = (f_1, \dots, f_\ell) \in k[\mathbf{X}]^m$ be a finite tuple of polynomials. We define matrix $M(F)$ as follows. First, let $\mathrm{Mon}(F)$ be the set of all monomials in F. We enumerate $\mathrm{Mon}(F)$ in the $\prec$-ascending order; i.e. $\mathrm{Mon}(F) = \{ \gamma_1 \succ \gamma_2 \succ \cdots \succ \gamma_\ell \}$. Then $M(F) = (a_{ij})_{i \le m, j \le \ell}$ is $m \times \ell$-matrix defined by:

$$a_{ij} := \text{the coefficient of } \gamma_j \text{ in } f_i.$$

Conversely, for any $M = (a_{ij})_{i,j} \in \mathrm{M}_{m,\ell}(k)$ and set $\Gamma = \{ \gamma_1 \succ \gamma_2 \succ \cdots \succ \gamma_\ell \}$, we define the set of *row polynomials* of M, denoted by $\mathrm{rows}(M, \Gamma)$ or $\mathrm{rows}(M)$ if Γ is obvious, as follows:

$$f_i := \sum_{j \le \ell} a_{ij} \gamma_j$$
$$\mathrm{rows}(M, \Gamma) := \{ f_i \mid i \le n, f_i \neq 0 \}.$$

Then, the F_4 algorithm is given as follows:

Algorithm 1.17 (**F_4 Algorithm**)

Input: $F = (f_1, \dots, f_m)$ a list of polynomials
Output: G, a Gröbner basis of $\langle f_1, \dots, f_m \rangle$
$G \leftarrow F$
$B \leftarrow \{ \{ i, j \} \mid i < j \leq m \}$
while $(B \neq \emptyset)$
  $B' \leftarrow$ any nonempty subset of B
  $B \leftarrow B \setminus B'$
  $L \leftarrow \left\{ \frac{\mathrm{LCM}(\mathrm{LM}(f_i), \mathrm{LM}(f_j))}{\mathrm{LT}(f_i)} f_i \;\middle|\; \{ i, j \} \in B' \right\}$
  $F \leftarrow \left\{ \frac{\mathbf{X}^\alpha}{\mathrm{LM}(f_i)} f_i \;\middle|\; \mathbf{X}^\alpha \in \mathrm{Mon}(L), i :$ the least with $\mathrm{LM}(f_i) \mid \mathbf{X}^\alpha \right\}$
  $M \leftarrow$ upper-triangular form of $M(F)$
  $G \leftarrow G \frown \mathrm{rows}(M)$
  $B \leftarrow B \cup \{ \{ i, m + j \} \mid i \leq m, 1 \leq j \leq |\mathrm{rows}(M)| \}$
  $m \leftarrow m + |\mathrm{rows}(M)|$
end
return G

Theorem 1.18 (**Faugère [17]**). *The F_4 terminates and correctly computes a Gröbner basis regardless of choice of B' at each step.*

Proof. See Faugère [17, Theorem 2.2] or Cox–Little–O'Shea [13, Theorem 2]. □

1.2.5 *F_5 and Signature-based Algorithms*

Faugère [18] proposed F_5 algorithm, which is known to be one of the fastest Gröbner basis computation algorithm. The central concept of the F_5 algorithm is a *signature*. The intuition behind signature-based algorithms is that making use of information of syzygy module can significantly reduce the burden of computing Gröbner basis. The exposition of signature-based algorithms is based on Gao–Iv–Wang [20].

Definition 1.19. For any tuple of polynomials $\mathbf{g} = (g_1, \dots, g_m)$, the *syzygy module* $\mathbf{H}(\mathbf{g})$ of $\mathbf{g}$ is the submodule of $k[\mathbf{X}]^m$ defined by:

$$\mathbf{H}(\mathbf{g}) := \left\{ \mathbf{u} = (u_1, \dots, u_m) \in k[\mathbf{X}]^m \;\middle|\; \mathbf{u} \cdot \mathbf{g} = \sum_{i=1}^{m} u_i g_i = 0 \right\}.$$

Let $M = M(\mathbf{g})$ be a submodule of $k[\mathbf{X}]^m \times k[\mathbf{X}]$ defined by:

$$M := \{ (\mathbf{u}, \mathbf{u} \cdot \mathbf{g}) \mid \mathbf{u} \in k[\mathbf{X}]^m \}$$

The idea behind signature-based algorithms is to keep track of non-trivial relations on generators by treating not only the ideal I but also a linear coefficients in $k[\mathbf{X}]$. Such information will be useful to predict unnecessary polynomial divisions. As a bonus, we can also compute basis for the syzygy module $\mathbf{H}$. To state the F_5 in detail, we must introduce some more definitions:

Notation. *We denote the i^{th} unit basis in $k[\mathbf{X}]^m$ by $\mathbf{e}_i$.*

Definition 1.20. 1. A *monomial* in $k[\mathbf{X}]^m$ is a element $\mathbf{X}^\alpha e_i \in k[\mathbf{X}]^m$ for any $\alpha \in \mathbb{N}^n$. Note that every element of $k[\mathbf{X}]^m$ can be expressed uniquely as a k-linear combination of module monomials.

2. A module monomial $\mathbf{X}^\alpha \mathbf{e}_i$ *divides* $\mathbf{X}^\beta \mathbf{e}_j$, denoted by $\mathbf{X}^\alpha \mathbf{e}_i \mid \mathbf{X}^\beta \mathbf{e}_j$, if $i = j$ and $\mathbf{X}^\alpha \mid \mathbf{X}^\beta$; the *quotient* is given by $\mathbf{X}^{\beta-\alpha}\mathbf{e}_i$.

3. A *module ordering* on $k[\mathbf{X}]^m$ is a well-order which is compatible with multiplication by monomials of $k[\mathbf{X}]$.

4. Let $\prec$ be a monomial ordering on $k[\mathbf{X}]$ and $\lhd$ module ordering on $k[\mathbf{X}]^m$. $\lhd$ is *compatible* with $\prec$ if, $\mathbf{X}^\alpha \prec \mathbf{X}^\beta$ holds if and only if, for any $i \leq m$, $\mathbf{X}^\alpha \mathbf{e}_i \lhd \mathbf{X}^\beta \mathbf{e}_i$.

5. For any $\mathbf{u}$, we define $\mathrm{LM}_\lhd(\mathbf{u})$ to be the $\lhd$-maximum monomial of $\mathbf{u}$. Likewise we define $\mathrm{LC}_\lhd(\mathbf{u})$ and $\mathrm{LT}_\lhd(\mathbf{u})$. For any pair $p = (\mathbf{u}, v) \in M$, $\mathrm{sig}(p) := \mathrm{LM}(\mathbf{u})$ is called the *signature* of p.

6. For any $p_i = (\mathbf{u}_i, v_i), r \in k[\mathbf{X}]^m \times k[\mathbf{X}]$ $(i = 1, 2)$, we define *top-reduction* relation $p_1 \xrightarrow[p_2]{} r$ by:

$$p_1 \xrightarrow[p_2]{} p_1 - \frac{\mathrm{LT}(v_1)}{\mathrm{LT}(v_2)} p_2 \iff \begin{cases} v_1, v_2 \neq 0, \mathrm{LM}(v_2) \mid \mathrm{LM}(v_1), \text{and} \\ \frac{\mathrm{LM}(v_1)}{\mathrm{LM}(v_2)} \mathrm{LM}(\mathbf{u}_2) \unlhd \mathrm{LM}(\mathbf{u}_1), \end{cases}$$

$$p_1 \xrightarrow[p_2]{} p_1 - \frac{\mathrm{LT}(\mathbf{u}_1)}{\mathrm{LT}(\mathbf{u}_2)} p_2 \iff \begin{cases} v_1 \neq 0, v_2 = 0, \mathbf{u}_2 \neq \mathbf{0}, \text{and} \\ \mathrm{LM}(\mathbf{u}_2) \mid \mathrm{LM}(\mathbf{u}_1). \end{cases}$$

7. $G \subseteq M$ is called a *signature Gröbner basis* if any non-zero $p \in M$ is top-reducible by some $q \in G$.

As the terminology suggests, signature Gröbner bases give Gröbner bases:

Theorem 1.21 (Gao–Iv–Wang [20, Proposition 2.2]). *Let $G \subseteq M$ be a signature Gröbner basis. Then* $\mathbf{G}_0 := \{ \mathbf{u} \mid \exists v\ (\mathbf{u}, v) \in G \}$ *is a Gröbner basis for the syzygy module* $\mathbf{H}$ *and* $G_1 := \{ v \mid \exists \mathbf{u}\ (\mathbf{u}, v) \in G \}$ *is for I.*

Proof. Clear by Theorem 1.3, the characterisation of Gröbner basis. □

Hence, finding a Gröbner basis for a given ideal reduces to find a signature Gröbner basis of associated M. We now see two main ingredients of signature-based algorithms: Syzygy and Signature criteria.

Definition 1.22. Let $p_i = (\mathbf{u}_i, v_i) \in k[\mathbf{X}]^m \times k[\mathbf{X}]$ $(i = 0, 1)$ and $G \subset k[\mathbf{X}]^m \times k[\mathbf{X}]$ be finite.

1. A top-reduction $p_1 \xrightarrow[p_2]{} r$ is *regular* if $v_2 \neq 0$ and $\mathrm{sig}(r) = \mathrm{sig}(p_1)$; we call it *super* otherwise.

2. A pair p_1 is *eventually super top-reducible* by G if it has at least one minimal G-regular top-reduction which can be super top-reduced by G.

3. p_1 is *covered* by p_2 if $\mathrm{LM}(\mathbf{u}_2) \mid \mathrm{LM}(\mathbf{u}_1)$, $t\,\mathrm{LM}(v_2) \prec \mathrm{LM}(v_1)$ where $t = \mathrm{LM}(\mathbf{u}_1) / \mathrm{LM}(\mathbf{u}_2)$. p_1 is covered by G if it is covered by some $q \in G$.

4. First put:

$$t := \mathrm{LCM}(\mathrm{LM}(v_1), \mathrm{LM}(v_2)), \quad t_i := \frac{t}{\mathrm{LM}(v_i)}\ (i = 1, 2)$$
$$c := \frac{\mathrm{LC}(v_1)}{\mathrm{LC}(v_2)}, \quad T := \max\{ t_1 \mathrm{LM}(\mathbf{u})_1, t_2 \mathrm{LM}(\mathbf{u}_2) \}.$$

And let j be such that $T = t_j \mathrm{LM}(\mathbf{u}_j)$. If $T = \mathrm{LM}(t_j \mathbf{u}_j - c t_{1-j} \mathbf{u}_{1-j})$, we define the *S-pair* $S_{\mathrm{pr}}(p_1, p_2)$ and *S-signature* $S_{\mathrm{sig}}(p_1, p_2)$ [1] of p_1 and p_2 by

$$S_{\mathrm{pr}}(p_1, p_2) := t_j p_j, \quad S_{\mathrm{sig}}(p_1, p_2) := T = \mathrm{LM}(t_j \mathbf{u}_j - c t_{1-j} \mathbf{u}_{1-j}).$$

1 In Gao–Iv–Wang [20], they are called "J-pair" and "J-signature".

Theorem 1.23 (Gao–Iv–Wang [20, Theorem 2.4]). *Let $G \subseteq M$ be finite, and suppose that for any term $T \in k[\mathbf{X}]^m$ there is some $(\mathbf{u}, v) \in G$ and monomial t such that $T = t\,\mathrm{LM}(\mathbf{u})$. Then the following are equivalent:*

1. *G is a signature Gröbner basis for M,*
2. *Every S-pair of G is eventually super top-reducible by G,*
3. *Every S-pair of G is covered by G.*

Proof. See Gao–Iv–Wang [20]. □

As in the polynomial ideal case, one can compute a signature Gröbner basis by adding S-pairs not reduced to zero. Furthermore, we can get the following criteria to discard pairs without actual reduction.

Corollary 1.24 (Syzygy Criterion, [20, Corollary 2.5]). *For any pairs $p_1, p_2 \in G$, if the S-pair $S_{\mathrm{pr}}(p_1, p_2)$ is top-reducible by some syzygy $(\mathbf{u}, 0) \in M$, then $S_{\mathrm{pr}}(p_1, p_2) \xrightarrow[G]{*} 0$.*

Corollary 1.25 (Signature Criterion, [20, Corollary 2.6]). *For any $p_1, p_2 \in G$, if the $S_{\mathrm{pr}}(p_1, p_2)$ is covered by G or other J-pair of G, then $S_{\mathrm{pr}}(p_1, p_2) \xrightarrow[G]{*} 0$. Hence, a signature Gröbner basis needs at most only one S-pair for each signature.*

By the above two criteria, one can compute a signature Gröbner basis in a similar way to Buchberger's Algorithm, but in more efficient way. Suppose one wants to compute a signature Gröbner basis for $\langle f_1, \dots, f_m \rangle$. Fist, one compute the list S of principal syzygies as $S := \left\{ f_j \mathbf{e}_i - f_i \mathbf{e}_j \;\middle|\; i < j \leq m \right\}$ and set $G := \{ (\mathbf{e}_i, f_i) \mid i \leq m \}$. Then, one iterates over S-pairs of elements of G. If the S-pair s corresponds to a syzygy, then one store it in S. Otherwise, one append it to G if s satisfies neither of above criteria. This process will terminate in finite steps, and gives a signature Gröbner basis as desired. For the complete description of algorithm, the readers can refer to Gao–Iv–Wang [20, Figure 3.1].

2

PURELY FUNCTIONAL PROGRAMMING IN HASKELL

In this chapter, we will review basic concepts and methods in Haskell programming language. Readers familiar with Haskell can skip this chapter.

2.1 OVERVIEW OF HASKELL

Haskell [25] is a lazy statically typed purely functional programming language which has been evolving for the decades.

Being *statically typed* means that every Haskell program will be tested if the whole program has the valid type. A type can be regarded as a tag representing their kind. Haskell has a very strong and expressive type-system which enforces correctness at compile-time type-checking.

In *Functional Programming* paradigm, one builds up a program by composing *functions* as building blocks recursively. Programs in functional programming language seems much *declarative*, that is to say, similar to definitions in mathematics.

The *purity* means that the every expression of Haskell *does not* have any *side-effects*, that is, returns the same result when given the same inputs. This doesn't mean that Haskell cannot treat side-effect; indeed, Haskell employs the concept of *monads* [58] to encapsulate side-effects while maintaining the purity. As a mental model, every effectful computation can be regarded as an abstract instruction, and then the compiler finally interprets it to the real I/O. It is this purity that enables Haskell to enjoy the lightweight parallelism and rewriting rules.

Expressions in Haskell are evaluated *lazily*; i.e. expression will not be evaluated until their value is actually needed. With this feature, one can encode *infinite* objects in Haskell intuitively. For example, we can define the infinite list of Fibonacci sequence as follows:

```
fibs :: [ℤ]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

Though evaluating `fibs` alone won't terminate, one can freely cut-out a finite segment; for example, "`take 5 fibs`" results in `[1,1,2,3,5]`. One can even print the contents of `fibs` indefinitely:

```
ghci> print fibs
— [1,1,2,3,5, … <priting elements indefinitely>
```

In this chapter, we will provide a introductory explanation of programming in Haskell. For more details, the author refer readers to standard textbooks, such as Bird [6], Hutton [30] or Lipovača [51]

2.1.1 *Notation*

In what follows in this thesis, we use symbols in Table 2.1 in code fragments for the sake of readability.

In Haskell, one use `$`-operator to save parenthesises. For example, `map succ $ tail $ 1 : [2] ⌢ [3,4,5]` is equivalent to `map succ (tail (1 : [2] ⌢ [3,4,5]))`.

Table 2.1: Symbols in Code Fragments

Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code
$\underline{\mathbb{N}}$	Nat or Natural	$\underline{\mathbb{Z}}$	Integer	$\underline{\mathbb{Q}}$	Rational	$\underline{\mathbb{F}_p}$	F p
$::$	::	$=$	==	$\neq$	/=	$\lambda\ \vec{x} \to e$	\x⃗ -> e
$\times$	*	$\ltimes$	!*	$\frown$	++	$\ominus$	%-
$\simeq$	:~:	$\sim$	~	$\to$	->	$\leftarrow$	<-
$\Longrightarrow$	==>	$\Rightarrow$	=>	$:=$	.=	$:\Leftarrow$	.%=
$\subseteq$	`Subset`	$\leq$	<=	$\circ$	.	$\wedge$	.&&.
$\bullet$	.*	<$>	<$>	<*>	<*>	$\forall$	forall

2.2 FUNCTIONAL AND DECLARATIVE PROGRAMMING IN HASKELL

As stated above, Haskell is a statically-typed purely functional programming language. In this section, we will skim through how these features help a programming a lot.

2.2.1 *Programming with Recursive and Higher-order Functions*

In functional programming, a program is a composition of *functions*. Even control structures, such as if-conditionals and while-loops, can be simulated by functions if one doesn't care about efficiency. The secret of such expressivity is to use *recursion* and *higher-order functions*. Briefly, a recursive function is one that calls itself inside its definition. For example, the following program computes a Fibonacci sequence by recursion:

```
fib :: ℤ → ℤ
fib n | n ≤ 1     = 1
      | otherwise = fib (n − 1) + fib (n − 2)
```

A higher-order function is one that takes other functions as arguments. For example, one can write function representing `while`-loop as follows:

```
while :: (a → Bool) → (a → a) → a → a
while p iter a =
  if p a
  then while p iter (iter a)
  else a
```

The first two arguments of `while` are functions corresponding to the loop-condition and the body of loop.

The next function represents n-times bounded loop construct:

```
loop :: (a → a) → a → ℤ → a
loop iter a n = body a n
  where
    body x 0 = x
    body x n = body (iter x) (n − 1)
```

Thus, `loop f a n` applies a function `f` to `a` n-times. One can define local subroutines using **`where`** -clause; in the above, since the loop-body function `iter` doesn't change across iterations, so we define inner loop `body :: ℤ → a → a` which only takes the remaining number of iterations and current value. One might wonder the main definition of `loop` must be `loop n iter x = body n x`.

Furthermore, one can drop the arguments `n` and `a` from the definition; i.e. one can replace Line 2 with:

```
2 loop iter = body
```

Such reduction of variables is called . Actually, in Haskell, a function type `a → b → c`, a type of functions taking two arguments of types `a` and `b` and returns `c`, is just a short hand for `a → (b → c)`; i.e. that of functions taking an argument of type `a` and returns a function of type `b → c`. In other words, every function in Haskell is *curried* by default. Since it defines a function taking an argument of type `a → a` and returns a function of type `a → ℤ → a`, the above new definition is completely valid and have the same meaning.

With `loop` function, one can even rewrite `fib`-function in imperative style:

```
fib :: ℤ → ℤ
fib n = fst (loop (λ (a,b) → (b, a + b)) (1,1) n)
–– where fst :: (a, b) → a is the canonical projection.
```

Here, an expression of form `(λ x → e)` is called a *λ-abstraction*; it corresponds to an anonymous function or closure in other languages; `(λ x → e)` expresses the function which takes an argument `x` and returns `e`, which can depend on `x`. So, the above `fib` iterates the operation of shifting and adding n-times to the initial value $(1, 1)$.

Further, one can simplify the above code using *function composition* and η-reduction as follows:

```
  fib n = fst (loop (λ (a,b) → (b, a + b)) (1,1) n)
        = (fst ∘ loop (λ (a, b) → (b, a + b)) (1,1)) n
∴ fib   = fst ∘ loop (λ (a, b) → (b, a + b)) (1,1)
```

2.2.2 *Types in Haskell*

In Haskell, one can define (ADTs), or *data-type* simply. ADTs can be regarded as a free object generated by relations given by sum of products[1]. The following illustrates examples of simple ADTs:

```
data Unit = U        –– A type with just one element
data Boole = FF | TT  –– Truth–value
data PN = Zero | Succ PN
      –– A type of naturals, expressed as Peano numerals.

