PUTTING TYPES TO GOOD USE

Arjen van Weelden

IPA dissertation series 2007-10
ISBN: 978-90-9022041-3

Typeset with $\LaTeX\ 2_\varepsilon$
Printed by Print Partners Ipskamp

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

PUTTING TYPES TO GOOD USE

Een wetenschappelijke proeve op het gebied van de
Natuurwetenschappen, Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. mr. S.C.J.J. Kortmann,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op woensdag 17 oktober 2007
om 15:30 uur precies

door

Arjen van Weelden

geboren op 21 oktober 1978
te Arnhem

Preface

I would like to thank all the people who in some way supported me and this thesis. Their influence, either in their professional role or as a friend, has improved me, my research, and this thesis. I thank Rinus for offering me this job as a researcher and his enthusiastic support from the start. Sjaak has my gratitude for his gentle supervision after my research focus shifted from Dynamics to Generics. For showing me how to teach and how to dress formally, credits must go to Erik. Diederik must be mentioned for this ability to turn unlimited amounts of coffee into constructive criticism. By showing me assembly code, John improved my understanding of pure and lazy programming languages. Furthermore, I thank all the people in the Software Technology group and the Computer Science department at the Radboud University Nijmegen. This thesis would not have been possible without my parents, who gave me the opportunity to pursue my studies. Last but not least, I thank Floor for her support these years. Anyone mistakenly absent here ought to blame it on the drinks we shared on one happy occasion or the other.

Arjen
Nijmegen, August 2007

Table of Contents

CHAPTER 1

Introduction

This thesis presents some of my research on improving software construction using relatively new techniques provided by functional programming languages. This chapter starts with the initial problem description in Sect. 1.1 and the motivation for using functional techniques in Sect. 1.2. Sections 1.3 and 1.4 introduce the two new techniques we apply to several software construction issues throughout the body of this thesis. The scope and focus of this research is further narrowed in Sect. 1.5. This research did not take place in isolation and the published papers, which constitute the main chapters of this work, are written in co-authorship with my promotor, copromotor, and several other researchers. Additionally, we would like to take this opportunity to thank all the anonymous reviewers for their helpful comments on the original papers. A more detailed description of the contents of this thesis is presented in Sect. 1.6, which also mentions my research contributions to each of the chapters.

1.1 The Complexity of Software

People are still not very good at writing correct software. The famous Dutch computer scientist Dijkstra[1] [Dij72] already warned of problems in constructing correct software as early as 1972. He argued that the main problem was the increasing number of abstraction layers required to reduce complexity, and to bridge the gap between the hours that programs ran and the milliseconds the individual

[1]For the readers who expect this thesis to be about computers, I would like to quote Dijkstra that "*Computer science is no more about computers than astronomy is about telescopes*".

computer instructions took. Computers have gotten, and will most likely continue to get, faster and increasingly parallel and distributed. Moreover, programmers try to add more and more features to (existing) software. Although many layers of abstraction can already be found in both software (e.g., extensive libraries, middle-ware, high-level domain specific programming languages) and hardware (e.g., virtual machines, out–of–order execution, multiple pipelines, threading), bridging the still growing gap between the running time of a program and the computer instructions remains difficult. Obviously, the related gap between user specifications and correct executable programs is also very wide.

Much literature about the problems of software construction has been published. The current state of software development is sometimes referred to as the "software crisis". A citation, taken from Royce [Roy91], which emphasizes software construction problems and mentions the "software crisis", predates the start of our research by more than a decade: "*The construction of new software that is both pleasing to the user/buyer and without latent errors is an unexpectedly hard problem. It is perhaps the most difficult problem in engineering today, and has been recognized as such for more than 15 years. It is often referred to as the "software crisis". It has become the longest continuing "crisis" in the engineering world, and it continues unabated*".

While timely correct software construction is difficult, others have argued that one can hardly call it a crisis [Gla98] and have called the idea of a "software crisis" paradoxical. Over the years since the first computers, much software has been written, delivered, bought, and used. Glass presents a convincing argument [Gla06] that both the Standish *Chaos Report* [Sta94] and the *GAO Study*, as pointed out by Blum [Blu91], are biased towards *failed* software projects. Glass [Gla00], as editor of several journals, warned researchers about presenting their research as solely a solution towards the "software crisis". Given the sheer number of companies that write, sell, and use software, it is a very successful industry! While computers and software requirements have become more complex, computers and software have been successful enough to become almost ubiquitous.

Nonetheless, it is still difficult to construct correct software in a timely fashion as soon as requirements, or the software itself, becomes complex. Software incorrectness is often only discovered after it has been constructed. The correctness of software can be shown by proving or, to a certain degree, by testing. Testing properties of more than trivial programs is already difficult. Giving a (partial) correctness proof is much harder and often practically impossible. Software to aid human testers and provers in their approach to ascertaining software correctness exists. Unfortunately, that software is affected by the same complexity problems and is therefore often incorrect, incomplete, and still no match for human reasoning. The main advantage of such tools is that they force one to be both formal and complete in writing specifications and proofs.

It is our belief that functional programming languages might alleviate some of these issues at the source of software construction. A functional language can get the programmer off to a good start by providing him or her with a programming language that has very good support for abstraction. This should make it easier to construct better abstraction layers to bridge the aforementioned gap.

1.2 Abstraction by Functional Programming

Functional programming languages are based on the mathematical concept of functions. Like mathematical functions, functional programs consist of functions applied to values and other functions. Functional languages use a declarative style, specifying *what* the program should compute without being very specific about *how* it should be done. This declarative programming style does not require explicit flow–of–control constructions and the programmer leaves decisions about the order of evaluation to the compiler.

For a compiler to change the flow–of–control without changing the result of a program, all expressions in the language must be free of observable side-effects. Side-effect free functional languages are called pure. Languages that use a data-driven evaluation order are called lazy. To prevent recomputing the result of an expression each time it is demanded, an implementation of a lazy language replaces each unevaluated expression by its result when it is evaluated due to a demand. This is only valid if the replaced expression is referentially transparent, i.e., without (in)direct side-effects.

Pure and lazy functional languages guarantee referential transparency, which helps reasoning about programs. Reasoning about imperative programs is difficult because equational reasoning does not work (easily) due to side-effects, such as the assignment of values to variables, which is at the basis of imperative programming. The use of side-effects can cause the program to change parts of its state unexpectedly. These languages provide no protection against parts of a program that implicitly and unexpectedly change the behavior of other parts.

The result of a mathematical function depends only on its input, not on any other implicit or hidden state. The typical strategy for handling state and side-effects in a pure and lazy language is explicit state threading. Regardless whether this is done using monads or unique environment passing, it makes the state of a functional program explicit. Since this is the basic construct in functional languages, it forces the programmer to be explicit about state changes and protects the programmer (with the help of guaranteed referential transparency) from unexpected or inconsistent behavior of the program.

Referential transparency has additional merits for reasoning about functional programs. The declarative functional style and the ability to replace an expression with its result are ideal for equational reasoning. Furthermore, the recursive structure of functional programming is a natural match for the use of proof by induction. The guarantee that a function will always evaluate to the same result given the same arguments, means that unit tests only have to test each combination of argument values once.

Functional programming languages also feature advanced type systems. This allows programmers to express properties of functions using types, which are checked at compile time. Type-checking prevents certain run-time errors: “square pegs will not go into round holes”. Languages with a strong type system have the property that a type can be assigned to every (sub)expression in a language. Functional languages are structured such that automatic type inference can be used, which reduces the need for manually specifying type signatures. Since the type checker can statically prove type safety, there is no need for run-time type-checking.

The support for abstraction and composition in functional language manifests itself clearly in the use of functions as arguments and as result of other functions. Such higher-order functions are first-class citizens. Functional arguments are not like C pointers to functions; instead they can consist of any expression that has a function type (i.e., a type containing an arrow). This includes currying: applying a function to some, but not all, of its arguments.

Functional languages also support easy generalization of functions that work on the high-level structure (e.g., the branches of a tree) of its argument without inspecting the values inside that structure (e.g., at the leafs). For instance, mapping a function over a list requires the inspection of the structure of a list, not the elements of that list. The (inferred) type of the function becomes polymorphic in the type of the elements of the list, and it will automatically work on any list regardless of the type of its elements.

For functions that are almost polymorphic but still require some operations on the elements in which it would otherwise be polymorphic, the languages usually provide overloading. Such operations are declared as members of classes and become overloaded in those classes for types of those elements. This system is not very different from using interfaces (named type classes in functional languages) and implementations (instances of type classes) in Java. Overloaded functions have a polymorphic type with the additional requirement that an instance of the class in which they are overloaded exists. The type-checker can often, but not always, deduce the type for which the programmer must provide an instance of a certain class. Overloading is similar to higher-order parameterization of a function. The required operations are passed as argument-functions, except that this parameterization is implicit and the compiler provides the actual argument function based on the requested type.

The last advantage of functional languages worth mentioning is that the semantics of those languages are usually more clearly, and definitely more formally, defined than imperative languages. As most functional languages come from the academia, they often have a strong backing in formal, and well understood, theories such as λ-calculus and term– and graph–rewriting. Many languages and programming techniques have found their (experimental) origins in functional programming languages.

In the recent past, we have seen more and more imperative programming languages with automatic memory management (garbage collection). Functional language implementations have been using this for a long time. In combination with recursive data structures, automatic memory management can make direct use of pointers obsolete. Software quality ought to improve by this means, since explicit memory management is a common source of bugs in complex software. Currently, we see a tendency of programmers in imperative languages to search for, and try to use, a pure and functional subset of their favorite programming language. Unfortunately, those programming languages have no support for enforcing such a subset. The programmers, and the users of the software that they release, have to rely on their self-discipline to get the safety features that functional languages freely provide.

1.3 Hybrid Static/Dynamic Typing

Type checking is a program analysis used in many programming languages. Usually, programming languages are either statically (at compile-time) or dynamically (at run-time) typed. Static type checking informs the programmer early of type errors and guarantees that a program will not give any type error at run-time. Furthermore, knowing the types at compile-time can improve the efficiency of the generated code.

Statically typed (correct) programs are a subset of all (correct) programs. Obviously, there are limits to the ability of type checkers in functional languages to prove type correctness of (i.e., type check) a program. We have heard implementers of functional compiler say: "if we add one more feature to the type system it becomes undecidable", which shows that the static type systems are already very advanced and close to being programming languages on their own. For instance, it has been shown that the type checker of some functional language compilers, with all their bells-and-whistles enabled [Wan98, Doc06], are Turing powerful.

Functional programming languages featuring a strong type system do not even require the programmer to specify a type for all parts of the program. The type checker found in such compilers can infer most of the types automatically. This removes the need to specify all types while the benefit of having the types for all expressions statically checked remains. The functional language Clean features an interesting hybrid type system that combines the static as well as the dynamic approaches to type checking.

Clean [PvE02] is a pure and lazy functional programming language developed at the Radboud University Nijmegen by the Software Technology group of the Institute for Computing and Information Sciences. It is slightly different from other functional languages such as Haskell [Pey03], (O'Ca)ML, Scheme and Lisp (ordered from pure and lazy to impure and strict). Its approach is to incorporate side-effects using unique environment passing instead of monads. Furthermore, Clean is known for the efficiency of the code it generates and for the speediness of the compiler.

To enable the elegant use of typing at run-time in combination with static typing, Clean has a hybrid type system. Using only dynamic type-checking would have incurred a high run-time cost and would counter the aspiration of early (static) error detection. The hybrid type system does static checking as much as possible. The programmer must specify which checks are performed at run-time. However, this is done such that the static type checker can already assume some things about the type after a successful run-time check. This allows one to use (previously unknown) types at run-time, without invalidating the proof of the static type checker. The Clean run-time system performs the actual type-checking on the run-time values. A dynamic linker is part of this system and it can read (and write) any Clean expression from (and to) disk. Reading and writing functions is fully supported, since lazy functional expressions can contain closures, i.e., unevaluated (possibly cyclic or infinite) expressions with references to functions.

An additional benefit of Clean's hybrid type system is that it enables incorporation of new types and new functions at run-time. This makes it possible, in combination with the run-time type checks, to do type-safe communication of data (values) and code (functions) of any type between different Clean programs. It also provides the means to incorporate plug-ins in a type safe manner.

1.4 Polytypic Programming

Programming simple but useful functions, like pretty-printing or comparison operators, for each and every data type one uses, is boring and error prone. In certain specialized cases, programs are better than humans are at reliably generating correct programs, as long as they themselves are correct. In principle, it alleviates the work of the programmer by reducing the amount of code, which not only needs to be written, but also maintained. If the source from which the actual program is generated is 'simpler' than its result, it could in principle also simplify proving.

Polytypic functions reduce the number of lines of code, and hopefully the related number of errors, by requiring merely a generic specification for a small set of types in order to work on any type. Polytypic programming [JJ97, Hin00b], also known as generic programming, uses polytypic functions that have to be defined for only a handful of types, but can be applied to values of any type. Given such a (usually) terse function specification, compilers can derive the function for all types automatically. Recently, pure and lazy functional programming languages have been extended with support for polytypic function definitions. They are now a part of Clean [AP03] and for Haskell there is the Generic Haskell [LCJ03] preprocessor, which can be seen as the successor of PolyP [JJ97].

Defining a polytypic function can be described as specifying a function for all types by specifying it only for a few simple generic types (i.e., the set of canonical types: the unit, sum, and product type). Since all (algebraic) data types can be described in terms of products, sums and certain basic types, a compiler can generate function alternatives for all (algebraic) data types, given alternatives for the basic types. These basic types are types like numbers, functions, handles, and other types predefined and built-in the compiler. The programmer can overrule the generated function for a specific type at any time. This is useful if one wants a function with generic behavior over most types but only a certain specialized action on some of the types. This has been coined "programming the exception".

Deriving programs using polytypic functions is somewhat similar to generating code via templates (as found in C++), except that it is not only type directed but also type safe. There is no need for type-checking the generated code if the polytypic function definition type checks. This is an advantage over templates, which are type checked after each instantiation. Type errors in polytypic functions are detected in the polytypic code written by the programmer, in contrast to templates, where the error messages are usually about the instantiated code.

There are advantages to generating parts of a program automatically, and with guaranteed type safety, especially in a programs that contain many different data types or when the data types change a lot during the development. One can easily obtain the required behavior of a program by overruling the generated code for a small strategically chosen set of types. This could improve the speed and quality of both the design and maintenance of software.

1.5 Scope of the Research

When we started the research presented in this thesis, two new programming techniques where emerging in the pure and lazy functional programming community: hybrid static/dynamic typing and polytypic programming (also known as generic programming). Overviews of those techniques have been presented in Sect. 1.3

and Sect. 1.4, respectively. The hybrid static/dynamic typing is important becasue it extends the type safety of a statically typed functional language into the run-time world. The polytypic programming approach to writing software is attractive in its promise to reduce the amount of code that needs to be written, better maintenance by adding code instead of rewriting, and elegant abstract algorithms that work for any type.

To investigate the usefulness of hybrid static/dynamic typing and polytypic programming, we tested these techniques on some common software/programming problems. Since both were very new approaches, we wanted to see them "in action" to assess applicability, performance, and expressiveness of the programming techniques themselves as well as their implementations in both Clean and (Generic) Haskell. Any sensible definition of "common programming problems" yields a set too large to investigate in the time given for this thesis. We therefore limited ourselves to a handful of subjects that we believed could benefit from strong typing at run-time or program generation based on types.

1.6 Contents of this Thesis

The chapters of this work consist of peer-reviewed published articles. The chapters are mostly unmodified versions of the published work. This resulted in some redundancy, most notably in Sect. 2.2 and Sect. 3.2, which are identical. The second section has therefore been replaced by a reference to the first. Likewise, introductory sections of chapters using the same technique have some overlap. We preferred not to rewrite entire chapters, but rather keep them as close as possible to the original articles as they were accepted by the program committees of the various conferences, which makes each chapter self-contained. However, we did make changes to the layout and visual style of the papers (most notably code fragments) to improve consistency.

The (many) program fragments presented in this work are written in Haskell and Clean. We assume some knowledge of these languages, which are very similar, and functional programming in general. Syntactical and some other essential differences between Clean and Haskell will be explained mostly in Sect. 2.2, Sect. 5.2, and on the fly where appropriate.

Chapter 2:

The main component that provides means of communication in any computer is the operating system. Additionally, the operating system provides a way to start new programs. In this chapter, we present Famke: a prototype/proof–of–concept implementation of a strongly typed functional operating system. We incorporate strong dynamic typing to ensure that all communication between processes is type safe. Furthermore, (de)serialization is not required by virtue of Clean's dynamic linker, which enables the transportation of any strong statically typed (unevaluated) expression.

Famke enables the creation and management of independent distributed Clean processes on a network of workstations. It uses Clean's dynamic type system and its dynamic linker to communicate values of any type, e.g., data, closures, and functions (i.e., compiled code), between running applications in a type safe way. Mobile processes can be implemented using Famke's ability to communicate functions.

We have built a simple interactive shell on top of Famke that enables user interaction. The shell uses a functional-style command language that allows construction of new processes, and it type checks the command line before executing it. A much improved version of the shell, developed after the work in this chapter was complete, is described in Chap. 3.

This chapter is a slight adaptation of a peer-reviewed publication [vWP02]. It was written in collaboration with Rinus Plasmeijer, who supported me in extending the research of my Master's thesis and improved the comprehensibility of the paper. It received the Peter Landin award for best paper at IFL'02.

Chapter 3:

Esther is the interactive shell of Famke, a prototype implementation of a strongly-typed operating system written in the functional programming language Clean. As usual, the shell can be used for manipulating files, applications, data, and processes at the command line. A special feature of Esther is that the shell language provides the basic functionality of a strongly typed lazy functional language, at the command line. The shell type-checks each command line and only executes well-typed expressions. Files are also typed; applications and commands are simply files with a function type.

The type-checking/inference performed by the shell uses the hybrid static/dynamic type system of Clean. The shell behaves like an interpreter, but it actually executes a command line by combining existing compiled code of functions (and programs from disk). Clean's dynamic linker is used to store (and retrieve) any expression (both data and code) with its type on disk. This linker is also used to communicate values of any type, e.g., data, closures, and functions (i.e., compiled code) between running applications in a type safe way.

The shell combines the advantages of interpreters (direct response) and compilers (statically typed and fast code). Applications (compiled functions) can be used, in a type safe way, in the shell. Functions defined at the command line can be used by any compiled Clean application.

This chapter is a slight adaptation of peer-reviewed publications [vWP03, PvW04] written in collaboration with Rinus Plasmeijer, who supervised the research and helped to structure the papers.

Chapter 4:

The Esther shell (presented in Chap. 3) uses bracket abstraction to translate command line expressions to Clean's dynamics, which are then used to type check the expressions. Bracket abstraction is an algorithm that transforms lambda expressions into combinator terms. There are several versions of this algorithm depending on the actual set of combinators used. Most of them have been proven correct with respect to the operational semantics. In this chapter, we focus on typability and prove that the shell's approach to typing expressions (after bracket abstraction) is valid.

We present a fully machine-verified proof of the property that bracket abstraction preserves types; the types assigned to an expression before and after performing bracket abstraction are identical. To our knowledge, this is the first time that (1) such proof has been given, and (2) the proof is verified by a theorem prover. The theorem prover used in the development of the proof is PVS [OSRS01].

This chapter is a slight adaptation of a peer-reviewed publication [SvW06]. It is written in collaboration with Sjaak Smetsers who performed the tedious job of actually proving the theorems and who wrote the PVS part of the paper.

Chapter 5:

We have eased GUI programming using polytypic functional programming techniques. This was done by constructing a programming toolkit with which one can create GUIs in an abstract and compositional way using type-directed Graphical Editor Components (*GEC*s). In this toolkit, the programmer specifies a GUI by means of a data model instead of low-level GUI programming. In earlier versions of this toolkit, the data model must have a first-order type.

In this chapter, we show that the programming toolkit can be extended in two ways, such that the data model can contain higher-order data structures. We added support for dynamic polymorphic higher-order editors using the functional shell Esther. We also added statically typed, higher-order editors. In principle, this solution extends our GUI programming toolkit with the full expressive power of functional programming languages.

This chapter is a slight adaptation of peer-reviewed publications [AvEPvW04b, AvEPvW04c]. They were written in collaboration with Peter Achten, Marko van Eekelen, and Rinus Plasmeijer who together designed and implemented the *GEC* system, into which I incorporated the shell as a library.

Chapter 6:

Polytypic functional programming has the advantage that it can automatically derive code for generic functions. However, it is not yet clear whether it can be useful for anything other than the textbook examples. Furthermore, the generated polytypic code is usually too slow for real-life programs. As a real-life test, we derive a polytypic parser for the Haskell 98 syntax and look into other front-end syntax-tree operations commonly found in functional language compilers.

We present a types–as–grammar approach, which uses polytypic programming (in both Generic Haskell and Clean) to derive the code for a parser automatically based on the syntax tree type, without using external tools. Moreover, we show that using polytypic programming can even be useful for data–specific syntax tree operations, such as scope checking and type inference.

Simple speed tests show that the performance of polytypic parsers can be abominable for real-life inputs. However, we show that much performance can be recovered by applying (extended) fusion optimization on the generated code. We have actually produced a derived parser using this technique whose speed is close to that of one generated by a specialized Haskell parser generator.

This chapter is a slight adaptation of a peer-reviewed publication [vWSP05]. It was written in collaboration with Rinus Plasmeijer and Sjaak Smetsers. I initiated this topic of research and contributed in implementing the parser combinators and grammars in Clean and Haskell. Sjaak Smetsers applied the fusion technique that makes it a viable approach to parser generation.

Chapter 7:

Functional programming, in a polytypic way, reduces the amount of code that needs to be written manually. Functional programming also provides tremendous support for abstraction. Using an abstraction called *arrows*, one can abstract over the flow of data in a functional program. In this chapter, we combine polytypic programming with abstraction over the order of applications, in an attempt to reduce the code that needs to be written manually even further. We present a way to program invertible functions by specifying merely one direction of computation. We further generalize this by specifying them in a polytypic way to make them useful for any data type.

Invertible programming occurs in the area of data conversion where it is required that the conversion in one direction is the inverse of the other. For that purpose, we introduce bidirectional arrows (*bi-arrows*). The bi-arrow class is an extension of Haskell's arrow class with an extra combinator that changes the direction of computation. The advantage of the use of bi-arrows for invertible programming is the automatic preservation of invertibility properties by using only invertible combinators. Programming with bi-arrows in a polytypic way exploits this aspect the most. Besides bidirectional polytypic examples, including invertible serialization, we define a monadic bi-arrow transformer, which we use to construct a bidirectional parser/pretty printer.

This chapter is a slight adaptation of a peer-reviewed publication [vWSP05]. It was written in collaboration with Artem Alimarine, Sjaak Smetsers, Marko van Eekelen, and Rinus Plasmeijer. My contribution consisted mainly of designing, implementing, and proving the polytypic invertible bi-arrows and writing the paper together with Sjaak Smetsers and Marko van Eekelen, based on the preliminary research of Artem Alimarine and Rinus Plasmeijer.

Chapter 8:

This chapter presents a case study on the use of formal methods in specification-based, black-box testing of a smart card applet. The system under test is a simple electronic purse application running on a Java Card platform. The applet's specification is given as a State Chart model, and transformed into a functional form to serve as the input for GAST, an on–the–fly test generation, execution, and analysis tool. The GAST tool is an automated test system that uses systematic polytypic derivation of test cases. We show that it can also be used for testing properties of an imperative program running on a (simulated) Java card. We show that automated, formal, specification-based testing of smart card applets is of high value, and that errors can be detected using this model-based testing.

This chapter is a slight adaptation of a peer-reviewed publication [vWOF$^+$05], written in collaboration with Martijn Oostdijk, Lars Frantzen, Pieter Koopman, and Jan Tretmans. My contribution mainly consisted of writing the Clean specification of the purse, extending the test system to enable it to interface with the Java applet, performing all tests, and presenting the results in the paper.

Chapter 9:

We look back upon the research presented in Chap. 9 and mention some related and future work that came up later in Sect. 9.1. The results of, and lessons learned during the research presented in this thesis are concluded in Sect. 9.2.

CHAPTER 2

Towards a Strongly Typed Functional Operating System

ARJEN VAN WEELDEN[*]
RINUS PLASMEIJER

2.1 Introduction

Functional programming languages like Haskell [Pey03] and Clean [PvE02] offer a very flexible and powerful static type system. Compact, reusable, and readable programs can be written in these languages while the static type system is able to detect many programming errors at compile time. However, this works only within a single application.

Independently developed applications often need to communicate with each other. One would like the communication of objects to take place in a type safe manner as well. Not only communication of simple objects, but objects of any type, including functions. In practice, this is not easy to realize. The compile-time type information is generally not kept inside a compiled executable, and therefore cannot be used at run-time. In real life therefore, applications often only communicate simple data types like streams of characters, ASCII text, or use some ad-hoc defined (binary) format. Although more and more applications use XML to communicate data together with the definitions of the data types used, most programs do not support run-time type unification, cannot use previously unknown data types or cannot exchange functions (i.e., code) between different

[*]This work was supported by STW as part of project NWI.4411.

programs in a type safe way. This is mainly because the used programming language has no support for such things.

In this chapter, we present a prototype implementation of a micro kernel, called *Famke* (*f*unction*a*l *m*icro *k*ernel *e*xperiment). It provides explicit non-deterministic concurrency and type safe message passing for all types to processes written in Clean. By adding servers that provide common operating system services, an entire strongly typed, distributed operating system can be built on top of Famke.

Clearly, we need a powerful dynamic type system [ACPP91] for this purpose and a way to extend a running application with new code. Fortunately, the new Clean system offers some of the required basic facilities: it offers a hybrid type system with static as well as dynamic typing (*dynamics*) [Pil97, Pil99], including run-time support for *dynamic linking* [VP03] (currently on Microsoft Windows only). To achieve type safe communication, Famke uses the above-mentioned facilities offered by Clean to implement *lightweight threads*, *processes*, *exception handling* and *type safe message passing* without requiring additional language constructs or run-time support.

It also makes use of an underlying operating system to avoid some low-level implementation work and to integrate better with existing software (e.g., resources such as the console and the file system). With Famke, we want to accomplish the following objectives without changing the Clean compiler or run-time system.

- Present an interface (API) for Clean programmers with which it is easy to create (distributed) processes that can communicate expressions of any type in a type safe way;
- Present an interactive shell with which it is easy to manage, apply and combine (distributed) processes, and even construct new processes interactively. The shell should type check the command line before executing it in order to catch errors early;
- Achieve a modular design using an extensible micro kernel approach;
- Achieve a reliable system by using static types where possible and, if static checking cannot be done (e.g., between different programs), dynamic type checks;
- Achieve a system that is easy to port to another operating system (if the Clean system supports it).

We will introduce the static/dynamic hybrid type system of Clean in Sect. 2.2. Sections 2.3 and 2.4 present the micro kernel of Famke, which provides cooperative thread scheduling, exception handling, and type safe communication. It also provides an interface to the preemptively scheduled processes of the underlying operating system. These sections are very technical, but necessary to understand the interesting sections that follow. On top of this micro kernel, an interactive shell has been implemented, which we describe in Sect. 2.5. The crucial role of dynamics will become apparent in these sections. Related work is discussed in Sect. 2.6, and we conclude and mention future research in Sect. 2.7.