–– Simple arithmetic expression
data Expr = Lit PN
          | Expr :+ Expr
          | Expr :× Expr
          | Negate Expr
```

Note that `PN` and `Expr` has the recursive definition.

One can *pattern-match* on ADTs; this can be seen as a case-analysis in mathematical proof. For example, one can write a simple evaluator on arithmetic expressions as follows:

```
evalPeano :: PN → ℤ
evalPeano Zero     = 0
evalPeano (Succ n) = 1 + evalPeano n
```

1 More rigorously, algebraic data-types in Haskell can be regarded as the initial and final F-(co)algebra, where F is an endofunctor.

```

evalExpr :: Expr → ℤ
evalExpr (Lit n)    = evalPeano n
evalExpr (e :+ d)   = evalExpr e + evalExpr d
evalExpr (e :× d)   = evalExpr e × evalExpr d
evalExpr (Negate e) = − evalExpr e
```

The type-system of Haskell is based on *polymorphic λ-calculus*; more precisely, Haskell is some kind of extension of *Hindley–Milner type-system* [26]. In particular, the type-system of Haskell admits both *parametric* and *ad-hoc polymorphism*.

Parametric polymorphism is inherited from Hindley–Milner type-system. This allows as to define *polymophic data-types* and *functions*. For example, we can define a list-type[2] generically:

```
data List a = Nil | Cons a (List a)
empty :: List a
empty = []

ints :: List ℤ
ints = Cons 1 (Cons 2 (Cons 3 Nil))
```

One can define a mapping function[3] as follows:

```
1 mapList :: (a → b) → List a → List b
2 mapList f Nil         = Nil
3 mapList f (Cons x xs) = Cons (f x) (mapList f xs)
```

This function is *polymorphic*, or *generic*, in a sense that it can accept arbitrary function and lists matching its type. For example, one can write `mapList succ ints` or `mapList intToString ints`, where `succ :: Int → Int` and `intToString :: ℤ → String`. Furthermore, one can even omit the type annotation in Line 1; the compiler can *infer* the type of `mapList` to be most generic one!

Ad-hoc polymorphism, on the other hand, is some kind of polymorphism with *constraints*. Consider the following function for taking the sum of the given list of integers:

```
sumInt :: [ℤ] → ℤ
sumInt []       = 0
sumInt (a : as) = a + sum as
```

One can also have the summation function for `Double`s:

```
sumDouble :: [Double] → Double
sumDouble []       = 0
sumDouble (a : as) = a + sum as
```

Although they have almost identical definitions, types are different. We cannot define generically solely with parametric polymorphism; summation function cannot be defined for *arbitrary* types, but those with *additive structure*. In other words, the summation function can be defined for the types which satisfies the *constraint* that it is endowed with addition. This is where ad-hoc polymorphism can play a role. To express such constrained form of polymorphism, Haskell provides a functionality called *type-classes*. For example, Haskell provides the type-class `Num` for "numeric" types:

```
class Num a where
  ( + ) :: a → a → a
```

2 Haskell has the built-in type `[a]` for the list-type. Here, we define it on our own for the sake of exposition.
3 The `map` built-in function in Haskell.

```
  ( × ) :: a → a → a
  ( − ) :: a → a
  fromInteger :: ℤ → a
  …
```

With this, we can define `sum` function for arbitrary types which is an *instance* of `Num`!

```
sum :: Num a ⇒ [a] → a
sum []       = fromInteger 0
sum (x : xs) = x + sum xs
```

Then, since we have `Num` instances for ℤ and `Double`, we can use `sum` function for lists of integers and doubles. One can add an instance for type-classes at anytime[4].

One can even extend the type-class by defining subclass. For example, in Haskell, `Integral` type-class is defined for integer-like `Num`-types:

```
class Num a ⇒ Integral a where
  toℤ :: a → ℤ
  divMod :: a → a → (a, a) −− Integral division
  …
```

2.3 IMPERATIVE PROGRAMMING IN HASKELL WITH MONADS

So far, we have seen the *pure* part of Haskell. But, as claimed above, Haskell can also treat *effectful* computations, that is, functions with *side-effects*.

In this section, we will see how we can use *monads* to handle side-effects.

2.3.1 *Monads as a Modular Semantics*

Monads were introduced to the realm of computer science first by Moggi [58]. Although the concept of monad comes from category theory, we only consider the definition of monad only in Haskell.

Definition 2.1 (monad). A unary type m is a *monad* if it is endowed with two functions

```
return :: a → m a
(>=>)  :: (a → m b) → (b → m c) → (a → m c)
```

such that the following equalities hold:

$$\mathtt{return} \mathrel{>\!\!=\!\!>} f = f = f \mathrel{>\!\!=\!\!>} \mathtt{return}$$
$$(f \mathrel{>\!\!=\!\!>} g) \mathrel{>\!\!=\!\!>} h = f \mathrel{>\!\!=\!\!>} (g \mathrel{>\!\!=\!\!>} h)$$

We call the operator $(\mathrel{>\!\!=\!\!>})$ a *monadic composition*.

Intuitively, monads abstracts the concept of *sequential executions*. We regard a function of type `a → m b` as an effectful function from `a` to `b` with side-effect cared with `m`. In this view, the monadic composition operation $(\mathrel{>\!\!=\!\!>})$ can be regarded as a composition operator of effectful functions, generalising mere composition $(\circ)$. Then, the first law states that `return` behaves as the left- and right-identity to the monadic composition, generalising the identity function. The second law requires monadic composition to be associative; i.e. composition doesn't depend on particular composition ordering.

4 Strictly speaking, one can prevent this by *hiding* class definition.

In Haskell, one can use **do**-*natation* to write monadic program in imperative-style. First we define the monadic application operator, $f \gg= g$, as follows:

```
(>>=) :: Monad m ⇒ m a → (a → m b) → m b
ma >>= f = (id >=> f) ma
```

The next code fragment illustrates how a **do**-notation look like and how compiler desugars it to a monadic expression:

```
do a ← calc x            calc x        >>= (λ a →
   doSomething      ⇒    doSomething   >>= (λ _ →
   b ← other a      ⇒    other a       >>= (λ b →
   return (f a b)        return (f a b) )))
```

See Haskell 2010 Language Report [24, §3.14] for the formal specification of **do**-notation.

2.3.2 *Examples of Monads*

To get a picture, we see some examples of monads.

2.3.3 *Handling Failures by Maybe-monad*

In Haskell, *Maybe-monad* is often used to express the computation which can possibly *fail*.

```
1 data Maybe a = Nothing | Just a
2 instance Monad Maybe where
3   return = Just
4   Nothing >>= _ = Nothing
5   Just a  >>= f = f a
```

Above, `Nothing` denotes the failure of computation and absence of result. `Just a`, on the other hand, represents a succeeded computation with a result value `a`. Since we assume that failed computation cannot be recovered, we define the monadic composition on `Maybe`s so that:

1. If the preceding computation is failed (i.e. `Nothing`), then the entire computation must be failed (Line 4),
2. If the previous computation returns successfully with result `a` (i.e. `Just a`), then just feed it to the next step (Line 5).

For example, the following function tries to find an element of the given list with specified property:

```
find :: (a → Bool) → [a] → Maybe a
find p [] = Nothing
find p (x : xs)
  | p x       = Just x
  | otherwise = find p xs
```

Then, one can nest `find` functions freely as follows:

```
doubleOddElementInEvenList :: [[ℤ]] → Maybe ℤ
doubleOddElementInEvenList xss = do
  -- Finds a list of integers of even length.
```

```
  list ← find (even ∘ length) xss
  i ← find odd list  —— Pick an odd number from the list
  return (2 × i)     —— Double the result.
```

This takes a nested list of integers, then finds and doubles an odd number in the even-length list. For example,

```
ghci> doubleOddElementInEvenList [[1,2,3],[4,8],[5,6,7]]
Nothing

ghci> doubleOddElementInEvenList [[1,2,3],[4,5],[6,9]]
Just 10
```

2.3.4 *Non-deterministic Computation with List-monads*

Actually, *lists* can be regarded as a monad.

```
instance Monad [] where
  return x = [x]
  xs ≫= f = concat (map f xs)

concat :: [[a]] → [a] —— Flattening map
```

Let's redefine the `find` function as follows:

```
find :: (a → Bool) → [a] → [a]
find p []       = []
find p (x : xs)
  | p x       = x : find p xs
  | otherwise = find p xs
```

Then, the `doubleOddElementInEvenList` above returns now lists and can return multiple results:

```
ghci> doubleOddElementInEvenList [[1,2,3],[4,8],[5,6,7]]
[]

ghci> doubleOddElementInEvenList [[1,2,3],[4,5],[6,9]]
[10,18]
```

2.3.5 *Handling I/O with Monads*

We can also handle I/O actions, such as random number generation, operations on mutable reference, or read/write inputs to files, etc, with monads. Actually, Haskell provides the *`IO`-monad* for that purpose. The `IO`-monad differs from `Maybe` and list monads in a such a way that the actual semantics of `IO`-monads are defined at the *meta-level*, not the *object-level*. That is to say, values of `IO` can be regarded as an abstract directive specifying computations and interpreted directly by the compiler.

And, expressing I/O computations with a type `IO a`, one can distinguish impure computations from pure ones. In contrast to `Maybe` and list-monads, one cannot retrieve pure values from `IO`-value[5]. In this way, monads provides a way to treat impure computations yet retaining purity and type-safety at the same time.

5 Actually, there is a function to convert value of type `IO a` to pure value of type `a`, but it is marked as "unsafe" and not recommended to be used frequently.

2.4 ADVANCED TOPICS

In this section, we will briefly review the advanced features eventually used in what follows. Indeed, features we will discuss in this section is not standard ones in Haskell, but those implemented in *Glasgow Haskell Compiler* [21] (GHC), a flagship compiler of Haskell.

2.4.1 *Higher Rank Polymorphism and ST-monads*

In the original Hindley–Milner type-system, one can treat only polymorphism of *rank* 1. In other words, higher functions can take a function, but an argument *cannot* be polymorphic, i.e. cannot have universal quantifiers over types. One might may that `map` function's type `(a → b) → [a] → [b]` is not rank-1 polymorphic. But, if we revive the universal quantifier over type variables, the actual type of `map` is indeed `∀ a b. (a → b) → [a] → [b]`, and hence it actually is rank-1 polymorphic.

This restriction is for the sake of decidability and completeness of type-inference. But, in many practical cases, type-inference can be done for higher rank polymorphism, and Haskell provides `Rank2Types` and `RankNTypes` language extensions which enables rank-2 and higher rank polymorphic functions.

As we will see in Section 4.5.1, higher polymorphism itself is particularly useful and improve the expressivity of the language. Furthermore, Launchbury and Peyton Jones [48] proposed the another way to exploit higher polymorphism to improve *type-safety* and *purity* at the same time: an *ST-monad*.

ST-monad provides a way to treat *mutable* states within *pure* computations. The problem that ST-monad solves is as follows. Although we can treat mutable states in IO-monad, one cannot retrieve the result of computations with mutable states in IO safely, even though the entire computation is pure at total. This restriction is inherent in IO-monad, since any unsafe operations can be taken place in IO-computation. Providing a unwrapping function just for a mutable reference doesn't make sense here, because if one have something like `readRef :: IO (IORef a) → a` and have value `nestedRef :: IORef (IORef Int)`, then one can extract mutable states outside by `readRef (return nestedRef)`, which breaks purity!

Code 1 The interface of ST-monad

```
type ST s a
data STRef s a
newSTRef   :: a → ST s (STRef s a)
writeSTRef :: a → STRef s a → ST s ()
readSTRef  :: STRef s a → ST s a
runST      :: (∀ s. ST s a) → a
```

ST-monad solves this problem with a trick using *Skolem variable* to encapsulate mutability and prevent them from leaking outside. Code 1 shows the basic interface of ST-monad. Intuitively, a type-variable `s` represents the internal state, or "world", of whole computation. The type `STRef s a` corresponds to a mutable state with internal state `s`. Functions `newSTRef`, `writeSTRef` and `readSTRef` correspond to operations of creating, writing and reading mutable states respectively. As the types indicate, each operation results in the monadic value inside `ST s` with the same type parameter `s` as the `s`. In other words, inside `ST s`-monad, one can freely access mutable states of type `STRef s a`.

We use `runST` function to retrieve the result of `ST s`-computation, where the Skolem trick comes into play. Note that `runST` has a higher polymorphic type; the first argument must be polymorphic, or generic, in type-variable `s`. The genericity constraint on `s` means that the entire computation must be agnostic about a specific internal state `s`; in particular, the result type `a` *cannot* depend a particular value of `s`. This can be described more rigorously by syntactic argument: `a` is bound the outermost position but `s` is bounded only inside the first argument of `runST`, hence `a` cannot depend on particular `s`. This situation is analogous to the eigenvariable, or *Skolem variable*, condition in formal logic: to deduce $\forall x\ \varphi$ from $\varphi[y/x]$, y must not occur in φ and this variable y is called eigenvariable or Skolem variable. In this way, rank-2 polymorphism can be used to prevent the leak of inside state.

2.4.2 *Generalised Algebraic Data-types and Dependent Types in Haskell*

In GHC, one can define *non*-parametric type with *Generalised Algebraic Data-types* (GADTs). We refer readers to Hinze [28] for more detailed explanation.

Consider the early example of data-type expressing arithmetic expressions:

```
data Expr = Lit PN
          | Expr :+ Expr
          | Expr :× Expr
          | Negate Expr
```

Suppose one wants to extend the expression with boolean conditionals and predicates, such as zero-test or boolean combinations. One simple modification is as follows:

```
data Expr = Lit PN
          | Expr :+ Expr
          | Expr :× Expr
          | Negate Expr
          | IsZero Expr -- Check if zero
          | IfThenElse Expr Expr Expr -- If-expression
          | Expr :&& Expr -- Booelean conjunction
          | Expr :|| Expr -- Boolean disjunction
          | Not Expr      -- Boolean negation
          | Boole Bool    -- Boolean literal
```

But this approach is rather *unsafe* in a sense that one can form ill-typed terms such as `True + 2` or `if 5 then True else 4`. One possible idea is to introduce a type-parameter to indicate the entire type of expression. But, in the standard Haskell, one can only define parametric type variable or completely *phantom* type, which doesn't occur in the actual definition of constructors. But with GADTs, one can control phantom type parameters in constructor definition as follows:

```
data Expr a where
  Lit        :: PN → Expr ℤ
  (:+)       :: Expr ℤ → Expr ℤ → Expr ℤ
  …
  IsZero     :: Expr ℤ → Expr Bool
  IfThenElse :: Expr Bool → Expr a → Expr a → Expr a
  (:&&)      :: Expr Bool → Expr Bool → Expr Bool
  Boole      :: Bool → Expr Bool
```

Then, the compiler rejects ill-typed terms at compile-time.

Another application of GADTs is to simulating *dependent types* in Haskell. Briefly speaking, types depending on expressions are called *Dependent Types*. GHC supports them via the *Promoted Data-types* language extension [70] since version 7.4.

For example, one can write *length-parametrised lists* as follows:

```
data PN = Zero | Succ
data Vec n a where
  Nil  :: Vec Zero a
  (:-) :: a → Vec n a → Vec (Succ n) a
```

So, the type `Vec` n `a` corresponds to the type of lists with exactly n elements of type `a`. With this, one can achieve a type-safe tail function as follows:

```
tailV :: Vec (Succ n) a → a
tailV (_ :- as) = a
```

In less-typed setting, `tail` function is partial; in particular `tail []` passes type-checking but halts with error at run-time. But in this setting, `tailV Nil` is rejected at compile-time, since `Nil` is of type `Vec Zero a`, which won't match with `Vec (Succ n) a`!

So far, so good. Next, consider the following `replicate` function, which returns a *simple* list consisting of a specified number of copies of the same element:

```
replicate :: ℕ → a → [a]
replicate 0 _ = []
replicate n a = a : replicate (n − 1) a
```

For example, `replicate 3` *True* returns `[`*True*`,` *True*`,` *True*`]`. How this function can be generalised to *length-parametrised* lists? One might write the following simple generalisation:

```
replicateV :: PN → a → Vec n a
replicateV Zero _ = Nil
replicateV (Succ n) a = a :- replicateV (n − 1) a
```

But this won't work well. Actually, the type parameter `n` in the result is *free* and independent of the first argument! So what we need in this situation is something like:

```
replicateV :: (n :: PN) → a → Vec n a
```

Then, the type parameter `n` is bound by the first argument and must coincide with the length of the result. In other words, the type of `replicateV` depends on the *value* `n`. Such a type can be expressed in type-systems with *full dependent-types*, but GHC doesn't support such type-level argument directly. But, one can use *singleton* GADTs [16] in place of such type-level arguments.

Intuitively, the singleton type `Sing n` of type `n :: T` has exactly one inhabitant for each `n`, which has an "isomorphic" structure to `n`. For example, one can define the singleton types for type-level Peano numerals by GADTs as follows:

```
data Sing (n :: PN) where
  SZero :: Sing Zero
  SSucc :: Sing n     → Sing (Succ n)
```

With this, Peano-numeral version of `replicateV` can be readily implemented:

```
replicateV :: Sing n → a → Vec n a
replicateV SZero      _ = Nil
replicateV (SSucc n) a = a :- replicateV n a
```

GHC also has the built-in type-level natural numbers, which is implemented in terms of primitive integers and not represented as Peano numerals. One can also use singletons for built-in naturals, but pattern-matching on them is impossible and we need additional dedicated facilities to treat them seamlessly. We will turn-back to and solve this problem in Section 4.2.3.

Part II

RESEARCH CONTRIBUTIONS

3

FREER MONADS, MORE EXTENSIBLE EFFECTS [1]

3.1 ABSTRACT

We present a rational reconstruction of extensible effects, the recently proposed alternative to monad transformers, as the confluence of efforts to make effectful computations compose. Free monads and then extensible effects emerge from the straightforward term representation of an effectful computation, as more and more boilerplate is abstracted away. The generalisation process further leads to freer monads, constructed without the Functor constraint. The continuation exposed in freer monads can then be represented as an efficient type-aligned data structure. The end result is the algorithmically efficient extensible effects library, which is not only more comprehensible but also faster than earlier implementations.

As an illustration of the new library, we show three surprisingly simple applications: non-determinism with committed choice (LogicT), catching IO exceptions in the presence of other effects, and the semi-automatic management of file handles and other resources through monadic regions.

We extensively use and promote the new sort of 'laziness', which underlies the left Kan extension: instead of performing an operation, keep its operands and pretend it is done.

3.2 INTRODUCTION

That monads do not compose was recognised as a problem early on [66]. Two independently-written expressions using different side-effects (and hence monads) are difficult to combine in one program. Modifying a small part of a large program to use a new side-effect (e.g., adding debug output) sends ripples of changes throughout the code base. The very same difficulty of adding and combining effects has plagued denotational semantics [8]. In fact, monads, introduced by Moggi as a way to structure denotational semantics, inherited that problem. One can identify three approaches to solving it. The most popular is monad transformers [49], implemented in the widely used monad transformer library (MTL). They are based on Moggi's original idea of "monads with a hole", adding to it the lifting of monad operations through the transformer stack. The second approach combines monads through a quite complicated co-product [53], whose simplification has lead to the free monad popularised in Data types à la carte [67]. The third, presented just before monad transformers, looked at effects as an interaction and introduced side-effect–request handlers [8]. That idea of effect handlers, generalising exception handlers, was picked up in [4, 62], and developed into the language Eff. In Haskell, it was implemented as extensible effects [42] and [38].

We present, in §3.3, a unifying view: we derive the free monad and extensible effects by progressively abstracting the straightforward term representation of an effectful computation. Extensible effects emerge as the combination of the ideas of free monads and open union. The unifying, rational

1 The contents of this chapter is based on the following article: O. Kiselyov and H. Ishii. Freer monads, more extensible effects. *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, 50(12):94–105, December 2015 [41]. © 2015 ACM New York, NY, USA. The final publication is available on ACM Digital Library with doi: 10.1145/2804302.2804319.

reconstruction is not only edifying: it pointed to the further generalisation in §3.3.4: freer monads, free even from the Functor constraint. Freer (or, free-er, for emphasis) monad is an algebraic data type that is a monad by the very construction, just like list is a monoid by construction.

Besides intellectually satisfying, the freer monads are more economical with memory, avoiding rebuilding of the request data structure on each bind operation. Mainly, by exposing the continuation the freer monads made it easier to represent it differently, as a type-aligned sequence data structure [61], which improved the performance algorithmically. §3.4 describes the improved extensible effects library and §3.5 demonstrates its better performance on several benchmarks, in comparison with MTL and other effect handler libraries.

Our contribution thus is *rationally deriving* – telling the compelling story – of the freer monad, which supports the easy addition, composition and also subtraction (that is, encapsulation) of effects. It is so far the most efficient and expressive extensible monad. We demonstrate the expressivity on three applications, which were previously considered difficult for extensible effects or monads in general. §3.6 shows the exceptionally straightforward implementation of non-determinism with committed choice (the LogicT monad). §3.7 presents the surprisingly simple implementation of catching IO errors in monads other than IO. At last IO exceptions behave, with regard to other effects (State, in particular), just as non-IO exceptions. Finally, §3.8 is the ultimate demonstration of effect encapsulation: monadic regions, re-implementing and simplifying the transformer-based library of [44].

§3.9 describes the related work. The complete code is available at `http://okmij.org/ftp/Haskell/extensible/Eff1.hs`.

3.3 DERIVATION OF FREE-ER MONAD

In this section we derive the freer and extensible monads by progressively removing boilerplate from the term representation of effects. The result, however elegant, has poor performance, to be improved in §3.4.

3.3.1 *Reader Effect*

We start with the simplest side effect: dynamic binding, or `Reader` in the MTL terminology. `Reader` computations depend on a value supplied by the environment, that is, their context. A side-effect can be understood [8] as an interaction of an expression with its context. The possible requests can be specified as a data type, which in our case is[2]

```
data It i a = Pure a
            | Get (i → It i a)
```

Such an algebraic modelling of possible operations was pioneered in Haskell by Hughes [29] and is now known in Haskell as 'operational' [1]. Hinze [27] gave the lucid demonstration of this technique, to derive backtracking monad transformers. (We also deal with non-determinism, in §3.6). The expression `Pure e` marks the computation `e` that makes no requests, silently working towards a value of the type `a`. The request `Get k` asks the context for the (current dynamically-bound) value of the type `i`. Having received the value `i`, the computation `k i :: It i a` continues, perhaps asking for more values from the context. One may hence call `Get`'s argument `k` a continuation.

The simplest asking computation is

```
ask :: It i i
ask = Get Pure
```

2 The choice of the name It should become clear shortly.

which immediately returns the received value. Bigger computations are built with the help of the monad bind (≫=): `It i` is a monad.

```
instance Monad (It i) where
  -- return :: a → It i a
  return = Pure

  -- (≫=) :: It i a → (a → It i b) → It i b
  Pure x ≫= k = k x
  Get k' ≫= k = Get (k' <=< k)
```

The last clause in the definition of bind says that a computation that waits for an input and then continues as `k'`, and after that, as `k` – is the computation that continues after waiting as the composition of `k'` and `k`. The operation (<=<), Kleisli composition, is the composition of effectful functions:

```
(<=<) :: Monad m ⇒ (a → m b) → (b → m c) → (a → m c)
f <=< g = (≫= g) ∘ f
```

Here are two examples of bigger Reader computations

```
addGet :: Int → It Int Int
addGet x = ask ≫= λi → return (i+x)

addN :: Int → It Int Int
addN n = foldl (<=<) return (replicate n addGet) 0
```

The latter asks for `n` numbers and returns their sum.

The computations `addGet` and `addN` make requests to the context. We need to define how to reply, that is, how to "run" these computations. The following interpreter gives the same value `i` on each request: `It i a` is indeed interpreted as the `Reader` monad.

```
runReader :: i → It i a → a
runReader _ (Pure v) = v
runReader x (Get k) = runReader x (k x)
```

Unlike the MTL Reader, `It i a` may be treated differently: each request gets a new value, as if read from an input stream:

```
feedAll :: [i] → It i a → a
feedAll _ (Pure v) = v
feedAll [] _       = error "end of stream"
feedAll (h : t) (Get k) = feedAll t (k h)
```

In this interpretation, `It i a` is called an iteratee and `feedAll` an enumerator [40].

3.3.2 *Reader/Writer Effect*

Let us add another effect: rather than asking a context for a value, we tell the context. This is a Writer, or tracing effect.

```
data IT i o a = Pure a
              | Get   (i → IT i o a)
              | Put o (() → IT i o a)
```

The `Put o k` request tells the value `o` to the context. After the context acknowledges with `()`[3] the computation continues as `k ()`. The extended `IT i o` is also a monad:

```
instance Monad (IT i o) where
  return = Pure
  Pure x    >>= k = k x
  Get k'    >>= k = Get   (k' <=< k)
  Put x k'  >>= k = Put x (k' <=< k)
```

Again, a computation that tells the context and continues as `k'` and then as `k`, really continues as `k' <=< k`.

In MTL's Writer monad, the told value must have a `Monoid` type. Our `IT i o` has no such constraints. If we write a `Writer`-like `IT` interpreter to accumulate the told values in a monoid, it will have the `Monoid o` constraint then:

```
runRdWriter :: Monoid o => i -> IT i o a -> (a,o)
runRdWriter i m = loop mempty m
 where
   loop acc (Pure x)  = (x,acc)
   loop acc (Get k)   = loop acc (k i)
   loop acc (Put o k) = loop (acc `mappend` o) (k ())
```

There are other ways of interpreting `IT i o a` requests, for example, keeping the last told value, or writing the told value to `stderr`. Yet another interpreter, of `IT s s` computation takes the told value as the one to give when next asked, thus treating `IT s s` as a `State` computation.

The `IT i o` computation is an extension of `It i`. Alas, data types are not extensible. Therefore, we had to change the data type name and hence modify (the signatures of) `addGet` and `addN`, even if their code does not care about the new writer effect and remains essentially the same.

3.3.3 *Free Monad*

A data type describing an effectful computation such as `It i a` and `IT i o a` follows a common pattern: It is a recursive data type, with the `Pure` variant for the absence of any requests, and the variants for requests, usually containing the continuation that receives the reply (except for exceptions that do not expect any reply). The recursive occurrences of the data type are always as the return type of the continuations, that is, in covariant positions. This pattern, of pure and effectful parts and covariant recursive occurrences can be captured as

```
data Free f a = Pure a
              | Impure (f (Free f a))
```

where `f` is a (categorical) functor, that is, in `f a`, the type `a` occurs covariantly. The latter phrase means that if we can convert a value of type `a` into some other value of type `b`, we can also turn `f a` into `f b`. The `Functor` type class captures that meaning literally:

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

The concrete instantiations of `f` define the types of requests and replies, that is, the effect signature of a particular effectful computation. This splitting of a recursive data type such as `It i` into a non-recursive "structure component" and the recursive tying the knot `Free f a` was pioneered in [65] (who also used extensible effectful interpreters as one of the examples).

3 In Haskell, this () acknowledgment is not needed, but it fits our story better.

The monad instances for `It i` and `IT i o` also look very much alike. It is a shame to keep writing such instances for each new effect and each combination of effects. The `Free f` data type lets us capture the common pattern:

```
instance Functor f ⇒ Monad (Free f) where
 return = Pure
 Pure a   ≫= k = k a
 Impure f ≫= k = Impure (fmap (≫= k) f)
```

Thus `Free f` for a functor `f` is a monad – the free monad. New effects will have new effect signatures `f`, but the single instance of `Monad (Free f)` will work for all of them, with no further re-writing.

As an example, the earlier `IT i o` computation may now be specified as

```
data ReaderWriter i o x = Get (i → x) | Put o (() → x)
instance Functor (ReaderWriter i o) where …
type IT i o a = Free (ReaderWriter i o) a
```

The word "free" in free monad refers to the category theory's construction of the left adjoint of a forgetful operation [2]. In English, if we take a monad, say, `State s` with its `return`, `bind`, `fmap`, `put` and `get` operations and forget the first two, we can recover the monad as `Free (State s)`, with prosthetic `return` and `bind`. In short, we get the Monad instance for free.

In general monads do not compose: if `M1 a` and `M2 a` are monads, `M1 (M2 a)` is generally not. Free monads however are a particular form of monads, defined via a functor. Functors do compose. We will exploit that fact after one more generalisation.

3.3.4 *Free-er Monads*

Let us look more carefully at the Monad instance for `Free f`. The purpose of `fmap` there is to extend the continuation, embedded somewhere within `(f (Free f a))`, by `(<<>)`-composing it with the new `k`. The operation `fmap` lets us generically modify the embedded continuation, for any request signature.

Since the continuation argument is being handled so uniformly, it makes sense to take it out of the request signature and place it right into the fixed request data structure, as the second argument of `Impure`:

```
data FFree f a where
  Pure   :: a → FFree f a
  Impure :: f x → (x → FFree f a) → FFree f a
```

The remaining part of the request signature `f x` tells the type `x` of the reply, to be fed into the continuation. Different requests have their own reply types, hence `x` is existentially quantified. Our Reader-Writer effect gets then the following signature:

```
data FReaderWriter i o x where
 Get :: FReaderWriter i o i
 Put :: o → FReaderWriter i o ()
```

It is a GADT: the type variable `x` in `FReaderWriter i o x` is instantiated depending on the type of the request. For `Get`, the reply type is `i`, and for `Put`, it is unit. The `IT i o a` is now

```
type IT i o a = FFree (FReaderWriter i o) a
```

The monad instance for `FFree f` no longer needs the `Functor` or any other constraint on `f`:

```
instance Monad (FFree f) where …
 Impure fx k' ≫= k = Impure fx (k' ≪> k)
```

`FFree f` is more satisfying since it abstracts more of the common pattern of accumulating continuation, compared to `Free`. It is more general, not imposing any constraints on `f` – it is "freer". Continuing our example of `State s` from the end of §3.3.3, we can now forget not only `return` and `bind` but also the `fmap` operation, and still recover the state monad through `FFree (State s)` construction. We no longer have to bother defining the basic monad and functor operations in the first place: We now get not only the `Monad` instance but also the `Functor` and `Applicative` instances for free.

Freer monad is also more economical in terms of memory (and running time) because the continuation can now be accessed directly rather than via `fmap`, which has to rebuild the mapped data structure. The explicit continuation of `FFree` also makes it easier to change its representation, which we will do in §3.4.

Marcelo Fiore has suggested in private communication that the above FFree construction is the left Kan extension.

To highlight this point we show another derivation of `FFree`. Recall, if `f :: × → ×` is a functor, we can convert `f x` to `f a` whenever we can map `x` values to `a` values. If `g :: × → ×` is not a functor, such a conversion is not possible. We can "cheat" however: although we cannot truly fmap `h :: x → a` over `g x`, we can keep its two operands as a pair, and assume the mapping as if it were performed:

```
data Lan (g :: × → ×) a where
  FMap :: (x → a) → g x → Lan g a
```

Any further mapping over `Lan g a` updates the original mapping, leaving `g x` intact. That is, `Lan g` is now a "formal" functor:

```
instance Functor (Lan g) where
  fmap h (FMap h' gx) = FMap (h ∘ h') gx
```

This `Lan` construction is the Left Kan extension. One may think of it as a free Haskell Functor – Functor by construction – just as a list is a free Monoid.

Let us see what `Free (Lan g)` is: substituting `f` in the type of `(f (Free f a)) → Free f a` of `Free.Impure` with `Lan g` gives us

```
exists x. (x → (Free (Lan g) a)) → g x → Free (Lan g) a
```

which is the type of `FFree.Impure`. Hence

```
type FFRee g  = Free (Lan g)
```

Incidentally, the type-aligned sequences, which we will use in §3.4, are essentially Free-er Applicative.

By analogy with the "free functor" `Lan g` we may also define a "free bifunctor"

```
data BiFree p a b where
  Bimap :: (a → b) → (c → d) → p a c → BiFree p b d
```

which is a generalisation of the bifunctor used in [36, §6.3].

One last generalisation step remains, to deliver the promised extensibility.

3.3.5 *From Free(er) Monads to Extensible Effects*

We have hinted in §3.3.3 that the form of free monads, built from functors, lends itself to composability since functors compose. This section demonstrates this composability on freer monads, built around left Kan extensions, which are functors by construction. There are two sides to composability: extensible monad type and modular interpreters. The latter part has been receiving less attention: for example, Data types à la carte [67] provides the former but not the latter.

A monad type is extensible if we can add a new effect without having to touch or even recompile the old code. The `Free f` or `FFree f` lets us do that: the monad type is indexed by the request signature `f`. Specifying this signature as an ordinary data type, such as `ReaderWriter` in §3.3.3 or GADT `FReaderWriter` in §3.3.4 is not extensible: an ordinary variant data type is a closed union, with the fixed number of variants. Open unions are relatively easy to construct, essentially by nesting the simplest union, the Either data type. The monad transformer paper [49] already showed such an implementation; Swierstra [67] used essentially the same.

We will use the open union that improves the previous implementations, including the one in [42]. It provides the (abstract) type `Union (r :: [* → *]) x` where the first argument `r` is a type-level list of effect labels, to be described shortly. The second argument is the response type, which depends on a particular request. The argument `r` lists all effects that are possible in a computation; a concrete `Union r x` value contains one request out of those listed in `r`.

It is crucial for extensibility to be able to talk about one effect without needing to list all others. For the sake of this effect polymorphism, our implementation provides a type class

```
class Member t r where
  inj :: t v → Union r v
  prj :: Union r v → Maybe (t v)
```

that asserts that a label `t` occurs in the list `r`. If an effect is part of the union, its request can be injected and projected. We also offer another function, not present in [49, 67], to "orthogonally project" from the union,

```
decomp :: Union (t ': r) v → Either (Union r v) (t v)
```

obtaining either a request labeled `t` or a smaller union, without `t`. This function is needed for effect encapsulation. The earlier extensible effects library [42] provided a similar open union, implemented using overlapping instances and Typeable. The latter in particular attracted a large number of complaints. Deriving Typeable is indeed an extra step for a library aiming to encourage using many custom effects. For applications like monadic regions, Typeable was quite an obstacle, as we discuss in §3.8. The current implementation uses neither overlapping, nor Typeable. It also does not provide the no longer needed Functor instance.

The extensible freer monad, the monad of extensible effects, is hence `FFree` with the open union:

```
data FEFree r a where
  Pure   :: a → FEFree r a
  Impure :: Union r x → (x → FEFree r a) → FEFree r a
```

A request label defines a particular effect and its requests. For example, the Reader and Writer effects have the following labels:

```
data Reader i x where
  Get :: Reader i i
data Writer o x where
  Put :: o → Writer o ()
```

Informally, we split the monolithic FReaderWriter request signature into its components (to be combined in the open union). The simplest Reader computation, ask of §3.3.1, can now be written as

```
ask :: Member (Reader i) r ⇒ Eff r i
ask = Impure (inj Get) return
```

The signature tells that ask is an Eff r i computation which includes the Reader i effect, without telling what other effects may be present. Unlike the old ask of §3.3.1, the new one can be used, *as it is*, without any adjustments to code or the signature, in programs with other effects. The new ask is thus extensible.

Making interpreters such as runRdWriter of §3.3.2 modular is just as important, and not always achieved in the past. We describe them §3.4.

3.3.6 *Performance Problem of Free(er) Monads*

Free (and freer) monads are certainly elegant and insightful, but poorly performing. Let us look again at the FFree f monad instance

```
instance Monad (FFree f) where …
 Impure fx k' ≫= k = Impure fx (k' ≪> k)
```

The bind operation traverses its left argument but merely passes around the right argument. Therefore, the performance of left-associated binds, like the performance of left-associated list appends, will be poor – algorithmically poor. For example, the running time of addN n, implemented either as the It i monad or the FEFree [Reader i] monad, is quadratic in n. This is because addN happens to associate addGets on the left. For example, addN 3 evaluates to

```
(((return ≪> addGet) ≪> addGet) ≪> addGet) 0
```

which takes 3 evaluation steps to

```
((Impure (inj Get) return ∘ (+0)) ≫= addGet) ≫= addGet
```

The two evaluations of bind then produce the final request

```
Impure (inj Get) ((return ∘ (+0) ≪> addGet x) ≪> addGet)
```

The continuation, the second argument to Impure, is the addGet chain we started with, only one link shorter. Processing the reply from the context will again take time linear in the size of the chain. Overall, processing n requests takes $O(n^2)$ time. We refer the reader to [61] for more illustration and discussion of this performance problem, and for the general solution: representing the continuation as an efficient data structure, a type-aligned sequence.

3.4 FINAL RESULT: FREER AND BETTER EXTENSIBLE EFF MONAD

This section describes our current, improved and efficient library of extensible effects. Thanks to the Freer monad and the new open union it became easier, compared to the version presented two years ago [42], to define a new effect and to write a handler for it. There is no longer any need for Functor and Typeable instances. The performance has also improved, algorithmically; see §3.4.3. Before showing off the library in §3.4.2 we describe the last key improvement, representing the continuation as an efficient data structure.

3.4.1 *Composed Continuation as a Data Structure*

The new library is based on the `FEFree` monad derived in §3.3.5 (repeated here for reference):

```
data FEFree r a where
  Pure   :: a → FEFree r a
  Impure :: Union r x → (x → FEFree r a) → FEFree r a
```

differing in one final respect: Now that the request continuation `x → FEFree r a` is exposed, it can be represented in other ways than just a function. The motivation for a new representation comes from looking at the monad instance for `FEFree f`

```
instance Monad (FFree f) where …
 Impure fx k' ≫= k = Impure fx (k' ≪> k)
```

which extends the request continuation `k'` with the new segment `k`. The lesson of [61] is to represent this *conceptual* sequence of extending the continuation with more and more segments as a concrete sequence. It would contain all the segments that should be functionally composed – without actually composing them! We shall see soon that the composing is not really needed: it was just a way of accumulating continuation segments, and not an efficient way at that. (Another motivation to look for a new representation of continuations is the performance problem of free(er) monads, described in §3.3.6).

We call the improved `FEFree r` monad `Eff r`, where `r`, as in §3.3.5, is the list of effect labels. The request continuation – which receives the reply `x` and works towards the final answer `a` – then has the type `x → Eff r a`. We define the convenient type abbreviation for such effectful functions, that is, functions mapping `a` to `b` that also do effects denoted by `r`.