2.2 Dynamics in Clean

This section gives an introduction into Clean's hybrid static/dynamic type system. We show how any statically typed expression in Clean can be converted into a statically typed object called *Dynamic*, which contains both the original expression and its static type. Dynamic objects can be written to disk and read by any other Clean program. Clean's dynamic linker takes care of incorporating any necessary code referenced by closures (unevaluated expressions) in a Dynamic.

Clean offers a hybrid type system: in addition to its static type system, it also has a (polymorphic) dynamic type system [ACPP91, ACP+92, Pil99, VP03]. This allows programs to use (statically) untyped values, so called *dynamics*. A dynamic in Clean is a value of static type *Dynamic*, which contains an expression as well as a representation of the (static) type of that expression.

Dynamics can be formed (i.e., lifted from the static to the dynamic type system) using the special constructor **dynamic** applied to a value and, optionally, its type. The type will be inferred if it is omitted[1].

```
dynamic 42 :: Int[2]
dynamic map fst :: A[3].a b: [(a, b)] → [a]
```

Function alternatives and case patterns can pattern match on (run-time) values of (the static) type `Dynamic`, i.e., bring them from the dynamic back into the static type system. Such pattern matches consist of a value pattern and a type pattern. In the example below, `matchInt` returns `Just` the value contained inside the dynamic if it has type `Int`; and `Nothing` if it has any other type. The pattern matches on types are translated into run-time type unifications. If the unification fails, the next alternative is tried, as in a common (value) pattern match.

```
::[4]Maybe a = Nothing | Just a
```

[1]Types containing universally quantified variables are currently not inferred by the compiler. We will sometimes omit these types for ease of presentation.

[2]Numerical denotations are not overloaded in Clean.

[3]Clean's syntax for the `forall` extension of Haskell.

[4]Defines a new data type in Clean, Haskell uses the `data` keyword.

```
matchInt :: Dynamic → Maybe Int
matchInt (x :: Int) = Just x
matchInt  other     = Nothing
```

A type pattern can contain type variables that are bound to the offered type, if the run-time unification succeeds. In the example shown below, `dynamicApply` tests whether the argument type of the function `f` inside the first argument of `dynamicApply` can be unified with the type of the value `x` inside the second argument. If this is the case then `dynamicApply` can safely apply `f` to `x`. The type variables `a` and `b` will be instantiated by the run-time unification. At compile time it is generally unknown what type `a` and `b` will be, but if the type pattern match succeeds, we can assume that applying `f` to `x` is type safe. This yields a value with the type that is bound to `b` by unification, which is wrapped in a dynamic.

```
dynamicApply :: Dynamic Dynamic → Dynamic[5]
dynamicApply (f :: a → b) (x :: a) = dynamic f x :: b[6]
dynamicApply df dx = dynamic "Error: cannot apply"
```

Type variables in dynamic patterns can also relate to a type variable in the static type of a function. Such functions are called type dependent functions [ACPP91]. A caret (^) behind a variable in a pattern associates it with the type variable with the same name in the static type of the function. The static type variable then becomes overloaded in the predefined `TC` (or type code) class. The `TC` class is used to 'carry' the type representation. The compiler automatically generates instances for this class, which contain the necessary methods to convert values to dynamics and vice versa. In the example below, the static type variable `t` will be determined by the (static) context in which it is used, and will impose a restriction on the actual type that is accepted at run-time by `matchDynamic`. It yields `Just` the value inside the dynamic (if the dynamic contains a value of the required context dependent type) or `Nothing` (if it does not).

```
matchDynamic :: Dynamic → Maybe t | TC t[7]
matchDynamic (x :: t^) = Just x
matchDynamic  other    = Nothing
```

[5]Clean separates argument types by whitespace, instead of →.

[6]The type *b* is also inferred by the compiler.

[7]Clean uses a | to announce context restrictions. In Haskell this would be written as `(TC t) ⇒ Dynamic → Maybe t`.

Reading and Writing of Dynamics

The dynamic run-time system of Clean supports writing dynamics to disk and reading them back again, possibly in another application or during another execution of the same application. This is not a trivial feature, since Clean is not an interpreted language: it uses compiled code. Since a dynamic may contain unevaluated functions, reading a dynamic implies that the corresponding code produced by the compiler has to be added to the code of the running application. To make this possible one needs a dynamic linker. Furthermore, one needs to be able to retrieve the type definitions and function definitions associated with a stored dynamic. The need for function definitions comes from lazy evaluation, which can introduce closures in values that refer to functions and their code. With the ability to read and write dynamics, type safe plug-ins can be realized in Clean relatively easily.

A dynamic can be written to a file on disk using the `writeDynamic` function.

```
writeDynamic :: String Dynamic *[8]World → (Bool, *World)
```

In the `producer` example below a dynamic is created which consists of the application of the function `sieve` to an infinite list of integers. This dynamic is then written to file using the `writeDynamic` function. Evaluation of a dynamic is done lazily. The `producer` does not demand the result of the application of `sieve` to the infinite list. Therefore, the application is written to file in its unevaluated form. The file therefore contains a calculation that will yield a potential infinite integer list of prime numbers.

```
producer :: *World → *World
producer world
    = writeDynamic "primes" (dynamic sieve [2..]) world
where
    sieve :: [Int] → [Int]
    sieve [prime:rest] = [prime:sieve filter]
    where
        filter = [h \\ h <- rest | h mod prime ≠ 0]
```

When the dynamic is stored to disk, not only the dynamic expression and its type has to be stored somewhere. To allow the dynamic to be used as a plug-in by any other application additional information has to be stored as well. One also has to store:

[8]This is a uniqueness attribute, indicating that the world environment is passed around in a single threaded way. Unique values allow safe destructive updates and are used for I/O in Clean. The value of type `World` corresponds to the hidden state of the `IO` monad in Haskell.

- the code corresponding to the function definitions that are referred to in closures inside the dynamic;

- the definitions of all the types involved needed to check type consistency when matching on the type of the dynamic in another Clean program.

The required code and type information will be generated by the compiler and is stored in a special database when an application is compiled and linked. For the detail of the bookkeeping of the code database, we refer to [VP03]. The code and type information is created and stored once at compile-time, while the dynamic value and dynamic type can be created and stored several times at run-time. The run-time system has to be able to find both kinds of information when a dynamic is read in.

A dynamic can be read from disk using the `readDynamic` function.

```
readDynamic :: String *World → (Bool, Dynamic, *World)
```

This `readDynamic` function is used in the `consumer` example below to read the earlier stored dynamic. The dynamic pattern match checks whether the dynamic expression is an integer list. In case of success the first 100 elements are taken. In case that the read in dynamic is not of the indicated type, the consumer aborts. Actually, it is not possible to do something with a read-in dynamic (besides passing it around to other functions or saving it to disk again), unless the dynamic matches some type or type scheme specified in the pattern match of the receiving application.

```
consumer :: *World → [Int]
consumer world
    #[9] (ok, dyn, world) = readDynamic "primes" world
    = take 100 (extract dyn)
where
    extract :: Dynamic → [Int]
    extract (list :: [Int]) = list
    extract other = abort "dynamic type check failed"
```

To turn a dynamically typed expression into a statically typed expression, the following steps are performed by the run-time system of Clean:

- The type of the dynamic and the type specified in the pattern are unified with each other. If the unification fails, the dynamic pattern match also fails.

[9]Clean uses environment passing, instead of monads, for side effects. It supports let-before (#) to increase readability.

- If the unification is successful, it is checked that the type definitions of types with the same name coming from different applications are equal. If one of the involved type definitions differs, the dynamic pattern match fails. Types are equivalent if and only if their type definitions are syntactically the same (modulo alpha-conversion and the order of algebraic data constructors).
- If all patterns match, the corresponding function alternative is chosen and evaluated.
- It is possible that for the evaluation of the, now statically typed, expression parts of its representation on disk are required. In that case, the expression is reconstructed out of the information stored in the dynamic on disk. The corresponding code needed for the evaluation of the expression is added to the running application, after which the expression can be evaluated.

Running `prog1` and `prog2` in the example below will write a function and a value to dynamics on disk. Running `prog3` will create a new dynamic on disk that contains the result of ‘applying’ (using the `dynamicApply` function) the dynamic with the name “function” to the dynamic with the name “value”. The closure `40 + 2` will not be evaluated until the `*` operator needs it. In this case, because the ‘dynamic application’ of `df` to `dx` is lazy, the closure will not be evaluated until the value of the dynamic on disk named “result” is needed. Running `prog4` tries to match the dynamic `dr`, from the file named “result”, with the type `Int`. After this succeeds, it displays the value by evaluating the expression, which is semantically equal to `let x = 40 + 2 in x * x`, yielding `1764`.

```
prog1 world = writeDynamic "function" (dynamic (*)) world

prog2 world = writeDynamic "value" (dynamic 40 + 2) world

prog3 world =
    let (ok1, df, world1) = readDynamic "function" world
        (ok2, dx, world2) = readDynamic "value" world1
    in writeDynamic "result" (dynamicApply df dx) world2

prog4 world =
    let (ok, dr, world1) = readDynamic "result" world
    in (case dr of (x :: Int) → x, world1)
```

A dynamic will be read in lazily only after successful run-time unification (triggered by a pattern match on the dynamic). The dynamic linker will take care of the actual linking of the code to the running application and the checking of the type definitions referenced by the dynamic being read. The dynamic linker is able to find the code and type definitions in the database in which they are stored at compile time. The amount of data and code that the dynamic linker will link depends on how far the dynamic expression is evaluated.

Dynamics written by one application program can safely be read by any other Clean application. Dynamics can always be read by both programs, but type pattern matches will only succeed when the type definitions used in the Dynamic match those in the program. The reading program is run-time extended with the definitions of the new types and the code of new functions found in the dynamic. Known types and constructors in the dynamic are mapped to the corresponding types and constructors in the program. Therefore, two Clean applications can communicate values of any type they like, including function types, in a type safe manner.

2.3 Threads in Famke

Here we show how a programmer can construct concurrent programs in Clean, using Famke's thread management and exception handling primitives.

Currently, Clean offers only very limited library support for process management and communication.

Old versions of Concurrent Clean [NSvEP91] did offer sophisticated support for parallel evaluation and lightweight processes, but no support for exception handling. Concurrent Clean was targeted at deterministic, implicit concurrency, but we want to build a system for distributed, non-deterministic, explicit concurrency.

Porting Concurrent Clean to Microsoft Windows is a lot of work, and still would not give us exactly what we want. Although Microsoft Windows offers threads to enable multi-tasking within a single process, there is no run-time support for making use of these preemptive threads in Clean. We could emulate threads using the preemptive processes that Microsoft Windows provides by multiple incarnations of the same Clean program, but this would make the threads unacceptably heavyweight, and it would prevent them from sharing the Clean heap, and we still would not have exception handling.

Therefore, Famke does her own scheduling of threads in order to keep them lightweight and to provide exception handling.

2.3.1 Thread Implementation

In order to implement cooperative threads we need a way to suspend running computations and to resume them later. Wand [Wan80] shows that this can be done using continuations and the call/CC construct offered by Scheme and other functional programming languages. We copy this approach using first class continuations in Clean. Because Clean has no call/CC construction, we have to write the continuation passing explicitly. Our approach closely resembles Claessen's concurrency monad [Cla99], but our primitives operate directly on the kernel state using Clean's uniqueness typing, and we have extended the implementation with easily extendable exception handling (see Sect. 2.3.2).

```
:: Thread a :== (a → KernelOp) → KernelOp[10]
:: KernelOp :== Kernel → Kernel

threadExample :: Thread a
threadExample = λcont kernel → cont x kernel'
where
  x = ···                        //[11] calculate argument for cont
  kernel' = ··· kernel ···       // operate on the kernel state
```

A function of the type `Thread`, such as the example function above, gets the tail of a computation (named `cont`; of type `a → KernelOp`) as its argument and combines that with a new computation step, which calculates the argument (named `x`) for the tail computation, to form a new function (of type `KernelOp`). This function returns, when evaluated on a kernel state (named `kernel`; of type `Kernel`), a new kernel state.

```
:: ThreadId    // abstract thread id

:: *Kernel[12] = {currentId  :: ThreadId, newId :: ThreadId,
                  ready :: [ThreadState], world :: *World}

:: ThreadState = {thrId :: ThreadId, thrCont :: KernelOp}

:: Void = Void    // written more elegantly as () in Haskell
```

The kernel state (of type `Kernel`) is a record that contains the information required to do the scheduling of the threads. It contains information like the current running thread (named `currentId`), the threads that are ready to be scheduled (in the `ready` list), and the `world` state which is provided by the Clean run-time system. Clean's uniqueness type system makes these types a little more complicated, but we will not show this in the examples in order to keep them readable.

[10]Clean uses `:==` to indicate a type synonym, whereas Haskell uses the `type` keyword.

[11]This is a single line comment in Clean, Haskell uses `--`

[12]Record types in Clean are surrounded by { and }. The `*` before `Kernel` indicates that the record must always be unique. Therefore, the `*` can then be omitted in the rest of the code.

```
newThread :: (Thread a) → Thread ThreadId
newThread thread = λcont k=:{newId, ready}[13] → cont newId
  {k & newId = inc newId, ready = [threadState:ready]}[14]
where
  threadState = {thrId = newId, thrCont = thread (λ_ k → k)}

threadId :: Thread ThreadId
threadId = λcont k=:{currentId} → cont currentId k
```

The `newThread` function starts the given thread concurrently with the other threads. Threads are evaluated for their effect on the kernel and the world state. They therefore do not return a result, hence the polymorphically parameterized `Thread a` type. It relieves our system from the additional complexity of returning the result to the parent thread. The communication primitives that will be introduced later enable programmers to extend the `newThread` primitive to deliver a result to the parent. Threads can obtain their thread identification with `threadId`.

Scheduling of the threads is done cooperatively. This means that threads must occasionally allow rescheduling using `yield`, and should not run endless tight loops. The `schedule` function then evaluates the next ready thread. `StartFamke` can be used like the standard Clean `Start` function to start the evaluation of the main thread.

```
yield :: KernelOp Kernel → Kernel
yield cont k=:{currentId, ready}
    = {k & ready = ready ++ [threadState]}
where
    threadState = {thrId = currentId, thrCont = cont}

schedule :: Kernel → Kernel
schedule k=:{ready = []} = k    // nothing to schedule
schedule k=:{ready = [{thrId, thrCont}:tail]} =
    let k`  = {k & ready = tail, currentId = thrId}
        k`` = thrCont k`    // evaluate the thread until it yields
    in schedule k``

StartFamke :: (Thread a) *World → *World
StartFamke mainThread world = (schedule kernel).world
where
```

[13]`r=:{f}` denotes the (lazy) selection of the field `f` in the record `r`. `r=:{f = v}` denotes the pattern match of the field `f` on the value `v`.

[14]`{r & f = v}` denotes a new record value that is equal to `r` except for the field `f`, which is equal to `v`.

```
firstId = ···    // first thread id
kernel = {currentId = firstId, newId = inc firstId,
          ready = [threadState], world = world}
threadState = {thrId=firstId, thrCont=mainThread (λ_ k → k)}
```

The thread that is currently being evaluated returns directly to the scheduler whenever it performs a `yield` action, because `yield` does not evaluate the tail of the computation. Instead, it stores the continuation at the back of the ready queue (to achieve round-robin scheduling) and returns the current kernel state. The scheduler then uses this new kernel state to evaluate the next ready thread.

Programming threads using a continuation style is cumbersome, because one has to carry the continuation along and one has to perform an explicit yield often. Therefore, we added thread-combinators resembling a more common monadic programming style. Our `return`, `>>=` and `>>` functions resemble the monadic `return`, `>>=` and `>>` functions of Haskell[15]. Whenever a running thread performs an atomic action, such as a return, control is voluntarily given to the scheduler using `yield`.

```
return :: a → Thread a
return x = λcont k → yield (cont x) k

(>>=) :: (Thread a) (a → Thread b) → Thread b
(>>=) l r = λcont k → l (λx → r x cont) k

(>>) l r = l >>= λ_ → r

combinatorExample = newThread (print ['hello']) >>
                    print ['world']
where
    print []     = return Void
    print [c:cs] = printChar c >> print cs
```

The `combinatorExample` above starts a thread that prints ”`hello`” concurrent with the main thread that prints ”`world`”. It assumes a low-level print routine `printChar` that prints a single character. The output of both threads is interleaved by the scheduler, and is printed as ”`hweolrllod`”.

[15]Unfortunately, Clean does not support Haskell's do-notation for monads, which would make the code even more readable.

2.3.2 Exceptions and Signals

Thread operations (e.g., `newThread`) may fail because of external conditions such as the behavior of other threads or operating system errors. Robust programs quickly become cluttered with lots of error checking code. An elegant solution for this kind of problem is the use of exception handling.

There is no exception handling mechanism in Clean, but our thread continuations can easily be extended to handle exceptions. Because of this, exceptions can only be thrown or caught by a thread. This is analogous to Haskell's `ioError` and `catch` functions, with which exceptions can only be caught in the `IO` monad.

In contrast to Haskell exceptions, we do not want to limit the set of exceptions to system defined exceptions and strings, but instead allow any value. Exceptions are therefore implemented using dynamics. This makes it possible to store any value in an exception and to extend the set of exceptions at compile-time or even at run-time. To provide this kind of exception handling, we extend the `Thread` type with a continuation argument for the case that an exception is thrown.

```
:: Thread a :== (SucCnt a) → ExcCnt → KernelOp
:: SucCnt a :== a → ExcCnt → KernelOp
:: ExcCnt   :== Exception → KernelOp

:: Exception :== Dynamic

throw :: e → Thread a | TC e
throw e = λsc ec k → ec (dynamic e :: e^) k

rethrow :: Exception → Thread a
rethrow exception = λsc ec k → ec exception k

try :: (Thread a) (Exception → Thread a) → Thread a
try thread catcher =
  λsc ec k → thread (λx _ → sc x ec) (λe → catcher e sc ec) k
```

The `throw` function wraps a value in a dynamic (hence the `TC` context restriction) and throws it to the enclosing `try` clause by evaluating the exception continuation (`ec`). `rethrow` can be used to throw an exception without wrapping it in a dynamic again. The `try` function catches exceptions that occur during the evaluation of its first argument (`thread`) and feeds it to its second argument (`catcher`). Because any value can be thrown, exception handlers must match against the type of the exception using dynamic type pattern matching.

The kernel provides an outermost exception handler (not shown here) that aborts the thread when an exception remains uncaught. This exception handler informs the programmer that an exception was not caught by any of the handlers, and shows the type of the occurring exception.

```
return :: a → Thread a
return x = λsc ec k → yield (sc x ec) k

(>>=) :: (Thread a) (a → Thread b) → Thread b
(>>=) l r = λsc ec k → l (λx → r x sc) ec k
```

The addition of an exception continuation to the thread type also requires small changes in the implementation of the `return` and `bind` functions. Note how the `return` and `throw` functions complement each other: `return` evaluates the success continuation while `throw` evaluates the exception continuation. This implementation of exception handling is relatively cheap, because there is no need to test if an exception occurred at every bind or return. The only overhead caused by our exception handling mechanism is the need to carry the exception continuation along with it.

```
:: ArithErrors = DivByZero | Overflow

exceptionExample = try (divide 42 0) handler

divide x 0 = throw DivByZero
divide x y = return (x / y)

handler (DivByZero :: ArithErrors) = return 0   // or any other value
handler  other                     = rethrow other
```

The `divide` function in the example throws the value `DivByZero` as an exception when the programmer tries to divide by zero. Exceptions caught in the body of the `try` clause are handled by `handler`, which returns zero on a `DivByZero` exception. Caught exceptions of any other type are thrown again outside the try, using `rethrow`.

In a distributed or concurrent setting, there is also a need for throwing and catching exceptions between different threads. We call this kind of inter-thread exceptions *signals*. Signals allow threads to throw kill requests to other threads. Our approach to signals, or *asynchronous exceptions* as they are also called, follows the semantics described by Marlow et al. in an extension of Concurrent Haskell [MPMR01]. We summarize our interface for signals below.

```
throwTo    :: ThreadId e → Thread Void | TC e
signalsOn  :: (Thread a) → Thread a
signalsOff :: (Thread a) → Thread a
```

Signals are transferred from one thread to the other by the scheduler. A signal becomes an exception again when it arrives at the designated thread, and can therefore be caught in the same way as other exceptions. To prevent interruption by signals, threads can enclose operations in a `signalsOff` clause, during which signals are queued until they can interrupt. Regardless of any nesting, `signalsOn` always means interruptible and `signalsOff` always means non-interruptible. It is therefore always clear whether program code can or cannot be interrupted. This allows easy composition and nesting of program fragments that use these functions. When a signal is caught, control goes to the exception handler and the interruptible state will be restored to the state before entering the try.

The try construction allows elegant error handling. Unfortunately, there is no automated support for identifying the exceptions that can be thrown by a function. This is partly because exception handling is written in Clean and not built in the language/compiler, and partly because exceptions are wrapped in dynamics and can therefore not be expressed in the type of a function. Furthermore, exceptions of any type can be thrown by any thread, which makes it hard to be sure that all (relevant) exceptions are caught by the programmer. However, the same can be said for an implementation that uses user-defined strings, in which non-matching strings are also not detected at compile-time.

2.4 Processes in Famke

In this section, we will show how a programmer can execute groups of threads using processes on multiple workstations, to construct distributed programs in Clean.

Famke uses Microsoft Windows processes to provide preemptive task switching between groups of threads running inside different processes. Once processes have been created on one or more computers, threads can be started in any one of them. We start by introducing Famke's message passing primitives for communication between threads and processes. The dynamic linker plays an essential role in getting the code of a thread from one process to another.

2.4.1 Process and Thread Communication

Elegant ways for type-safe communication between threads are Concurrent Haskell's M-Vars [PGF96] and Concurrent Clean's lazy graph copying [NSvEP91]. Unfortunately, M-Vars do not scale very well to a distributed setting because of two problems, described by Stolz and Huch in [SH01]. The first problem is that M-Vars require distributed garbage collection because they are first class objects, which is hard in a distributed or mobile setting. The second problem is that the location of the M-Var is generally unknown, which complicates reasoning about them in the context of failing or moving processes. Automatic lazy graph copying allows processes to work on objects that are distributed over multiple (remote) heaps, and suffers from the same two problems.

Distributed Haskell [HN01] solves the problem by implementing an asynchronous message-passing system using ports. Famke uses the same kind of ports. Ports in Famke are channels that vanish as soon as they are closed by a thread, or when the process containing the creating thread dies. Accessing a closed port yields an exception. Using ports as the means of communication, it is always clear where a port resides (at the process of the creating thread) and when it is closed (explicitly or because the process died). In contrast with Distributed Haskell, we do not limit ports to a single reader (which could be checked at compile-time using Clean's uniqueness typing). The single reader restriction also implies that the port vanishes when the reader vanishes but we find it too restrictive in practice.

```
:: PortId msg   // abstract port id
:: PortExceptions = UnregisteredPort | InvalidMessageAtPort | ···

newPort   :: Thread (PortId msg)         | TC msg
closePort :: (PortId msg) → Thread Void | TC msg

writePort :: (PortId msg) msg → Thread Void | TC msg
writePort port m
    = windowsSend port (dynamicToString (dynamic m :: msg^))

readPort :: (PortId msg) → Thread msg | TC msg
readPort port = windowsReceive port >>= λmaybe →
                case maybe of
                   Just s  → case stringToDynamic s of
                                (True, (m :: msg^)) → return m
                                _ → throw InvalidMessageAtPort
                   Nothing → readPort port   // make it appear blocking

registerPort :: (PortId msg) String → Thread Void | TC msg
lookupPort   :: String → Thread (PortId msg)      | TC msg

dynamicToString :: Dynamic → String
stringToDynamic :: String → (Bool, Dynamic)
```

All primitives on ports operate on typed messages. The `newPort` function creates a new port and `closePort` removes a port. `writePort` and `readPort` can be used to send and receive messages. The dynamic run-time system is used to convert the messages to and from a dynamic. Because we do not want to read and write files each time we want to send a message to someone, we will use the low-level `dynamicToString` and `stringToDynamic` functions from the dynamic run-time system library. These functions are similar to Haskell's `show` and `read`, except that they can (de)serialize functions and closures. They should be handled with care, because they allow you to distinguish between objects that should be indistinguishable (e.g., between a closure and its value). The actual sending and receiving of these strings is done via simple message (string) passing primitives of the underlying operating system. The `registerPort` function associates a unique name with a port, by which the port can be looked up using `lookupPort`.

Although Distributed Haskell and Famke both use ports, our system is capable of sending and receiving functions (and therefore closures) using Clean's dynamic linker. The dynamic type system also allows programs to receive, through ports of type `(PortId Dynamic)`, previously unknown data structures, which can be used by polymorphic functions or functions that work on dynamics such as the `dynamicApply` functions in Sect. 2.2. An asynchronous message passing system, such as presented here, allows programmers to build other communication and synchronization methods (e.g., remote procedure calls, semaphores and channels).

Here is a skeleton example of a database server that uses a port to receive functions from clients and applies them to the database.

```
:: DBase = ···    // list of records or something like that

server :: Thread Void
server = openPort >>= λport →
         registerPort port "MyDBase" >>
         handleRequests emptyDBase
where
  emptyDBase = ···  // create new database
  handleRequests db = readPort port >>= λf →
                      let db‘ = f db in   // apply function to database
                      handleRequests db‘

client :: Thread Void
client = lookupPort "MyDBase" >>= λport →
         writePort port mutateDatabase
where
  mutateDatabase :: DBase → DBase
  mutateDatabase db = ···    // change the database
```

The server creates, and registers, a port that receives functions of the type `DBase → DBase`. Clients send functions that perform changes to the database to the registered port.

The server then waits for functions to arrive and applies them to the database `db`. These functions can be safely applied to the database because the dynamic run-time system guarantees that both the server and the client have the same notion of the type of the database (`DBase`), even if they reside in different programs. This is also an example of a running program that is dynamically extended with new code.

2.4.2 Process Management

Since Microsoft Windows does preemptive scheduling of processes, our scheduler does not need any knowledge about multiple processes. Instead of changing the scheduler, we let our system automatically add an additional thread, called the *management thread*, to each process upon creation. This management thread is used to handle signals from other processes and to route them to the designated threads. On request from threads running at other processes, it also handles the creation of new threads inside its own process. This management thread, in combination with the scheduler and the port implementation, forms the micro kernel that is included in each process.

```
:: ProcId                  // abstract process id
:: Location :== String

newProc      :: Location → Thread ProcId
newThreadAt  :: ProcId (Thread a) → Thread ThreadId
```

The `newProc` function creates a new process at a given location and returns its process id. The creation of a new process is implemented by starting a pre-compiled Clean executable, the *loader*, which becomes the new process. The loader is a simple Clean program that starts a management thread. The function `newThreadAt` starts a new thread in another process. The thread is started inside the new process by sending it to the management thread at the given process id via a typed port. When the management thread receives the new thread, it starts it using the local `newThread` function. The dynamic linker on the remote computer then links in the code of the new thread automatically.