```
type Arr r a b = a → Eff r b
```

The job of the monad bind is to accumulate the request continuation, by `(≪>)`-composing it with further and further `Arr r a b` segments. Rather than really doing the composition, we assume it as performed, and merely accumulate the pieces being composed in a data structure. The left Kan extension used the same ‘pretend the operation performed’ trick. The data structure has to be heterogeneous, actually, type-aligned [61]: the `Arr r a b` being composed have different `a` and `b` types, and the result type of one function must match the argument type of the next. The type-aligned sequences enforce this invariant by construction. We chose the sequence `FTCQueue` of the following interface

```
type FTCQueue (m :: ∗ → ∗) a b
tsingleton :: (a → m b) → FTCQueue m a b
(|>) :: FTCQueue m a x → (x → m b) → FTCQueue m a b
(><) :: FTCQueue m a x → FTCQueue m x b → FTCQueue m a b
data ViewL m a b where
  TOne  :: (a → m b) → ViewL m a b
  (:|)  :: (a → m x) → (FTCQueue m x b) → ViewL m a b
tviewl :: FTCQueue m a b → ViewL m a b
```

`FTCQueue m a b` represents the composition of one or more functions of the general shape `a → m b`. The operation `tsingleton` constructs a one-element sequence, `(▷)` adds a new element at the right edge and `(><)` concatenates two sequences; `tviewl` removes the element from the left edge. All operations have constant or average constant running time. Our `FTCQueue` may be regarded as the minimalistic version of a more general fast type-aligned queue `FastTCQueue`: see [61] and type-aligned on Hackage. Thus the composition of functions (continuation segments) `a → Eff r t1`, `t1 → Eff r t2`, …, `tn → Eff r b` is represented as

```
type Arrs r a b = FTCQueue (Eff r) a b
```

and the `Eff r` monad has the following form

```
data Eff r a where
  Pure   :: a → Eff r a
  Impure :: Union r x → Arrs r x a → Eff r a
```

A composition of functions is a function itself; likewise `Arrs r a b` is isomorphic to the single `Arr r a b` (or `a → Eff r b`). In one direction,

```
singleK :: Arr r a b → Arrs r a b
singleK = tsingleton
```

the conversion builds the sequence with one element. In the other direction,

```
qApp :: Arrs r b w → b → Eff r w
qApp q x = case tviewl q of
   TOne k   → k x
   k :| t → bind' (k x) t
 where bind' :: Eff r a → Arrs r a b → Eff r b
       bind' (Pure y) k     = qApp k y
       bind' (Impure u q) k = Impure u (q >< k)
```

The `qApp` operation applies the argument `x` to a composition of functions denoted by the sequence `Arrs r a b`. To be precise, it applies `x` to the head of the sequence `k` and 'tacks in' the tail `t` (if any) as it was. That is the performance advantage of the new representation for continuation. The `bind'` operation is like monad bind (`≫=`) but with the continuation represented as the sequence `Arrs r a b` rather than the `a → Eff r b` function. If the application `k x` runs in constant time, the whole `qApp q x` takes on average constant time.

Finally, in the monad instance of `Eff r`

```
instance Monad (Eff r) where
  return = Pure
  Pure x      ≫= k = k x
  Impure u q ≫= k = Impure u (q |> k)
```

the bind operation grows the sequence `Arrs r x a` of continuations by appending another segment, `k`, which takes constant time.

3.4.2 *Library Showcase: Defining and Interpreting Effects*

We now demonstrate the extensible effects library: writing and composing effectful computations with the `Eff` monad. We re-do the reader and writer example §3.3.1, §3.3.2 to show that now adding the writer does not have to change the earlier code.

An effect is defined first by listing its requests and the corresponding reply types. For the `Reader i` effect, the request merely asks for a reply of the type `i`.

```
data Reader i x where
  Get :: Reader i i
```

The simplest client that returns the received reply is hence

```
ask :: Member (Reader i) r ⇒ Eff r i
ask = Impure (inj Get) (tsingleton Pure)
```

Recall, `tsingleton` creates the singleton sequence. The following library function makes the sending of requests even easier:

```
send :: Member t r ⇒ t v → Eff r v
send t = Impure (inj t) (tsingleton Pure)
```

The other `Reader` computations `addGet` and `addN` of §3.3.1 are expressed in terms of `ask` and monad operations; their code is hence unchanged. Here they are, for the ease of reference:

```
addGet :: Member (Reader Int) r ⇒ Int → Eff r Int
addGet x = ask ≫= λi → return (i+x)

addN :: Member (Reader Int) r ⇒ Int → Eff r Int
addN n = foldl (≪>) return (replicate n addGet) 0
```

Their types however become more general: `addN n` has the `Reader` effect *and* can be used in computations that do other effects.

Interpreters of `Reader` requests now have to keep in mind there may be other request types, for other interpreters to deal with. Here is the new version of `runReader` from §3.3.1:

```
runReader :: i → Eff (Reader i ': r) a → Eff r a
runReader i m = loop m where
 loop (Pure x)     = return x
 loop (Impure u q) = case decomp u of
        Right Get → loop $ qApp q i
        Left  u   → Impure u (tsingleton (qComp q loop))
```

The type signature says that `runReader i` receives the `Eff` computation with the `Reader i` effect, and returns the `Eff` computation without. The `Reader i` effect is thus handled, or encapsulated. The code indeed replies to the `Get` request – leaving other requests for other interpreters, see the *`Left`* `u` case. After that other interpreter replies, the program resumes and may make further `Get` requests. That is why we append the reader interpreter `loop` to the reply continuation `q`, using the function `qComp`:

```
qComp :: Arrs r a b → (Eff r b → Eff r' c) → Arr r' a c
qComp g h = h ∘ qApp g
```

The result continuation has the different list of effect labels `r'` since some of the effects will be handled by the interpreter `h`.

The common request handling code is factored out in the following function provided by the library:

```
handle_relay :: (a → Eff r w) →
                (∀ v. t v → Arr r v w → Eff r w) →
                Eff (t ': r) a → Eff r w
handle_relay ret _ (Pure x)     = ret x
handle_relay ret h (Impure u q) = case decomp u of
  Right x → h x k
  Left  u → Impure u (tsingleton k)
 where k = qComp q (handle_relay ret h)
```

The first two arguments of `handle_relay` are like `return` and `bind`. The reader interpreter can be thus written simply as

```
runReader i = handle_relay return (λGet k → k i)
```

The last part of handle_relay's signature, `Eff (t ': r) a → Eff r w`, shows that the label `t` of the handled effect must be at the top of the list of effect labels `r`. Whereas effectful functions like `addN` above or `rdwr` below regard `r` truly as a set of effect labels, with no particular order, handlers impose the order. This fact is noticeable already in the interface of `Union` in §3.3.5: in the signatures of `inj` and `prj`, effects are represented by the type variable `r`, with a `Member` constraint. On the other hand, `decomp` takes the collection of effects to be specifically a list, with the projected effect `t` at its head. In our experience so far, this imposition of order by the handlers has not been a problem. It is theoretically unsatisfying. Although we could avoid it by playing with Constraint types, the required type annotations made the result impractical. Unfortunately, there does not seem to be any convenient way in Haskell to discharge one type class constraint by submitting the corresponding dictionary. (Implicit parameters do come very close.)

To run the `Eff` computation after all effects have been handled by the corresponding interpreters, the library provides

```
run :: Eff '[] a → a
run (Pure x) = x
```

The `Impure` case is unreachable since `Union '[] a` has no (non-bottom) values. Thus we run `addGet 1` as

```
run ∘ runReader 10 $ addGet 1
```

Let us add the writer effect, of telling the context the value of type `o`:

```
data Writer o x where
  Put :: o → Writer o ()

tell :: Member (Writer o) r ⇒ o → Eff r ()
tell o = send $ Put o
```

The type of `tell` lets it be combined in any effectful computation with the `Writer o` effect. Here is a sample combined reader-writer computation

```
-- rdwr :: (Member (Reader Int) r, Member (Writer String) r)
--    ⇒ Eff r Int
rdwr = do{ tell "begin"; r ← addN 10; tell "end"; return r }
```

whose inferred type is shown in the comments. Because the type of `addN` is polymorphic in `r`, we could use `addN` as it was in a computation with more effects (and similarly, for tell).

In §3.3.2, the interpreter for Reader-Writer computations was the monolithic `runRdWriter`, which handled both types of requests. Now we can interpret only the `Writer` requests

```
runWriter :: Eff (Writer o ': r) a → Eff r (a,[o])
runWriter =
  handle_relay (λx → return (x,[]))
     (λ(Put o) k → k () ≫= λ(x,l) → return (x,o:l))
```

and literally compose it with the previously written `runReader`. The sample reader-writer computation `rdwr` is thus run as

```
(run ∘ runReader 10 ∘ runWriter) rdwr
```

Since the reader and writer effects commute, the order of the interpreters can be switched without affecting the result.

One may write other reader and writer interpreters, for example, handling Reader and Writer requests together; the value last told becomes the value to give on the next Reader request. We thus implement `State`, by decomposing it into the reading and mutating parts. It becomes easier to tell, just from their inferred type, which parts of the computation mutate the state.

```
runStateR :: Eff (Writer s ': Reader s ': r) w → s → Eff r (w,s)
runStateR m s = loop s m where
  loop :: s → Eff (Writer s ': Reader s ': r) w → Eff r (w,s)
  loop s (Pure x) = return (x,s)
  loop s (Impure u q) = case decomp u of
    Right (Put o) → k o ()
    Left  u   → case decomp u of
      Right Get → k s s
      Left u → Impure u (tsingleton (k s))
   where k s = qComp q (loop s)
```

3.4.3 *Improved Performance*

This section re-analyses the performance of the freer monad after changing the representation of the request continuation, on the problematic example from §3.3.6. As before, `addN 3` evaluates to

```
(((return ≪> addGet) ≪> addGet) ≪> addGet) 0
```

and then to

```
((Impure (inj Get) [return ∘ (+0)]) >>= addGet) >>= addGet
```

The two evaluations of bind produce the request

```
Impure (inj Get) [return (+0), addGet, addGet]
```

(where we used the list notation for the type-aligned sequence for clarity). So far, the process and its result seem similar to that for the non-optimised monad in §3.3.5. The fact that the continuation of the `Get` request is now represented as an efficient sequence makes the difference. When a `runReader` interpreter replies, say, with the value `v1`, it does the following operations that eventually produce a new request. For emphasis we denote as `t` the tail of the request continuation (in our example, `t` is the singleton sequence `[addGet]`):

```
    qApp (return (+0) : addGet : t) v1
⟹  return v1 `bind'` (addGet : t)
⟹  addGet v1 `bind'` t
⟹  Impure (inj Get) (return (+v1) : t)
```

The above reduction sequence has dealt only with the two head elements of the entire continuation of the original request. The tail `t` was merely passed around and not even looked at. Furthermore, all `FTCQueue` operations involving `t` such as concatenation, etc., were constant-time. Therefore, the entire sequence of reductions above runs in time independent of the length of `t`. The run-time of the entire `addN n` computation is thus linear in `n`. Compared with the previous version §3.3.5, we obtain the algorithmic improvement in performance, from quadratic to linear. The key to the performance is the ability to look at and remove initial segments from the accumulated request continuation. If the continuation is represented as a composition of functions, we cannot ‘uncompose’ them – but we can deconstruct a data structure.

3.5 PERFORMANCE EVALUATION

This section reports on several micro-benchmarks used to evaluate the performance of extensible effects (EE) relative to monad transformer library MTL, Kammar's et al. "Handlers in action (HIA)" [38] and the old version of EE presented in [42].

The benchmark code was compiled with GHC 7.8.3 with the flag `-threaded -O -rtsopts`. We ran the benchmark on an Intel Core i7 (2.8GHz) laptop with 16GB of RAM. The Criterion framework was used to report the run-time.

3.5.1 *Deep-monad-stack Benchmarks*

First, we ran two benchmark computations with many effects (deep monad stacks). These benchmarks do a simple stateful computation with many `Reader` layers under or over the target `State` layer. The core `State` computation is as follows:

```
benchS ns = foldM f 1 ns where
 f acc x | x `mod` 5 = 0 = do
                            s ← get
                            put $! (s+1 :: ℤ)
                            return $! max acc x
 f acc x = return $! max acc x
```

Strictness annotations are to avoid space leaks.

Figure 3.1 shows the results. If we add the extra `Reader` layers *under* the `State` (the top of Fig.3.1), EE runs in constant time, while the MTL version takes linear time in the number of layers. Our EE is about 12% faster than HIA, and 40% faster than the old EE. If the `State` layer is at the bottom of the monad stack (the bottom of Fig.3.1) the run-time of HIA and EE versions is linear in the number of layers, whereas MTL and the old EE are quadratic. The results confirm the analyses of performance in §3.3.6 and §3.4.3: the EE library presented in this paper indeed algorithmically improves the performance over the old version – as well over MTL for deep monad stacks. The overhead of MTL can indeed be severe for deep stacks. We also see that EE is competitive with HIA.

Monad Stack Depth and Memory Consumption

We also evaluated the memory efficiency of the two deep-monad-stack benchmarks by taking the memory profile, using GHC with RTS options `-N2 -prof -p -N2 -p -hm`. Figure 3.2 shows the result.

Adding `Reader` layers under the `State` layer (the top of Fig.3.2) affects the memory consumption of effect libraries (EE and HIA) very little. The memory use does increase linearly with the number of layers, but by such a small amount that it is very difficult to see in the figure. In contrast, the amount of allocated memory for MTL is quadratic in the number of layers, and is quite large compared to the effect libraries. If we add `Reader` over the `State` layer (the bottom of Fig.3.2), the linear increase in allocated memory for effect libraries becomes quite more noticeable. The MTL memory use is again quadratic in the number of layers. The results confirm our expectation of the memory efficiency of the EE library presented in this paper.

3.5.2 *Single-effect Benchmark*

We have just seen that EE can overcome the overhead of handling very many effects. To see how EE and MTL compare for a single effect, we ran a simpler benchmark, with the single `State` or the single `Error` effect (table 3.1).

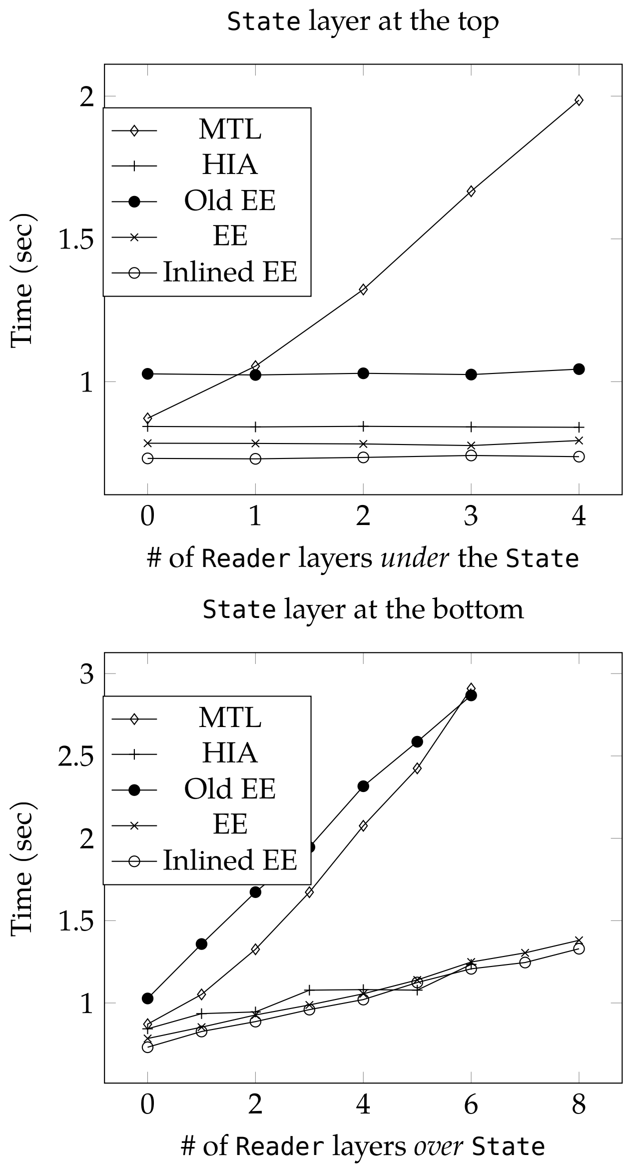

Figure 3.1: Runtime in seconds for MTL, HIA, Old EE, EE and Inlined EE. x-axis corresponds to the number of `Reader` layers *under* (top) or *over* (bottom) the target `State` layer.

The single `State` benchmark counts down from $10,000,000$ to 0, using the `State` monad. The EE version is much slower than the MTL and HIA, 30 and 60 times correspondingly. This is because the `State` monad enjoys the preferential treatment by GHC, with dedicated optimisation passes. Likewise, GHC is very good at optimising simple CPS code employed in simple instances of HIA. Thus for the single `State` effect, our EE approach is not so suitable. The new library is still noticeably faster than the original EE version two years ago.

In contrast, for the `Error` monad EE and MTL have almost the same performance and notably, three times, faster than HIA and the old approach. The `Error` benchmark takes the product of $10,000,000$ copies of 1 and 0, raising a exception when the zero factor is found.

Thus for the single or few-layered monadic computations, EE can compete with individual single specialised monads in general, but for some monads, like `State`, it runs much more slowly.

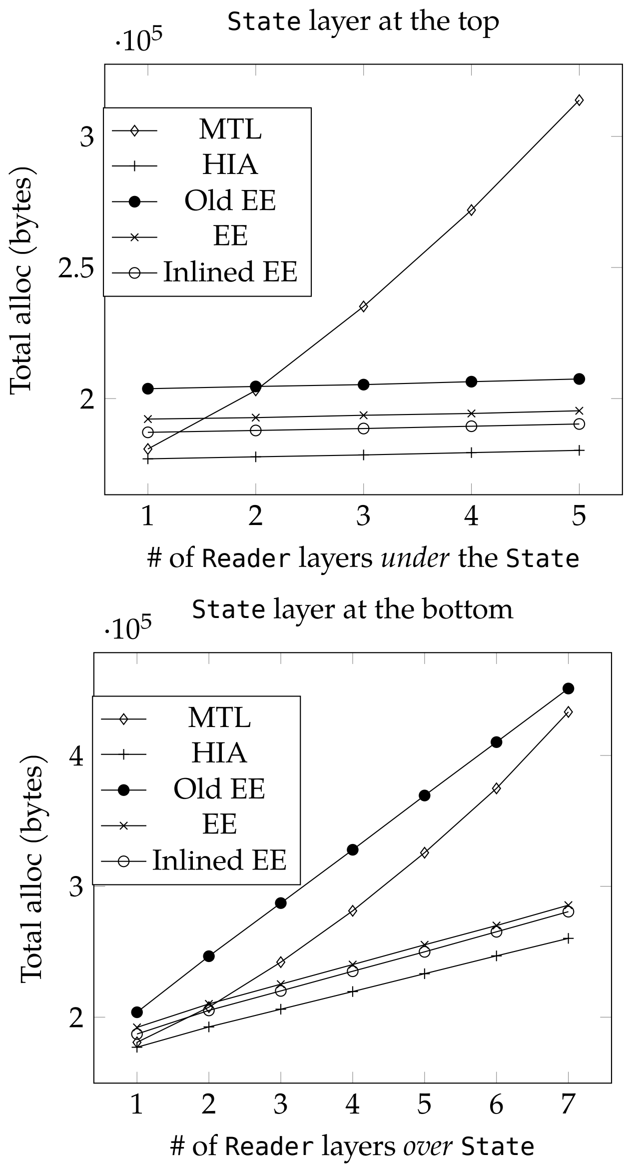

Figure 3.2: Total allocation in 10^5 bytes for MTL, HIA, Old EE, EE and Inlined EE. x-axis corresponds to the number of `Reader` layers *under* (top) or *over* (bottom) the target `State` layer.

3.5.3 *Non-determinism Benchmarks*

We have run another series of benchmarks, for the non-determinism effect, to be discussed in detail in §3.6.

The first benchmark (the top of Fig.3.3) searches for Pythagorean triples up to the given bound with non-deterministic brute-force:

```
iota k n = if k > n then mzero else return k `mplus` iota (k+1) n

pyth1 :: MonadPlus m ⇒ Int → m (Int, Int, Int)
pyth1 upbound = do
  x ← iota 1 upbound
  y ← iota 1 upbound
  z ← iota 1 upbound
  if x×x + y×y = z×z then return (x,y,z) else mzero
```

Table 3.1: A simple benchmark with a single layer (msec).

	pure	MTL	HIA	Old EE	EE	Inlined EE
`State`	-	15.2	7.16	840	579	488
`Error`	46.4	218	648	644	204	216

For the MTL version, we use the continuation monad transformer `ContT`. The result shows that our EE is much faster than old EE, but slightly slower than MTL and HIA. We should stress that our EE library, unlike HIA and MTL, implements the more general LogicT effect: non-determinism with committed choice.

The next benchmark adds to the previous one counting of the all attempted choices, using the `State` effect. The result at the bottom of Figure 3.3 shows that our EE approach is faster than the other alternatives.

The results confirm the good performance of EE, also for more complicated computations with many layers of effects.

3.5.4 *Comparison with "Fusion for Free"*

Very recently, Wu and Schrijvers [68] introduced the "Fusion for Free" approach for algebraic event handlers. Since their implementation is not yet published as a library, we will briefly compare performance in a qualitative manner. Specifically, we ran the benchmarks `count_1` and `count_2` from [68] for MTL, EE and Inlined EE. The result is shown in Table 3.2.

Table 3.2: Runtime in milliseconds for Counting benchmarks from Wu and Schrijvers [68]

		MTL	EE	Inlined
`count_1`	10^3	0.000694	0.0592	0.0488
	10^4	0.00692	0.593	0.489
	10^5	0.0689	5.87	4.81
`count_2`	10^3	0.202	0.306	0.287
(`Writer`	10^4	4.84	6.40	6.51
bottom)	10^5	54.4	84.7	80.8
`count_2`	10^3	0.0859	0.345	0.316
(`Writer`	10^4	2.85	6.57	6.67
top)	10^5	37.3	85.2	84.2

Here, `count_1` is the single `State`-effect benchmark, counting down in the `State` monad, similar to our benchmark in §3.5.2. This is the singular most unfavourable case for EE compared to MTL, since GHC has several optimisations that benefit the MTL State monad. The table shows that in all the cases, the run-time increases with the count not just linearly but proportionally. This qualitatively reproduces the behaviour reported by Wu and Schrijvers in [68].

The next `count_2` benchmark counts down using `State`, and also logs every intermediate value in the `Writer` monad. Wu and Schrijvers did not indicate which layer is on top, so we ran both cases, which proved to make little difference for EE (in contrast to MTL, however). The run-times again seem linear, but not proportional. In [68], the run-time of "Fusion" is proportional. Although

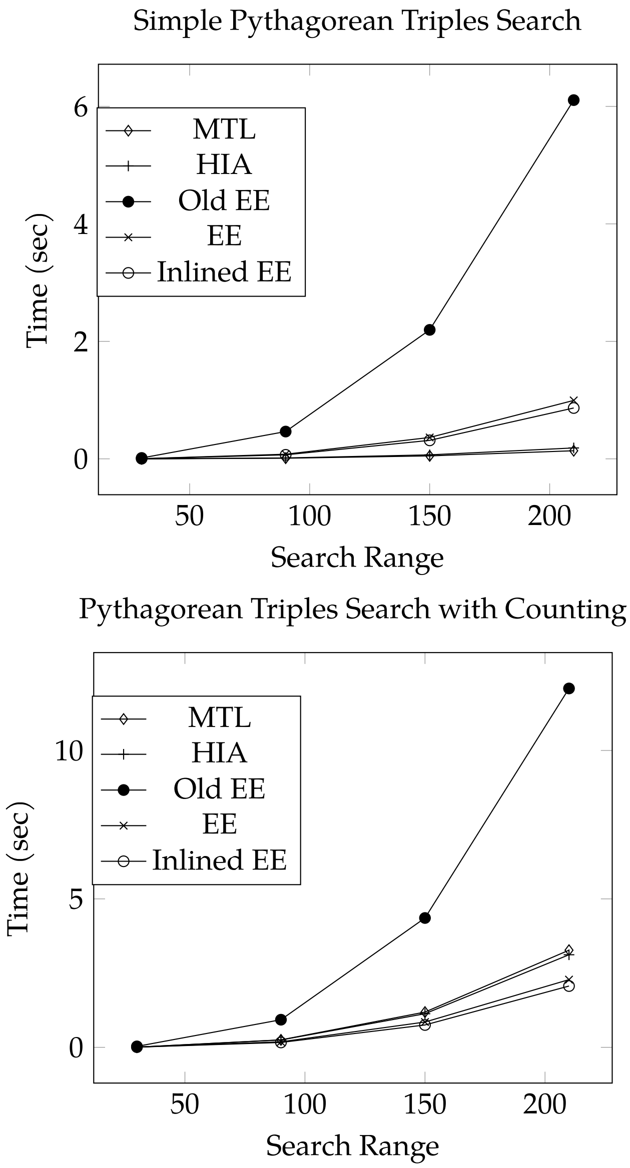

Figure 3.3: Runtime in seconds for MTL, HIA, Old EE, EE and Inlined EE. x-axis corresponds to the search range for Pythagorean triples.

the qualitative behaviour again seems similar, quantitative comparison is clearly needed. We defer it to future work, when the code for [68] becomes available.

3.5.5 *Inlining of Key Functions*

The key functions of the EE library such as `handle` described in §3.4, contain a recursive reference but not a recursive invocation. These functions are hence safe to inline. To see if it makes any difference we added the `INLINEABLE` pragma for these functions. The pragma had almost no effect. The performance has improved slightly only when we inlined `tviewl` into `qApp` by hand (these functions are defined in different modules).

3.6 NON-DETERMINISM WITH COMMITTED CHOICE

Non-determinism, with its inherent balancing of several continuations, may seem impossible to express as a freer monad, which explicitly deals with a single continuation. This section shows that not only the `Eff` monad can represent non-deterministic choice, but also that the representation preserves the sharing of continuations, lost in the standard free monad approach.

Free monad models non-determinism with the following request functor:

```
data Ndet x = MZero | MPlus x x
instance Functor Ndet where …
```

`MZero`, like an exception, requests abandoning the current line of computation as unsuccessful; `MPlus` asks the context to choose between the two `Ndet` computations. This request signature comes straight from the interface for non-deterministic computations in Haskell: `MonadPlus` or `Alternative`:

```
instance MonadPlus (Free Ndet) where
  mzero       = Impure MZero
  mplus m1 m2 = Impure $ MPlus m1 m2
```

The `MPlus` constructor has two continuation arguments. How are we going to separate them into the single continuation argument of `FFree`? Let us consider the non-deterministic choice in context:

```
   (mplus m1 m2 ≫= k1) ≫= k2
   {The bind of the Free monad}
⟹ Impure (fmap (≫= k1) (MPlus m1 m2) ≫= k2
   {The fmap from the derived Functor instance}
⟹ Impure (MPlus (m1 ≫= k1) (m2 ≫= k1)) ≫= k2
   {Repeating for k2}
⟹ Impure (MPlus ((m1 ≫= k1) ≫= k2) ((m2 ≫= k1) ≫= k2))
```

That is, the two continuations collected by `MPlus` in fact have the common `k1`, `k2` suffix. That suffix, albeit common, is not shared: although the two MPlus continuations share the common segments, they are independently composed. It is this common suffix that the freer monad will factor out and share.

After the common continuation suffix is separated out, what remains of `MPlus` is the request to the context to pick and return one of the two choices. There is no need to include the choices themselves in the request then. Hence in the `Eff` framework, the non-determinism effect has the following signature:

```
data NdetEff a where
  MZero :: NdetEff a
  MPlus :: NdetEff Bool

instance Member NdetEff r ⇒ MonadPlus (Eff r) where
  mzero       = send MZero
  mplus m1 m2 = send MPlus ≫= λx → if x then m1 else m2
```

To complete the implementation, we add an interpreter, such as the following, mapping the `NdetEff` -effect non-determinism to Alternative:

```
makeChoiceA :: Alternative f ⇒
               Eff (NdetEff ': r) a → Eff r (f a)
makeChoiceA = handle_relay (return ∘ pure) $ λm k → case m of
    MZero → return empty
    MPlus → liftM2 (<|>) (k True) (k False)
```

One may recognise in this code the "flip oracle" of [15, §3], which non-deterministically returns a boolean value. Just as Danvy and Filinski's code, we are capturing the context of `mplus`, represented as `k` above, and plugging first *`True`* and then *`False`* into the very same context.

The advantage of `NdetEff` over Alternative is not only the ability to mix `NdetEff` with other (non-applicative) effects, for example, State. It also supports the so-called "committed choice" [59], such as logical "if-then-else" (called "soft-cut" in Prolog):

```
ifte :: Member NdetEff r ⇒
        Eff r a → (a → Eff r b) → Eff r b → Eff r b
```

Declaratively, `ifte t th el` is equivalent to `t ≫= th` if the non-deterministic computation `t` succeeds at least once. Otherwise, `ifte t th el` is equivalent to `el`. The difference between `ifte t th el` and the seemingly equivalent ``(t ≫= th) `mplus` el`` is that in the latter `el` is a valid choice even if `t` succeeds. In the former, `el` is chosen if and only if `t` is the total failure. One of the examples of `ifte` is the `many` parser combinator with 'maximal munch': `many p` should keep applying the argument parser `p` for as long as it succeeds. The following is another, easier to explain albeit more contrived, example: computing primes

```
test_ifte = do
  n ← gen
  ifte (do d ← gen
           guard $ d < n && n `mod` d = 0)
    (λ_ → mzero)
    (return n)
 where gen = msum ∘ fmap return $ [2..30]
msum :: MonadPlus m ⇒ [m a] → m a –– choose one from a list
```

Here `gen` non-deterministically produces a candidate prime and a candidate divisor. The prime candidate is accepted if all attempts to divide it fail. For example,

```
test_ifte_run :: [Int]
test_ifte_run = run ∘ makeChoiceA $ test_ifte
–– [2,3,5,7,11,13,17,19,23,29]
```

gives the result shown in the comment.

We actually implement not just `ifte` but the general

```
msplit :: Member NdetEff r ⇒
          Eff r a → Eff r (Maybe (a, Eff r a))
```

which expresses all other committed choice operations [45]. One may think of `msplit` as "inspecting" the argument computation, to see if it can succeed. If a computation gives an answer, it is returned along with the computation that may produce further answers. The implementation is so straightforward and small that it can be listed in its entirety:

```
msplit = loop [] where
 loop jq (Pure x)      = return (Just (x, msum jq))
 loop jq (Impure u q)  = case prj u of
  –– The current choice fails (requested abort)
  Just MZero → case jq of
                  –– check if there are other choices
                  []     → return Nothing
                  (j:jq) → loop jq j
  Just MPlus → loop ((qApp q False):jq) (qApp q True)
  _  → Impure u (tsingleton k) where k = qComp q (loop jq)
```

In words, `msplit t` intercepts the `NdetEff` requests of `t`. If `t` asks to choose, `MPlus`, one choice is pursued immediately and the other is saved in the work list `jq` of possible choices. The function finishes when the watched computation succeeds (the worklist is the collection of the remaining choices then) or when all possible choices failed.

We have demonstrated the most straightforward `Eff` implementation of not just non-determinism but non-determinism with committed choice (or, LogicT) [45].

3.7 CATCHING IO EXCEPTIONS

Handling IO errors in the presence of other effects abounds in subtleties. It was also thought to be a challenge for the `Eff` library. Not only has `Eff` met the challenge, it improves on MTL. With extensible effects, the state of the computation at the point of an exception is available to the handler. In MTL, an exception handler only has access to the state that existed at the point where it was installed (that is, `catch` was entered). Any further changes, up to the point of the exception, are lost.

Capturing IO errors in general `MonadIO` computations (not just the bare `IO` monad) has been a fairly frequently requested feature, going back to 2003[4]. An early approach[5] has been improved and polished through many packages (such as MonadCatchIO) and eventually de facto standardised in `exceptions` . The solution, although very useful in many circumstances is not without problems. For example, consider the following computation with the Writer and IO effects

```
do tell "begin"; r ←faultyFn; tell "end"; return r
`catch` (λe → return ∘ show $ (e::SomeException))
```

where `faultyFn` throws an IO or a user-defined dynamic exception. With MTL, any Writer updates that happened after `catch` up to the point of the exception are lost. That is, after the above code finishes the accumulated trace has neither “end” nor “begin”. Such a transactional semantics is useful – but not when the Writer is meant to accumulate the debug trace. Alas, MTL does not give us the easy choice.

To understand the MTL behaviour, recall that its `WriterT String IO a` monad is `IO (a,String)`: it is the computation that produces the value `a` along with the contribution to the writer string. The catch is implemented as (see `liftCatch` in `mtl`).

```
 catch h m = m `IO.catch` λe → h e
```

When an IO exception is raised, the value produced by `m`, including its Writer contribution, is lost. MTL’s `liftCatch` for the State monad has the similar behaviour of discarding the state accumulated since the catch is entered. In general, effect interaction in MTL depends on the order of the transformer layers; the IO monad is not a transformer however and must always be at the bottom of the stack.

If we execute the same code with the extensible-effect IO error handling [6] the trace accumulated by the writer of course has no `"end"` but it does have `"begin"`. Here is the whole code for catching IO exceptions

```
catchDynE :: ∀ e a r.
             (MemberU2 Lift (Lift IO) r, Exc.Exception e) ⇒
             Eff r a → (e → Eff r a) → Eff r a
catchDynE m eh = interpose return h m
```

4 http://www.haskell.org/pipermail/glasgow-haskell-users/2003-September/005660.html http://haskell.org/pipermail/libraries/2003-February/000774.html
5 http://okmij.org/ftp/Haskell/misc.html#catch-MonadIO
6 http://okmij.org/ftp/Haskell/extensible/EffDynCatch.hs

```
 where
  h :: Lift IO v → Arr r v a → Eff r a
  h (Lift em) k = lift (Exc.try em) ≫= λx → case x of
        Right x → k x
        Left  e → eh e
```

In the extensible effects library, IO computations are requested with the `Lift IO` effect

```
newtype Lift m a = Lift (m a)
```

whose interpreter

```
runLift :: Monad m ⇒ Eff '[Lift m] w → m w
```

is necessarily the last one, which is signified by the special `MemberU2 Lift (Lift IO) r` constraint. The library function `interpose` is a version of `handle_relay` that does not consider an effect handled although it does reply to its requests: `interpose` may also 're-throw' effect's request. The function `catchDynE` intercepts IO requests to wrap them into the `Exception.try` to reify possible exceptions. Therefore, IO errors are instantly caught and do not immediately discard their continuation. The effect handlers in scope and their state are thus preserved.

We can also easily implement transactional behaviour: an exception rolling-back the state to what it was when the exception handler was installed; see the source code for details.

3.8 REGIONS

Monadic Regions were introduced by Fluet and Morrisett [19] as a surprisingly simple version of the type-safe region memory management system. It may be thought of as a nested `ST` monad while also allowing reference cells allocated in a parent region to be used, relatively hassle-free, in any child region. Lightweight monadic regions [44] is the Haskell implementation of the extended version of Fluet and Morrisett's system, which was applied to IO resources such as file handles rather than memory cells, and is simpler to use. Lightweight regions statically ensure that every accessible file handle is open, while providing timely closing. The original Monadic Regions used an atomic monad, indexed by a unique region name; the lightweight version was built by iterating an `ST`-like monad transformer. Extensible effects, with its atomic `Eff` monad indexed by effects tempted one to re-implement lightweight regions closer to Fluet and Morrisett's original style while still avoiding the inconvenience of passing around parent-child–relationship witnesses. This challenge was set as future work in [42].

Implementing monadic regions with extensible effects was certainly a challenge. To ensure that an allocated resource such as a memory cell or a file handle do not escape from their region, Monadic Regions – like the `ST s` monad – mark the types of the computation and its resources with a quantified (or rigid, in GHC parlance) type variable. Defining `Typeable` instances for such types was the first challenge. More worrisome, any type-level programming with types that include rigid variables never meant to be instantiated is fragile. Sometimes, incoherent instances [5] are needed, which is a rather worrisome extension that we are keen to avoid. Finally, lightweight monadic regions, although based on monad transformers, intentionally prohibited any lifting and hence the addition of other effects. Exceptions and non-determinism are clearly incompatible with the region discipline. On the other hand, State and Reader are benign and should be allowed.

All these challenges have been met[7]. Below we describe the salient points of the implementation.

Since the new version of extensible effects no longer uses `Typeable`, the first challenge disappears. The second one was difficult indeed. The most straightforward realisation of Fluet and Morrisett's

7 http://okmij.org/ftp/Haskell/extensible/EffRegion.hs http://okmij.org/ftp/Haskell/extensible/EffRegionTest.hs

idea is to provide a `RegionEff s` effect indexed by the rigid type variable `s` taken to be the name of the region. File handles allocated within the region will be marked by that region's name:

```
newtype SHandle s = SHandle Handle
data RegionEff s a where
  RENew :: FilePath → IOMode → RegionEff s (SHandle s)
```

The data constructors are private and not exported. (The actual implementation is a bit more complex because it supports bequeathing of file handles to an ancestor region, see [44] for more discussion.)

The operation to allocate the new file handle will send a `RENew` request and obtain the handle marked with the region's name.

```
newSHandle :: Member (RegionEff s) r ⇒ -- simplified
              FilePath → IOMode → Eff r (SHandle s)
newSHandle fname fmode = send (RENew fname fmode)
```

The list of constraints is a bit simplified, omitting the type-level computation that scans the list of effect labels `r` and finds the name of the closest, that is, innermost, region. The interpreter of the requests

```
newRgn :: (∀ s. Eff (RegionEff s ': r) a) → Eff r a
```

like `runST`, has higher-rank type: informally, it allocates a fresh rigid type variable `s`, the fresh name for the region. The interpreter keeps the list of handles it was asked to allocate, closing all of them upon normal or exceptional exit. An operation using the handle has the type

```
shGetLine :: Member (RegionEff s) r ⇒
      SHandle s → Eff r String
```

that enforces that the region named `s` owning the handle is active: its name is among the current effect labels `r`. Incidentally, the signature *automatically* allows the handle allocated in any ancestor region to be used in a child region.

The outlined implementation indeed works, save for two subtleties. It is indeed tempting to think of the rigid type variable `s` as the name for the region `RegionEff s`. Alas, the ever-present `Member (RegionEff s) r` constraint, checking that the `RegionEff s` effect is part of the current effect list `r`, cannot distinguish two types `RegionEff s1` and `RegionEff s2` that differ only in the rigid type variable. Although these variables will never be instantiated and hence never can be the same, the constraint-solving part of GHC does not know or understand this fact. Therefore, we have to give regions another name, a type-level numeral, which the constraint-solver can distinguish. Therefore, the signatures of `newSHandle` and `newRgn` (but not `shGetline`, etc) are slightly more complex than shown.

The second subtlety is allowing other effects besides `RegionEff`. Since all possible effects of a `Eff r` computation are listed in `r`, we merely need to look through the list to check if the effect is known to be benign. The implementation provides such `SafeForRegion` constraint, treating `Reader` and `State` as safe[8]. `Exc SomeException` is also allowed since `newRgn` specifically listens for this request.

The rest of the implementation is straightforward. It passes the old Lightweight Regions regression tests with minimal modifications.

8 Since the user may write their own interpreter, they may well treat Reader as an exception, which is not safe. We may prevent such a behaviour by not exporting the data constructor for the Reader request.

3.9 RELATED WORK

The library of extensible effects reported in this paper is the simplification and improvement of the library presented two years ago [42]. `Eff` was a co-density-transformed free monad – which was not made clear in that paper. The co-density transform is regarded as an optimisation – alas, it does not work for modular interpreters, which have to reflect the continuation when relaying a request to another handler. The incompatibility of reflection with the co-density optimisation was described in detail in [61]. We now use the simpler and quite better performing Freer monad with type-aligned sequences. The new `Eff` also uses the new implementation of open unions without the objectionable features: `Typeable` and overlapping instances. The applications described in §§3.6,3.7,3.8 are also new, for extensible effects.

One of the most common questions about Extensible Effects is their relation to Swierstra's well-known "Data types à la carte" [67]. Similarities are indeed striking: free monads, open unions, 'modular' monads leading to a type-and-effect system. Although the à la carte approach provides extensible monad type, it does not provide modular interpreters with modular effects and hence effect encapsulation. Related to the lack of composability are problems with type inference, requiring cumbersome and what should be unnecessary annotations. (The ambiguity in the definition of the subsumption relation on collection of types, which caused the inference problems, has been rooted out in the novel approach by [3].) See http://okmij.org/ftp/Haskell/extensible/extensible-a-la-carte.html for a detailed comparison with the old `Eff` library. The present paper moves past the free monad to freer monad.

In comparison with monad transformers, the interaction of effects in `Eff` depends not on the statically fixed order of transformers but on the order of effect interpreters and can even be adjusted dynamically (by interpreters that listen to and intercept other requests). See [42] for more extensive comparison with MTL. That paper relates `Eff` with other effect systems known at that time. In the following we compare `Eff` with the systems introduced since.

The effect system of Idris [7] is an implementation of algebraic effects in the dependently-typed setting. The paper [7] introduces a domain-specific language – a notation – for describing effectful computations and demonstrates the easy combination of effects. The handlers are specifies as instances of a type class. The effect order is globally fixed and effects are interpreted essentially at the top level; there is no encapsulation of effects. The paper makes an excellent case that effect handlers provide a more flexible and cleaner alternative to monad transformers. We disagree about limitations: as we show in our implementation, the effect approach is more, rather than less expressive than monad transformers.

The closely related to our work is Kammar's et al. "Handlers in action" [38]. Whereas our library manages sets of effects using both type-level constraints and type-level lists, Kammar et al. rely only on type-class constraints. Constraints truly represent an unordered set. Using constraints exclusively however requires all effect handler definitions be top-level since Haskell does not support local type class instances. Handlers in Action rely on Template Haskell to avoid much of the inconvenience of type-class encoding and provide a pliable user interface. The provided library has excellent performance, which can also been seen from our benchmarks in §3.5. The use of Template Haskell however significantly hinders the practical use of Handlers in Action. The present paper demonstrates that many of Kammar's et al. benefits can be attained in a simple to develop and to use library, staying entirely within Haskell.

The freer or freer-like monads have already appeared before, yet connecting all the dots took long time. The origins of freer monads can be traced to the pioneering work of Hughes [29], Claessen [9] and Hinze [27], who introduced and explained the term representation of effectful monadic computations. That representation was fully developed in the monad construction toolkit Unimo [50]:

```haskell
data Unimo r a =
```

```
    Unit a
  | Effect (r (Unimo r) a)
  | ∀ b. Bind (Unimo r b) (b → Unimo r a)
```

It is quite close to `FFree` of §3.3.4, in particular, the `Bind` constructor whose second argument accumulates the continuation. Dedicating a variant `Effect` for effect requests proved to be a drawback, requiring the interpreter of `Unimo r a` monad to deal with two separate but very similar cases: `Effect e` and `Bind (Effect e) k`. One can think of free monads as eliminating this boilerplate – and throwing away the explicit continuation argument in the process. Our `FFree` brings the explicit continuation back. Unimo aimed to provide extensibility by emulating monad transformers. The Operational tutorial [1] introduces

```
data Program instr a where
  Then   :: instr a → (a → Program instr b) → Program instr b
  Return :: a → Program instr a
```

which is exactly like our `FFree`. The tutorial correctly observed that `Program instr` is a monad (although without proof). Alas the paper mis-characterised `Program` as a GADT (it is not: it is a mere existential data type in GADT notation) and has not made the connection with the free monad. It is this connection that proves that `Program instr` really is a monad. More recently, Kammar's et al. also came within an inch of the freer monad: [38, Figure 5] contains the following definition

```
 data Comp h a where
  Ret :: a → Comp h a
  Do :: (h `Handles` op ) e  ⇒
    op e u  → (Return (op e u ) →  Comp h a ) →  Comp h a
```

where `Return` is a type family and `Handles` is a three-parameter type class. It is very, very similar to `FFree` of §3.3.4, but with constraints. The very similar data type, also with the constraints, appears in [64] as the data type `NM ctx t a` for constrained monad computations. Handlers in Action did not seem to have recognised that removing all the constraints gives a new algebraic data structure that is a monad by construction. The paper describes `Comp` in the traditional way: "the monad `Comp h`, which is simply a free monad over the functor defined by those operations `op` that are handled by `h` (i.e. such that (h 'Handles ' op) e is defined for some type `e`)". `FFree` in §3.3.4 requires no functors and has no constraints or preconditions; it is a monad, period.

The papers [1] and [38] have noted the performance problem of free monads and attempted to overcome it with some sort of continuation passing – which works only up to the (reflection) point, as explained in [61]. The latter paper, which introduced type-aligned sequences, also applied them to speed up free monads. The implementation was quite complex, with the mutually recursive `FreeMonad` and `FreeMonadView`. It tried hard to fit the type-aligned sequences into the traditional free monads, rather than overcoming them. The main lesson of that paper – representing a conceptual sequence of binds as an efficient data structure – is expressed most clear in the new `Eff` monad in §3.4.

The recent 'Handlers in scope' [69] gives the more traditional introduction of extensible effects based on Data types à la carte. It also introduces the notion of a handler scope and two ways to support it. The underlying idea seems to be to run an effectful computation at the place of its handler, so to speak. The detailed investigation of the notion of scope deserves its own paper.

Compared to the right Kan extensions, left Kan extensions seem to have found so far fewer applications in functional programming. A notable application is Johann and Ghani's [36], which used a specific form of left Kan extension (only with the equality GADTs) to develop the initial algebra semantics for GADTs.

3.10 CONCLUSIONS

We have rationally reconstructed the simplified and more efficient version of the extensible effects library and illustrated it with three new challenging applications: non-determinism, handling IO errors in the presence of other effects, and monadic regions. The new library is based on the freer monad, a more general and more efficient version of the traditional free monads. To improve efficiency we systematically applied the lesson of the left Kan extension: instead of performing an operation, record the operands in the data structure and pretend it done.

The ambition is for `Eff` to be the only monad in Haskell. Rather than defining new monads programmers will be defining new effects, that is, effect interpreters.

This work was partially supported by JSPS KAKENHI Grants 22300005, 25540001, 15H02681.

4

A PURELY FUNCTIONAL COMPUTER ALGEBRA SYSTEM EMBEDDED IN HASKELL[1]

We demonstrate how methods in *Functional Programming* can be used to implement a computer algebra system. As a proof-of-concept, we present the `computational-algebra` package. It is a computer algebra system implemented as an embedded domain-specific language in *Haskell*, a purely functional programming language. Utilising methods in functional programming and prominent features of Haskell, this library achieves safety, composability, and correctness at the same time. To demonstrate the advantages of our approach, we have implemented advanced Gröbner basis algorithms, such as Faugère's F_4 and F_5, in a composable way.

4.1 INTRODUCTION

In the last few decades, the area of computer algebra has grown larger. Many algorithms have been proposed, and there have emerged plenty of computer algebra systems. Such systems must achieve *correctness*, *composability* and *safety* so that one can implement and examine new algorithms within them. More specifically, we want to achieve the following goals:

COMPOSABILITY means that users can easily implement algorithms or mathematical objects so that they work seamlessly with existing features.

SAFETY prevents users and implementors from writing "wrong" code. For example, elements in different rings, e.g. $\mathbb{Q}[x, y, z]$ and $\mathbb{Q}[w, x, y]$, should be treated differently and must not directly be added. Also, it is convenient to have handy ways to convert, inject, or coerce such values.

CORRECTNESS of algorithms, with respect to prescribed formal specifications, should be guaranteed with a high assurance.

We apply methods in the area of *functional programming* to achieve these goals. As a proof-of-concept, we present the `computational-algebra` package [33]. It is implemented as an embedded domain-specific language in the *Haskell* Language [25]. More precisely, we adopt the *Glasgow Haskell Compiler* (GHC) [21] as our hosting language. We use GHC because: its *type-system* allows us to build a safe and composable interface for computer algebra; *lazy evaluation* enables us to treat infinite objects intuitively; *declarative style* sometimes reduces a burden of writing mathematical programs; *purity* permits a wide range of equational optimisation; and there is a plenty of libraries for functional methods, especially *property-based testing*. These methods are not widely adopted in this area; an exception is *DoCon* [56], a pioneering work combining Haskell and computer algebra. Our system is designed with more emphasis on safety and correctness than DoCon, adding more

1 The contents of this chapter is based on the following article: H. Ishii. A Purely Functional Computer Algebra System Embedded in Haskell. *Computer Algebra in Scientific Computing – 20th International Workshop, CASC 2018*, 288–303, August 2018 [32]. © Springer Nature Switzerland AG 2018. The final publication is available on SpringerLink with doi: `10.1007/978-3-319-99639-4_20`.

ingredients. Although we use a functional language, some methods in this paper are applicable in imperative languages.

This paper is organised as follows. In Section 4.2, we discuss how the progressive type-system of GHC enables us to build a safe and expressive type-system for a computer algebra. Then, in Section 4.4, we see how the method of *property-based testing* can be applied to verify the correctness of algebraic programs in a lightweight and top-down manner. To demonstrate the practical advantages of Haskell, Section 4.5 gives a brief description of the current implementations of the Hilbert-driven, F_4 and F_5 algorithms. We also take a simple benchmark there. We summarise the paper and discuss related and future works in Section 4.6.

4.2 TYPE SYSTEM FOR SAFETY AND COMPOSABILITY

In this section, we will see how the progressive type-level functionalities of GHC can be exploited to construct a safe, composable and flexible type-system for a computer algebra system. There are several existing works on type-systems for computer algebra, such as in Java and Scala [47, 37], and DoCon. However, none of them achieves the same level of safety and composability as our approach, which utilises the power of *dependent types* and *type-level functions*.

4.2.1 *Type Classes to Encode Algebraic Hierarchy*

We use *type-classes*, an ad-hoc polymorphism mechanism in Haskell, to encode an algebraic hierarchy. This idea is not particularly new (for example, see Mechveliani [56] or Jolly [37]), and we build our system on top of the existing `algebra` package [46], which provides a fine-grained abstract algebraic hierarchy.

Code 2 Group structure, coded in the `algebra` package

```
1 class Additive a where
2   (+) :: a → a → a
3 class Additive a ⇒ Monoidal a where
4   zero :: a
5 class Monoidal a ⇒ Group a where
6   negate :: a → a
```

Code 2 illustrates a simplified version of the algebraic hierarchy up to Group provided by the `algebra` package. Each statement between **class** or ⇒ and **where**, such as `Additive a` or `Monoidal a`, expresses the constraint for types. For example, Lines 1 and 2 express "a type `a` is `Additive` if it is endowed with a binary operation +", and Lines 3 and 4 that "a type `a` is `Monoidal` if it is `Additive` and has a distinguished element called `zero`".

Note that, none of these requires the "proof" of algebraic axioms. Hence, one can accidentally write a non-associative `Additive`-instance, or non-distributive `Ring`-instance[2]. This sounds rather "unsafe", and we will see how this could be addressed reasonably in Section 4.4.

4.2.2 *Classes for Polynomials and Dependent Types*

Expressing algebraic hierarchy using type-class hierarchy, or class inheritance, is not so new and they are already implemented in DoCon or JAS. However, these systems lack a functionality to

2 Indeed, one can use *dependent types*, described in the next subsection, to require such proofs. However, this is too heavy for the small outcome, and does not currently work for primitive types.

Code 3 A type-class for polynomials

```
class (Module (Coeff poly) poly, Commutative poly, Ring poly,
       CoeffRing (Coeff poly), IsMonomialOrder (MOrder poly))
    ⇒ IsOrdPoly poly where
  type Arity  poly :: ℕ
  type MOrder poly :: Type
  type Coeff  poly :: Type
  liftMap :: (Module (Scalar (Coeff poly)) alg, Ring alg)
          ⇒ (ℕ_<Arity poly → alg) → poly → alg
  leadTerm :: poly → (Coeff poly, OrdMonom (MOrder poly) n)
  ...
```

Code 4 Examples for polynomial instances

```
1  instance (IsMonomialOrder ord, CoeffRing r)
2        ⇒ IsOrdPoly (OrdPoly r ord n) where
3    type Arity  (OrdPoly r ord n) = n
4    type MOrder (OrdPoly r ord n) = ord
5    type Coeff  (OrdPoly r ord n) = r
6    ...
7
8  f :: OrdPoly ℚ Grevlex 3
9  f = let [x,y,z] = vars in x ^ 2 × y + 3 × x + z + 1
10
11 instance (CoeffRing r) ⇒ IsOrdPoly (Unipol r) where
12   type Arity  (OrdPoly r ord n) = 1
13   type MOrder (OrdPoly r ord n) = Lex
14   type Coeff  (OrdPoly r ord n) = r
15   ...
```

distinguish the arity of polynomials or the denominator of a quotient ring. In particular, DoCon uses sample arguments to indicate such parameters, and they cannot be checked at compile-time. To overcome these restrictions, we use *Dependent Types*.

For example, Code 3 presents the simplified definition of the class `IsOrdPoly` for polynomials. We provide an abstract class for polynomials, not just an implementation, to enable users to choose appropriate internal representations fitting their use-cases.

The class definition includes not only functions, but also *associated types*, or *type-level functions*: `Arity`, `MOrder` and `Coeff`. Respectively, they correspond to the number of variables, the monomial ordering and the coefficient ring.

Note that `liftMap` corresponds to the universality of the polynomial ring $R[X_1, \ldots, X_n]$; i.e. the free associative commutative R-algebra over $\{1, \ldots, n\}$. In theory, this function suffices to characterise the polynomial ring. However, for the sake of efficiency, we also include some other operations in the definition.

Code 4 shows example instance definitions for the standard multivariate and univariate polynomial ring types. Note that, in Lines 8 and 12, number literal *expressions* 1 and 3 occur in *type* contexts. As we had seen in Section 2.4.2, types depending on expressions are called *Dependent Types* in type theory. Our library heavily uses this functionality, and achieves the type-safety preventing users from unintendedly confusing elements from different rings.

4.2.3 *Proofs in Dependent Types and Type-driven Casting Function*

Code 5 Various casting function, with simplified type-signatures

```
convPoly :: (Coeff r ∼ Coeff r', MOrder r ∼ MOrder r',
             Arity r ∼ Arity r')
         ⇒ r → r'
injVars :: (Arity r ≤ Arity r', Coeff r ∼ Coeff r')
        ⇒ r → r'
injVarsOffset :: (n + Arity r ≤ Arity r', Coeff r ∼ Coeff r')
              ⇒ Sing n → r → r'
```

In theory, we can use `liftMap` to cast between any elements of "compatible" polynomial rings. To reduce the burden to write boilerplate casting functions, our library comes with smart functions, as shown in Code 5. The `convPoly` function maps a polynomial into one with the same setting but different representation; e.g. `OrdPoly ℚ Lex 1` into `Unipol ℚ`. The next `injVars` function maps an element of $R[X_1, \ldots, X_n]$ into another polynomial ring with the same coefficient ring, but with more number of variables, e.g. $R[X_1, \ldots, X_{n+m}]$, regardless of ordering. For example, it maps `Unipol ℚ` into `OrdPoly ℚ Grevelx 3`. Then, `injVarsOffset` is a variant of `injVars` which maps variables with offset; for example,

```
injVarsOffset [sn|3|] :: Unipol ℚ → Polynomial ℚ 5
```

maps $\mathbb{Q}[X]$ into $\mathbb{Q}[X_0, \ldots, X_4]$ with $X \mapsto X_3$. Here, `[sn|3|]` is a *singleton*, which we had seen in Section 2.4.2, for the type-level natural number 3. More precisely, for any *type-level natural* `n`, there is the unique *expression* `sing :: Sing n` and we can use it as a tag for type-level arguments.

To work with type-level naturals, we sometimes have to *prove* some constraints. For example, suppose we want to write a variant of `injVars` mapping variables to *the end of* those of the target polynomial ring, instead of *the beginning*. We might first write it as follows:

```
injVarsAtEnd :: (Arity r ≤ Arity r', Coeff r ∼ Coeff r')
             ⇒ r → r'
injVarsAtEnd =
  let sn = sing :: Sing (Arity r)
      sm = sing :: Sing (Arity r')
  in injVarsOffset (sm ⊖ sn) −− Errors!
```

However, GHC cannot see `Arity r' − Arity r + Arity r ≤ Arity r'` even if side-condition `Arity r ≤ Arity r'` was given. Although this constraint is rather clear to us, we have to give the compiler its proof. We have developed the `type-natural` package [35] which includes typical "lemmas". For example, we can use the `minusPlus` lemma to fix this:

```
−− From type−natural:
minusPlus :: Sing n → Sing m
          → IsTrue (m ≤ n) → ((n − m) + m) ≃ n

injVarsAtEnd :: (Arity r ≤ Arity r', Coeff r ∼ Coeff r')
             ⇒ r → r'
injVarsAtEnd =
  let sn = sing :: Sing (Arity r)
      sm = sing :: Sing (Arity r')
  in withRefl (minusPlus sm sn Witness) $
```

```
    injVarsOffset (sm ⊖ sn)
```

Since giving such a proof each time is rather tedious, we can use type-checker plugins to let the compiler try to prove constraints automatically. In particular, the author developed the `ghc-typelits-presburger` plugin [34] to resolve propositions in Presburger arithmetic at compile time. With a help from this plugin, one can just use the first implementation of `injVarsAtEnd` by just adding the following line at the beginning of source-code:

```
{-# OPTIONS_GHC -fplugin GHC.TypeLits.Presburger #-}
```

This pragma tells the compiler to call the type-checker plugin at the compile time, which resolves the constraint that $n \leq m \implies m - n + n = m$, since this proposition is a theorem in Presburger arithmetic. Indeed, a large part of `type-natural` package is built on top of this type-checker plugin. The `type-natural` package also provides a way to pattern-match on GHC's builtin type-level naturals, which is impossible without any trick. For example, the `replicateV` example can be also written with builtin naturals, not with Peano numerals as we did in Section 2.4.2:

```
{-# OPTIONS_GHC -fplugin GHC.TypeLits.Presburger #-}
import Data.Type.ℕ.Builtin (ZeroOrSucc(Zero, Succ))
replicateV :: Sing (n :: ℕ) → a → Vec n a
replicateV Zero     _ = Nil
replicateV (Succ n) a = a :- replicateV n a
```

Our library also provides the `LabPoly` type, which converts existing polynomial types into "*labelled*" ones. For example, one can write as follows:

```
f :: LabPoly (Polynomial ℚ 3) '["x", "y", "z"]
f = 5 × #x ^ 2 × #y ^ 3 − #y × #z + 1
```

This relies on the `DataKinds` and `OverloadedLabels` language extensions of GHC. GHC's type system is strong enough to reject illegal terms and types, such as `#w :: LabPoly (Unipol ℚ) '["a"]` (w is not listed as a variable) or `LabPoly (Polynomial ℚ 3) '["x", "y", "x"]` (the variable x occurs twice). Using the type-level information, one can invoke the canonical inclusion maps naturally as follows:

```
f :: LabPoly' ℚ Grevlex '["x", "y", "z"]
f = #x × #y × #z + 2 × #y − 3  × #z × #x + 1
g :: LabPoly' ℚ Lex '["w", "z", "y", "u", "x"]
g = canonicalMap f

-- Where:
canonicalMap :: (xs ⊆ ys, Wraps xs poly, Wraps ys poly',
                 IsPolynomial poly, IsPolynomial poly',
                 Coeff poly ~ Coeff poly')
             ⇒ LabPoly poly xs → LabPoly poly' ys
```

4.2.4 *Optimising Casting Functions with Rewriting Rules*

Since the casting functions are implemented generically, they sometimes introduce unnecessary overhead. For example, if one uses `injVars` with the *same* source and target types, it should just be the identity function. Fortunately, we can use the type-safe *Rewriting Rule* functionality of GHC to achieve this:

```
{-# RULES "injVars/identity" injVars = id #-}
```

Each rewriting rule fires at compile-time, if there is a term matching the left-hand side of the rule and having the same type as the right-hand side.

In Haskell, it suffices just to consider algebraic laws to write down custom rewriting rules. This is due to the *purity* of Haskell. That is, every expression in Haskell is pure, in a sense that they evaluate to the same result when given the same arguments. Note that this does not mean that Haskell cannot treat values with side-effects; indeed, the type-system of Haskell distinguishes pure and impure values at type-level, and one can treat impure operations without violating purity as a whole. The trick behind this situation is to describe side-effects as some kind of abstract instructions, instead of treating impure values directly. Hence, for example, duplicating the same term does not make any difference in its meaning, provided that it is algebraically correct. Such a rewriting rule is used extensively in Haskell. For example, Stream Fusion [12] uses them to eliminate unnecessary intermediate expressions and fuse complicated functions into efficient one-path constructions. Yet, DoCon did not do any optimisation using rewriting rules.

In our library, we also use rewriting rules to remove idempotent applications such as "grading" a monomial ordering twice, e.g:

```
{-# RULES "graded/graded" ∀ ord.
  graded (graded ord) = graded ord #-}
```

4.2.5 *Notes on Applicability in Imperative Languages*

The safety we achieved in this section cannot be achieved at compile-time without dependent types and type-level functions. Existing works using type-classes or class inheritance to encode algebraic hierarchy, such as JAS or DoCon, lack this level of safety. In theory, one can achieve the same level of safety even in a statically-typed *imperative* language, if it supports a kind of dependent types. For example, in C++, templates with non-type arguments can be used to simulate dependent types. On the other hand, in Java, Generics do not allow non-type arguments and we need to mimic Peano numerals with classes. In either case, it requires much effort to prove the properties of naturals within them, because they lack dedicated support for type-level naturals or type-checker plugins.

On the other hand, to make use of rewriting rules, we need purity as discussed above.

4.3 TYPE-SAFE QUOTIENT RINGS WITH IMPLICIT CONFIGURATION

In existing less-typed approaches, such as DoCon or JAS, one can treat elements of quotient rings, but cannot distinguish the denominator, because their type-systems are not strong enough to express denominators at the type-level. So, if $R = k[X, Y]$, $I = (X^2 + Y^2)$ and $J = (X, Y)$, we cannot distinguish R/I from R/J at type-level and hence library users can easily add elements from R/I and R/J unintendedly!

If we can use *fully-dependent types*, the problem would be completely solved; we can express a quotient ring of R by an ideal I as the type like "`Quot r i`"; i.e. we can make a type also dependent on a denominator ideal. Then, if I and J are distinct ideals, the quotient rings have the different types, say "`Quot r i`" and "`Quot r j`".

But, how can we achieve the same distinction in our *weakly* dependently-typed setting? Since we use only naturals and symbols as type-level values, one cannot lift ideals to type-levels directly[3]. We use a method of *implicit configuration* [43] to overcome this situation. Code 6 illustrates the API

3 Indeed, the current GHC allows us to lift-up some ideals up to type-level. But, such a lifting easily violates the implementation hiding policy and unboxed values such as `Double`s cannot be lifted.

Code 6 Basic interface for quotient rings

```
1 data Quot r i
2 modIdeal :: Reifies i (Ideal r) ⇒ r → Quot r i
3 withQuot :: Ideal poly
4          → (∀ i. Reifies i (Ideal poly) ⇒ Quot poly i)
5          → poly
6
7 instance (Reifies i (Ideal r), IsOrdPoly r) ⇒ Ring (Quot r i)
```

to treat quotient rings. Intuitively, the type `Quot r i` corresponds to the quotient ring of R by I as above. But here, one cannot specify the value of I directly; in particular, the type parameter `i` is *not* a lifted ideal expression. So how can we "encode" the value of ideal to the type parameter `i`?

One trick playing a role here is to use the `Reifies` type-class to express such information. Intuitively, we read the type-constraint `Reifies i (Ideal r)` as "`i` must carry information about an ideal on the ring `r`". So, Line 7 defines a ring instance for `Quot r i` if and only if `i` carries information of some ideal on the ring `r`. For example, consider the following (pseudo)code:

```
data MyIdeal

instance Reifies MyIdeal (Ideal ℚ[X,Y,Z]) where
  reflect @MyIdeal _ = ⟨X^2 + Y^2 − Z, X + Y − 1⟩
```

Then one can use `Quot ℚ[X,Y,Z] MyIdeal` as the type corresponding to the quotient ring $\mathbb{Q}[X,Y,Z]/\langle X^2+Y^2-Z, X+Y-1\rangle$ and one can access the content of an ideal with `reflect` function.

In this way, one can treat quotient rings with *specific* ideals by providing custom `Reifies` instances. But, how can we treat quotient rings by *general* ideals? The trick using higher-rank polymorphism can help here. Indeed, in Code 6, the `withQuot` function gives us a way to temporarily reifying an arbitrary ideal to the type parameter and do the computation in the corresponding quotient ring. The function `withQuot` takes an ideal `j` and an element of, or computation `f` in a quotient ring and returns the remainder `f` mod `j` of the whole computation `f` modulo `i`. Here, the second argument `f` must be *polymorphic*, or *generic*, in the *type* parameter `i`. That is to say, the second argument `f` for `withQuot` must be completely generic and agnostic about the specific information of an ideal except for that it is an ideal. Then `withQuot` virtually defines a temporary instance for `Reifies` coding information of `j` and do the computation `f` instantiating the value of `i` with `j` and returns the representative element of the result of computation.

If one nests the `withQuot`s, then by the genericity of type parameter `i`'s prevent us from adding or multiplying elements of quotient rings by the different `j`'s.

In this way, we can treat quotient rings type-safely yet limiting the use of dependent-types only to the weak form.

4.4 LIGHTWEIGHT CORRECTNESS: PROPERTY-BASED TESTING

4.4.1 *Property-based Testing Introduced*

In this section, we will address the correctness issue, in a top-down, or *lightweight* manner. Especially, we apply the method of *property-based testing* [10] to verify the correctness of our implementation. The idea is that one specifies the formal properties that the implemented algorithms and types must satisfy, and checks if they hold by testing them against randomly or exhaustively generated inputs.

Although it is not as rigorous as a theorem proving, it still gives a guarantee of the correctness at high assurance, after repeating tests time after time.

Code 7 Formal Specification of Algebraic Programs

```
1  prop_division :: ℚ → Property
2  prop_division q =
3      q ≠ 0 ⟹ (recip q × q = 1 ∧ q × recip q = 1)
4    ∧ q × 1 = q ∧ 1 × q = q
5
6  prop_passesSTest n =
7    forAll (idealOfArity n) $ λ ideal →
8    let gs = calcGroebnerBasis (toIdeal ideal)
9    in all (isZero ∘ (`modPoly` gs))
10          [sPoly f g | f ← gs, g ← gs, f ≠ g]
```

Code 7 presents the example specifications for algebraic programs. In Lines 1 through 4, `prop_division` states that the implementation of $\mathbb{Q}$ must satisfy the axioms of division ring. The `prop_passesSTest` function demand the result of `calcGroebnerBasis` to pass the S-test. The tester accepts the specifications above, generates a specified number of inputs (default: 100) and tests against them. If all the inputs satisfy the specifications, it successfully halts; otherwise, it reports counterexamples, which is useful while debugging.

4.4.2 *Discussion*

There are several libraries for property-based testing adopting different strategies to generate inputs. For example, QuickCheck [10] generates inputs randomly, while SmallCheck [63] exhaustively enumerates inputs in the depth-increasing order. Even though there are other implementations of property-based testers in languages other than Haskell [31], it does not seem that it is applied in existing systems, such as Singular [23], JAS or DoCon.

By its *generative* nature, property-based testing has several drawbacks and pitfalls. First, evidently, it cannot assure the validity as rigorously as the *formal theorem proving*, unless the input space is finite. There are several pieces of research that combine formal theorem proving and computer algebra to rigorously certify correctness of implementations (for example, [57, 11]). These first formalise the theory of Gröbner basis in the constructive type-theory. Then, execute them within the host theorem proving language, or extract the program into other languages. However, by its nature, this approach requires everything to be proven formally. It is not so easy a task to prove the correctness of every part of a program, even with help from automatic provers. Even if one manages to finish the proof of the validity of some algorithm, when one wants to optimise it afterwards, then one must prove the "equivalence" or validity of that optimisation. Moreover, it is sometimes the case that the validity, or even termination, of the algorithm remains unknown when it is implemented; e.g. the correctness and termination of Faugerè's F_5 [18] are proven very recently [60]. Furthermore, there is an obvious restriction that we can extract programs only into the languages supported by the theorem prover. We consider these conditions too restrictive, and decided to adopt theorem proving only in trivial arity arithmetic.

Secondly, if the algorithm has a bad time complexity, property-based tests can easily explode. Specifically, since Gröbner bases have double-exponential worst time complexity, randomly generated input can take much time to be processed. One might reduce the burden by combining randomised and enumerative generation strategies carefully, but there is still a possibility that there are small inputs which take much time. To avoid such a circumstance, one can reduce the number of inputs, however it also reduces the assurance of validity.

Finally, they are not so good at treating *existential properties*. Although SmallCheck provides the existential quantifier in its vocabulary, it just tries to find solutions up to a prescribed depth. If solutions are relatively "larger" than its inputs, this results in *false-negative* failures. For example, one can write the following specification that demands each element of the result of `calcGroebnerBasis` to be a member of the original ideal, however it does not work as expected:

```
prop_gbInc ideal =
  let j = calcGroebnerBasis ideal
  in exists $ λ cs →
     and (zipWith (λ f gs → f == dot ideal gs) j cs)
```

In the above, `dot i g` denotes the "dot-product". As a workaround, we currently combine inter-process communication with property-based testing. More specifically, we invoke a reliable existing implementation, such as SINGULAR, inside the spec as follows:

```
prop_gbInc = forAll arbitrary $ λ i → monadicIO $ do
  let gs = calcGroebnerBasis i
  is ← evalSingularIdealWith [] [] $
        funE "reduce" [
          idealE gs, funE "groebner" [idealE i]]
  return $ all isZero is
```

Thus, if the existential property in question is decidable and has an existing reliable implementation, then it might be better to call it inside specifications.

4.5 CASE STUDY: THE HILBERT-DRIVEN, F_4 AND F_5 ALGORITHMS

In this section, we will focus on three algorithms as case-studies: the Hilbert-driven, F_4 and F_5 algorithms. Firstly, we demonstrate the power of laziness and parallelism by the Hilbert-driven algorithm. Then by the F_4 interface, we illustrate the practical example of composability. Finally, we skim through the simplified version of the main routine of F_5 and see how imperative programming with mutable states can be written purely in Haskell. For our purpose, we will discuss only a fragment of implementations that elucidates the advantages of Haskell, rather than the entire implementation and theoretical details.

4.5.1 *Homogenisation and Hilbert-driven Basis Conversion*

As we have seen in Section 1.2.2, *homogenisation* is a powerful tool in Gröbner basis computation. Code 8 is an API for (de-)homogenisation and Gröbner basis computation. The type `Homogenised poly` represents polynomials obtained by homogenising polynomials of type `poly`. Then `calcGBViaHomog calc i` first checks if the input `i` is homogeneous. If it is so, then it applies the argument `calc` to its input directly (Line 15); otherwise, it first homogenises the input, applies `calc`, and then unhomogenises it to get the final result (Line 16). Note that, though it uses the same term `calc` in both cases, they have different types. In the first case, since it just feeds an input directly, `calc` has type `Ideal poly → [poly]`. On the other hand, in the non-homogeneous case, it is applied *after* homogenisation, hence it is of type `Ideal (Homogenised poly) → [Homogenised poly]`. Thus, `calcGBViaHomog` takes a *polymorphic function* as its first argument and this is why we

Code 8 Basic API for homogenisation

```
data Homogenised poly
instance IsOrdPoly poly ⇒ IsOrdPoly (Homogenised poly) where
  type Arity  (Homogenised poly) = 1 + Arity poly
  type MOrder (Homogenised poly) = HomogOrder (MOrder poly)
  type Coeff  (Homogenised poly) = Coeff poly
  …
homogenise   :: IsOrdPoly poly ⇒ poly → Homogenised poly
unhomogenise :: IsOrdPoly poly ⇒ Homogenised poly → poly

calcGBViaHomog :: (Field (Coeff poly), IsOrdPoly poly)
               ⇒ (∀ r. (Field (Coeff r), IsOrdPoly r)
                       ⇒ Ideal r → [r])
               → Ideal poly → [poly]
calcGBViaHomog calc i
  | all isHomogeneous i = calc i
  | otherwise = map unhomogenise (calc (fmap homogenise i))
```

have ∀ inside the type of the first argument. Such a nested polymorphic type is called a *rank n polymorphic type*, which we had seen in Section 2.4.1[4].

Code 9 Data-type of and operations on Hilbert–Poincaré series

```
1  data HPS n = HPS { taylor :: [ℤ], hpsNumerator :: Unipol ℤ }
2
3  instance Eq (HPS a) where
4    (==) = (==) `on` hpsNumerator
5  instance Additive (HPS n) where
6    HPS cs f + HPS ds g = HPS (zipWith (+) cs ds) (f + g)
7  instance LeftModule (Unipol ℤ) (HPS n) where
8    f • HPS cs g = HPS (conv (taylor f ++ repeat 0) cs) (f × g)
9
10 conv :: [ℤ] → [ℤ] → [ℤ]
11 conv (x : xs) (y : ys) =
12   let parSum a b c = a `par` b `par` c `seq` (a + b + c) in
13   x × y :
14    zipWith3 parSum (map (x×) ys) (map (y×) xs) (0 : conv xs ys)
```

For example, one can use the *Hilbert-driven algorithm*, which we reviewed in 1.2.3, as the first argument to `calcGBViaHomog`. It first computes a Gröbner basis w.r.t. a lighter monomial ordering, compute the Hilbert–Poincaré series (HPS) with it and use it to compute Gröbner basis w.r.t. the heavier ordering. In this procedure, we need the following operations on HPS: Equality test on HPS's, n^{th} Taylor coefficient of the given HPS, and the $\mathbb{Z}[X]$-module operation on HPS. Code 9 illustrates such an interface for HPS. For equality test, we use the numerator `hpsNumerator` of the closed form, and an *infinite list* `taylor` maintains Taylor coefficients. By the *lazy* nature of Haskell, we can intuitively treat infinite lists and write a convolution on them. In Line 12, `par` and `seq` specify the *evaluation strategy*. Briefly, expressions x and y in "x `par` y" (resp. `seq`) are evaluated

4 This can be achieved in object-oriented language with subtyping and Generics.

parallelly (resp. *sequentially*). Since every expression is pure in Haskell, we can safely take advantage of parallelism, without a possibility of changing results.

4.5.2 *A Composable Implementation of F_4*

Code 10 Matrix classes and the F_4 function

```
1  class MMatrix mat a where
2    fromRows :: [Vector a] → ST s (mat s a)
3    scaleRow :: Multiplicative a ⇒ Int → a → mat s a → ST s ()
4    …
5
6  class MMatrix (Mutable mat) a ⇒ Matrix mat a where
7    type Mutable mat :: ⋆ → ⋆
8    freeze :: Mutable mat s a → ST s (mat a)
9    …
10   gaussReduction :: Field a ⇒ mat a → mat a
11
12 type Strategy f w = f → f → w
13 f4 :: (Ord w, IsOrdPoly poly, Field (Coeff poly),
14        Matrix mat (Coeff poly))
15    ⇒ proxy mat → Strategy poly w → Ideal poly → [poly]
```

As mentioned in Section 1.2.4, F_4 is one of the most efficient algorithms for Gröbner basis computation and introduced by Faugère [17]. Briefly, F_4 reduces more than two polynomials at once, replacing S-polynomial remaindering in the Buchberger Algorithm with the *Gaussian elimination* of the matrices. This means that the efficiency of F_4 reduces to that of Gaussian elimination and the internal representation of matrices. Thus, it is useful if we can easily switch internal representations and elimination algorithms. For this purpose, we provide type-classes for mutable and immutable matrices which admit row operations and a dedicated Gaussian elimination. Code 10 demonstrates the interface for immutable and mutable matrices (`Matrix` and `MMatrix`) and the type signature of our F_4 implementation (`f4`). In Lines 1 and 6, the last type argument `a` of `Matrix` and `MMatrix` corresponds to the type of coefficients. Note that, one can give different instance definitions for the same `mat` but different coefficient types `a`. For example, one can implement efficient Gaussian elimination on $\mathbb{F}_p$ for `Matrix Mat` $\mathbb{F}_p$, and then use it in the definition of `Matrix Mat` $\mathbb{Q}$, with the Hensel lifting or Chinese remaindering.

In Line 15, the first argument of `f4` of type `proxy mat` specifies the internal representation `mat` of matrices. In addition, `f4` takes a *selection strategy* as the second argument. Here, the selection strategy is abstracted as a weighting function to some ordered types, and we store intermediate polynomials in a heap and select all the polynomials with the minimum weight at each iteration.

4.5.3 *The F_5 Algorithm*

Finally, we present the simplified version of the main routine of Faugère's F_5 [18] (Code 11), which we reviewed in Section 1.2.5. Readers may be surprised that the code looks much imperative. This is made possible by the *ST monad* [48], which we had seen in Section 2.4.1, encapsulating side-effects introduced by mutable states and prevents them from leaking outside. We use a functional heap to choose the polynomial vectors with the least signature, demonstrating the fusion of functional and imperative styles.

Code 11 Main Routine of the F_5 Algorithm