Below you will see an example of starting a thread at a remote process and getting the result back to the parent. The `remote` function creates a port to which the result of the given thread must be sent. It then starts a child thread at the remote location `pid` that calculates the result and writes it to the port, and returns the port enclosed in a `Remote` node to the parent. When the parent decides that it wants the result, it can use `join` to get it and to close the port.

```
:: *Remote a = Remote (PortId a)
```

```
remote :: ProcId (Thread a) → Thread (Remote a) | TC a
remote pid thread =
    newPort >>= λport →
    newThreadAt pid (thread >>= writePort port) >>
    return (Remote port)

join :: (Remote a) → Thread a | TC a
join (Remote port) = readPort port >>= λresult →
                     closePort port >>
                     return result
```

The extension of our system with this kind of heavyweight process enables the programmer to build distributed concurrent applications. If one wants to run Clean programs that contain parallel algorithms on a farm of workstations, this is a first step. However, non-trivial changes are required to the original program to accomplish this. These changes include splitting the program code into separate threads and making communication between the threads explicit. The need for these changes is unfortunate, but our system was primarily designed for explicit distributed programs (and eventually mobile programs), not to speedup existing programs by running them on multiple processors.

This concludes our discussion of the micro kernel and its interface that provides support for threads (with exceptions and signals), processes and type-safe communication of values of any type between them. Now it is time to present the first application that makes use of these strongly typed concurrency primitives.

2.5 Interacting with Famke: the Shell

In this section, we introduce our shell that enables programmers to construct new (concurrent) programs interactively.

A shell provides a way to interact with an operating system, usually via a textual command line/console interface. Normally, a shell does not provide a complete programming language, but it does enable users to start pre-compiled programs. Although most shells provide simple ways to combine multiple programs, e.g., pipelining and concurrent execution, and support execution-flow controls, e.g., if-then-else constructs, they do not provide a way to construct new programs. Furthermore, they provide very limited error checking before executing the given command line. This is mainly because the programs mentioned at the command line are practically untyped because they work on, and produce, streams of characters. The intended meaning of these streams of characters varies from one program to the other.

Our view on pre-compiled programs differs from common operating systems in that they are dynamics that contain a typed function, and not untyped executables. Programs are therefore typed and our shell puts this information to good use by actually type-checking the command line before performing the specified actions. For example, it could test if a printing program (:: `WordDocument` → `PostScript`) matches a document (:: `WordDocument`).

The shell supports function application, variables, and a subset of Clean's constant denotations. The shell syntax closely resembles Haskell's do-notation, extended with operations to read and write files.

Here follow some command line examples with an explanation of how they are handled by the shell.

```
> map (add 1) [1..10]
```

The names `map` and `add` are unbound (do not appear in the left hand side of a let of lambda expression) in this example and our shell therefore assumes that they are names of files (dynamics on disk). All files are supposed to contain dynamics, which together represent a typed file system. The shell reads them in from disk, practically extending its functionality with these functions, and inspects the types of the dynamics. It uses the types of `map` (let us assume that the file `map` contains the type that we expect: `(a → b) [a] → [b]`), `add` (let us assume: `Int Int → Int`) and the list comprehension (which has type: `[Int]`) to type-check the command line. If this succeeds, which it should given the types above, the shell applies the partial application of `add` with the integer one to the list of integers from one to ten, using the `map` function. The application of one dynamic to another is done using the `dynamicApply` function from Sect. 2.2, extended with better error reporting. With the help of the `dynamicApply` function, the shell constructs a new function that performs the computation `map (add 1) [1..10]`. This function uses the compiled code of `map`, `add`, and the list comprehension. Our shell is a hybrid interpreter/compiler, where the command line is interpreted/compiled to a function that is almost as efficient as the same function written directly in Clean and compiled to native code. Dynamics are read in before executing the command line, so it is not possible to change the meaning of a part of the command line by overwriting a dynamic.

```
> inc <- add 1; map inc [2,4..10]
```

Defines a variable with the name `inc` as the partial application of the `add` function to the integer one. Then it applies the `map` function using the variable `inc` to the list of even integers from two to ten. The dynamic linker detects that `map` and `add` are already linked in, and reuses their code.

```
> inc <- add 1; map inc ['a'..'z']
```

Defines the variable `inc` as in the previous example, but applies it, using the `map` function, to the list of all the characters in the alphabet. This obviously fails with the usual type error: `Cannot unify [Int] with [Char]`.

```
> write "result" (add 1 2); x <- read "result"; x
> add 1 2 > result; x < result; x
```

Both the above examples do the same thing, because the `<` (read file) and `>` (write file) shell operators can be expressed using predefined `read` and `write` functions. The sum of one and two is written to the file with the name `result`. The variable `x` is defined as the contents of the file with the name `result`, and the result of the command line is the contents of the variable `x`. In contrast to the `add` and `map` functions, which are read from disk by the shell before type-checking and executing the command line, `result` is read in during the execution of the command line.

```
> newThread server;
> p <- lookupPort "MyDBase"; writePort p (insertDBase MyRecord)
```

The first line in the example above creates a new thread that executes the `server` from Sect. 2.4.1. Let us assume that we have two dynamics on disk: one with the name `insertDBase` containing a function that can insert a record into a database, and one with the name `MyRecord` containing a record for the database. In the second line, we get the port of the server by looking it up using the name MyDBase. We send the function `insertDBase` applied to `MyRecord` to the server by writing the closure to the port. This example shows how we can interactively communicate with threads in a type safe way.

2.6 Related Work

There are concurrent versions of both Haskell and Clean. Concurrent Haskell [PGF96] offers lightweight threads in a single UNIX process and provides M-Vars as the means of communication between threads. Concurrent Clean [NSvEP91] is only available on multiprocessor Transputers and on a network of single-processor Apple Macintosh computers. Concurrent Clean provides support for native threads on Transputer systems. On a network of Apple computers, it ran the same Clean program on each processor, providing a virtual multiprocessor system. Concurrent Clean provided lazy graph copying as the primary communication mechanism. Both concurrent systems cannot easily provide type safety between different programs or between multiple incarnations of a single program.

Another difference between Famke and the concurrent versions of Haskell and Clean is the choice of communication primitives. Neither lazy graph copying nor

M-Vars scale very well to a distributed setting because they require distributed garbage collection. This issue has led to a distributed version of Concurrent Haskell [HN01] that also uses ports. However, its implementation does not allow functions or closures to be sent over ports, because it cannot serialize functions. Support for this could be provided by a dynamic linker for Concurrent Haskell.

Both Cooper [CM90] and Lin [Lin98] have extended Standard ML with threads (implemented as continuations using call/CC) to form a small functional operating system. Both systems implement the basics needed for a stand-alone operating system. However, none of them supports the type-safe communication of any value between different computers.

Erlang [AVWW96] is a functional language specifically designed for the development of concurrent processes. It is completely dynamically typed and primarily uses interpreted byte-code, while Famke is mostly statically typed and executes native code generated by the Clean compiler. A simple spelling error in a token used during communication between two processes is often not detected by Erlang's dynamic type system, sometimes causing deadlock.

Back et al. [BTS$^+$98] built two prototypes of a Java operating system. Although they show that Java's extensibility, portable byte code and static/dynamic type system provides a way to build an operating system where multiple Java programs can safely run concurrently, Java lacks the power of polymorphic and higher-order functions and closures (to allow laziness) that our functional approach offers.

Haskell provides exception handling without changing its pure and lazy foundation. In [MPMR01] support for asynchronous exceptions has been added to Concurrent Haskell. Our implementation of signals closely follows their approach.

The Scheme Shell [Shi94] integrates a shell into the programming language in order to enable the user to use the full expressiveness of Scheme. Es [HR93] is a shell that supports higher-order functions and allows the user to construct new functions at the command line. Neither shell provides a way to read and write typed objects from and to disk, not do they provide type safety, because they operate on untyped executables.

2.7 Conclusions and Future Work

In this chapter, we presented the basics of our prototype functional operating system called Famke. Famke is written entirely in Clean and provides lightweight threads, exceptions and heavyweight processes, and a type safe communication mechanism, using Clean's dynamic type system and dynamic linking support. Furthermore, we have built an interactive shell that type checks the command line before executing it. With the help of these mechanisms, it becomes feasible to build distributed concurrent Clean programs running on a network. Programs can easily be extended with new code at run-time using the dynamic run-time system of Clean.

We can extend our kernel in a modular way by putting all extensions in separate dynamics, which would allow us to tailor our system (at run-time) to a given situation. Nevertheless, there remain issues that need further research.

We would like to give the programmer more information about what exceptions a function may throw. Unfortunately, we have not yet found a way to do this without compromising the flexibility of our approach.

The implementation of ports given in this chapter does not check if the name is unique (when registering) or even exists (when looking up), entrusting this responsibility upon the programmer. Fortunately, this situation will be detected at run-time because it causes an exception at the receiving end. We intend to repair it in a more mature implementation.

The current focus of further research on Famke is to increase the power and usability of the shell.

CHAPTER 3

A Functional Shell that Operates on Typed and Compiled Applications

ARJEN VAN WEELDEN[*]
RINUS PLASMEIJER

3.1 Introduction

Functional programming languages like Haskell [Pey03] and Clean [PvE02] offer a very flexible and powerful static type system. Compact, reusable, and readable programs can be written in these languages while the static type system is able to detect many programming errors at compile time. However, this works only within a single application.

Independently developed applications often need to communicate with each other. One would like the communication of objects to take place in a type safe manner as well. Not only communication of simple objects, but objects of any type, including functions. In practice, this is not easy to realize. The compile-time type information is generally not available to the compiled executable at run-time. In real life therefore, applications often only communicate simple data types like streams of characters, ASCII text, or use some ad-hoc defined (binary) format.

Programming languages, especially pure and lazy functional languages like Clean and Haskell, provide good support for abstraction (e.g., subroutines, overloading, polymorphic functions), composition (e.g., application, module system,

[*]Part of this work was supported by InterNLnet.

higher-order functions), and verification (e.g., strong type-checking and inference). In contrast, command line languages used by operating system shells usually have little support for abstraction, composition, and especially verification. They do not provide higher-order subroutines, complex data structures, type inference, or type-checking *before* evaluation. Given their limited set of types and their specific area of application (in which they have been, and still are, very successful), this is not experienced as a serious problem.

Nonetheless, we think that command line languages can benefit from some of the programming language facilities, as this will increase their flexibility, reusability, and security. We have previously done research on reducing run-time errors (e.g., memory access violations, type errors) in operating systems by implementing a micro kernel in Clean that provides type safe communication of any value of any type between functional processes, called Famke (*F*unction*A*l *M*icro *K*ernel *E*xperiment) [vWP02]. This has shown that (moderate) use of dynamic typing [ACPP91], in combination with Clean's dynamic run-time system and dynamic linker [Pil99, VP03], enables processes to communicate any data (and even code) of any type in a type safe way.

During the development of a shell/command line interface for our prototype functional operating system it became clear that a normal shell cannot really make use (at run-time) of the type information derived by the compiler (at compile-time). To reduce the possibility of run-time errors during execution of scripts or command lines, we need a shell that supports abstraction and verification (i.e., type-checking) in the same way as the Clean compiler does. In order to do this, we need a better integration of compile-time (i.e., static typing) and run-time (i.e., interactivity) concepts.

Both the shell and micro kernel are built on top of Clean's hybrid static/dynamic type system and its dynamic I/O run-time support. It allows programmers to save any Clean expression, i.e., a graph that can contain data, references to functions, and closures, to disk. Clean expressions can be written to disk as a *dynamic*, which contains a representation of their (polymorphic) static type, while preserving sharing. Clean programs can load dynamics from disk and use run-time type pattern matching to reintegrate it into the statically typed program. In this way, new functionality (e.g., plug-ins) can be added to a running program in a type safe way. This chapter stresses type safety and assumes that we can trust the compiler.

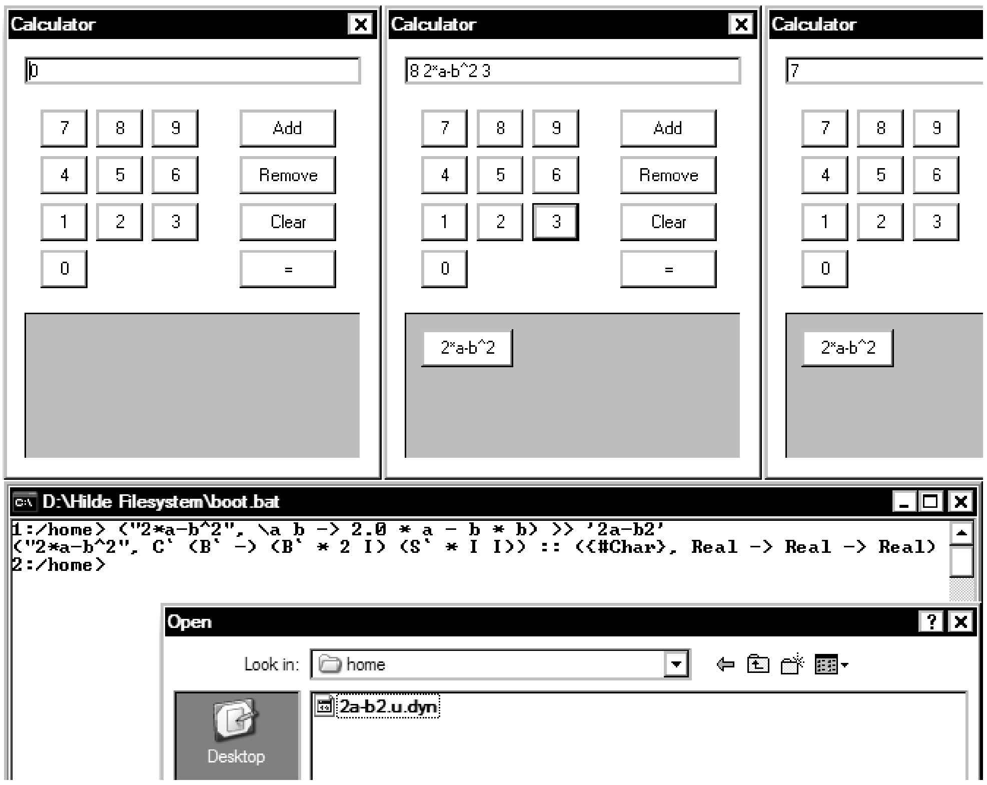
The shell is called Esther (*E*xtensible *S*hell with *T*ype c*H*ecking *E*xpe*R*iment), and is capable of:

- reading an expression from the console, using Clean's syntax for a basic, but complete, functional language. It offers application, lambda abstraction, recursive let, pattern matching, function definitions, and even overloading;

- using compiled Clean programs as typed functions at the command line;
- defining new functions, which can be used by other compiled Clean programs (without using the shell or an interpreter);
- extracting type information (and indirectly, code) from dynamics on disk;
- type-checking the expression and resolving overloading, *before* evaluation;
- constructing a new dynamic containing the correct type and code of the expression.

3.1.1 Esther Example: an Application Uses a Shell Function

Figure 3.1 shows a sequence of screenshots of a calculator program written in Clean. Initially, the calculator has no function buttons. Instead, it has buttons to add and remove function buttons. These will be loaded dynamically after adding dynamics that contain tuples of `String` and `Real Real` → `Real`.



```
1:/home> ("2*a-b^2", \a b -> 2.0 * a - b * b) >> '2a-b2'
("2*a-b^2", C` (B` -> (B` * 2 I) (S` * I I)) :: ({#Char}, Real -> Real -> Real)
2:/home>
```

Figure 3.1: A combined screenshot of the calculator and Esther in action.

The lower half of Fig. 3.1 shows a command line in the Esther shell that writes such a tuple as a dynamic named “2a-b2.u.dyn” to disk. Its button name is `2*a-b^2` and the function is `λa b → 2.0*a - b*b`. Pressing the `Add` button on the calculator opens a file selection dialog, shown at the bottom of Fig. 3.1. After selecting the dynamic named “2a-2b.u.dyn”, it becomes available in the calculator as the button `2*a-b^2`, and it is applied to `8` and `3` yielding `7`.

The calculator itself is a separately compiled Clean executable that runs without using Esther. Alternatively, one can write the calculator, which has the type `[(String, Real Real → Real)] *World → *World`, as a dynamic to disk. The calculator can then be started from Esther, either in the current shell or as a separate process.

3.1.2 Overview

First, we introduce the static/dynamic hybrid type system and dynamic I/O of Clean in Sect. 3.2. The type-checking and combining of compile code features of Esther are directly derived from Clean’s dynamic implementation. In Sect. 3.3, we give an overview of the expressive power of the shell command language using tiny examples of commands that can be given. In Sect. 3.4 we show how to construct a dynamic for each kind of sub-expression such that it has the correct semantics and type, and how to compose them in a type checked way. Related work is discussed in Sect. 3.5 and we conclude and mention future research in Sect. 3.6. We assume that reader is familiar with Haskell, and will indicate syntactic difference with Clean in footnotes. The implementation has been done in Clean because it has more support for (de)serializing dynamics than Haskell. Unfortunately, Clean’s dynamic linker, which is required for Esther, has only been implemented for Microsoft Windows. The implementation, which is reasonably stable but always under development, can be downloaded from the author’s web site[1].

3.2 Dynamics in Clean

The reader is kindly referred to Sect. 2.2 of the previous chapter.

3.3 Overview of the Shell

Like any other shell, our Esther shell enables users to start pre-compiled programs, provides simple ways to combine multiple programs, e.g., pipelining and concur-

[1]`http://www.cs.ru.nl/A.vanWeelden/`

rent execution, and supports execution-flow controls, e.g., if-then-else constructs. It provides a way to interact with the underlying operating system and the file system, using a textual command line/console interface.

A special feature of the Esther shell is that it offers a complete typed functional programming language with which programs can be constructed. The shell type checks a command line before performing any actions. Traditional shells provide very limited error checking before executing the given command line. This is mainly because the applications mentioned at the command line are practically untyped because they work on, and produce, streams of characters. The intended meaning of these streams of characters varies from one program to the other. The choice to make our shell language typed also has consequences for the underlying operating system and file system: they should be able to deal with types as well.

In this section, we give a brief overview of the functionality of the Esther shell and the underlying operating system and file system it relies on.

3.3.1 Famke: a Type Safe Micro Kernel

A shell has to be able to start applications and to provide a way to connect applications (e.g., by creating a pipeline) such that they can communicate. Since our shell is typed, process communication should be type safe as well. The Windows Operating System that we use does not provide such a facility. We therefore have created a micro kernel on top of Windows. Our micro-kernel, Famke, provides Clean programs with ways to start new (possibly distributed running) processes, and the ability to communicate any value in a type safe way. It should be no surprise that Famke uses dynamics for this purpose. Dynamics can be sent between applications as strings (see [VP03]), which makes it possible to use conventional interprocess communication system such as TCP/IP for the actual communication.

3.3.2 A Typed File System

A shell works on applications and data stored on disk. Our shell is typed; it can only work if all files it operates on are typed as well. We therefore assume that all files have a proper type.

For applications written in Clean this can be easily realized. Any data value, function, or even any large complete Clean application (which is a function as well) can be written as dynamic to disk, thus forming a rudimentary typed file system.

Applications written in other languages are usually untyped. We can in principle incorporate such an application into in our typed file system, by writing a properly typed Clean wrapper application around it, which is then stored again as dynamic on disk. We could also wrap them automatically, or via a function, and give them the type `String String` → `String` (commandline, stdin → stdout). Obviously, this type does not really help type-checking and is therefore not implemented in this prototype that promotes type-checking.

We assume that all documents and compiled applications are stored in a dynamic of appropriate type. Applications in our file system are just dynamics that contain a function type. This typed file system makes it possible for the shell to ensure (given an ideal world where all programs are stored as dynamics) that it is type safe to apply a printing program (`print` :: `WordDocument` → `PostScript`) to a document (`myDocument` :: `WordDocument`). The Clean dynamic type system will ensure that the types will indeed fit. This is, of course, not a very realistic example, and it is for illustration purposes only.

Normal directory manipulation operations aside, one no longer reads bytes from a file. Instead, one reads whole files (only conceptually, the dynamic linker reads it lazily), and one can pattern match on the dynamic to check the type. This removes the need for explicit (de)serialization, as data structures are stored directly as graphs in dynamics. Serialization, parsing, and printing are often significant parts of existing software (up to thirty percent), which may be reduces by providing these operations in the programming language and/or operating system.

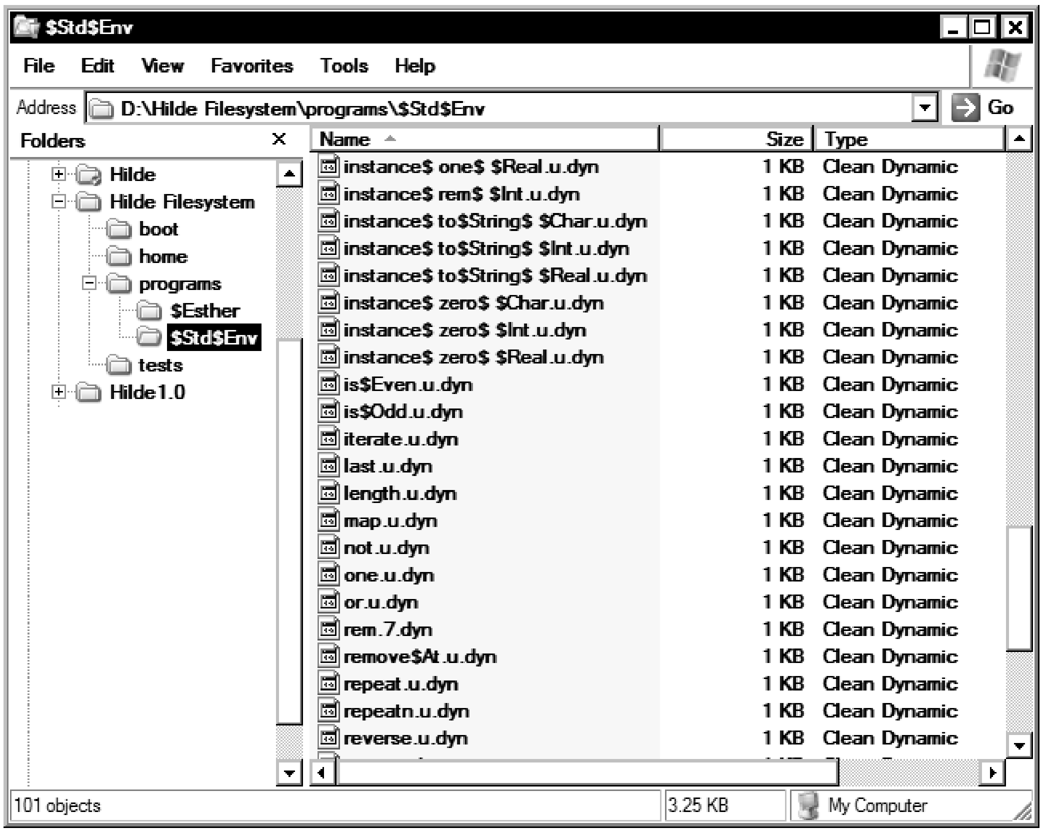
The shell contains no built-in commands. The commands it knows are determined by the files (dynamics) stored on disk. To find a command, the shell searches its directories in a specific order as defined in its search paths, looking for a file with that name.

The shell is therefore useless unless a collection of useful dynamics has been stored. When the system is initialized, a standard file system is created (see Fig. 3.2) in a Windows folder. It contains:

- almost all functions from the Clean standard environment[2], such as `+`, `-`, `map`, and `foldr` (stored as dynamic on disk);
- common commands to manipulated the file system (`mkdir`, `rmdir`, and the like);
- commands to create processes directly based on the functionality offered by Famke (`famkeNewProcess`, and the like).

All folders are common Window folders, all files contain dynamics created by Clean applications using the `writeDynamic`-function. The implementation of dynamics on disk is organized in such a way ([VP03]) that a user can safely rename, copy or delete files, either using the Esther shell or directly using Windows.

[2]Similar to Haskell's Prelude.



D:\Hilde Filesystem\programs\$Std$Env

Name	Size	Type
instance$ one$ $Real.u.dyn	1 KB	Clean Dynamic
instance$ rem$ $Int.u.dyn	1 KB	Clean Dynamic
instance$ to$String$ $Char.u.dyn	1 KB	Clean Dynamic
instance$ to$String$ $Int.u.dyn	1 KB	Clean Dynamic
instance$ to$String$ $Real.u.dyn	1 KB	Clean Dynamic
instance$ zero$ $Char.u.dyn	1 KB	Clean Dynamic
instance$ zero$ $Int.u.dyn	1 KB	Clean Dynamic
instance$ zero$ $Real.u.dyn	1 KB	Clean Dynamic
is$Even.u.dyn	1 KB	Clean Dynamic
is$Odd.u.dyn	1 KB	Clean Dynamic
iterate.u.dyn	1 KB	Clean Dynamic
last.u.dyn	1 KB	Clean Dynamic
length.u.dyn	1 KB	Clean Dynamic
map.u.dyn	1 KB	Clean Dynamic
not.u.dyn	1 KB	Clean Dynamic
one.u.dyn	1 KB	Clean Dynamic
or.u.dyn	1 KB	Clean Dynamic
rem.7.dyn	1 KB	Clean Dynamic
remove$At.u.dyn	1 KB	Clean Dynamic
repeat.u.dyn	1 KB	Clean Dynamic
repeatn.u.dyn	1 KB	Clean Dynamic
reverse.u.dyn	1 KB	Clean Dynamic

Figure 3.2: A screenshot of the typed file system; implemented as dynamic on disk.

3.3.3 Esther: a Type-Checking Shell

The last example of Sect. 3.2 shows how one can store and retrieve values, expressions, and functions of any type to and from the file system. It also shows that the `dynamicApply` function can be used to type check an application at run-time using the static types stored in dynamics. Combining both in an interactive ‘read expression – apply dynamics – evaluate and show result’ loop gives a very simple shell that already supports the type checked run-time application of programs to documents. For maximum flexibility, the shell contains almost no built-in functions. Any Clean function can be saved to disk using dynamics, and can thus be used by Esther.

Esther performs the following steps in a loop:

- it reads a string from the console and parses it like a Clean expression. It supports Clean’s basic and predefined types, application, infix operators, lambda abstraction, functions, overloading, let(rec), and case expressions;

- identifiers that are not bound by a lambda abstraction, a let(rec), or a case pattern are assumed to be names of dynamics on disk, and they are read from disk;
- type checks the expression using dynamic run-time unification and type pattern matching, which also infers types;
- if the command expression does not contain type errors, Esther displays the result of the expression and the inferred type. Esther will automatically be extended with any code necessary to display the result (which requires evaluation) by the dynamic linker.

For instance, if the user types in the following expression:

```
> map ((+) 1) [1..10]
```

the shell reacts as follows:

```
[2,3,4,5,6,7,8,9,10,11] :: [Int]
```

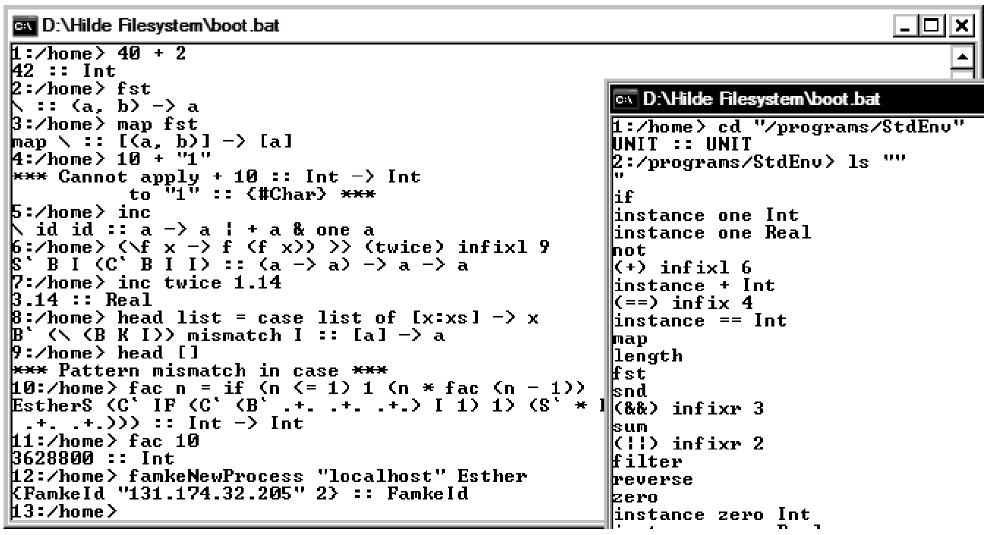
Roughly, the following happens. The shell parses the expression. The expression consists of typical Clean-like syntactical constructs (such as `(`, `)`, and `[ .. ]`), constants (such as `1` and `10`), and identifiers (such as `map` and `+`).

The names `map` and `+` are unbound (do not appear in the left hand side of a let, case, lambda expression, or function definition) in this example, and the shell therefore assumes that they are names of dynamics on disk. They are read from disk (with help of `readDynamic`), practically extending its functionality with these functions, and inspects the types of the dynamics. It uses the types of `map` (let us assume that the file `map` contains the type that we expect: `(a → b) [a] → [b]`), `+` (for simplicity, let us assume: `Int Int → Int`) and the list comprehension (which has type: `[Int]`) to type-check the command line. If this succeeds, which it should given the types above, the shell applies the partial application of `+` with the integer one to the list of integers from one to ten, using the `map` function. The application of one dynamic to another is done using the `dynamicApply` function from Sect. 3.2, extended with better error reporting. How this is done exactly, is explained in more detail in Sect. 3.4. With the help of the `dynamicApply` function, the shell constructs a new function that performs the computation `map ((+) 1) [1..10]`. This function uses the compiled code of `map`, `+`, and `[ .. ]`, which is implemented as a generator function called `_from_to` in Clean.

Our shell can therefore be regarded as a hybrid interpreter/compiler, where the command line is interpreted/compiled to a function that is almost as efficient as the same function written directly in Clean and compiled to native code. If functions, such as `map` and `+`, are used in other commands later on, the dynamic linker will notice that they are already have been used and linked in, and it will reuse their code. Consequently, the shell will react even quicker, because no dynamic linking is required anymore in such a case. For more details on Clean's dynamic linker, we refer to Vervoort and Plasmeijer [VP03].

3.3.4 The Esther Command Language

Here follow some command line examples with an explanation of how they are handled by the shell. Figure 3.3 show two example sessions with Esther. The right Esther window in Fig. 3.3 shows the same directory as the Windows Explorer window in Fig. 3.2. We explain Esther's syntax by example below. Like a common UNIX shell, the Esther shell prompts the user with something like `1:/home>` for typing in a new command. For readability we use only `>` in the examples below.



```
1:/home> 40 + 2
42 :: Int
2:/home> fst
\ :: (a, b) -> a
3:/home> map fst
map \ :: [(a, b)] -> [a]
4:/home> 10 + "1"
*** Cannot apply + 10 :: Int -> Int
         to "1" :: {#Char} ***
5:/home> inc
\ id id :: a -> a | + a & one a
6:/home> (\f x -> f (f x)) >> (twice) infixl 9
S` B I (C` B I I) :: (a -> a) -> a -> a
7:/home> inc twice 1.14
3.14 :: Real
8:/home> head list = case list of [x:xs] -> x
B` (\ (B K I)) mismatch I :: [a] -> a
9:/home> head []
*** Pattern mismatch in case ***
10:/home> fac n = if (n <= 1) 1 (n * fac (n - 1))
EstherS (C` IF (C` (B` .+. .+. .+.) I 1) 1) (S` * I
 .+. .+.))) :: Int -> Int
11:/home> fac 10
3628800 :: Int
12:/home> famkeNewProcess "localhost" Esther
{FamkeId "131.174.32.205" 2} :: FamkeId
13:/home>
```

```
1:/home> cd "/programs/StdEnv"
UNIT :: UNIT
2:/programs/StdEnv> ls ""
"
if
instance one Int
instance one Real
not
(+) infixl 6
instance + Int
(==) infix 4
instance == Int
map
length
fst
snd
(&&) infixr 3
sum
(||) infixr 2
filter
reverse
zero
instance zero Int
```

Figure 3.3: A combined screenshot of the two concurrent sessions with Esther.

Expressions

Here are some more examples of expressions that speak for themselves. Application:

```
> map
map :: (a -> b) [a] -> [b]
```

Expressions that contain type errors:

```
> 40 + "5"
*** cannot apply + 40 :: Int -> Int to  "5" :: {#Char} ***
```

Saving Expressions to Disk

Expressions can be stored as dynamics on disk using `>>`:

```
> 2 >> two
2 :: Int
> two
2 :: Int

> (+) 1 >> inc
+ 1 :: Int -> Int
> inc 41
42 :: Int
```

The `>>` operator is introduced mostly for convenience. Most expressions of the form `expr >> name` can be written as:

`writeDynamic` ”name” (**`dynamic`** `expr) world`

Unfortunately, it does not work to specify the `>>` operator as

`(>>) expr name` = `writeDynamic` ”name” (**`dynamic`** `expr)`

because this can require a rank-2 type that neither Esther, nor the Clean's compiler, can infer. Furthermore, `>>` uses `writeDynamic`, which is a function with side-effects and therefore it has type `*World` → `(Bool, *World)`. Functions of this type must be treated specially by Esther, which must pass them the world environment[3]. By defining the `>>` operator at the syntax level, we circumvent these problems.

Overloading

Esther resolves overloading in almost the same way as Clean. It is currently not possible to define new classes at the command line, but they can be introduced using a simple Clean program that stores the class as an overloaded function. It is also not possible to save overloaded command-line expressions using the `>>` described above. Arithmetic operations are overloaded in Esther, just as they are in Clean:

[3]Execute them in the `IO` monad.

```
> +
+ :: a a -> a | + a

> one
one :: a | one a

> (+) one
(+) one :: a -> a | + a & one a
```

Function Definitions

One can define new functions at the command line:

```
> dec x = x - 1
dec :: Int -> Int
```

This defines a new function with the name `dec`. This function is written to disk in a file with the same name (”dec”) such that from now on it can be used in other expressions.

```
> fac n = if (n < 2) 1 (n * fac (dec n))
S (C` IF (C` < I 2) 1) (S` * I (B (S .+. .+.) (C` .+. .+. .+.))
) :: Int -> Int
```

The factorial function is constructed by Esther using combinators (see Sect. 3.4), which explains why Esther responds in this way. The internals of the function shown by Esther proved useful while debugging Esther itself, but this may change in the future.

Functions cannot only be reused within the shell itself, but also used by any other Clean program. Such a function is a dynamic and can be used (read in, dynamically linked, copied, renamed, communicated across a network) as usual.

Notice that dynamics are read in before executing the command line, so it is not possible to change the meaning of a part of the command line by overwriting a dynamic.

Lambda Expressions

It is possible to define lambda expressions, just as in Clean.

```
> (\f x -> f (f x)) ((+) 1) 0
2 :: Int

> (\x x -> x) "first-x" "second-x"
"second-x" :: String
```

Esther parses the example above as: λx → (λx → x). This is not standard, and may change in the future.

Let Expressions

To introduce sharing and to construct both cyclic and infinite data structures, one can use let expressions.

```
> let x = 4 * 11 in x + x
88 :: Int

> let ones = [1:ones] in take 10 ones
[1,1,1,1,1,1,1,1,1,1] :: [Int]
```

Case Expressions

It is possible to do a simple pattern match using case expressions. Nested patterns are not yet supported, but one can always nest case expressions by hand. An exception `Pattern mismatch in case` is raised if a case fails.

```
> hd list = case list of [x:xs] -> x
B' (\ (B K I)) mismatch I :: [a] -> a

> hd [1..]
1 :: Int

> hd []
*** Pattern mismatch in case ***

> sum l = case l of [x:xs] -> x + sum xs; [] -> 0
B' (\ (C' (B' .+.) I (B .+. .+.))) (\ 0 mismatch) I
:: [Int] -> Int
```

The interpreter understands Clean denotations for basic types like `Int`, `Real`, `Char`, `String`, `Bool`, tuples, and lists. How can one perform a pattern match on a user-defined constructor defined in some application? It is not (yet) possible to define new types in the shell itself. However, one can define the types in any Clean module, and construct an application writing the constructors as dynamic to disk.

```
module IntroduceNewType
```

```
:: Tree a = Node (Tree a) (Tree a) | Leaf a

Start world
    # (ok, world) = writeDynamic "Node"
        (dynamic Node :: ∀a: (Tree a) (Tree a) → Tree a) world
    # (ok, world) = writeDynamic "Leaf"
                    (dynamic Leaf :: ∀a: a → Tree a) world
    # (ok, world) = writeDynamic "myTree"
                    (dynamic Node (Leaf 1) (Leaf 2)) world
    = world
```

These constructors can then be used by the shell to pattern match on a value of that type.

```
> leftmost tree = case tree of Leaf x -> x; Node l r ->
leftmost l
leftmost :: (Tree a) -> a

> leftmost (Node (Node myTree myTree) myTree)
1 :: Int
```

Typical Shell Commands

Esther's search path also contains a directory with common shell commands, such a file system operations:

```
> mkdir "foo"
UNIT :: UNIT
```

Esther displays `UNIT` because `mkdir` has type `World → World`, i.e., has a side effect, but no result. Functions that operate on the Clean's `World` state are applied to the world by Esther.

More operations on the file system:

```
> cd "foo"
UNIT :: UNIT

> 42 >> bar
42 :: Int

> ls ""
" bar " :: {#Char}
```

Processes

Examples of process handling commands:

```
> famkeNewProcess "localhost" Esther
{FamkeId "131.174.32.197" 2} :: FamkeId
```

This starts a new, concurrent, incarnation of Esther at the same computer. IP addresses can be used to start processes on other computers. `famkeNewProcess` yields a new process id (of type `FamkeId`). It is necessary to have the Famke kernel running on the other computer, e.g., by starting a shell there, to be able to start a process on another machine. Starting Esther on another machine does not give remote access. However, the console of the new incarnation of Esther is displayed and accessible on the other machine.

3.4 Implementation of Esther Using Dynamics

In this section, we explain how one can use the type unification of Clean's dynamic run-time system to type check a shell command, and we show how the corresponding Clean expression is translated effectively using combinations of already existing compiled code.

Obviously, we could have implemented type-checking ourselves using one of the common algorithms involving building and solving a list of type equations. Another option would be to use the Clean compiler to do the type-checking and compilation of the command line expressions. Instead, we decided to use Clean's dynamic run-time unification, because we want to show the power of Clean's dynamics, and because this has several advantages:

- Clean's dynamics allow us to do type safe and lazy I/O of expressions;
- we do not need to convert between the (hidden) type representation used by dynamics and the type representation used by our type-checking algorithm;
- it shows whether Clean's current dynamics interface is powerful enough to implement basic type inference and type-checking;
- we get future improvements of Clean's dynamics interface for free, e.g., uniqueness attributes or overloading.

The parsing of a shell command line is trivial and we will assume here that the string has already been successfully parsed.

In order to support a basic, but complete, functional language in our shell we need to support function definitions, lambda, let(rec), and case expressions.

We will introduce the syntax tree constructors piecewise and show for each kind of expression how to construct a dynamic that contains the corresponding Clean expression and the type for that expression. Names occurring free in the command line are read from disk as dynamics before type-checking. The expression can contain references other dynamics, and therefore references the compiled code of functions, which will be automatically linked by Clean's run-time system.

3.4.1 Application

Suppose we have a syntax tree for constant values and function applications that looks like:

```
:: Expr = (@) infixl 9[4] Expr Expr   //[5] Application
        | Value Dynamic               // Constant or dynamic value from disk
```

We introduce a function `compose`, which constructs the dynamic containing a value with the correct type that, when evaluated, will yield the result of the given expression.

```
compose :: Expr → Dynamic
compose (Value d) = d
compose  (f @ x)  = case (compose f, compose x) of
      (f :: a → b, x :: a) → dynamic f x :: b
      (df, dx)             → raise[6] ("Cannot apply " +++ typeOf df
                                         +++ " to " +++ typeOf dx)
typeOf :: Dynamic → String
typeOf dyn = toString (typecodeOfDynamic dyn) // pretty print type
```

Composing a constant value, contained in a dynamic, is trivial. Composing an application of one expression to another is a lot like the `dynamicApply` function of Sect. 3.2. Most importantly, we added error reporting using the `typeOf` function for pretty printing the type of a value inside a dynamic.

In an application, both dynamic arguments contain references to compiled code. If we write the resulting **dynamic** `(f x)` to disk, it contains references to the compiled code of `f` and `x`. Furthermore, we added a reference to the code of Clean's application. The resulting dynamic is self-contained in the sense that the shell is not required to execute the code inside the dynamic. When the resulting dynamic is used in another Clean program, the referenced code is linked into the running program. In essence, we have combined existing code to form

[4] This defines an infix constructor with priority 9 that is left associative.

[5] This a Clean comment to end–of–line, like Haskell's `--`.

[6] For easier error reporting, we implemented imprecise user-defined exceptions à la Haskell [PRH+99]. We used dynamics to make the set of exceptions extensible.

the new code of the resulting expression. This shows that the resulting code is not interpreted, but actually compiled via Clean's dynamics, even if not all conversions to dynamics (more are presented later) are as efficient one expects from a real compiler.

3.4.2 Lambda Expressions

Next, we extend the syntax tree with lambda expressions and variables.

```
:: Expr = ···                            // Previous def.
        | (~>) infixr 0 Expr Expr        // Lambda abstraction: λ .. → ..
        | Var String                     // Variable
        | S | K | I                      // Combinators
```

At first sight, it looks as if we could simply replace a ~> constructor in the syntax tree with a dynamic containing a lambda expression in Clean:

```
compose (Var x ~> e) = dynamic (λy → composeLambda x y e :: ?)
```

The problem with this approach is that we have to specify the type of the lambda expression before the evaluation of `composeLambda`. Furthermore, the application of `composeLambda` will not be evaluated until the lambda expression is applied to an argument. This problem is unavoidable because we cannot get 'around' the lambda. Fortunately, bracket abstraction [Sch24, CF58] solves both problems.

Applications and constant values are `compose`d to dynamics in the usual way. We translate each lambda expression to a sequence of combinators (`S`, `K`, and `I`) and applications, with the help of the function `ski`.

```
compose ···         // Previous def.
compose (x ~> e) = compose (ski x e)
compose I        = dynamic λx → x
compose K        = dynamic λx y → x
compose S        = dynamic λf g x → f x (g x)

ski :: Expr Expr → Expr    // common bracket abstraction
ski  x (y ~> e)              = ski x (ski y e)
ski (Var x) (Var y) |[7] x == y = I
ski  x      (f @ y)          = S @ ski x f @ ski x y
ski  x       e               = K @ e
```

Composing lambda expressions uses `ski` to eliminate the ~> and `Var` syntax constructors, leaving only applications, dynamic values, and combinators. Com-

[7] If this guard fails, we end up in the last function alternative.

posing a combinator simply wraps its corresponding definition and type as a lambda expression into a dynamic.

Special combinators and combinator optimization rules are often used to improve the speed of the generated combinator code by reducing the number of combinators [CF58]. One has to be careful not to optimize the generated combinator expressions in such a way that the resulting type becomes too general. In an untyped world, this is allowed because they preserve the intended semantics when generating untyped (abstract) code. However, our generated code is contained within a dynamic and is therefore typed. This makes it essential that we preserve the principal type of the expression during bracket abstraction. Adding common η-reduction, for example, results in a too general type for `Var "f"` ⇝ `Var "x"` ⇝ `f x`: `a → a`, instead of: `(a → b) → a → b`. Such optimizations might prevent us from getting the principal type for an expression. Simple bracket abstraction using `S`, `K`, and `I`, as performed by `ski`, does preserve the principal type [HS86].

Code combined by Esther in this way, is not as fast as code generated by the Clean compiler. Combinators introduced by bracket abstraction are the main reason for this slowdown. Additionally, all applications are lazy and not specialized for basic types. However, these disadvantages only hold for the small (lambda) functions written at the command line, which are mostly used for plumbing. If faster execution is required, one can always copy-paste the command line into a Clean module that writes a dynamic to disk and running the compiler.

In order to reduce the number of combinators in the generated expression, our current implementation uses Diller's algorithm C [Dil88] without η-conversion in order to preserve the principal type, while reducing the number of generated combinators from exponential to quadratic. Our current implementation seems to be fast enough, so we did not explore further optimizations by other bracket-abstraction algorithms.

3.4.3 Irrefutable Patterns

Here we introduce irrefutable patterns, e.g., (nested) tuples, in lambda expressions. This is a preparation for the upcoming let(rec) expressions.

```
:: Expr = ···                       // Previous def.
       | Tuple Int                  // Tuple constructor

tupleConstr :: Int → Dynamic
tupleConstr 2 = dynamic λx y → (x, y)
tupleConstr 3 = dynamic λx y z → (x, y, z)
tupleConstr ··· // and so on...[8]
```

[8]...until 32. Clean does not support functions or data types with arity above 32.

```
compose ···           // Previous def.
compose (Tuple n) = tupleConstr n

ski :: Expr Expr → Expr
ski (f @ x)    e = ski f (x ~> e)
ski (Tuple n) e = Value (matchTuple n) @ e
ski  ···  // previous def.

matchTuple :: Int → Dynamic
matchTuple 2 = dynamic λf t → f (fst t) (snd t)
matchTuple 3 = dynamic λf t → f (fst3 t) (snd3 t) (thd3 t)
matchTuple  ···  // and so on...
```

We extend the syntax tree with `Tuple` *n* constructors (where *n* is the number of elements in the tuple). This makes expressions like

```
Tuple 2 @ Var "x" @ Var "y" ~> Tuple 2 @ Var "y" @ Var "x"
```

valid expressions. This example corresponds with the Clean lambda expression `λ(x, y) → (y, x)`.

When the `ski` function reaches an application in the left-hand side of the lambda abstraction, it processes both sub-patterns recursively. When the `ski` function reaches a `Tuple` constructor it replaces it with a call to the `matchTuple` function. Note that the right-hand side of the lambda expression has already been transformed into lambda abstractions, which expect each component of the tuple as a separate argument. We then use the `matchTuple` function to extract each component of the tuple separately. It uses lazy tuple selections (using `fst` and `snd`, because Clean tuple patterns are always eager) to prevent non-termination of recursive let(rec)s in the next section.

3.4.4 Let(rec) Expressions

Now we are ready to add irrefutable let(rec) expressions. Refutable let(rec) expressions must be written as cases, which will be introduced in next section.

```
:: Expr = ···                           // Previous def.
       | Letrec [Def] Expr              // let(rec) .. in ..
       | Y                              // Combinator

:: Def =
         (:=:) infix 0 Expr Expr        // .. = ..
```

```
compose ···              // Previous def.
compose (Letrec ds e) = compose (letRecToLambda ds e)
compose Y              = dynamic let y = f y
                                 in  y :: ∀a: (a → a) → a

letRecToLambda :: [Def] Expr → Expr
letRecToLambda ds e = let (p :=: d) = combine ds
                      in  ski p e @ (Y @ ski p d)

combine :: [Def] → Def
combine [p :=: e]       = p :=: e
combine [p1 :=: e1:ds] = let (p2 :=: e2) = combine ds in
                          Tuple 2 @ p1 @ p2 :=: Tuple 2 @ e1 @ e2
```

When `compose` encounters a let(rec) expression it uses `letRecToLambda` to convert it into a lambda expression. The `letRecToLambda` function `combines` all (possibly mutually recursive) definitions by pairing definitions into a single (possibly recursive) irrefutable tuple pattern. This leaves us with just a single definition that `letRecToLambda` converts to a lambda expression in the usual way [Pey87].

3.4.5 Case Expressions

Composing a case expression is done by transforming each alternative into a lambda `Expr`ession that takes the expression to match as an argument. This yields a function that, once composed, results in a dynamics that contains a functions that expects a value to match to. If the expression matches the pattern, the right-hand side of the alternative is taken. When it does not match, the lambda expression corresponding to the next alternative is applied to the expression, forming a cascade of **if**-then-else constructs. This will result in a single lambda expression that implements the case construct, and we apply it to the expression that we wanted to match against.

```
:: Expr = ···                          // Previous def.
       | Case Expr [Alt]              // case .. of ..

:: Alt = (:>) infix 0 Expr Expr      // .. → ..

compose ···              // Previous def.
compose (Case e as) = compose (altsToLambda as @ e)
```

We translate the alternatives into lambda expressions using `altsToLambda`, as shown below. If the pattern consists of an application, we do bracket abstraction for each argument, just as we did for lambda expressions, in order to deal with each subpattern recursively. Matching against an irrefutable pattern, such as variables of tuples, always succeeds and we reuse the code of `ski` that does the matching for lambda expressions. Matching basic values is done using `ifEqual` that uses Clean's built-in equalities for each basic type. We always add a default alternative using the `mismatch` function, which informs the user that none of the patterns matched the expression.

```
altsToLambda :: [Alt] → Expr
altsToLambda []                  = Value mismatch
altsToLambda [f @ x :> e:as]     = altsToLambda [f :> ski x e:as]
altsToLambda [Var x :> e:_]      = Var x ~> e
altsToLambda [Tuple n :> e:_]    = Tuple n ~> e
altsToLambda [Value dyn :> th:as] = case dyn of
    (i :: Int)  → Value (ifEqual i) @ th @ el
    (c :: Char) → Value (ifEqual c) @ th @ el
    ···  // for all basic types
where
    el = altsToLambda as

ifEqual :: a → Dynamic | TC a & Eq a
ifEqual x = dynamic λth el y → if (x == y) th (el y)
                    :: ∀b: b (a^ → b) a^ → b

mismatch = dynamic raise "Pattern mismatch" :: ∀a: a
```

Matching against a constructor contained in a dynamic takes more work. For example, if we put Clean's list constructor `[:]` in a dynamic we find that it has type `a → [a] → [a]`, which is a function type. In Clean, one cannot match closures or functions against constructors. Therefore, using the function `makeNode` below, we construct a node that contains the right constructor by adding dummy arguments until it has no function type anymore. The function `ifMatch` uses some low-level code to match two nodes to see if the constructor of the pattern matches the outermost constructor of the expression. If it matches, we need to extract the arguments from the node. This is done by the `applyTo` function, which decides how many arguments need to be extracted (and what their types are) by inspection of the type of the curried constructor. Again, we use some low-level auxiliary code to extract each argument while preserving laziness.

Some low-level code is necessary, because selecting an arbitrary argument of a node is not expressible in Clean. To keep the Clean language referential transparent, this cannot be expressed in general. Since we know that the resulting code and types are correct, we chose to write this in Clean's machine independent assembly code: ABC-code. Another option is to request destructors/machting function from the user. We chose to implement this hack using ABC-code because all information is available, from the Clean run-time system and the data and heap storage areas. We decided that the users convenience has priority over an elegant implementation in this case.

```
altsToLambda [Value dyn :> th:as] = case dyn of
    ··· // previous definition for basic types
    constr → Value (ifMatch (makeNode constr))
                              @ (Value (applyTo dyn) @ th) @ el
where
    el = altsToLambda as

ifMatch :: Dynamic → Dynamic
ifMatch (x :: a) = dynamic λth el y → if (matchNode x y) (th y)
                              (el y) :: ∀b: (a → b) (a → b) a → b

makeNode :: Dynamic → Dynamic
makeNode (f :: a → b) = makeNode (dynamic f undef :: b)
makeNode (x :: a)     = dynamic x :: a

applyTo :: Dynamic → Dynamic
applyTo ···                  // and so on, most specific type first...
applyTo (_ :: a b → c) = dynamic λf x → f (arg1of2 x)(arg2of2 x)
                                :: ∀d: (a b → d) c → d
applyTo (_ :: a → b)   = dynamic λf x → f (arg1of1 x)
                                :: ∀c: (a → c) b → c
applyTo (_ :: a)       = dynamic λf x → f :: ∀b: b a → b

matchNode :: a a → Bool // low-level code; compares two nodes.

argiofn :: a → b // low-level code; selects i^th argument of an n-ary node
```

Pattern matching against user-defined constructors requires that the constructors are available from, i.e., stored in, the file system. Esther currently does not support type definitions at the command line, and the Clean compiler must be used to introduce new types and constructors into the file system. For an example of this, we refer to the description of the use of case expressions in Sect. 3.3.4.

3.4.6 Overloading

Support for overloaded expressions within dynamics in Clean is not yet implemented (e.g., one cannot write **dynamic** `(==) :: ∀a: a a → Bool | Eq a`). If a future dynamics implementation would support overloading, it cannot be used in a way that suits Esther. We want to solve overloading using instances/dictionaries from the file system, which may change over time, and which is something we cannot expect from Clean's dynamic run-time system out of the box.

Below is the Clean version of the overloaded functions `==` and `one`. We will use these two functions as a running example.

```
class Eq  a where (==) infix 4 :: a a → Bool
class one a where one :: a

instance Eq  Int where (==) x y = ···  // low-level comparison code
instance one Int where one      = 1
```

To mimic Clean's overloading, we introduce the type `Overloaded` to differentiate between 'overloaded' dynamics and 'normal' dynamics. This type as shown below, has three type variables that represent: the dictionary type `d`, the 'original' type of the expression `t`, and the type of the structure representing the context restrictions of the overloaded function `o`, which also contains the variable the expression is overloaded in. Values of the type `Overloaded` consists of a infix constructor `|||` followed by the overloaded expression (of type `d → t`), and the context restrictions (of type `o`). A term `Class c` of type `Context v` is used for a single context restriction of the class `c` on the type variable `v`. Multiple context restrictions are combined in a tree of type `Contexts`.

```
:: Overloaded d t o = (|||) infix 1 (d → t) o
:: Contexts a b     = (&&&) infix 0 a b
:: Context v        = Class String

(==) = dynamic id ||| Class "Eq"
       :: ∀a: Overloaded (a a → Bool) (a a → Bool) (Context a)
one  = dynamic id ||| Class "one"
       :: ∀a: Overloaded a a (Context a)

instance_Eq_Int  = dynamic λx y → x == y :: Int Int → Bool
instance_one_Int = dynamic 1             :: Int
```

The dynamic `(==)`, in the example above, is Esther's representation of Clean's overloaded function `==`. The overloaded expression itself is the identity function because the result of the expression *is* the dictionary. The name of the class is `Eq`. The dynamic `(==)` is overloaded in a single variable `a`, the type of the dictionary is `a → a → Bool` as expected, the 'original' type is the same, and the type of the name is `Context a`. Likewise, the dynamic `one` is Esther's representation of Clean's overloaded function `one`.