```
f5 :: (Field (Coeff pol), IsOrdPoly pol)
   ⇒ Vector pol → [(Vector pol, pol)]
f5 (map monoize → i0) = runST $ do
  let n = length i0
  gs ← newSTRef []
  ps ← newSTRef $ H.fromList [ basis n i | i ← [0..n−1]]
  syzs ← newSTRef
    [ sVec (i0 ! m) (i0 ! n) | m ← [0..n−1], n ← [0..j−1] ]
  whileJust_ (H.viewMin <$> readSTRef ps) $
  λ (Entry sig g, ps') → do
    ps := ps'
    (gs0, ss0) ← (,) <$> readSTRef gs <*> readSTRef syzs
    unless (standardCriterion sig ss0) $ do
      let (h, ph) = reduceSignature i0 g gs0
          h' = map (× injectCoeff (recip $ leadingCoeff ph)) h
      if isZero ph then syzs :⇐ (mkEntry h : )
        else do
        let adds = fromList $ mapMaybe (regSVec (ph, h')) gs0
        ps :⇐ H.union adds
        gs :⇐ ((monoize ph, mkEntry h') :)
  map (λ (p, Entry _ a) → (a, p)) <$> readSTRef gs
```

4.5.4 *Benchmarks*

Table 4.1: Benchmark results (ms)

	I_1 (Lex)	I_1 (Grevlex)	I_2 (Lex)	I_2 (Grevlex)	I_3 (Grevlex)
B	1.820×10^0	1.593×10^1	1.400×10^1	4.129×10^0	6.689×10^2
DbyD	6.364×10^1	9.162×10^2	1.147×10^2	5.647×10^1	4.125×10^2
Hilb	1.644×10^2	2.313×10^2	5.265×10^1	3.414×10^1	9.645×10^3
F_5	1.851×10^0	4.314×10^2	7.129×10^0	2.648×10^0	1.290×10^3
S(gr)	2.300×10^0	8.493×10^{-1}	2.651×10^0	8.210×10^{-1}	9.511×10^{-1}
S(sba)	2.279×10^{-1}	8.711×10^{-1}	2.343×10^{-1}	7.958×10^{-1}	1.541×10^{-1}

$$\begin{aligned}
I_1 &:= \langle 35y^4 - 30xy^2 - 210y^2z + 3x^2 + 30xz - 105z^2 + 140yt - 21u, \\
&\quad 5xy^3 - 140y^3z - 3x^2y + 45xyz - 420yz^2 + 210y^2t - 25xt + 70zt + 126yu\rangle \\
I_2 &:= \langle w + x + y + z, wx + xy + yz + zw, wxy + xyz + yzw + zwx, wxyz - 1\rangle \\
I_3 &:= \langle x^{31} - x^6 - x - y, x^8 - z, x^{10} - t\rangle
\end{aligned}$$

We also take a simple benchmark and the result is shown in Table 4.1 (examples are taken from Giovini et al. [22]). This compares the algorithms implemented in our `computational-algebra` package and Singular. The first four rows correspond to the algorithms implemented in our library; i.e. the Buchberger algorithm optimised with syzygy and sugar strategy (B), the degree-by-degree algorithm for homogeneous ideals (DbyD), the Hilbert-driven algorithm (Hilb), and F_5. S(gr) and S(sba) stand for the `groebner` and `sba` functions in the Singular computer algebra system 4.0.3. The complete source-code is available on GitHub [33][5]. The benchmark program is compiled with GHC 8.2.2 with flags `−O2 −threaded −rtsopts −with−rtsopts=−N`, and ran on an Intel Xeon

5 More specifically, we used the implementation in commit `70e6e7b`.

E5-2690 at 2.90 GHz, RAM 128GB, Linux 3.16.0-4 (SMP), using 10 cores in parallel. We used the Gauge framework to report the run-time of our library, and the `rtimer` primitive for Singular. For actual benchmark codes, see `http://bit.ly/hbench1` and `hbench2`. Unfortunately, in our system, F_4 takes much more computing time, hence we did not include the result. The results show that, among the algorithms implemented in our system, F_5 works fine in general, though it takes much time in some specific cases. Nevertheless, there remains much room for improvement to compete with the state-of-the-art implementations such as Singular, although there is one case where our implementation is slightly faster than Singular's `groebner` function.

4.6 CONCLUSIONS

In this paper, we have demonstrated how we can adopt the methods developed in the area of functional programming to build a computer algebra system. Some of these methods are also applicable in imperative languages.

In Section 4.2, we presented a type-system strong enough to detect algebraic errors at compile-time. For example, our system can distinguish number of variables of polynomial rings at type-level thanks to dependent types. It also enables us to automatically generate casting functions and we saw how their overhead can be reduced using rewriting rules. As for type-systems for a computer algebra system, there are several existing works [47, 56]. However, these systems are not safe enough for discriminating variable arity at type-level and don't make use of rewriting rules.

In Section 4.4, we successfully applied the method of *property-based testing* for verification of the implementation, which is lightweight compared to the existing theorem-prover based approach [11, 57]. Although property-based testing is not as rigorous as theorem proving, it is lightweight and can be applied to algorithms not yet proven to be valid or terminate and available also for imperative languages.

We have seen that, in Section 4.5, other features of Haskell, such as higher-order polymorphism, parallelism and laziness, can also be easily applied to computer algebra by actual examples. Even though they are shown as fragments of code, we expect them to be convincing.

Since some of the methods in this paper, such as dependent types or property-based testing, are not limited to the functional paradigm, it might be interesting to investigate their applicability in the imperative settings.

From the viewpoint of efficiency, there are much to be done. For example, efficiency of our current F_4 implementation is far inferior to that of the naïve Buchberger algorithm, and other algorithms are far much slower than state-of-the-art implementations such as Singular. To optimise implementations, we can make more use of Rewriting Rules and efficient data structures. Also, the parallelism must undoubtedly play an important role. Fortunately, there are plenty of the parallel computation functionalities in Haskell, such as Regular Parallel Arrays [39] and `parallel` package [55], and another book by Marlow [54] on general topics in parallelism in Haskell. Also, there is an existing work by Lobachev et al. [52] on parallel symbolic computation in Eden, a dialect of Haskell with parallelism support. Although Eden is retired, the methods introduced there might be helpful.

BIBLIOGRAPHY

[1] Heinrich Apfelmus, *The Operational monad tutorial*, The Monad.Reader **15** (2010), ed. by Brent Yorgey, pp. 37–56.

[2] Steve Awodey, *Category theory*, **49**, Oxford Logic Guides, Oxford University Press, 2006, ISBN: 978-0199237180.

[3] Patrick Bahr, *Composing and decomposing data types: a closed type families implementation of data types à la carte*, Workshop on Generic programming at ICFP 2014, Gothenburg, Sweden, Aug. 2014, pp. 71–82, ISBN: 978-1-4503-3042-8, DOI: 10.1145/2633628.2633635.

[4] Andrej Bauer and Matija Pretnar, *Programming with algebraic effects and handlers*, Journal of Logical and Algebraic Methods in Programming (2012), DOI: 10.1016/j.jlamp.2014.02.001, arXiv: 1203.1539 [cs.PL].

[5] Jean-Philippe Bernardy and Nicolas Pouillard, *Names for free: polymorphic views of names and binders*, Proceedings of the 2013 ACM SIGPLAN symposium on Haskell, 2013, pp. 13–24, ISBN: 978-1-4503-2383-3, DOI: 10.1145/2503778.2503780.

[6] Richard Bird, *Introduction to functional programming*, 2nd, Prentice Hall, 1998, ISBN: 978-0134843469.

[7] Edwin Brady, *Programming and reasoning with algebraic effects and dependent types*, Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, 2013, pp. 133–144, ISBN: 978-1-4503-2326-0, DOI: 10.1145/2500365.2500581.

[8] Robert Cartwright and Matthias Felleisen, *Extensible denotational language specifications*, Theoretical Aspects of Computer Software, ed. by Masami Hagiya and John C. Mitchell, Springer Berlin Heidelberg, 1994, pp. 244–272, ISBN: 978-3-540-48383-0, DOI: 10.1007/3-540-57887-0_99.

[9] Koen Claessen, *Parallel parsing processes*, Journal of Functional Programming **14**.6 (2004), pp. 741–757, DOI: 10.1017/S0956796804005192.

[10] Koen Claessen and John Hughes, *QuickCheck: a lightweight tool for random testing of Haskell programs*, Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00, New York, NY, USA: ACM, 2000, pp. 268–279, ISBN: 1-58113-202-6, DOI: 10.1145/351240.351266, URL: http://doi.acm.org/10.1145/351240.351266.

[11] Thierry Coquand and Henrik Persson, *Gröbner bases in type theory*, Types for Proofs and Programs, ed. by Thorsten Altenkirch, Bernhard Reus, and Wolfgang Naraschewski, Berlin, Heidelberg: Springer, 1999, pp. 33–46, ISBN: 978-3-540-48167-6, DOI: 10.1007/3-540-48167-2_3.

[12] Duncan Coutts, Roman Leshchinskiy, and Don Stewart, *Stream Fusion: From lists to streams to nothing at all*, Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP '07, 2007, ISBN: 978-1-59593-815-2, DOI: 10.1145/1291151.1291199.

[13] David A Cox, John Little, and Donal O'Shea, *Additional Gröbner basis algorithms*, Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, Springer, 2015, chap. 10, pp. 539–591, ISBN: 978-3319167206, DOI: 10.1007/978-3-319-16721-3_10.

[14] ———, *Ideals, varieties, and algorithms*, Springer, 2015, ISBN: 978-3319167206, DOI: 10.1007/978-3-319-16721-3.

[15] Olivier Danvy and Andrzej Filinski, *Abstracting control*, Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, 1990, pp. 151–160, ISBN: 0-89791-368-X, DOI: 10.1145/91556.91622.

[16] Richard A. Eisenberg and Stephanie Weirich, *Dependently typed programming with singletons*, ACM SIGPLAN Notices - Haskell '12 **47**.12 (Sept. 2012), pp. 117–130, ISSN: 0362-1340, DOI: 10.1145/2430532.2364522.

[17] Jean-Charles Faugère, *A new efficient algorithm for computing Gröbner bases* (F_4), Journal of Pure and Applied Algebra **139**.1-3 (1999), pp. 61–88, ISSN: 0022-4049, DOI: 10.1016/S0022-4049(99)00005-5.

[18] Jean-Charles Faugère, *A new efficient algorithm for computing Gröbner bases without reduction to zero* (F_5), Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation, Lille, France: ACM, 2002, pp. 75–83, ISBN: 1-58113-484-3, DOI: 10.1145/780506.780516.

[19] Matthew Fluet and J. Gregory Morrisett, *Monadic regions*, Journal of Functional Programming **16**.4–5 (2006), pp. 485–545, DOI: 10.1017/S095679680600596X.

[20] Shuhong Gao, Frank Volny, and Mingsheng Wang, *A new framework for computing Gröbner bases*, Mathematics of Computation **85** (Jan. 2015), DOI: 10.1090/mcom/2969.

[21] GHC Team, *The Glasgow Haskell Compiler*, 2018, URL: https://www.haskell.org/ghc/ (visited on 2018).

[22] Alessandro Giovini et al., *"One sugar cube, please" or selection strategies in the Buchberger algorithm*, Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation, ISSAC'91, ACM, 1991, pp. 5–4, ISBN: 0-89791-437-6, DOI: 10.1145/120694.120701.

[23] Gert-Martin Greuel and Gerhard Pfister, *A singular introduction to commutative algebra*, 2nd, Springer, 2007, ISBN: 9783540735410, DOI: 10.1007/978-3-662-04963-1.

[24] Haskell Committee, *Haskell 2010 language report*, ed. by Simon Marlow, 2010, URL: https://www.haskell.org/onlinereport/haskell2010/ (visited on 12/06/2018).

[25] ———, *The Haskell programming language*, URL: http://haskell.org/.

[26] R. Hindley, *The principal type-scheme of an object in combinatory logic*, Transactions of the American Mathematical Society **146** (1969), pp. 29–60, ISSN: 00029947, DOI: 10.2307/1995158, URL: http://www.jstor.org/stable/1995158.

[27] Ralf Hinze, *Deriving backtracking monad transformers*, Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, 2000, pp. 186–197, ISBN: 1-58113-202-6, DOI: 10.1145/351240.351258.

[28] ———, *Fun with phantom types*, The Fun of Programming, ed. by Jeremy Gibbons, Oege de Moor, and Geraint Jones, Palgrave, Dec. 2003, chap. 12, ISBN: 978-0333992852, URL: https://www.cs.ox.ac.uk/publications/books/fop/.

[29] John Hughes, *The design of a pretty-printing library*, First Intl. Spring School on Adv. Functional Programming Techniques, London, UK, UK: Springer-Verlag, 1995, pp. 53–96, ISBN: 3-540-59451-5, DOI: 10.1007/3-540-59451-5_3.

[30] Graham Hutton, *Programming in Haskell*, 2nd, Cambridge University Press, 2016, ISBN: 978-1316626221, DOI: 10.1017/CBO9781316784099.

[31] Hypothesis, *Most testing is ineffective - hypothesis*, 2018, URL: https://hypothesis.works (visited on 06/05/2018).

[32] Hiromi Ishii, *A purely functional computer algebra system embedded in Haskell*, Computer Algebra in Scientific Computing (Lille, France), ed. by Vladimir P. Gerdt, Wolfram Koepf, and Werner M. Seiler, vol. 11077, Lecture Notes in Computer Science, Springer, Cham, 2018, pp. 288–303, ISBN: 978-3-319-99638-7, DOI: 10.1007/978-3-319-99639-4_20, arXiv: 1807.01456.

[33] ———, *The `computational-algebra` package*, 2018, URL: https://konn.github.io/computational-algebra.

[34] ———, *The `ghc-typelits-presburger` package*, 2017, URL: http://hackage.haskell.org/package/ghc-typelits-presburger.

[35] ———, *The `type-natural` package*, 2013, URL: http://hackage.haskell.org/package/type-natural.

[36] Patricia Johann and Neil Ghani, *Foundations for structured programming with GADTs*, Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2008, pp. 297–308, ISBN: 978-1-59593-689-9, DOI: 10.1145/1328438.1328475.

[37] Raphaël Jolly, *Categories as type classes in the Scala Algebra System*, Computer Algebra in Scientific Computing, ed. by Vladimir P. Gerdt et al., Cham: Springer, 2013, pp. 209–218, ISBN: 978-3-319-02297-0, DOI: 10.1007/978-3-319-02297-0_18.

[38] Ohad Kammar, Sam Lindley, and Nicolas Oury, *Handlers in action*, Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, 2013, pp. 145–158, ISBN: 978-1-4503-2326-0, DOI: 10.1145/2500365.2500590.

[39] Gabriele Keller et al., *Regular, shape-polymorphic, parallel arrays in Haskell*, Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP '10, Baltimore, Maryland, USA: ACM, 2010, pp. 261–272, ISBN: 978-1-60558-794-3, DOI: 10.1145/1863543.1863582.

[40] Oleg Kiselyov, *Iteratees*, Proceedings of the 11th International Symposium on Functional and Logic Programming, 2012, pp. 166–181, ISBN: 978-3-642-29821-9, DOI: 10.1007/978-3-642-29822-6_15.

[41] Oleg Kiselyov and Hiromi Ishii, *Freer monads, more extensible effects*, Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell, Haskell '15, Vancouver, BC, Canada: ACM, 2015, pp. 94–105, ISBN: 978-1-4503-3808-0, DOI: 10.1145/2804302.2804319.

[42] Oleg Kiselyov, Amr Sabry, and Cameron Swords, *Extensible effects: an alternative to monad transformers*, Proceedings of the 2013 ACM SIGPLAN symposium on Haskell (Boston, Massachusetts, USA), ACM New York, NY, USA, 2013, pp. 59–70, DOI: 10.1145/2503778.2503791.

[43] Oleg Kiselyov and Chung-chieh Shan, *Functional pearl: implicit configurations - or, type classes reflect the values of types*, Proceedings of the 2004 ACM SIGPLAN workshop on Haskell (Snowbird, Utah, USA), Jan. 2004, pp. 33–44, DOI: 10.1145/1017472.1017481.

[44] ———, *Lightweight monadic regions*, Proceedings of the First ACM SIGPLAN Symposium on Haskell, 2008, pp. 1–12, ISBN: 978-1-60558-064-7, DOI: 10.1145/1411286.1411288.

[45] Oleg Kiselyov et al., *Backtracking, interleaving, and terminating monad transformers (functional pearl)*, Proceedings of the tenth ACM SIGPLAN international conference on Functional Programming, 2005, pp. 192–203, ISBN: 1-59593-064-7, DOI: 10.1145/1086365.1086390.

[46] Edward A. Kmett, *The* `algebra` *package*, 2011, URL: http://hackage.haskell.org/package/algebra (visited on 2018).

[47] Heinz Kredel and Raphael Jolly, *Generic, type-safe and object oriented computer algebra software*, Computer Algebra in Scientific Computing, ed. by Vladimir P. Gerdt et al., Berlin, Heidelberg: Springer, 2010, pp. 162–177, ISBN: 978-3-642-15274-0, DOI: 10.1007/978-3-642-15274-0_14.

[48] John Launchbury and Simon L. Peyton Jones, *Lazy functional state threads*, Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI '94, Orlando, Florida, USA: ACM, 1994, pp. 24–35, ISBN: 0-89791-662-X, DOI: 10.1145/178243.178246.

[49] Sheng Liang, Paul Hudak, and Mark Jones, *Monad transformers and modular interpreters*, Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (San Francisco, CA), 1995, pp. 333–343, ISBN: 0-89791-692-1, DOI: 10.1145/199448.199528.

[50] Chuan-kai Lin, *Programming monads operationally with Unimo*, Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming, 2006, pp. 274–285, ISBN: 1-59593-309-3, DOI: 10.1145/1159803.1159840.

[51] Miran Lipovača, *Learn you a Haskell for great good: A beginner's guide*, No Starch Press, 2011, ISBN: 978-1593272838, URL: http://learnyouahaskell.com (visited on 12/06/2018).

[52] Oleg Lobachev and Rita Loogen, *Implementing data parallel rational multiple-residue arithmetic in Eden*, Computer Algebra in Scientific Computing, ed. by Vladimir P. Gerdt et al., Berlin, Heidelberg: Springer, 2010, pp. 178–193, ISBN: 978-3-642-15274-0, DOI: 978-3-642-15274-0_15.

[53] Christoph Lüth and Neil Ghani, *Composing monads using coproducts*, Proceedings of the seventh ACM SIGPLAN international conference on Functional Programming, 2002, pp. 133–144, ISBN: 1-58113-487-8, DOI: 10.1145/581478.581492.

[54] Simon Marlow, *Parallel and concurrent programming in Haskell: techniques for multicore and multithreaded programming*, O'Reilly Media, 2013, ISBN: 9781449335908.

[55] Simon Marlow et al., *Seq no more: better strategies for Parallel Haskell*, Proceedings of the Third ACM Haskell Symposium on Haskell, Haskell '10, Baltimore, Maryland, USA: ACM, 2010, pp. 91–102, ISBN: 978-1-4503-0252-4, DOI: 10.1145/1863523.1863535, URL: http://doi.acm.org/10.1145/1863523.1863535.

[56] Serge D. Mechveliani, *Computer algebra with Haskell: applying functional–categorial–"lazy" programming*, Proceedings of International Workshop CAAP, 2001, pp. 203–211.

[57] Sergei D. Mechveliani, *Docon-a a provable algebraic domain constructor*, 2018, URL: http://www.botik.ru/pub/local/Mechveliani/docon-A/2.02/manual.pdf (visited on 06/05/2018).

[58] E. Moggi, *Computational lambda-calculus and monads*, LICS, Piscataway, NJ, USA: IEEE Press, 1989, pp. 14–23, DOI: 10.1109/LICS.1989.39155.

[59] Lee Naish, *Pruning in logic programming*, tech. rep. 95/16, Department of Computer Science, University of Melbourne, 1995.

[60] Senshan Pan, Yupu Hu, and Baocang Wang, *The termination of the F5 algorithm revisited*, Proceedings of the 38th International Symposium on Symbolic and Algebraic Computation, ISSAC '13, Boston, Maine, USA: ACM, 2013, pp. 291–298, ISBN: 978-1-4503-2059-7, DOI: 10.1145/2465506.2465520.

[61] Atze van der Ploeg and Oleg Kiselyov, *Reflection without remorse: revealing a hidden sequence to speed up monadic reflection*, Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell, 2014, pp. 133–144, ISBN: 978-1-4503-3041-1, DOI: 10.1145/2633357.2633360.

[62] Gordon Plotkin and Matija Pretnar, *Handlers of algebraic effects*, Programming Languages and Systems, York, UK: Springer-Verlag, 2009, pp. 80–94, ISBN: 978-3-642-00590-9, DOI: 10.1007/978-3-642-00590-9_7.

[63] Colin Runciman, Matthew Naylor, and Fredrik Lindblad, *SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values*, Proceedings of the First ACM SIGPLAN Symposium on Haskell, Haskell '08, Victoria, BC, Canada: ACM, 2008, pp. 37–48, ISBN: 978-1-60558-064-7, DOI: 10.1145/1411286.1411292, URL: http://doi.acm.org/10.1145/1411286.1411292.

[64] Neil Sculthorpe et al., *The constrained-monad problem*, Proceedings of the 18th ACM SIGPLAN international conference on Functional programming, 2013, pp. 287–298, ISBN: 978-1-4503-2326-0, DOI: 10.1145/2500365.2500602.

[65] Tim Sheard and Emir Pašalić, *Two-level types and parameterized modules*, Journal of Functional Programming **14**.5 (Sept. 2004), pp. 547–587, DOI: 10.1017/S095679680300488X.

[66] Guy L. Steele Jr., *Building interpreters by composing monads*, Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1994, pp. 472–492, ISBN: 0-89791-636-0, DOI: 10.1145/174675.178068.

[67] Wouter Swierstra, *Data types à la carte*, Journal of Functional Programming **18**.4 (July 2008), pp. 423–436, DOI: 10.1017/S0956796808006758.

[68] Nicolas Wu and Tom Schrijvers, *Fusion for free: efficient algebraic effect handlers*, Mathematics of Program Construction 2015, 2015, ISBN: 978-3-319-19797-5, DOI: 10.1007/978-3-319-19797-5_15, URL: /Research/papers/mpc2015.pdf.

[69] Nicolas Wu, Tom Schrijvers, and Ralf Hinze, *Effect handlers in scope*, Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell, 2014, pp. 1–12, ISBN: 978-1-4503-3041-1, DOI: 10.1145/2633357.2633358.

[70] Brent A. Yorgey et al., *Giving Haskell a promotion*, Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation, TLDI '12, Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 53–66, ISBN: 978-1-4503-1120-5, DOI: 10.1145/2103786.2103795.

SYMBOLS

INDEX