By separating the different parts of the overloaded type (the expression, the dictionary, and the variable), we obtain direct access to the variable in which the expression is overloaded. This makes it easy to detect if the overloading has been resolved: the variable no longer unifies with `∀a: a.a`. By separating the dictionary type and the 'original' type of the expression, it becomes easier to check if the application of one overloaded dynamic to another is allowed. We can check if a value of type `Overloaded _ (a → b) _` can be applied to a value of type `Overloaded _ a _`).

To apply one overloaded dynamic to another, we combine the overloading information using the `Contexts` type in the way shown below in the function `applyOverloaded`.

```
applyOverloaded :: Dynamic Dynamic → Dynamic
applyOverloaded (f ||| of :: Overloaded df (a → b) cf) (x :: a)
    = dynamic (λd_f → f d_f x) ||| of :: Overloaded df b cf
applyOverloaded (f :: a → b) (x ||| ox :: Overloaded dx a cx)
    = dynamic (λd_x → f (x d_x)) ||| ox :: Overloaded dx b cx
applyOverloaded (f ||| of :: Overloaded df (a → b) cf)
                (x ||| ox :: Overloaded dx a cx)
    = dynamic (λ(d_f, d_x) → f d_f (x d_x)) ||| of &&& ox
      :: Overloaded (df, dx) b (Contexts cf cx)
```

`applyOverloaded` applies an overloaded function to a value, a function to an overloaded value, or an overloaded function to an overloaded value. The `compose` function from the beginning of this section is extended in the same way to handle 'overloaded dynamics'.

We use the (private) data type `Contexts` instead of tuples because this allows us to differentiate between a pair of two context restrictions and a single variable that has been unified with a tuple.

Applying `applyOverloaded` to `(==)` and `one` yields an expression semantically equal to `isOne` below. The overloaded expression `isOne` needs a pair of dictionaries to build the expression `(==) one` and has two context restrictions on the same variable. The 'original' type is `a → Bool`, and it is overloaded in `Eq` and `one`. Esther will pretty print this as: `isOne :: a → Bool | Eq a & one a`.

```
isOne = dynamic (λ(d_Eq, d_one) → id d_Eq (id d_one))
                ||| Class "Eq" &&& Class "one"
        :: ∀a: Overloaded ((a a → Bool, a) (a → Bool))
                (Contexts (Context a) (Context a)
```

Applying `isOne` to the integer `42` will bind the variable `a` to `Int`. Esther is now able to choose the right instance for both `Eq` and `one`. It searches the file system for the files named "instance Eq Int" and "instance one Int", and applies the code of `isOne` to the dictionaries after applying the overloaded expression to `42`. The result will look like `isOne42` in the example below, where all overloading has been removed from the type.

```
isOne42 = dynamic (λ(d_Eq, d_one) → id d_Eq (id d_one) 42)
                  (instance_Eq_Int, instance_one_Int)     :: Bool
```

Although overloading is resolved in the example above, the plumbing/dictionary passing code is still present. This will increase evaluation time, and it is not clear yet how this can be prevented.

3.5 Related Work

We have not yet seen an interpreter or shell that equals Esther's ability to use precompiled code, and to store expressions as compiled code, which can be used in other already compiled programs, in a type safe way.

Es [HR93] is a shell that supports higher-order functions and allows the user to construct new functions at the command line. A UNIX shell in Haskell [Mat] by Jim Mattson is an interactive program that also launches executables, and provides pipelining and redirections. Tcl [Ous90] is a popular tool to combine programs, and to provide communications between them. None of these programs provides a way to read and write typed objects, other than strings, from and to disk. Therefore, they cannot provide our level of type safety.

A functional interpreter with a file-system manipulation library can also provide functional expressiveness and either static or dynamic type-checking of part of the command line. For example, the Scheme Shell (ScSh) [Shi94] integrates common shell operations with the Scheme language to enable the user to use the full expressiveness of Scheme at the command line. Interpreters for statically typed functional languages, such as Hugs [JR02], even provide static type-checking in advance. Although they do type check source code, they cannot type check the application of binary executables to documents/data structures because they work on untyped executables.

The BeanShell [Nie] is an embeddable Java source interpreter with object scripting language features, written in Java. It is capable of inferring types for variables and to combine shell scripts with existing Java programs. While Esther generates compiled code via dynamics, the BeanShell interpreter is invoked each time a script is called from a normal Java program.

Run-time code generation in order to specialize code at run-time to certain parameters is not related to Esther. Esther only combines existing code into new code, by adding code for function application and combinators in between, using Clean's dynamic I/O system.

There are concurrent versions of both Haskell and Clean. Concurrent Haskell [PGF96] offers lightweight threads in a single UNIX process and provides M-Vars as the means of communication between threads. Concurrent Clean [NSvEP91] is only available on multiprocessor Transputers and on a network of single-processor Apple Macintosh computers. Concurrent Clean provides support for native threads on Transputer systems. On a network of Apple computers, it runs the same Clean program on each processor, providing a virtual multiprocessor system. Concurrent Clean provided lazy graph copying as the primary communication mechanism. Neither concurrent system can easily provide type safety between different programs or between multiple incarnations of a single program.

Both Lin [Lin98] and Cooper and Morrisett [CM90] have extended Standard ML with threads (implemented as continuations using call/CC) to form a small functional operating system. Both systems implement the basics needed for a stand-alone operating system. However, none of them support the type-safe communication of any value between different computers.

Erlang [AVWW96] is a functional language specifically designed for the development of concurrent processes. It is completely dynamically typed and primarily uses interpreted byte-code, while Famke is mostly statically typed and executes native code generated by the Clean compiler. A simple spelling error in a token used during communication between two processes is often not detected by Erlang's dynamic type system, sometimes causing deadlock.

Back et al. [BTS$^+$98] built two prototypes of a Java operating system. Although they show that Java's extensibility, portable byte code and static/dynamic type system provides a way to build an operating system where multiple Java programs can safely run concurrently, Java does not support dynamic type unification, higher-order functions, and closures in the comfortable way that our functional approach does.

3.6 Conclusions

We have shown how to build a shell that provides a simple, but powerful strongly typed functional programming language. We were able to do this using only Clean's support for run-time type unification and dynamic linking, albeit syntax transformations and a few low-level functions were necessary. The shell named Esther supports type-checking and type inference before evaluation. It offers application, lambda abstraction, recursive let, pattern matching, and function definitions: the basics of any functional language. Additionally, infix operators and support for overloading make the shell easy to use.

By combining code from compiled functions/programs, Esther allows the use of any pre-compiled program as a function in the shell. Because Esther stores functions/expressions constructed at the command line as a Clean dynamic, it supports writing compiled programs at the command line. Furthermore, these expressions written at the command line can be used in any pre-compiled Clean program. The evaluation of expressions using recombined compiled code is not as fast as using the Clean compiler. Speed can be improved by introducing fewer combinators during bracket abstraction, but it seams unfeasible to make Esther perform the same optimizations as the Clean compiler. In practice, we find Esther responsive enough, and more optimizations do not appear worth the effort at this stage. One can always construct a Clean module using the same syntax and use the compiler to generate a dynamic that contains more efficient code.

CHAPTER 4

Bracket Abstraction Preserves Typability

A formal proof of Diller–algorithm–C in PVS

SJAAK SMETSERS
ARJEN VAN WEELDEN

4.1 Introduction

Pioneer functional programming languages used combinators [CF58], such as *S*, *K*, and *I*, originally developed by Schönfinkel [Sch24], to define the semantics of expressions. The function definitions are translated via lambda expressions to combinators, this last step has become known as bracket abstraction. The first implementations of interpreters and compilers, e.g., SASL [Tur76], also used combinators to evaluate or compile untyped functional programs. Designing the 'best' set of combinators has temporarily been a competitive sport. The race to construct the fastest evaluators has spawned dozens of additional combinators and complex bracket-abstraction algorithms, e.g., Abdali [Kam76], Turner [Tur79], Diller [Dil88], and Bunder [Bun90]. Super combinators, cf. Peyton Jones [Pey87], were later introduced and program optimizations were defined using extensible super combinators instead of a fixed set of combinators. Eventually, combinators were discarded in favor of generating efficient native machine code. Nowadays, compilers for strongly typed functional languages generate code comparable in efficiency to C.

Using the strongly typed functional programming language Clean [PvE02], the authors have written an interactive shell [PvW04] that can type check functional command-line expressions before executing them. The translation of command-line expressions, which support all basic functional language constructions, uses a variant of bracket abstraction. We wanted to show that this is possible using merely Dynamics with their apparently limited set of operations. This forced us to do the type inference after bracket abstraction. We are convinced that the algorithm used is sound with respect to the operational semantics, since it has been used innumerably as an implementation of functional languages. However, to our knowledge, nobody has ever shown that this variant of bracket abstraction preserves the principal type.

In this chapter, we present a formal proof, using the theorem prover PVS [OSRS01], which indicates that Diller–algorithm–C [Dil88] without η-conversion preserves typability. One would never attempt such a proof entirely by hand as it contains too many cases, while its reasonable complexity allows it to be rigorously specified in a formal way. All the proofs presented in this chapter can be downloaded from the following website[1].

Further on in this chapter, we proceed to explain a few things about Dynamics (Sect. 4.2), the translation from functional expressions to combinators (Sect. 4.3), and the theorem prover PVS (Sect. 4.4). We also share some interesting issues of the proof itself (Sect. 4.5). Related work is discussed in Sect. 4.6, and we conclude and mention future work in Sect. 4.7.

4.2 Dynamics and the Shell Written in Clean

Clean [PvE02] is a strongly typed, pure, and lazy functional programming language, much like Haskell [Pey03], and not entirely unlike strict functional languages. Such languages are based on the concept of functions and consist of expressions (usually without side effects), function definitions, algebraic data type definitions, pattern matching, and (data) recursion. They usually feature complex static type systems and checkers, including type inference.

Clean features a hybrid type system, where run-time type-checking is integrated into the static (compile-time) type system. The system is based on the theories of Abadi [ACPP91] and Pil [Pil99]. Any expressions can be wrapped, together with their static (polymorphic) type, into an object with the static type *Dynamic*. As usual, those expressions are compiled and statically type-checked with respect to the functions and types of the program that defines them. The

[1]`http://www.cs.ru.nl/A.vanWeelden/bracket/`

example below shows the definition of the factorial function, which is wrapped in a dynamic and extracted again using a type pattern match.

```
fac 0 = 1                                  // factorial function
fac n = fac (n - 1) * n

dynamicFac = dynamic fac :: Int → Int      // wrap in a dynamic

matchDynamic (f :: Int → Int) = f          // unwrap by matching

example = matchDynamic dynamicFac 10       // apply; yields 3628800
```

Dynamics can be serialized (written to disk or over a network) while preserving sharing and cycles. The expressions inside those dynamics contain data and code, i.e., functions and closures. Therefore, their serialization will often contain references to the compiled code of the defining program. Dynamics can also be deserialized (read from disk) in other programs. Any necessary code will be automatically and lazily linked into the reading program as implemented by Vervoort [VP03]. Once the Dynamic object exists inside a running program, the program can pattern match on the original static type of the expression contained within the Dynamic.

```
dynamicApply dynf dynx = case (dynf, dynx) of
    (f :: a → b, x :: a) → dynamic f x :: b
    (g, y) → abort "Cannot unify formal and actual argument"
```

In the example above, we show a more complex example where we apply one Dynamic to another at run-time in a type-safe way. The type checker can statically check the usage of type pattern variables a and b in the application of a function that matches the type pattern $a \rightarrow b$ on an argument that matches type a. Obviously, this results in something that matches some type b. At run-time, an attempt is made to unify the type of the argument of f (from the first Dynamic) with the type of x (from the second Dynamic). If it succeeds, the substitution required for that unification is also applied to the type of the result of f, which is used as the resulting type of the application. The result after application can only be stored in a Dynamic again, since the actual type at run-time is unknown to the static type checker.

Using Dynamics to type only applications, in the manner done in the examples above, our shell [PvW04] is capable of type-checking/inferring any expression of a simple functional language. This is enabled by existing translation schemes, cf. Peyton Jones [Pey87], which can transform any language construct in a functional programming language to lambda expressions with explicit letrec-sharing, cf.

Hindley [HS86]. Threfore, the source language used throughout this chapter contains only applications, lambda expressions, variables and letrec expressions.

$$\begin{array}{rcl} \mathit{Expr} & ::= & \mathit{Expr}\ \mathit{Expr} \mid \lambda\, \mathit{Var}\ .\ \mathit{Expr} \mid \mathit{Var} \mid \textbf{letrec}\ \mathit{Var} = \mathit{Expr}\ \textbf{in}\ \mathit{Expr} \\ \mathit{Var} & ::= & x \mid y \mid z \mid \cdots \end{array}$$

Using a syntax tree, we can infer the types of applications and constants:

```
:: Expr = Con Dynamic | App Expr Expr | Lam Var Expr | Var Var
:: Var = Identifier String

type (Con dyn)    = dyn
type (App e1 e2)  = case (type e1, type e2) of
                        (f :: a → b, x :: a) → dynamic f x :: b
type (Lam v b)    = case type b of (e :: a) → dynamic λx.e :: ?
type (Var v)      = abort "cannot type a single variable"
```

However, static type inference of lambda expressions is problematic. The type of the body of the lambda expression (b in the code above) cannot be inferred because it is an open expression (contains the variable v). To infer the type using Dynamics, we need the run-time value of the variable, which is not available at compile time. We solved this problem using bracket abstraction, which removes all variables from an expression. This forced us to do type inference after bracket abstraction.

4.3 From Expression to Combinators

The combinators used in our target language, both in this chapter and in the shell, are from the algorithm–C by Diller [Dil88] and are shown in Fig. 4.1. The target language consists of variables, applications, and constants, which are the combinators and Dynamics. Of course, bracket abstraction should remove all variables from a closed expression. The implementation of the shell uses only Dynamics as constants. The combinators can easily be expressed using lambda expressions[2] and be put in a Dynamics with their static types. In contrast, the algorithm used in the proof does not use Dynamics as constants. This does not influence typability because Dynamics are already typed and not altered in any way by bracket abstraction. One could easily add any Dynamic to the set of combinator constants, much like the super-combinator approach.

Translation from the source language to combinators uses lambda expressions as an intermediate step. The translation $[\![e]\!]$ of an expression e in the source

[2]The shell written in Clean actually implements the fixed point combinator Y by the recursive binding **let** `x = f x` **in** `x` to construct efficient cycles.

$$\begin{array}{rcl}
I\,x \Rightarrow & x & : \alpha \to \alpha \\
K\,x\,y \Rightarrow & x & : \alpha \to \beta \to \alpha \\
S\,f\,g\,x \Rightarrow & f\,x\,(g\,x) & : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma \\
B\,f\,g\,x \Rightarrow & f\,(g\,x) & : (\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta \\
C\,f\,g\,x \Rightarrow & f\,x\,g & : (\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma \\
S'\,h\,f\,g\,x \Rightarrow & h\,(f\,x)\,(g\,x) & : (\alpha \to \beta \to \gamma) \to (\delta \to \alpha) \to (\delta \to \beta) \to \delta \to \gamma \\
B'\,h\,f\,g\,x \Rightarrow & h\,f\,(g\,x) & : (\alpha \to \beta \to \gamma) \to \alpha \to (\delta \to \beta) \to \delta \to \gamma \\
C'\,h\,f\,g\,x \Rightarrow & h\,(f\,x)\,g & : (\alpha \to \beta \to \gamma) \to (\delta \to \alpha) \to \beta \to \delta \to \gamma \\
Y\,f \Rightarrow & f\,(Y\,f) & : (\alpha \to \alpha) \to \alpha
\end{array}$$

Figure 4.1: Combinator reduction rules and their types.

language is shown in Fig. 4.2. We show translations for both non-recursive *let* and recursive *letrec*s, for which there is a monomorphic and a polymorphic variant. The shell implements the monomorphic *letrec* and it writes polymorphic function definitions to disk using Dynamics. By reading them back in when they are used in other expression, it achieves the same effect as the polymorphic *letrec*. The expression inside a Dynamic is shared, not substituted, but the (polymorphic) type can be instantiated multiple times.

$$\begin{array}{rcll}
[\![e_1\ e_2]\!] & = & [\![e_1]\!][\![e_2]\!] & \text{(application)} \\
[\![\lambda x.e]\!] & = & [\![\,[\![e]\!]\,]\!]_x & \text{(abstraction)} \\
[\![x]\!] & = & x & \text{(variable)} \\
[\![let\ x = e_1\ in\ e_2]\!] & = & [\![\lambda x.e_2]\!]\ [\![e_1]\!] & \text{(non-recursive let)} \\
[\![letrec\ x = e_1\ in\ e_2]\!] & = & [\![\lambda x.e_2]\!]\ (Y\ [\![\lambda x.e_1]\!]) & \text{(monomorphic let)} \\
[\![letrec\ x = e_1\ in\ e_2]\!] & = & [\![e_2]\!][Y[\![\lambda x.e_1]\!]/x] & \text{(polymorphic let)}
\end{array}$$

Figure 4.2: Translation from expressions to combinators via lambda expressions.

The bracket-abstraction algorithm $[\![\cdot]\!]_x$, over a variable x used in the shell and throughout this chapter is defined in Fig. 4.3. It is almost the same as algorithm–C and very similar to Abs/Dash/4 by Joy et al. [JRB85], who show that it produces good code and that improvements are hard and yield little effect. In contrast to algorithm–C, we do not use η-conversion because it obviously does not preserve the principal type, as indicated by the following example:

$$\lambda x.(\lambda y.xy) : (\alpha \to \beta) \to \alpha \to \beta \leadsto_\eta \lambda x.x : \alpha \to \alpha \leadsto_{[\![\ ]\!]} I : \alpha \to \alpha.$$

$$\begin{array}{rcll}
[\![x]\!]_x & = & I & \\
[\![e]\!]_x & = & K\,e & \text{if } x \notin FV(e) \\
[\![(e_1\,e_2)\,e_3]\!]_x & = & B'\,e_1\,e_2\,[\![e_3]\!]_x & \text{if } x \notin FV(e_1) \wedge x \notin FV(e_2) \\
[\![(e_1\,e_2)\,e_3]\!]_x & = & C'\,e_1\,[\![e_2]\!]_x\,e_3 & \text{if } x \notin FV(e_1) \wedge x \notin FV(e_3) \\
[\![(e_1\,e_2)\,e_3]\!]_x & = & S'\,e_1\,[\![e_2]\!]_x\,[\![e_3]\!]_x & \text{if } x \notin FV(e_1) \\
[\![e_1\,e_2]\!]_x & = & B\,e_1\,[\![e_2]\!]_x & \text{if } x \notin FV(e_1) \\
[\![e_1\,e_2]\!]_x & = & C\,[\![e_1]\!]_x\,e_2 & \text{if } x \notin FV(e_2) \\
[\![e_1\,e_2]\!]_x & = & S\,[\![e_1]\!]_x\,[\![e_2]\!]_x &
\end{array}$$

Figure 4.3: Our variant of bracket abstraction, based on priority from the top down.

4.4 The Theorem Provers PVS

As a short introduction to PVS, we will briefly recall the basics (c.f. [OSRS01]). PVS (*P*rototype *V*erification *S*ystems) offers an interactive environment for the development and analysis of formal specifications. The system consists of a specification language and a theorem prover. The specification language of PVS is based on classical, typed higher-order logic. Both the use of basic types, like integers, booleans and reals, and compound types (built with type constructors such as records, tuples, and function types) are permitted. New, possibly recursive, data types can be introduced via algebraic data type definitions. As an example of a user-defined data type, consider the following parameterized definition of a binary tree:

```
BinTree[V : TYPE] : DATATYPE BEGIN
  leaf   : leaf?
  node(el:V, left, right: BinTree) : node?
END BinTree
```

The data type has two *constructors*, `leaf` and `node`, with which trees can be built. Additionally, two *recognizers* `leaf?` and `node?` are defined (observe that PVS allows question marks as constituents of identifiers), which can be used as predicates to test whether or not a tree object starts with the corresponding constructor. The field names `el`, `left` and `right` can be used as *accessors* to extract these components from a node. However, for extraction purposes, it is often more convenient to use the built-in pattern matching via CASES expressions. Consider, for example, the following function `tree2List` that collects all elements of a tree and places them in a list.

```
tree2List(t:BinTree) : RECURSIVE list[V] =
    CASES t OF
        leaf: null,
        node(e,l,r): append(tree2List(l),cons(e,tree2List(r)))
    ENDCASES
MEASURE size(t)
```

The MEASURE specification is mandatory when defining a recursive function, such as the tree2List function shown above. In PVS, it is required that all functions are total. This measure is used to show that the function terminates. This is realized by generating a proof obligation (a so-called *TCC*, *T*ype *C*orrectness *C*ondition) indicating that the measure strictly decreases at each recursive call. Obviously, in this case the size of the tree fulfills this property. How can an appropriate measure be given for the function size itself? A solution is to have data type definitions, which cause PVS to generate a number of functions and axioms, which can be used freely in a theory importing that data type. Among them, one example is the ordering $\ll$ on trees indicating whether the first argument tree is a subtree of second one. In a measure specification, $\ll$ can be used as follows:

```
size(t:BinTree) : RECURSIVE int  =
    CASES t OF
        leaf: 0,
        node(e,l,r): size(l) + size (r) + 1
    ENDCASES
MEASURE t BY <<
```

PVS specifications are organized into parameterized theories that may contain declarations of functions, axioms, theorems, etc.. The PVS language provides the customary arithmetic and logical operators, function application, lambda abstraction, and quantifiers. Names may be overloaded, including those of built-in operators such as $<$ and $+$.

Higher-order logic forms the base for the theorem prover of PVS, i.e., PVS admits to quantify over predicates. For example, consider the following lemma that enables the use of course–of–values induction on the size of trees in a proof.

```
tree_size_induction : LEMMA
    (∀ (p:pred[BinTree]): (∀ (g:BinTree): (∀ (s:BinTree):
        size(s) < size(g) ⇒ p(s)) ⇒ p(g)) ⇒ (∀ (c:BinTree): p(c)))
```

This can be proven with the predefined induction principle NAT_induction. In order to facilitate the development of proofs, PVS provides a collection of proof commands and predefined combinations of proof commands, so-called *proof strategies*. During the construction of a proof, PVS constructs and maintains a proof tree. The goal of a proof is to apply proof commands/strategies such that all the leaves of the proof tree are recognized as true. Therefore, the proof itself is just a sequence of proof strategies that converts the initial proof tree into a complete one. Such proof sequences are available in textual form making them easy to edit or display, or even to rerun. Usually, they are kept in a separate proof file and can be inspected at any time during a proof session.

4.5 The Proof in PVS

In this section, we will use PVS to prove that bracket abstraction preserves typability. We will first concentrate on the *monomorphic* case.

4.5.1 Bracket Abstraction

We start the description of the proof by introducing two data types to represent the source language `FUNC` and destination language `COMB` of our translation.

Our source language is essentially the lambda calculus enriched with a `letr` construct to specify single recursive definitions. The addition of multiple recursion and pattern matching is straightforward, but is left out for reasons of simplicity.

```
FUNC [V : TYPE] : DATATYPE BEGIN
  vari (id:V)                      : vari?
  appl (fun, arg:FUNC)             : appl?
  lamb (l_var:V, l_body:FUNC)      : lamb?
  letr (b_var:V, b_def, b_body:FUNC): letr?
END FUNC
```

The destination language consists of combinator expressions. Our representation is parametric in both variables and combinators.

```
COMB[V : TYPE, C : TYPE] : DATATYPE BEGIN
    c_var   (v_id: V) : c_var?
    c_const (c_id: C) : c_const?
    c_appl  (c_fun, c_arg: COMB) : c_appl?
END COMB
```

The concrete combinators are introduced via an enumeration type (see Fig. 4.1).

```
SKI: TYPE = {S,K,I,Y,B,C,S1,B1,C1}
```

We shall now present the bracket-abstraction algorithm `lam2Ski` (see Fig. 4.2):

```
lam2Ski(e:FUNC) : RECURSIVE COMB =
    CASES e OF
        vari(v)      : c_var(v),
        appl(f, a)   : c_appl(lam2Ski(f),lam2Ski(a)),
        lamb(v,e)    : abstr(lam2Ski(e),v),
        letr (v,d,e) : c_appl(abstr(lam2Ski(e),v),
                        c_appl(c_const(Y),abstr(lam2Ski(d),v)))
    ENDCASES
MEASURE e BY <<
```

where abstr is a recursive function that builds up the combinator expression for the distribution of a parameter v (see Fig. 4.3). CAppl2 and CAppl3 are just helper functions introduced to improve readability.

```
CAppl2(c:SKI, a1,a2: COMB): COMB = c_appl(c_appl(c_const(c),a1),a2)
CAppl3(c:SKI, a1,a2,a3: COMB): COMB = c_appl(CAppl2(c,a1,a2),a3)

abstr(e:COMB,v:V) : RECURSIVE COMB =
     IF fvs(e)(v)
     THEN CASES e OF
             c_var(w)     : c_const(I),
             c_appl(f,a)  :
                 IF c_appl?(f) ∧  ¬fvs(c_fun(f))(v)
                 THEN LET g = c_fun(f), b = c_arg(f) IN
                     IF ¬fvs(b)(v)
                     THEN CAppl3 (B1, g, b, abstr(a,v))
                     ELSIF ¬fvs(a)(v)
                     THEN CAppl3 (C1, g, abstr(b,v), a)
                     ELSE CAppl3 (S1, g, abstr(b,v), abstr(a,v))
                     ENDIF
                 ELSIF ¬fvs(f)(v)
                 THEN CAppl2 (B, f, abstr(a,v))
                 ELSIF ¬fvs(a)(v)
                 THEN CAppl2 (C, abstr(f,v), a)
                 ELSE CAppl2 (S, abstr(f,v), abstr(a,v))
                 ENDIF
          ENDCASES
     ELSE c_appl(c_const(K),e)
     ENDIF
MEASURE e BY <<
```

The standard subterm order of FUNC and COMB are used as measures in lam2Ski and in abstr, respectively. In both cases, these orders are denoted by $\ll$. The predicate fvs indicates whether a given variable freely occurs in an expression. It is defined using the reduce function for the COMB data type. This (fold-like) operation is internally generated by PVS and can often be used as a substitute for recursion.

```
fvs:[COMB → PRED[V]] = reduce(singleton, λ(c:C):∅, ∪)
```

The equivalence between lam2Ski and abstr and the specifications of these transformations in Sect. 4.3, is self-evident.

4.5.2 Typing

In order to represent types for both combinators and lambda expressions, we introduce the following data type.

```
TYPES[V : TYPE]: DATATYPE BEGIN
    t_var  (t_var:V)            : t_var?
    t_arr  (t_arg, t_res:TYPES) : t_arr?
END TYPES
```

This definition is self-explanatory, as well as the definitions of *substitution* and *(substitution) instance*. The latter is denoted as a binary predicate ≤ on types. Here `subst` is the customary lifting of substitutions to types. It can be easily expressed in terms of `reduce`.

```
Substitution : TYPE = [V → TYPES]
subst(s:Substitution): [TYPES → TYPES] = reduce(s,t_arr);

t1, t2: VAR TYPES
≤(t1, t2) : bool = ∃(s:Substitution) : t2 = subst(s)(t1)
```

The type system for `FUNC` terms is a straightforward extension of simple Curry typing. We make use of the possibility in PVS to define *inductive predicates*. The expression `typableE(b)(e,t)` should be read as “In the context of a base *b*, the expression *e* has type *t*”.

```
BASE : TYPE = [X → TYPES] typableE(b:BASE)(e:FUNC, tr:TYPES) :
INDUCTIVE bool =
    CASES e OF
        vari(v)     : b(v) = tr,
        appl(f, a)  : ∃(ta:TYPES): typableE(b)(f,t_arr(ta,tr)) ∧
                                    typableE(b)(a,ta),
        lamb(v,e)   : t_arr?(tr) ∧
                       typableE(b WITH [v := t_arg(tr)])(e,t_res(tr)),
        letr (v,d,e): ∃(ft:TYPES): LET nb = b WITH [v := ft] IN
                          typableE(nb)(d,ft) ∧  typableE(nb)(e,tr)
    ENDCASES
```

Observe that the type system is *monomorphic*: in the term `letr (v,d,e)`, all occurrences of `v` in `e` should have identical types. In Sect. 4.5.3 we describe how to extend the system with *polymorphic* `letr` constructs.

For typing `COMB` expressions, we assume that combinator symbols are supplied with a type (or actually a type scheme) by a so-called *type environment*. The type system is defined as a PVS theory parameterized with that environment.

```
typingCOMB [V, X, C:TYPE, %type and term variables, and combinator symbols
    (IMPORTING TYPES[V]) env:[C → TYPES[V]]]: THEORY %type environment
```

```
BEGIN
  typableC(b:BASE)(e:COMB, t:TYPES) : INDUCTIVE bool =
      CASES e OF
          c_var(w)      : b(w) = t,
          c_const(c)    : env(c) ≤ t,
          c_appl(f, a) : ∃(t1:TYPES):
                          typableC(b)(f,t_arr(t1,t)) ∧  typableC(b)(a,t1)
      ENDCASES
END typingCOMB
```

We can now formulate our main theorem that relates the typing of FUNC to the typing of COMB.

```
type_preserving : THEOREM ∀(e:FUNC,b:BASE,t:TYPES):
                           typableE(b)(e,t) ⇔ typableC(b)(lam2Ski(e),t)
```

In order to be able to prove this theorem, it is necessary to have a concrete type environment for SKI. We simply choose natural numbers as names for the type variables appearing in the type schemes.

```
 alp : TYPES = t_var(0)
 bet : TYPES = t_var(1)
 gam : TYPES = t_var(2)
 del : TYPES = t_var(3)

 TArr2(t1,t2,t3: TYPES)       : TYPES = t_arr(t1,t_arr(t2,t3))
 TArr3(t1,t2,t3,t4: TYPES)    : TYPES = t_arr(t1,t_TArr2(t2,t3,t4))
 TArr4(t1,t2,t3,t4,t5: TYPES) : TYPES = t_arr(t1,t_TArr3(t2,t3,t4,t5))

 env_ski(c:SKI): TYPES =
    CASES c OF
       I : t_arr(alp,alp),
       K : TArr2(alp,bet,alp),
       S : TArr3(t_arr(alp,t_arr(bet,gam)),t_arr(alp,bet),gam,bet),
       Y : t_arr(t_arr(alp,alp),alp),
       B : TArr3(t_arr(alp,bet),t_arr(gam,alp),gam,bet),
       C : TArr3(t_arr(alp,t_arr(bet,gam)),bet,alp,gam),
       S1: TArr4(t_arr(alp,t_arr(bet,gam)),t_arr(del,alp),
                 t_arr(del,bet),del,gam),
       B1: TArr4(t_arr(alp,t_arr(bet,gam)),alp,t_arr(del,bet),del,gam),
       C1: TArr4(t_arr(alp,t_arr(bet,gam)),t_arr(del,alp),bet,del,gam)
    ENDCASES
```

It is not difficult to show that each combinator written as a lambda expression is typable according to the typableE predicate, using an arbitrary base and the type given by env_ski. For example, the following properties can be proven in merely a few steps.

```
kterm: FUNC = lamb(0,lamb(1,vari(0)))
yterm: FUNC = lamb(0,letr(1,appl(vari(0),vari(1)))),vari(1))
K_typable: LEMMA ∀(bas:Base): typableE(bas)(kterm,env_ski(K))
Y_typable: LEMMA ∀(bas:Base): typableE(bas)(yterm,env_ski(Y))
```

The proof of our main theorem takes more effort. It requires the following property concerning the typing of abstractions.

```
type_abstr : LEMMA    ∀(e:COMB,b:BASE,ta,tr:TYPES,v:V):
 typableC(b WITH [v:=ta])(e,tr) ⇔ typableC(b)(abstr(e,v),t_arr(ta,tr))
```

This lemma is proven by course–of–values induction on the size of e. The proof itself is actually not difficult, but its size is quite extensive. It requires approximately 500 proof steps, as can be seen in the PVS proof files. This example also clearly shows that it is almost impossible to perform such a proof without the assistance of a theorem prover.

4.5.3 A Polymorphic Type System

In order to express letr polymorphism we need quantified types, also known as *type schemes*. These type schemes are of the form $\forall\alpha_1,\ldots,\alpha_n : \sigma$ in which the α_i are type variables and σ is a type. Instead of formalizing a type scheme as a pair consisting of a set of type variables and a type, we use a more explicit representation as for instance can be found in Naraschewski and Nipkow [NN99]. As such, the type scheme is formalized as an algebraic data type containing separate constructors for free and bound variables.

```
SCHEME[V : TYPE]: DATATYPE BEGIN
    ts_bv  (bv:V): ts_bv?
    ts_fv  (fv:V): ts_fv?
    ts_arr (arg, res:SCHEME): ts_arr?
END SCHEME

discard(x:V): PRED[V] = ∅

bvs:[SCHEME → PRED[V]] = reduce(singleton,discard,∪)
fvs:[SCHEME → PRED[V]] = reduce(discard,singleton,∪)
```

The predicates bvs and fvs determine the set of bound and free type variables for a given scheme, respectively. We will use several conversions between types and type schemes. A type can be obtained from a scheme by means of *instantiation*. This operation replaces the bound variables of a scheme with types and leaves the free variables unaltered.

```
inst(s:Substitution): [SCHEME → TYPES] =
    reduce(s,λ(v:V):t_var(v),t_arr)
```

A type can be converted into a scheme via *generalization*. The result of a generalization step depends on the context in which this operation is performed, in particular on the type variables appearing in the used base: only type variables not appearing free in a base can be universally quantified. Observe that in the polymorphic case a base associates term variables with schemes rather than with types.

```
BASE : TYPE = [X → SCHEME]

fvs(b:BASE): PRED[V] = { v:V | ∃(x:X) : fvs(b(x))(v) }
gen(b:BASE): [TYPES → SCHEME] =
    reduce(λ(v:V):IF fvs(b)(v)THEN ts_fv(v) ELSE ts_bv(v) ENDIF,ts_arr)
```

The adjustment of the `typableE` predicate leads to a type system almost equivalent to the system presented by Naraschewski and Nipkow [NN99]. There is only one small difference: in the term `letr(v,b,e)` we distinguish between two cases depending on whether or not *v* occurs in *e*. If *v* is present in *e* then the whole term is treated as usual. If not, the subterm *b* is ignored. It does not matter whether *b* is typable or not if it is not used in *e*. We will explain the reason for this refinement later. The test for the presence of *v* in *e* is done via `fvs`.

```
fvs:[FUNC → PRED[V]] = reduce(singleton, ∪, remove,
        λ(v:V,b,e:PRED[V]):IF e(v) THEN remove(v,∪(b,e)) ELSE e ENDIF)

type2Scheme: [TYPES  → SCHEME] = reduce(ts_fv,ts_arr)

typableE(b:BASE)(e:FUNC, tr:TYPES) : INDUCTIVE bool =
    CASES e OF
        vari(v)    : ∃(s:Substitution): inst(s)(b(v)) = tr,
        appl(f, a) : ∃(ta:TYPES): typableE(b)(f,t_arr(ta,tr)) ∧
                                  typableE(b)(a,ta),
        lamb(v,e)  : t_arr?(tr) ∧ typableE(b
                     WITH [v := type2Scheme(t_arg(tr))])(e,t_res(tr)),
        letr(v,d,e): IF fvs(e)(v)
                     THEN ∃(t:TYPES) :
                          typableE(b WITH [v := type2Scheme(t)])(d,t) ∧
                          typableE(b WITH [v := gen(b)(t)])(e,tr)
                     ELSE typableE(b)(e,tr)
                     ENDIF
    ENDCASES
```

The operation `type2Scheme` converts a type into a fully monomorphic scheme, i.e., a scheme with no bound variables.

The question remains of how to change the type system for combinators such that it can handle multiple occurrences of a recursive function introduced by a `letr`. Normally, (the combinator version of) this function is distributed over the corresponding expression via the *S* combinator. However, the polymorphic usage of arguments requires polymorphism of a higher rank. Observe that the schemes we have introduced are essentially of rank 1, as well as the types provided by the type environment, which is used for assigned types to combinators. Instead of allowing universal quantifiers at arbitrary levels, we adjust the transformation rule for `letr` expressions in the following way:

```
lam2Ski(e:FUNC) : RECURSIVE COMB =
    CASES e OF
        vari(v)     : c_var(v),
        appl(f, a)  : c_appl(lam2Ski(f),lam2Ski(a)),
        lamb(v,e)   : abstr(lam2Ski(e),v),
        letr(v,d,e): subst(single(v,c_appl(c_const(Y),
                             abstr(lam2Ski(d),v)))) (lam2Ski(e))
    ENDCASES
MEASURE e BY <<

single(v:V,e:COMB) : [V → COMB] =
    λ(w:V): IF v = w THEN e ELSE c_var(v) ENDIF
subst(s:[V → COMB]) : [COMB → COMB] = reduce(s,c_const,c_appl)
```

Here `single` and `subst` are substitutions on combinator terms. The disadvantage of the transformation is of course, that if the function is used more than once, it will be duplicated.

If the function is not used at all, it will disappear due to the substitution. For this reason, we made the case distinction in `typableE` for the `letr` construct. In this way, we are able to preserve typability in all cases and are not obliged to make an exception for cases that probably rarely occur.

The only part of the type system for combinators, `typableC`, that needs to be adjusted is the rule for variables. Instead of using the base type of the variable, we now allow instantiation of the scheme provided by the base. Since this adjustment is self-evident, we do not show it here.

The main goal of this section is to prove the following theorem again:

```
type_preserving : THEOREM ∀(e:FUNC,b:BASE,t:TYPES):
                            typableE(b)(e,t) ⇔ typableC(b)(lam2Ski(e),t)
```

At first sight, the extension of our system with type schemes seems to have low impact. However, as already has been noticed by Naraschewski and Nipkow [NN99], reasoning about type schemes is much more subtle than reasoning about (monomorphic) types. The proof of the theorem depends on the following two properties concerning term substitutions:

```
type_subst1 : LEMMA    ∀(e1,e2:COMB,b:BASE,t1,t2:TYPES, v:V):
        typableC(b)(e1,t1) ∧ typableC(b WITH [v := gen(b)(t1)])(e2, t2)
             ⇒ typableC(b)(subst(single(v,e1))(e2),t2)

type_subst2 : CONJECTURE
∀(e1,e2:COMB,b:BASE,t2:TYPES,v:(fvs(e2))):
        typableC(b)(subst(single(v,e1))(e2),t2) ⇒
             ∃(t1:TYPES): typableC(b)(e1,t1) ∧
                        typableC(b WITH [v := gen(b)(t1)])(e2, t2)
```

The first one is used to prove the ⇒ part of the theorem; the second one to prove the ⇐ part. Momentarily, we have finished the proof of `type_subst1`. This proof is performed by induction on the structure of the term `e2`, see the proof files. Currently we are working on a full formal proof of `type_subst2`.

4.6 Related Work

In his book, Hindley [Hin97] writes about *strong type-invariance* and shows that certain sets of combinators form a *typable basis*. He informally proves that a set of lambda expressions imitating combinators preserves typability. This is done by construction, using small variations on common bracket-abstraction algorithms. This approach is not unlike ours. However, we use a larger set, a more complex algorithm, and Hindley's proof is neither formal nor are all the details shown.

The type inference algorithm $\mathcal{W}$ by Damas and Milner [DM82] has been proven formally and mechanically by Nazareth and Nipkow [NN96] (monomorphic), Naraschewski and Nipkow [NN99] (polymorphic), using Isabelle/HOL, and by Dubois and Ménissier-Morain [DM99] (polymorphic) using Coq. Although we prove equivalence of type inference before and after bracket abstraction in PVS instead of type inference itself, the process of proving has a lot in common. Everybody runs into issues with alpha conversion. Furthermore, using a proof assistant/checker forces one to formalize everything explicitly and prove every minute detail, where one may adversely use hand waving to sweep it under the rug in an informal proof.

Hindley [Hin69] and, almost simultaneously, Curry [Cur69] showed that one can derive a principal type inference algorithm on a system of combinators, using reduction on type combinators. This already shows that one can do type inference using only application, since the combinators can be imitated using lambda calculus. Their (informal) proof is based on the *S* and *K* combinator, while we needed proof that it works for algorithm-C and that it infers the exact same types as functional languages usually do (using variants of algorithm $\mathcal{W}$).

4.7 Conclusions

We have shown how to specify type derivation and complex bracket abstraction in a rigorously formal way using PVS, at least for the monomorphic case. This enabled us to formally prove that Diller–algorithm–C without η-conversion preserves typability. Our approach to derive types after bracket abstraction in a type-checking command-line shell, requires this property. This proof also confirms that we were correct to assume that a type checker/inferrer can really be constructed using merely Dynamics of the functional programming language Clean. We now conclude that the seemingly limited interface of Dynamics is powerful enough for type inference such as done by our shell.

For future aspirations, it rests to complete the the $\Leftarrow$ part of the proof for the polymorphic case.

CHAPTER 5

Automatic Generation of Editors for Higher-Order Data Structures

PETER ACHTEN
MARKO VAN EEKELEN
RINUS PLASMEIJER
ARJEN VAN WEELDEN

5.1 Introduction

In the last decade, Graphical User Interfaces (GUIs) have become *the* standard for user interaction. Programming these interfaces can be done without much effort when the interface is rather static, and for many of these situations excellent tools are available. However, when there is more dynamic interaction between interface and application logic, such applications require tedious manual programming in any programming language. Programmers need to be skilled in the use of a large programming toolkit.

The goal of the *Graphical Editor* project is to obtain a concise programming toolkit that is *abstract*, *compositional*, and *type-directed*. Abstraction is required to reduce the size of the toolkit, compositionality reduces the effort of putting together (or altering) GUI code, and type-directed automatic creation of GUIs allows the programmer to focus on the data model. In contrast to visual programming environments, programming toolkits can provide ultimate flexibility, type safety, and dynamic behavior within a single framework. We use a *pure func-*

tional programming language (Clean [PvE02]) because functional programming languages have proven to be very suited for creating abstraction layers on top of each other. Additionally, they have strong support for type definitions and type safety.

Our programming toolkit utilizes the *Graphical Editor Component* (*GEC*) [AvEP04b] as universal building block for constructing GUIs. A *GEC*$_{\mathtt{t}}$ is a graphical editor for values of any *monomorphic first-order* type `t`. This type-directed creation of *GEC*s has been obtained by *generic programming* techniques [AP03, HP01, Hin00b]. Generic programming is extremely beneficial when applied to composite custom types. With generic programming, one defines a family of functions that depend on the *structure* of types. Although one structural element is the *function type* constructor (→), it is fundamentally impossible to define a generic function that edits these higher-order values directly, because pure functional programs cannot look inside functions without losing referential-transparency (for instance by distinguishing λx → x+1 from λx → 1+x).

In this chapter, we extend the *GEC* toolkit in two ways, such that it can construct higher-order value editors. The first extension uses run-time *dynamic typing* [ACP^{+}92, Pil99], which allows us to include them in the *GEC* toolkit, but this does not allow type-directed GUI creation. It does, however, enable the toolkit to use polymorphic higher-order functions and data structures. The second extension uses compile-time static typing, in order to gain monomorphic higher-order type-directed GUI creation of *abstract* types. It uses the *abstraction mechanism* of the *GEC* toolkit [AvEP04a].

Both extensions require a means of using functional expressions, entered by the user, as functional values. Instead of writing our own parser/interpreter/type-inference system, we use the *functional Esther shell* [PvW04] (Chap. 3), which provides type-checking at the command line and can use compiled functions from disk. These functions can have arbitrary size and complexity, and even interface with the imperative world. Esther makes extensive use of dynamic types. Dynamic types turn arbitrary (polymorphic, higher-order) data structures (for instance of type `[Int → Int]` or `(Tree a) → a`) into a *first-order* data structure of type `Dynamic` without losing the original type.

Contributions of this chapter are:

- We provide type-safe *expression* editors, which are needed for higher-order value editors.

  We obtain, as a bonus, the ability to edit first-order values using *expressions*.

  Another bonus: within these expressions, one can use compiled functions from disk, incorporating *real world* functionality.

- The programming toolkit can now create polymorphic dynamically typed, and monomorphic statically typed, higher-order value editors.

- The programming toolkit is type-safe and type-directed.

This chapter is structured as follows. Section 5.2 contains an overview of the first-order *GEC* toolkit. In Sect. 5.3 we present the first extension, in which we explain how Esther incorporates *expressions* as *functional values* using dynamic types. We present in Sect. 5.4 the second extension, and explain how we obtain higher-order type-directed GUI creation using the *abstraction mechanism* of the *GEC* toolkit. Section 5.5 gives examples of the new system that illustrate its expressive power. We discuss related work in Sect. 5.6 and conclude in Sect. 5.7.

Finally, a note on the implementation and the examples in this chapter. The project has been realized in Clean. Familiarity with Haskell [Pey03] is assumed, relevant differences between Haskell and Clean are explained in footnotes. The GUI code is mapped to Object I/O [AP98], which is Clean's library for GUIs. Given sufficient support for dynamic types, the results of this project can be transferred to Generic Haskell [CL03], using the Haskell port of Object I/O [AP01]. The complete code of all examples (including the complete *GEC* implementation in Clean) can be downloaded from the GEC toolkit website[1].

5.2 The GEC Programming Toolkit

With the *GEC* programming toolkit [AvEP04b], one constructs GUI applications in a *compositional* way using a high level of *abstraction*. The basic building block is the Graphical Editor Component (*GEC*). It is generated by a *generic* function, which makes the approach *type-directed*.

Before explaining *GEC*s in more detail, we need to point out that Clean uses an explicit multiple environment passing style [Ach96] for I/O programming. As *GEC*s are integrated with Clean Object I/O, the I/O functions that are presented in this chapter are state transition functions on the program state (`PSt ps`). The program state represents the external world of an interactive program, tailored for GUI operations. In this chapter, the identifier `env` is a value of this type. The uniqueness type system [BS99] of Clean ensures single threaded use of the environment. To improve the readability, uniqueness type attributes that actually appear in the type signatures are not shown. Furthermore, the code has been slightly simplified, leaving out a few details that are irrelevant for this chapter.

[1]`http://clean.cs.ru.nl/gec`

Graphical Editor Components

A *GEC*$_{\mathtt{t}}$ is an editor for values of type `t`. It is generated with a *generic* function [Hin00b, AP03]. A generic function is a meta-function that works on a description of the structure of types. For any concrete type `t`, the compiler is able to automatically derive an instance function of this generic function for the type `t`. The power of a generic scheme is that we obtain an editor for free for any monomorphic data type. This makes the approach particularly suited for *rapid prototyping*.

The generic function `gGEC` creates *GEC*s. It takes a *definition* (`GECDef t env`) of a *GEC*$_{\mathtt{t}}$ and *creates* the *GEC*$_{\mathtt{t}}$ object in the environment. It returns an *interface* (`GECInterface t env`) to that *GEC*$_{\mathtt{t}}$ object. The environment `env` is in this case `(PSt ps)`, since `gGEC` uses Object I/O.

```
generic[2] gGEC t :: (GECDef t (PSt ps)) (PSt ps)
                 → (GECInterface t (PSt ps), PSt ps)[3]
```

The (`GECDef t env`) consists of three elements. The first is a string that identifies the top-level Object I/O element (window or dialog) in which the editor must be created. The second is the initial value of type `t` of the editor. The third is a callback function of type `t` → `env` → `env`. This callback function tells the editor which parts of the program need to be informed of user actions. The editor uses this function to respond to changes to the value of the editor.

```
::[4] GECDef t env :==[5] (String,t,CallBackFunction t env)
:: CallBackFunction t env :== t → env → env
```

The (`GECInterface t env`) is a record that contains all *methods* of the newly created *GEC*$_{\mathtt{t}}$.

```
:: GECInterface t env = { gecGetValue :: env → (t,env)
                        , gecSetValue :: t → env → env }[6]
```

The `gecGetValue` method returns the current value, and `gecSetValue` sets the current value of the associated *GEC*$_{\mathtt{t}}$ object. Programs can be constructed combining editors by tying together the various `gecSetValues` and `gecGetValues`. We are working on an arrow combinator library that abstracts from the necessary plumbing [AvEPvW04a]. For the examples in this chapter, it is sufficient to use the following tying function:

[2]**generic** $f\ t$ :: $T(t)$ introduces a generic function f with type scheme $T(t)$. Keywords are type-set in **bold**.

[3]Clean separates function arguments by whitespace, instead of `->`.

[4]Type definitions are preceded by `::`.

[5]`:==` introduces a synonym type.

[6]$\{f_0$ :: t_0, ..., f_n :: $t_n\}$ denotes a record with field names f_i and types t_i.

```
selfGEC :: String (t → t) t (PSt ps) → (PSt ps) |[7] gGEC{|*|} t
selfGEC s f v env = env1 where ({gecSetValue},env1) = gGEC{|*|}
                                  (s,f v,λx → gecSetValue(f x)) env
```

Given an `f` of type `t → t` on the data model of type `t` and an initial value `v` of type `t`, `selfGEC gui f v` creates the associated *GEC*$_{\texttt{t}}$ using `gGEC` (hence the context restriction). `selfGEC` creates a feedback loop that sends every edited output value back as an input to the same editor, after applying the function `f`.

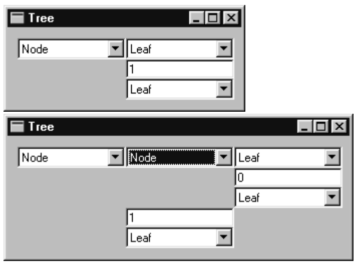
Example 1:

The standard appearance of a *GEC* is given by the following program that creates an editor for a *self-balancing* binary tree:

```
module Editor import StdEnv, StdIO, StdGEC

Start :: *World → *World
Start world = startIO MDI Void
                myEditor world

myEditor :: (PSt ps) → (PSt ps)
myEditor = selfGEC "Tree" balance
               (Node Leaf 1 Leaf)
```



```
:: Tree a = Node (Tree a) a (Tree a) | Leaf
```

In this example, we create a *GEC*$_{\texttt{Tree Int}}$ which displays the indicated initial value `Node Leaf 1 Leaf` (upper screen shot). The user can manipulate this value in any desired order, producing new values of type `Tree Int` (e.g., turning the upper `Leaf` into a `Node` with the pull-down menu). Each time a new value is created or edited, the feedback function `balance` is applied. `balance` takes a argument of type `Tree a` and returns the tree after balancing it. The shape and layout of the tree being displayed adjusts itself automatically. Default values are generated by the editor when needed.

Note that the only things that need to be specified by the programmer are the initial value of the desired type, and the feedback function. In all remaining examples, we only modify `myEditor` and the type for which an instance of `gGEC` is derived.

The tree example shows that a *GEC*$_{\texttt{t}}$ explicitly reflects the structure of type `t`. For the creation of GUI applications, we need to model both specific GUI elements (such as buttons) and layout control (such as horizontal, vertical layout). This

[7]In a function type, | introduces all overloading class restrictions.

has been done by *specializing* gGEC [AvEP04b] for a number of types that either represent GUI elements or layout. Here are the types and their gGEC specialization that are used in the examples in this chapter:

```
:: Display a = Display a     // a non-editable GUI: e.g.,
:: Hide    a = Hide    a     // an invisible GUI, useful for state.
:: UpDown    = UpPressed | DownPressed | Neutral   // a spin button: .
```

5.3 Dynamically Typed Higher-order GECs

In this section, we show how to extend *GEC*s with the ability to deal with functions and expressions. Because functions are opaque, the solution requires a means of interpreting functional expressions as functional values. Instead of writing our own parser/interpreter/type-inference system, we use the *Esther* shell [PvW04] (c.f. Sect. 5.3 and Chap. 3).

Esther enables the user to enter expressions (using a subset of Clean) that are dynamically typed, and transformed into values and functions using compiled code. It is also possible to reuse earlier created functions, which are stored on disk. Its implementation relies on the *dynamic type system* [ACP+92, Pil99, VP03] of Clean.

The shell uses a text-based interface, and hence it makes sense to create a special *string*-editor (Sect. 5.3.2), which converts any string into the corresponding dynamically typed value. This special editor has the same power as the Esther command interpreter and can deliver any dynamic value, including higher-order polymorphic functions.

5.3.1 Dynamics in Clean

A *dynamic* is a value of static type `Dynamic`, which contains an expression as well as a representation of its static type, e.g., **dynamic** `42 :: Int`, **dynamic** `map fst :: ∀a b: [(a, b)] → [a]`. Basically, dynamic types turn every (first and higher-order) data structure into a first-order structure, while providing run-time access to the original type and value.

Function alternatives and case patterns can match on values of type `Dynamic`. Such a pattern match consists of a value pattern and a type pattern, e.g., `[4, 2] :: [Int]`. The compiler translates a pattern match on a type into a run-time type-unification. If the unification is successful, type variables in a type pattern are bound to the offered type. Applying dynamics at run-time will be used to create an editor that changes according to the type of entered expressions (Sect. 5.3.2, Example 2).

```
dynamicApply :: Dynamic Dynamic → Dynamic
dynamicApply (f :: a → b) (x :: a) = dynamic f x     :: b
dynamicApply     df         dx     = dynamic "Error" :: String
```

dynamicApply tests whether the argument type of the function f, inside its first argument, can be unified with the type of the value x, inside the second argument. dynamicApply can safely apply f to x, if the type pattern match succeeds. It yields a value of the type that is bound to the type variable b by unification, wrapped in a dynamic. If the match fails, it yields a string in a dynamic.

Type variables in type patterns can also relate to type variables in the static type of a function. A ^ behind a variable in a pattern associates it with the same type variable in the static type of the function.

```
matchDynamic :: Dynamic → t | TC t
matchDynamic (x :: t^) = x
```

The static type variable t, in the example above, is determined by the static context in which it is used, and imposes a restriction on the actual type that is accepted at run-time by matchDynamic. The function becomes overloaded in the predefined TC (type code) class. This makes it a type dependent function [Pil99].

The dynamic run-time system of Clean supports writing dynamics to disk and reading them back again, possibly in another program or during another execution of the same program. This provides a means of type safe communication, the ability to use compiled plug-ins in a type safe way, and a rudimentary basis for mobile code. The dynamic is read in lazily after a successful run-time unification. The amount of data and code that the dynamic linker links is therefore determined by the evaluation of the value inside the dynamic.

```
writeDynamic :: String Dynamic env → (Bool,env) | FileSystem env
readDynamic  :: String env → (Bool,Dynamic,env) | FileSystem env
```

Applying dynamics at run-time will be used to create an editor that changes according to the type of entered expressions (Sect. 5.3.2, Example 2).

Programs, stored as dynamics, have Clean types and can be regarded as a typed file system. We have shown that dynamicApply can be used to type check any function application at run-time using the static types stored in dynamics. Combining both in an interactive ‘read expression – apply dynamics – evaluate and show result’ loop, already gives a simple shell that supports the type checked run-time application of programs to documents. The composeDynamic function below, taken from the Esther shell, applies dynamics and infers the type of an expression.

```
composeDynamic   :: String env → (Dynamic,env) | FileSystem env
showValueDynamic :: Dynamic → String
```

Applying `composeDynamic` to *expr env* parses *expr*. Unbound identifiers in *expr* are resolved by reading them from the file system. Additionally, overloading is resolved. Using the parse tree of *expr* and the resolved identifiers, the `dynamicApply` function is used to construct the (functional) value *v and* its type τ. These are packed in a **`dynamic`** $v :: \tau$ and returned by `composeDynamic`. In other words, **if** $env \vdash expr :: \tau$ and $[\![expr]\!]_{env} = v$ **then** `composeDynamic` *expr env* = $(v :: \tau, env)$. The `showValueDynamic` function yields a string representation of the value inside a dynamic.

5.3.2 Creating a GEC for the type Dynamic

With the `composeDynamic` function, an editor for dynamics can easily be constructed. This function needs an appropriate environment to access the dynamic values and functions (plug-ins) that are stored on disk. The standard `(PSt ps)` environment used by the generic `gGEC` function (Sect. 5.2) is such an environment. This means that we can simply use `composeDynamic` in a specialized editor to offer the same functionality as the command line interpreter. Instead of Esther's console, we use a `String` editor as interface to the application user. Additionally, we need to convert the provided string into the corresponding dynamic. We therefore define a composite data type `DynString` and a specialized `gGEC`-editor for this type (a $GEC_{\texttt{DynString}}$) that performs the required conversions.

```
:: DynString = DynStr Dynamic String
```

The choice of the composite data type is motivated mainly by simplicity and convenience: the string can be used by the application user for typing in the expression. It also stores the original user input, which cannot be extracted from the dynamic when it contains a function.

Now we specialize `gGEC` for this type `DynString`. The complete definition of `gGEC{|DynString|}` is given below.

```
gGEC{|DynString|} (gui,DynStr _ expr,dynStringUpdate) env
   #8 (stringGEC,env) = gGEC{|*|} (gui,expr,stringUpdate
                                              dynStringUpdate) env
   = ({ gecSetValue = dynSetValue stringGEC.gecSetValue
      , gecGetValue = dynGetValue stringGEC.gecGetValue }, env)
where
      dynSetValue stringSetValue (DynStr _ expr) env
         = stringSetValue expr env
      dynGetValue stringGetValue env
         # (nexpr,env) = stringGetValue env
```

[8]This is Clean's 'do-notation' for environment passing.

```
          # (ndyn, env) = composeDynamic nexpr env
          = (DynStr ndyn nexpr,env)
       stringUpdate dynStringUpdate nexpr env
          # (ndyn,env) = composeDynamic nexpr env
          = dynStringUpdate (DynStr ndyn nexpr) env
```

The created $GEC_{\texttt{DynString}}$ displays a box for entering a string by calling the standard generic gGEC{|*|} function for the value expr of type String, yielding a stringGEC. The DynString-editor is completely defined in terms of this String-editor. It only has to take care of the conversions between a String and a DynString. This means that its gecSetValue method dynSetValue simply sets the string component of a new DynString in the underlying String-editor. Its gecGetValue method dynGetValue retrieves the string from the String-editor, converts it to the corresponding Dynamic by applying composeDynamic, and combines these two values in a DynString-value. When a new string is created by the application user, the callback function stringUpdate is evaluated, which invokes the callback function dynStringUpdate (provided as an argument upon creation of the DynString-editor), after converting the String to a DynString.

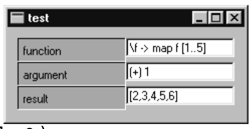
It is convenient to define a constructor function mkDynStr that converts any input *expr*, which has value v of type τ, into a value of type DynString guaranteeing that if $v :: \tau$ and $[\![expr]\!] = v$, then (DynStr (v::τ) *expr*) :: DynString.

```
mkDynStr :: a → DynString | TC a
mkDynStr x = let dx = dynamic x
             in DynStr dx (showValueDynamic dx)
```

Example 2:

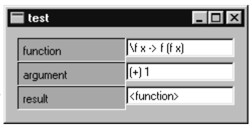
We construct an interactive editor that can be used to test functions. It can be a newly defined function, say λx → x^2, or any existing function stored on disk as a Dynamic. Hence, the tested function can vary from a small function, say factorial, to a large complete application.

```
:: MyRecord = { function :: DynString
              , argument :: DynString
              , result   :: DynString }
myEditor = selfGEC "test" guiApply (initval id 0)
where
   initval f v = { function = mkDynStr f
                 , argument = mkDynStr v
                 , result   = mkDynStr (f v) }
   guiApply  r=:[9]{ function = DynStr (f::a → b) _
                   , argument = DynStr (v::a)     _ }
               = {r &[10] result = mkDynStr (f v)}
   guiApply  r = r
```



test

function	\f -> map f [1..5]
argument	(+) 1
result	[2,3,4,5,6]



test

function	\f x -> f (f x)
argument	(+) 1
result	<function>

The type `MyRecord` is a record with three fields, `function`, `argument`, and `result`, all of type `DynString`. The user can use this editor to enter a function definition and its argument. The `selfGEC` function will ensure that each time a new string is created with the editor `test`, the function `guiApply` is applied that provides a new value of type `MyRecord` to the editor. The function `guiApply` tests, in a similar way as the function `dynamicApply` (see Sect. 5.3.1), whether the type of the supplied function and argument match. If so, a new result is calculated. If not, nothing happens.

This editor can only be used to test functions with one argument. What happens if we edit the function and the argument in such a way that the result is not a plain value but a function itself? Take, e.g., as function the twice function `λf x → f (f x)`, and as argument the increment function `((+) 1)`. Then the result is also a function `λx → ((+) 1) ((+) 1 x)`. The editor displays `<function>` as result. There is no way to pass an argument to the resulting function.

With an editor like the one above, the user can enter expressions that are automatically converted into the corresponding `Dynamic` value. As in the shell, unbound names are expected to be dynamics on disk. Illegal expressions result in a `Dynamic` containing an error message.

To have a properly higher-order dynamic application example, one needs an editor in which the user can type in functions of arbitrary arity, and subsequently enter arguments for this function. The result is then treated such that, if it is a function, editors are added dynamically for the appropriate number of arguments. This is explained in the following example.

[8] $x =: e$ binds x to e.

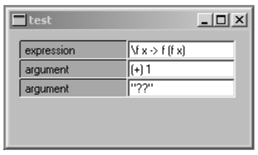
[10] $\{r$ & f_0=v_0, ..., f_n=$v_n\}$ is a record equal to r, except that fields f_i have value v_i.

Example 3:

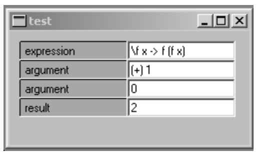
We construct a test program that accepts arbitrary expressions and adds the proper number of argument editors, which again can be arbitrary expressions. The number of arguments cannot be statically determined and has to be recalculated each time a new value is provided. Instead of an editor for a record, we therefore create an editor for a list of tuples. Each tuple consists of a string used to prompt to the user, and a `DynString`-value. The tuple elements are displayed below each other using the predefined list editor `vertlistAGEC` and access operator `^^`, which will be presented in Sect. 5.4.1. The `selfGEC` function is used to ensure that each change made with the editor is tested with the `guiApply` function and the result is shown in the editor.

```
myEditor = selfGEC "test" (guiApply o (^^))
    (vertlistAGEC [show "expression " 0])
where
  guiApply [f=:(_,(DynStr d _)):args]
   = vertlistAGEC [f:check (fromDynStr d) args]
  where
    check (f::a → b) [arg=:(_,DynStr (x::a) _):args]
          = [arg : check (dynamic f x) args]
    check (f::a → b) _ = [show "argument " "??"]
    check (x::a)      _ = [show "result "   x]

  show s v = (Display s,mkDynStr v)
```


test

expression	\f x -> f (f x)
argument	(+) 1
argument	"??"


test

expression	\f x -> f (f x)
argument	(+) 1
argument	0
result	2

The key part of this example is formed by the function `check`, which calls itself recursively on the result of the dynamic application. As long as function and argument match, and the resulting type is still a function, it will require another argument, which will be checked for type consistency. If function and argument do not match, `??` is displayed, and the user can try again. As soon as the resulting type is a plain value, it is evaluated and shown using the data constructor `Display`, which creates a non-editable editor that just displays its value. With this editor, any higher-order polymorphic function can be entered and tested.

5.4 Statically Typed Higher-order *GEC*s

The editors presented in the previous section are flexible because they deliver a `Dynamic` (packed into the type `DynString`). They have the disadvantage that the programmer has to program a check, such as the `check` function in the previous example, on the type consistency of the resulting `Dynamics`.

In many applications, it is statically known what the type of a supplied function must be. In this section, we show how the run-time type check can be replaced by a compile-time check, using the abstraction mechanism for *GEC*s. This gives us a second solution for higher-order data structures that is statically typed, which allows therefore type-directed generic GUI creation.

5.4.1 Abstract Graphical Editor Components

The generic function gGEC derives a GUI for its instance type. Because it is a function, the appearance of the GUI is completely determined by that type. This is in some cases much to rigid. One cannot use different visual appearances of the same type within a program. For this purpose *abstract GEC*s (*AGEC*) [AvEP04a] have been introduced. An instance of gGEC for *AGEC* has been defined. Therefore, an $AGEC_{\mathtt{d}}$ can be used as a $GEC_{\mathtt{d}}$, i.e., it behaves as an editor for values of a certain *domain*, say of type d. However, an $AGEC_{\mathtt{d}}$ never displays nor edits values of type d, but rather a *view* on values of this type, say of type v. Values of type v are shown and edited, and internally converted to the values of domain d. The view is again generated automatically as a $GEC_{\mathtt{v}}$. To makes this possible, the ViewGEC d v record is used to define the relation between the domain d and the view v.

```
:: ViewGEC d v
   = { d_val        :: d                       // initial domain value
     , d_oldv_to_v :: d →(Maybe v)→ v// convert domain value to view value
     , update_v     :: v → v                   // correct view value
     , v_to_d       :: v → d }                 // convert view value to domain value
```

It should be noted that the programmer does not need to be knowledgeable about Object I/O programming to construct an $AGEC_{\mathtt{d}}$ with a view of type v. The specification is only in terms of the involved data domains. The complete interface to *AGEC*s is given below.

```
:: AGEC d                                        // abstract data type
mkAGEC        :: (ViewGEC d v) → AGEC d | gGEC{|*|} v
(^^)          :: (AGEC d) → d                    // Read current domain value
(^=) infixl :: (AGEC d) d → AGEC d               // Set new domain value
```

The ViewGEC record can be converted to the abstract type AGEC, using the function mkAGEC above. Because *AGEC* is an abstract data type we need access functions to read (^^) and write (^=) its current value. *AGEC*s allow us to define arbitrarily many editors $gec_{\mathtt{i}}$ :: $AGEC_{\mathtt{d}}$ that have a private implementation of type $GEC_{\mathtt{v_i}}$. Because *AGEC* is abstract, code that has been written for editors that manipulates some type containing $AGEC_{\mathtt{d}}$, does not change when the value of type $AGEC_{\mathtt{d}}$

is exchanged for another *AGEC*$_{\mathtt{d}}$. This facilitates experimenting with various designs for an interface without changing any other code.

We built a collection of functions creating abstract editors for various purposes. Below, we summarize only those functions of the collection that are used in the examples in this chapter:

```
vertlistAGEC :: [a] → AGEC [a]  | gGEC{|*|} a  // all elements in a column
counterAGEC  ::  a  → AGEC  a   | gGEC{|*|}, IncDec a  // a counter
hidAGEC      ::  a  → AGEC  a                  // identity, no editor
displayAGEC  ::  a  → AGEC  a   | gGEC{|*|} a  // identity, non-editable
```

The *counter* editor below is a typical member of this library.

Example 4:

```
counterAGEC :: a → AGEC a | gGEC{|*|}, IncDec a
counterAGEC j = mkAGEC { d_val=j,d_oldv_to_v=λi _ → (i,Neutral)
                       , update_v=updateCounter,v_to_d=fst      }
where updateCounter (n,UpPressed)   = (n+one,Neutral)
      updateCounter (n,DownPressed) = (n-one,Neutral)
      updateCounter (n,Neutral)     = (  n  ,Neutral)
```

A programmer can use the counter editor as an integer editor, but because of its internal representation it presents the application user with an edit field combined with an up-down, or spin, button. The `updateCounter` function is used to synchronize the spin button and the integer edit field. The right part of the tuple is of type `UpDown` (Sect. 5.2), which is used to create the spin button.

5.4.2 Adding Static Type Constraints to Dynamic GECs

The abstraction mechanism provided by *AGEC*s is used to build type-directed editors for higher-order data structures, which check the type of the entered expressions dynamically. These statically typed higher-order editors are created using the function `dynamicAGEC`. The full definition of this function is specified and explained below.

```
dynamicAGEC :: d → AGEC d | TC d
dynamicAGEC x = mkAGEC {d_val=x           ,d_oldv_to_v=toView
                       ,update_v=updView x,v_to_d=fromView x }
where
    toView newx Nothing  = let dx = mkDynStr newx
                               in (dx,hidAGEC dx)
    toView _ (Just oldx) = oldx
```

```
fromView :: d (DynString,AGEC DynString) → d | TC d
fromView _ (_,oldx) = case ^^oldx of DynStr (x::d^) _ → x

updView :: d (DynString,AGEC DynString)
          → (DynString,AGEC DynString) | TC d
updView _ (newx=:(DynStr (x::d^) _),_) = (newx,hidAGEC newx)
updView _ (_,oldx)                     = (^^oldx,oldx)
```

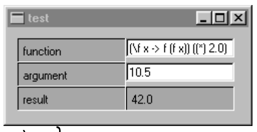
The abstract Dynamic editor, which is the result of the function dynamicAGEC initially takes a value of some statically determined type d. It converts this value into a value of type DynString, such that it can be edited by the application user as explained in Sect. 5.3.2. The application user can enter an expression of arbitrary type, but now it is ensured that only expressions of type d are approved.

The function updView, which is called in the abstract editor after any edit action, checks, using a type pattern match, whether the newly created dynamic can be unified with the type d of the initial value (using the ^-notation in the pattern match as explained in Sect. 5.3.1). If the type of the entered expression is different, it is rejected[11] and the previous value is restored and shown. To do this, the abstract editor has to remember the previously accepted correctly typed value. Clearly, we do not want to show this part of the internal state to the application user. This is achieved using the abstract editor hidAGEC (Sect. 5.4.1), which creates an invisible editor, i.e., a store, for any type.

Example 5:

Consider the following variation of Example 2:

```
:: MyRecord a b = { function :: AGEC (a → b)
                  , argument :: AGEC a
                  , result   :: AGEC b }
myEditor = selfGEC "test" guiApply (initval ((+) 1.0) 0.0)
where
   initval f v = { function = dynamicAGEC f
                 , argument = dynamicAGEC v
                 , result   = displayAGEC (f v) }
   guiApply myrec=:{ function = af, argument = av }
      = {myrec & result = displayAGEC ((^^af) (^^av))}
```



function	(\f x -> f (f x)) ((*) 2.0)
argument	10.5
result	42.0

The editor above can be used to test functions of a certain statically determined type. Due to the particular choice of the initial values ((+) 1.0 :: Real → Real

[11]There is currently no feedback on *why* the type is rejected. Generating good error messages as in [HJSA02] certainly improves the user interface.

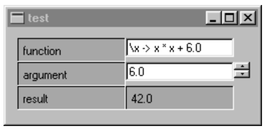
and `0.0` :: `Real`), the editor can only be used to test functions of type `Real` → `Real` applied to arguments of type `Real`. Notice that it is now statically guaranteed that the provided dynamics are correctly typed. The `dynamicAGEC`-editors take care of the required checks at run-time and they reject ill-typed expressions. The programmer therefore does not have to perform any checks anymore. The abstract `dynamicAGEC`-editor delivers a value of the proper type just like any other abstract editor.

The code in the above example is not only simple and elegant, but it is also very flexible. The `dynamicAGEC` abstract editor can be replaced by any other abstract editor, provided that the statically derived type constraints (concerning `f` and `v`) are met. This is illustrated by the next example.

Example 6:

If one prefers a counter as input editor for the argument value, one only has to replace `dynamicAGEC` by `counterAGEC` in the definition of `initval`:

```
initval f v = { function = dynamicAGEC f
              , argument = counterAGEC v
              , result = displayAGEC (f v) }
```


test

function	\x -> x * x + 6.0
argument	6.0
result	42.0

The `dynamicAGEC` is typically used when *expression* editors are preferred over *value* editors of a type, and when application users need to be able to enter functions of a statically fixed monomorphic type.

One can create an editor for any higher-order data structure τ, even if it contains polymorphic functions. It is required that all higher-order parts of τ are abstracted, by wrapping them with an *AGEC* type. Basically, this means that each part of τ of the form `a` → `b` must be changed into `AGEC (a` → `b)`. For the resulting type τ' an edit dialog *can* be automatically created, e.g., by applying `selfGEC`. However, the initial value that is passed to `selfGEC` must be monomorphic, as usual for any instantiation of a generic function. Therefore, editors for polymorphic types cannot be created automatically using this statically typed generic technique. As explained in Sect. 5.3.2 polymorphic types can be handled with dynamic type-checking.

5.5 Applications of higher-order *GEC*s

The ability to generate editors for higher-order data structures greatly enhances the applicability of *GEC*s. Firstly, it becomes possible to create applications in which functions can be edited as part of a complex data structure. Secondly, these functions can be composed dynamically from earlier created compiled functions on disk. Both are particular useful for rapid prototyping purposes, as they can add real-life functionality.

In this section, we discuss one small and one somewhat larger application. Even the code for the latter application is still rather small (just a few pages). The code is omitted in this chapter due to space limitations, but it can be found at the GEC Toolkit website[12]. Screen shots of the running applications are given in Sect. 5.8.

An Adaptable Calculator.

In the first example, we use *GEC* to create a 'more or less' standard calculator. The default look of the calculator was adapted using the aforementioned *AGEC* customization techniques. Special about this calculator is that its functionality can be easily extended at run-time: the application user can add his or her own buttons with a user-defined functionality. In addition to the calculator editor, a *GEC* editor is created, which enables the application user to maintain a list of button definitions consisting of button names with corresponding functions. Since the type of the calculator functions are statically known, a statically typed higher-order *GEC* is used in this example. The user can enter a new function definition using a lambda expression, but it is also possible to open and use an earlier created function from disk. Each time the list is changed with the list editor, the calculator editor is updated and adjusted accordingly. For a typical screen shot see Fig. 5.1.

A Form Editor.

In the previous example, we have shown that one can use one editor to change the look and functionality of another. This principle is also used in a more serious example: the form editor. The form editor is an editor with which electronic forms can be defined and changed. This is achieved using a meta-description of a form. This meta-description is itself a data structure, and therefore, we can generate an editor for it. One can regard a form as a dedicated spreadsheet, and with the form editor, one can define the actual shape and functionality of such a spreadsheet. With the form editor, one can create and edit fields. Each field can be used for a certain purpose. It can be used to show a string, it can be used as editor for a value of a certain basic type, it can be used to display a field in a certain way by assigning an abstract editor to it (e.g., a counter or a calculator), and it can be used to calculate and show new values depending on the contents of other fields. For this purpose, the application user has to be able to define functions that have the contents of other fields as arguments. The form editor uses a mixed mode strategy. The contents of some fields can be statically determined (e.g., a field for editing

[12]`http://clean.cs.ru.nl/gec`

an integer value). However, the form editor can only dynamically check whether the argument fields of a specified function are indeed of the right type. The output of the form editor is used to create the actual form in another editor, which is part of the same application. By filling in the form fields with the actual value, the application user can test whether the corresponding form behaves as intended. For a typical screen shot see Fig. 5.2.

5.6 Related Work

In the previous sections we have shown that we can create editors that can deal with higher order data structures. We can create dynamically typed higher-order editors, which have the advantages that we can deal with polymorphic higher order data structures and overloading. This has the disadvantage that the programmer has to check type safety in the editor. The compiler can ensure type correctness of higher-order data structures in statically typed editors, but they can only edit monomorphic types. Related work can be sought in three areas:

Grammars instead of types:

Taking a different perspective on the type-directed nature of our approach, one can argue that it is also possible to obtained editors by starting from a grammar specification instead of a type. Such toolkits require a grammar as input and yield an editor GUI as result. Projects in this flavor are for instance the recent *Proxima* project [Sch04], which relies on *XML* and its *DTD* (Document Type Definition language), and the *Asf+Sdf Meta-Environment* [vdBvDH$^+$01] which uses an *Asf* syntax specification and *Sdf* semantics specification. The major difference with such an approach is that these systems need both a grammar and some kind of interpreter. In our system higher-order elements are immediately available as a functional value that can be applied and passed to other components.

GUI programming toolkits:

From the abstract nature of the *GEC* toolkit, it is clear that we need to look at GUI toolkits that also offer a high level of abstraction. Most GUI toolkits are concerned with the low-level management of widgets in an imperative style. One well-known example of an abstract, compositional GUI toolkit based on a combinator library is *Fudgets* [CH93]. These combinators are required for plumbing when building complex GUI structures from simpler ones. In our system far less plumbing is needed. Most work is done automatically by the generic function `gGEC`. The only plumbing needed in our system is for combining the *GEC*-editors themselves.

Furthermore, the Fudget system does not provide support for editing function values or expressions.

Because a *GEC*$_{\mathtt{t}}$ is a `t`-stateful object, it makes sense to have a look at object-oriented approaches. The power of abstraction and composition in our functional framework is similar to *mixin*s [FKF98] in object-oriented languages. One can imagine an OO GUI library based on compositional and abstract mixins in order to obtain a similar toolkit. Still, such a system lacks higher-order data structures.

Visual programming languages:

Due to the extension of the *GEC* programming toolkit with higher-order data structures, *visual programming languages* have come within reach as *application domain*. One interesting example is the *Vital* system [Han02] in which Haskell-like scripts can be edited. Both systems allow direct manipulation of expressions and custom types, allow customization of views, and have guarded data types (like the `selfGEC` function). In contrast with the Vital system, which is a dedicated system and has been implemented in Java, our system is a general purpose toolkit. We could use our toolkit to construct a visual environment in the spirit of Vital.

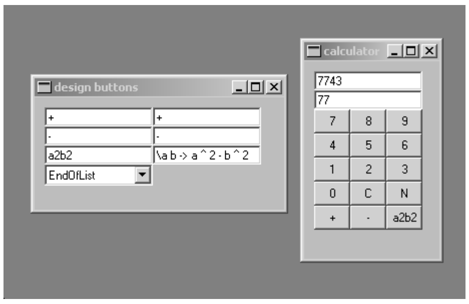
5.7 Conclusions

With the original *GEC*-toolkit one can construct GUI applications without much programming effort. This is done on a high level of abstraction, in a fully compositional manner, and type-directed. It can be used for any monomorphic first-order data type. In this chapter, we have shown how the programming toolkit can be extended in such a way that *GEC*s can be created for *higher-order* data structures. We have presented two methods, each with its own advantage and disadvantage.

We can create an editor for higher-order data structures using dynamic typing, which has as advantage that it can deal with polymorphism and overloading, but with as disadvantage that the programmer has to ensure type safety at run-time. We can create a editor for higher-order data structures using the static typing such that type correctness of entered expressions or functions is guaranteed at compile-time. In that case, we can only cope with monomorphic types, but we can generate type-directed GUIs automatically.

As a result, applications constructed with this toolkit can manipulate the same set of data types as modern functional programming languages can. The system is type-directed and type safe, as well as the GUI applications that are constructed with it.

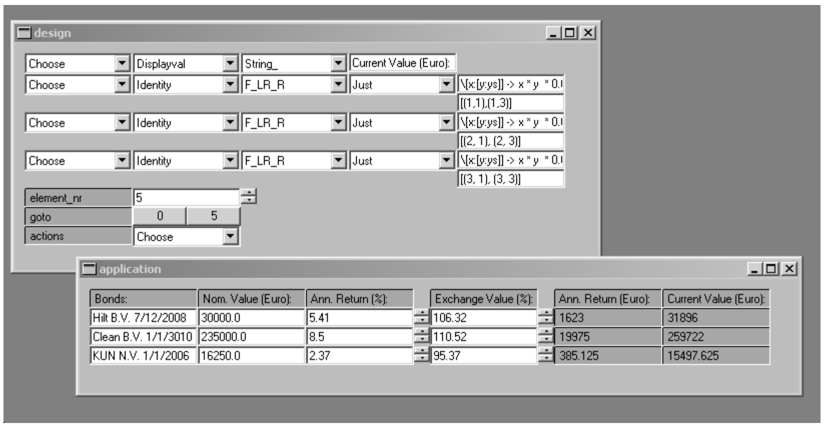
5.8 Screen Shots of Example Applications



+	+
-	-
a2b2	\a b -> a ^ 2 - b ^ 2
EndOfList	

7743		
77		
7	8	9
4	5	6
1	2	3
0	C	N
+	-	a2b2

Figure 5.1: A screen shot of the adaptable calculator. Left the editor for defining button names with the corresponding function definitions. Right the resulting calculator editor.



Choose	Displayval	String_	Current Value (Euro):	
Choose	Identity	F_LR_R	Just	\[x:[y:ys]] -> x * y * 0.(
				[(1,1),(1,3)]
Choose	Identity	F_LR_R	Just	\[x:[y:ys]] -> x * y * 0.(
				[(2, 1), (2, 3)]
Choose	Identity	F_LR_R	Just	\[x:[y:ys]] -> x * y * 0.(
				[(3, 1), (3, 3)]

element_nr	5	
goto	0	5
actions	Choose	

Bonds:	Nom. Value (Euro):	Ann. Return (%):	Exchange Value (%):	Ann. Return (Euro):	Current Value (Euro):
Hilt B.V. 7/12/2008	30000.0	5.41	106.32	1623	31896
Clean B.V. 1/1/3010	235000.0	8.5	110.52	19975	259722
KUN N.V. 1/1/2006	16250.0	2.37	95.37	385.125	15497.625

Figure 5.2: A screen shot of the form editor. The form editor itself is shown in the upper left window; the corresponding editable spreadsheet-like form is shown in the other.

CHAPTER 6

Polytypic Syntax Tree Operations

Arjen van Weelden
Sjaak Smetsers
Rinus Plasmeijer

6.1 Introduction

The construction of complex software often starts by designing suitable data types to which functionality is added. Some functionality is data type specific, other functionality only depends on the structure of the data type. Polytypic programming is considered an important technique to specify such generic functionality. It enables the specification of functions on the structure of data types, and therefore, it is characterized as type dependent (type indexed) programming. The requested overall functionality is obtained by designing your data types such that they reflect the separation of specific and generic functionality. By overruling the polytypic instantiation mechanism for those parts of the data type that correspond to specific functionality, one obtains the desired overall behavior. In essence, a programmer only has to *program the exception* and a small polytypic scheme, since polytypic functions automatically work for the major part of the data types. Examples of such generic operations are equality, traversals, pretty-printing, and serialization.

The number of such generic operations in a specific program can be quite small, and hence the applicability of polytypic programming seems limited. Polytypic functions that are data specific only make sense if the involved data types themselves are complex or very big. Otherwise, the definition of the polytypic version of an operation requires more effort than defining this operation directly. Moreover, the data-dependent functionality should be restricted to only a small portion of the data type, while the rest can be treated generically.

This chapter investigates the suitability of polytypic programming as a general programming tool, by applying it to (a part of) compiler construction. Compilers involve both rich data structures and many, more or less complex, operations on those data structures. We focus on the front-end of compilers: parsing, post-parsing, and type inference operations on the syntax tree. There exist many special tools, e.g., parser generators and compiler compilers, which can be used for constructing such a front-end. We show that polytypic programming techniques can be used to elegantly specify parsers. This has the advantage that the polytypic functional compiler can generate most of the code. Another advantage is that one can specify everything in the functional language itself, without synchronization issues between the syntax tree type and an external grammar definition.

We have implemented polytypic parsers in both Generic Haskell [LCJ03] (a preprocessor for Haskell [Pey03]) and Clean [AP03]. We use (Generic) Haskell to present our implementation in this chapter; the Clean code is very similar. The polytypic parser we use in this chapter differs from those commonly described in papers on polytypic programming [JJ99, JJ02a]. Our parser is based on the *types–as–grammars* approach: the context-free syntax of the language to parse is specified via appropriate data type definitions. The types–as-grammar approach was previously used to construct a new version of the Esther shell originally described in Weelden and Plasmeijer [vWP03]. The shell uses polytypic programming to specify the parser and post-parsing operations on expressions the size of a single command-line. This chapter tackles larger inputs and grammars, including the Haskell syntax.

Apart from its expressiveness, a programming technique is not very useful if the performance of the generated code is inadequate. The basic code generation schema used in the current implementations of polytypic systems produces inefficient code. We asses the efficiency of both the Generic Haskell and the Clean implementations and compare them with the code generated by an optimization tool by Alimarine and Smetsers [AS05]. This tool takes a polytypic Clean program as input and produces a Haskell/Clean-like output without the polytypic overhead.

To summarize, the main contributions of this chapter are:

- We show that polytypic programming, introduced in Sect. 6.2, is not only suited for defining more or less inherently generic operations, but also for specifying data specific functionality.
- We describe a technique that allows us to derive a parser for context-free languages automatically from the definition of a syntax tree in Sect. 6.3. The technique is based on the idea to interpret types as grammar specifications.

- We show that the same technique applies to several related syntax tree operations in Sect. 6.4. As operations become more data specific, we gain less from using polytypic programming. However, we show that it is not totally unsuitable for non-generic algorithms.
- As most polytypic programmers know, polytypic programs (including our parsers) have serious performance problems. Fortunately, we show in Sect. 6.5 that an appropriate optimization tool recovers a lot of efficiency, and that our parsers can approach the speed of parsers generated by external tools.

Related work is discussed in Sect. 6.6, and we conclude in Sect. 6.7.

6.2 Polytypic Programming

Specifying polytypic functions is a lot like defining a type class and its instances. The main difference is that a polytypic compiler can derive most of the instances automatically, given a minimal fixed set of instances for three or four (generic) types. The programmer can always overrule the derived instance for a certain type by specifying the instance manually. This powerful derivation scheme even extends to kinds (the types of types), which we will neither use nor explain in this chapter.

The fact that polytypic functions can be derived for most types is based on the observation that any (user-defined) algebraic data type can be expressed in terms of eithers, pairs, and units. This generic representation developed by Hinze [Hin00a]. It is encoded in Generic Haskell by the following Haskell types:

```
data Sum a b = Inl a | Inr b — either/choice between (In)left and (In)right
data Prod a b = a :*: b      — pair/product of two types, left associative
data Unit = Unit             — the unit type
```

A data type and its generic representation are isomorphic. The corresponding isomorphism can be specified as a pair of conversion functions. E.g., for lists the generic representation and automatically generated associated conversion functions are as follows.

```
type GenericList a = Sum (Prod a [a]) Unit

fromList :: [a] → GenericList a    toList :: GenericList a → [a]
fromList (x:xs) = Inl (x :*: xs)   toList (Inl (x :*: xs)) = x:xs
fromList []     = Inr Unit         toList (Inr Unit)       = []
```

Note that the generic representation type `GenericList` is not recursive and still contains the original list type. A polytypic function instance for the list type can be constructed by the polytypic system using the generic representation. The derived instance for the list type uses the given instances for `Sum`, `Prod`, `Unit`, and once again the currently deriving instance for lists. This provides the recursive call, which one would expect for a recursive type such as lists.

To define a polytypic function, the programmer has to specify its function type, similar to a type class, and only the instances for the generic types (`Prod`, `Sum`, and `Unit`) and non-algebraic types (like `Int` and `Double`). The polytypic instances for other types that are actually used inside a program are automatically derived. Polytypic functions are therefore most useful if a large collection of (large) data types is involved, or if the types change a lot during development.

To illustrate polytypic programming we use the following syntax tree excerpt:

```
data Expr = Apply Expr Expr               | Lambda Pattern Expr
          | Case Expr [(Pattern, Expr)]   | Variable String
          | If Expr Expr Expr             | ···

data Pattern = Var String | Constructor String [Pattern] | ···

data ···
```

We define a Generic Haskell function `print` of type `a → String` that is polytypic in the type variable `a`, similar to Haskell's `show :: Show a ⇒ a → String` that is overloaded in `a`. Instead of instances for the `Show` class, we define type instances for `print` using the special parentheses {| |}.

```
print{|a|} :: a → String

print{|Int|}       i          = show i          — basic type instance

print{|Unit|}      Unit       = ""              — unit instance

print{|Sum a b|} (Inl l)      = print{|a|} l    — left either instance
print{|Sum a b|} (Inr r)      = print{|b|} r    — right either instance

print{|Prod a b|} (l :*: r)                     — pair instance
        = print{|a|} l ++ " " ++ print{|b|} r

print{|Con d a|} (Con x)                        — instance for constructors
        = "(" ++ conName d ++ " " print{|a|} x ++ ")"
```

To print the parameterized type `Sum` and `Prod`, `print` requires printing functions for the parameter types `a` and `b`. These are automatically passed under the hood by Generic Haskell, similar to dictionaries in the case of overloading.

print{|Sum a b|} can refer to these hidden dictionary functions using print{|a|} and print{|b|}. Furthermore, the type Con, used in this example, was added to the set of generic types in Generic Haskell as well as in Clean. Run-time access to some information about the original data constructors is especially convenient when writing trace functions, such as print, for debugging purposes.

```
data Con a = Con a;    data ConDescr = { conName :: String, ··· }
```

When used in Generic Haskell, Con *appears* to get an additional argument d. This is not a type argument but a denotation that allows the programmer to access information about the constructor, which is of type ConDescr. In the example print{|Con d a|} applies conName to d to retrieve the name of the constructor.

Observe that this polytypic print function does not depend on the structure of the syntax tree type. If this type definition changes during development, the underlying system will automatically generate a proper version of print. This implementation of print is quite minimal, with superfluous parentheses and spaces. It is easy to adjust the definition to handle these situations correctly, see for example Jansson and Jeuring [JJ99].

It is not difficult to specify the polytypic inverse of the print function. Using a monadic parser library, with some utility functions such as symbol(s) and parseInt that take care for low-level token recognition, one could specify a polytypic parse function (similar to Haskell's read) as follows:

```
type Parser a = ···  — some monadic parser type

parse{|a|} :: Parser a

parse{|Unit|}     = return Unit

parse{|Sum a b|}  = mplus (parse{|a|} >>= return . Inl)
                          (parse{|b|} >>= return . Inr)

parse{|Prod a b|} = do    l ← parse{|a|}
                          r ← parse{|b|}
                          return (l :*: r)

parse{|Con d a|}  = do    symbol '('
                          symbols (conName d)
                          symbol ' '
                          x ← parse{|a|}
                          symbol ')'
                          return (Con x)
parse{|Int|}      = parseInt
```

Such a simple parser follows the print definition very closely and is easy to understand. `parse` is obviously `print`'s inverse, and it can only parse input generated by the print function, including redundant spaces and parentheses.

6.3 Polytypic Parsing of Programming Languages

This section introduces the types–as–grammar approach to polytypically derive a parser. This parser builds on a small layer of monadic parser combinators, to abstract from the lower level token recognition machinery. We use very naive parser combinators (shown below) because they are easy to use and explain. To abstract from the parsing issues at the lexical level, we assume a separated scanner/lexer and that the parser will work on a list of tokens. Later in Sect. 6.5, we will test the efficiency of the polytypic parser using also a set of continuation parser combinators that improve the error messages. The naive monadic parser, using the `Maybe` monad, is implemented as follows.

```
newtype Parser a = Parser {parser :: [Token] → Maybe (a,[Token])}

data Token = IdentToken String
           | LambdaToken | ArrowToken
           | IfToken | ThenToken | ElseToken
           | ···                                      — all tokens

token :: Token → Parser Token
token tok = Parser (λts → case ts of
                           (t:ts') | t == tok → Just (t, ts')
                            _                 → Nothing

instance Monad Parser where
    return x = Parser (λts → Just (x, ts))          — success parser
    l >>= r  = Parser (λts → case parser l ts of    — sequence parser
                             Just (x, ts') → parser (r x) ts'
                             Nothing       → Nothing          )

instance MonadPlus Parser where
    mzero     = Parser (λts → Nothing)               — fail parser
    mplus l r = Parser (λts → case parser l ts of    — choice parser
                              Just (x, ts') → Just (x, ts')
                              Nothing       → parser r ts  )
```

The `mplus` instance above defines a deterministic (exclusive) choice parser: if the left argument of `mplus` parses successfully, the right argument is never tried. This is done out of speed considerations and, if the parsers are written in the right way, it does not matter for deterministic grammars. Algebraic data constructors have unique names, which makes the grammar deterministic. This is also reflected in the `Parser` type, i.e., the parser returns a `Maybe` result, which shows that it returns at most one result.

To parse real programming languages we should not parse the constructor names that occur in the syntax tree type. Instead, we should parse all kinds of tokens such as **`if`**, λ, and →. This requires writing most of the instances for the polytypic function `parse` by hand. Another option is adding these tokens to the abstract syntax tree, which becomes a non-abstract, or rich, syntax tree. Since we instruct the polytypic parser using types, we cannot reuse the (constructors of the) `Token` data type. Instead, we specify each token as a separate data type. This gives us the ability to parse our own tokens, without the constructors getting in the way. We can now define, for example, a nicer parser for lists that uses the `[]` and `_:_` notation.

```
data List a = Cons a ColonToken (List a)
            | Nil EmptyListToken

data ColonToken     = ColonToken
data EmptyListToken = EmptyListToken

parse{|ColonToken|}     = symbol ':'   ≫  return ColonToken
parse{|EmptyListToken|} = symbols "[]" ≫  return EmptyListToken

parse{|Con d a|}        = parser{|a|}  ≫= return . Con

intListParser = parse{|List Int|}     — automatically derived by the system
```

We partly reuse the `parse` definition from Sect. 6.2. We do not want to parse the constructor names. Therefore, we replace the `Parse{|Con d a|}` alternative from Sect. 6.2 with the one shown above. Not parsing constructor names means that the order of alternatives is important. Since `parse{|Sum a b|}` uses the exclusive `mplus`, it gives priority to the `Inl`(eft) alternative over the `Inr`(ight) alternative. Therefore, the textual order of the constructors of an algebraic data type determines the order of parsing, which is similar to function definitions with multiple alternatives in Haskell and Clean.

One can parse any context-free syntax by specifying the grammar using algebraic data types. The grammar below is an excerpt of a small functional programming language. It uses the convention that N*type* represents the non-terminal *type* and T*type* represents a terminal symbol *type*.

```
data Nexpression = Apply Nexpression Nexpression
                 | Lambda Tlambda Nvariable Tarrow Nexpression
                 | If Tif Nexpression Tthen Nexpression
                                        Telse Nexpression
                 | Variable Nvariable
                 | Value Nvalue

data Nvariable = Variable String

data Nvalue = Integer Int | Boolean Bool

data Tlambda = Tlambda; data Tarrow = Tarrow
data Tif     = Tif;     data Tthen  = Tthen; data Telse = Telse

parse{|Con d a|} = parse{|a|}      >>= return . Con
parse{|String|}  = identifierToken >>= λ(IdentToken s) → return s

parse{|Tlambda|} = token LambdaToken >> return Tlambda
parse{|Tarrow|}  = token ArrowToken  >> return Tarrow
parse{|Tif|}     = token IfToken     >> return Tif
parse{|Tthen|}   = token ThenToken   >> return Tthen
parse{|Telse|}   = token ElseToken   >> return Telse
```

If we remove all constructors from the type definitions above, we end up with something that looks very similar to the following grammar description in BNF notation:

```
<expression> ::= <expression> <expression>
              |  "λ" <variable> "→" <expression>
              |  "if" <expression> "then" <expression>
                                   "else" <expression>
              |  <variable>
              |  <value>

<variable>   ::= String

<value>      ::= Int | Bool
```

It is also easy to support extended BNF (EBNF) notation by introducing some auxiliary data types: Plus to mimic $(\cdots)^+$, Option to mimic $[\cdots]$, and Star to mimic $(\cdots)^{\star}$. The parsers for all of them can be derived automatically.

```
data Plus a      = Plus a (Plus a) | One a
type Star a      = Option (Plus a)
type Option a    = Maybe a

data Nexpression = ···
        | Lambda Tlambda (Plus Nvariable) Tarrow Nexpression
        | ···
```

The use of parameterized data types, such as `Plus`, can make the definition of the syntax tree type very concise. It is similar to two-level or van Wijngaarden grammars [vW65]. We can now specify a lambda expression with multiple arguments using `Plus` as shown above. Clearly, this corresponds to the following EBNF grammar:

```
        <expression> ::= ···
                       | "λ" <variable>+ "→" <expression>
                       | ···
```

An issue with this *types–as–grammar* approach is left-recursive type definitions. Most parser combinator libraries do not support left-recursive parser definitions and run out of heap or stack space. Recently, Baars and Swierstra developed parser combinators [BS04] that detect and remove left-recursion automatically . Our current solution is manually removing the (few occurrences of) left-recursion by splitting the left-recursive type, as shown below. Only `Nexpression` is (mutually) left-recursive because it has no argument of type `T`*token* before the `Nexpression` arguments. We write a small parser for the left-recursive part, making sure that most of the parser is still derived automatically.

```
data Nexpression = Apply Nexpression Nexpression
                 | Term Nterm             — separate non-recursive part

data Nterm = Lambda Tlambda (Plus Nvariable) Tarrow Nexpression
           | ···

parse{|Nexpression|} = parse{|Plus Nterm|} >>= return . app
 where
    app (One t)      = Term t
    app (Plus t ts) = app' (Term t) ts
    app' acc (One t)      = Apply acc t
    app' acc (Plus t ts) = app' (Apply acc t) ts
```

We extended this example to a basic functional language grammar, to test our generated parser. Moreover, as a larger test, we converted Haskell's grammar to types and derived a parser for it. The results of those tests appear in Sect. 6.5.

6.4 Other Polytypic Syntax Tree Operations

Polytypic parsing and several other polytypic syntax tree operations are used in the current version of the Esther shell [vWP03], which is written using Clean's generics. The Esther shell offers a basic lazy functional language as shell syntax. Its grammar is specified as a type, using the approach of Sect. 6.3. This section uses excerpts from the Esther shell to give an impression about how data specific syntax tree operations, written using polytypic programming techniques, improve conciseness, modularity, and allow easy changes to the syntax by adding and rearranging types.

6.4.1 Restructuring Infix Expressions

A common syntax tree operation is re-parsing expressions that contain user-defined infix operators. Because they are user defined, they cannot be correctly parsed during the first parse. The usual solution is to restructure the syntax tree after parsing, once the precedence and associativity information is available.

```
data FixityInfo = ···  — precedence and associativity information

fixInfix{|a | m|} :: (Functor m, Monad m) ⇒ a → FixityInfo → m a

fixInfix{|Int|}        i        ops = return i
fixInfix{|Unit|}       Unit     ops = return Unit
fixInfix{|Sum a b|}    (Inl l)  ops = do   l′ ← fixInfix{|a|} l ops
                                           return (Inl l′)
fixInfix{|Sum a b|}    (Inr r)  ops = do   r′ ← fixInfix{|b|} r ops
                                           return (Inr r′)
fixInfix{|Prod a b|} (l :*: r) ops = do    l′ ← fixInfix{|a|} l ops
                                           r′ ← fixInfix{|b|} r ops
                                           return (l′ :*: r′)
fixInfix{|Nexpression|} (Term t) ops = do
                                       t′ ← fixInfix{|Nterm|} t ops
                                       return (Nterm t)
fixInfix{|Nexpression|} (Apply e1 e2) ops = ··· — rebuild expression tree
```

We overloaded `fixInfix` with the `Monad` class because this operation can fail due to conflicting priorities. Generic Haskell requires mentioning this type variable `m` at the left side of the function type definition. The polytypic restructuring `fixInfix` function can be derived for all types except `Nexpression`, which is where we intervene to restructure the syntax tree. Note that manually removing the left-recursion and splitting the `Nexpression` type, allows us to override the polytypic function derivation at exactly the right spot. We lack the space to show exactly how to restructure the expression tree. This can be found in the current version of the Esther shell [vWP03].

The traversal code in the instances for the generic representation types is a common occurring pattern. This shows that we can elegantly and concisely specify a syntax tree operation that operates on a very specific part of the tree. There is no need to specify traversal code for any other type in the syntax tree, these are all automatically derived.

6.4.2 Adding Local Variable Scopes

Another common operation is checking variable declarations in the context of local scope. Scope can easily be added into the syntax tree using polytypic programming. We simply define the `Scope` data type below and inject it into the syntax tree where appropriate.

```
data Scope a   = Scope a

data Nterm     = LambdaWithScope (Scope Nlambda)
               | ···
data Nlambda = Lambda Tlambda (Plus Npattern) Tarrow Nexpression
data Ncase     = Case Tcase Nexpression Tof
                                  (Plus (Scope Nalt, Tsemicolon))
data Nalt      = Alternative (Plus Npattern) Tarrow Nexpression
data Npattern= ···
               | VariablePattern Nvariable
```

We overrule the derived polytypic code for `chkVars` at the following positions in the syntax tree types: `Nvariable` is an applied occurrence, except for occurrences after a `VariablePattern` constructor (part of the `Npattern` type), where it is a defining occurrence. Furthermore, we override the polytypic instance for `Scope`, which ends a variable scope after lambda expressions and case alternatives.

```
chkVars{|a | m|} :: (Functor m,Monad m) ⇒ a → [String] → m [String]

chkVars{|Unit|}       _       vs = return vs
chkVars{|Int|}        _       vs = return vs
```

```
chkVars{|Prod a b|} (l :*: r) vs = chkVars{|a|} l vs >>= chkVars{|b|} r
chkVars{|Sum a b|}  (Inl l)   vs = chkVars{|a|} l vs
chkVars{|Sum a b|}  (Inr r)   vs = chkVars{|b|} r vs

chkVars{|Nvariable|} (Variable v) vs
        | v 'elem' vs = return vs
        | otherwise   = fail ("unbound variable: " ++ v)

chkVars{|case VariablePattern|} (VariablePattern (Variable v)) vs
        = return (v:vs)          -- polytypic instance for a single constructor

chkVars{|Scope a|} (Scope x) vs = chkVars{|a|} x vs >> return vs
```

We make use of a Generic Haskell feature in the `chkVars` example above, which is not found in Clean: overriding the generic scheme at the constructor level. Instead of writing code for all constructors of the `Npattern` type, we only specify the semantics for the `VariablePattern` (hence the use of the **case** keyword) and let Generic Haskell derive the code for the other alternatives of the type.

6.4.3 Type Inference

As the compilation process proceeds, syntax tree operations tend to be less generic and more data specific. Program transformations and code generation, but also type-checking, usually require writing polymorphic instances for almost all types, since each type must be treated differently. At first sight, it seems as if polytypic programming is no longer useful to implement such operations. In this section, we will show that even for more data specific functions a polytypic definition improves modularity because it splits the specification per type, even if there is little profit from the automatic derivation mechanism. As an example, we specify a type inference algorithm in a polytypic way. Type inference is much more data specific than any other example in this chapter, nevertheless, it illustrates the way to polytypically specify syntax operations that occur later in the compilation process.

The algorithm is based on the idea of strictly separating type inference into the generation of constraints (in the form of type equations), and solving these constraints by standard unification. We restrict ourselves to the generation part, which is usually done by traversing the syntax tree and collecting constraints corresponding to each syntactical construct. Such an algorithm not only takes the syntax tree as input but also an environment containing type declarations for functions, constructors, and variables. Moreover, during the generation process we sometimes need fresh type variables, e.g., to instantiate a function's type scheme or to create local type variables used to express dependencies between derivations. Therefore, we supply the generation function with a heap data structure and we use an accumulator to collect type equations. This leads to the following polytypic function type and auxiliary type definitions.

```
data Type = TVar String | TBasic TBasic  | TApp String [Type]
          | TTVar VHeap | TArr Type Type | TAll [String] Type

data TBasic = TBool | TInt
data Equ    = Equ Type Type

type TypeState a = State (VHeap, [Equ]) a     — a state monad

gtype{|t|} :: t → Envs → TypeState Type
```

The VHeap is used to allocate fresh type variables. Mostly it suffices to generate unique integers to distinguish different type variables. These fresh variables are represented by the TTVar-alternative in the definition of Type. The other alternatives are used to represent type variables, basic types, type constructor applications, arrow types, and type schemes, respectively.

The type equations are represented as a list of Equ elements. Together with the VHeap, they form the state of the polytypic function. For convenience, the implementation of the polytypic gtype function is based on the standard State monad. For creating fresh variables and for extending the list of type equations, we introduce the following functions.

```
freshVar :: TypeState VHeap
freshVar = State {runState = λ(vh, eqs) → (vh, (vh+1, eqs))}

newEqu :: Type → Type → TypeState ()
newEqu dt ot
    = State {runState = λ(vh, eqs) → ((), (vh, Equ dt ot:eqs))}
```

The polytypic instance declarations are straightforward. We chose to interpret a Prod of two terms as an application of the first to the second. The advantage is that we can derive the instance for the type Nexpression automatically.

```
gtype{|Sum a b|}  (Inl l)  env = gtype{|a|} l env
gtype{|Sum a b|}  (Inr r)  env = gtype{|b|} r env
gtype{|Prod a b|} (x :*: y) env = do tx ← gtype{|a|} x env
                                     ty ← gtype{|b|} y env
                                     fv ← freshVar
                                     newEqu (TArr ty (TTVar fv)) tx
                                     return (TTVar fv)
```

Clearly, there are not many other types for which we use the polytypic version; most of the instances have to be given explicitly. E.g., for `TfunctionId` we can use the following definition:

```
gtype{|TfunctionId|} (FunctionId name) env
        = freshType name (fun_env env)
```

The overall environment has three separate environments: for functions, for constructors, and for type variables.

```
type Env  = String → Type
data Envs = Envs {fun_env :: Env, cons_env :: Env, var_env :: Env}
```

The function `freshType` takes care of the instantiation of the environment type. It can be defined easily, using the `freshVar` function, for type variables introduced by a `TAll` type. Another example is the alternative for `Nif`. Again, its definition is straightforward.

```
gtype{|Nif|} (If Tif c Tthen t Telse e) env = do
        tc ← gtype{|Nterm|} c env
        newEqu tc (TBasic TBool)
        tt ← gtype{|Nterm|} t env
        te ← gtype{|Nterm|} e env
        newEqu tt te
        return tt
```

Although we have to specify many instances explicitly, it is not inconvenient to use a polytypic specification: it splits the implementation into compact polytypic instances, which are easy to write while the resulting structure of the algorithm remains clear.

Concluding this section, we want to remark that polytypic programming allowed easy changes to the syntax by adding and rearranging types. Usually, this was done by *adding* types and instances to polytypic functions, instead of *rewriting* existing instances.

6.5 Performance of Polytypic Parsers

In this section, we investigate the efficiency of the generated parsers for two different grammars/languages. Our elegant types–as-grammar technique is of little practical use if the resulting programs perform poorly because of the automatically derived code by the polytypic system. Who cares about the advantage of not having to use an external tool, when the polytypic parsers perform an order of magnitude worse than parser generator based parsers.

6.5.1 A Basic Functional Language Parser

The first example is the derived parser for the basic functional language from Sect. 6.3. Since we are not interested in lexical analysis, we have tokenized the test input for the parser manually resulting in a list of 663 tokens representing 45 small functions in this language. The programs under test copy the input list of tokens 100 times and parse the resulting list 100 times. The results are shown in Table 6.1. For Haskell we used Generic Haskell (GH) 1.42, which requires the GHC 6.2.2 compiler. For Clean we used the Clean 2.1.1 distribution.

	Execution time (s)	Garbage collection (s)	Total time (s)	Total heap allocation (MB)
GH+GHC	27.2	1.4	28.6	3,500
Clean	45.0	6.7	51.8	11,600

Table 6.1: Performance figures for the derived basic functional language parser, using `Maybe` parsers.

All programs were run with a heap size of 256MB. It is remarkable to see that the Haskell version used only a quarter of the heap allocated by the Clean version. At first glance, it might not be clear that the generated executables are very slow and consume huge amounts of memory. Both Generic Haskell and Clean have some built-in specific optimization techniques to improve the performance of the derived functions. Moreover, these derived functions also benefit from standard optimizations, such as dictionary elimination, higher-order removal, etc. However, it appears that this is insufficient to obtain any acceptable performance.

6.5.2 Improving the Automatically Derived Code

In [AS05] Alimarine and Smetsers present an optimization technique, called *fusion*, of which they claim that it removes all the overhead introduced by the compilation scheme for polytypic functions (developed by Hinze [Hin00a]) that is used both in Generic Haskell and in Clean. Like *deforestation*, fusion aims at removing intermediate data used for passing information between function calls. This is done by combining nested pairs of *consumer* and *producer* calls into a single function application, making the construction of intermediate data structures from the producer to the consumer superfluous.

Fusion is not implemented in the Clean compiler, but incorporated in a separate source–to–source translator. The input language for this translator is a basic functional language extended with syntactical constructs for specifying polytypic functions. The translator first converts polytypic definitions into ordinary function definitions and optimizes these generated functions, by eliminating data conversions that are necessary to convert each object from and to its generic representation. The optimized output is both Clean and Haskell syntax compatible, so it was easy to include performance figures using both compilers as a back–end. These figures are shown in Table 6.2.

	Execution time (s)	Garbage collection (s)	Total time (s)	Total heap allocation (MB)
Fusion+GHC	4.3	0.03	4.5	340
Fusion+Clean	6.3	0.4	6.7	1,500

Table 6.2: Execution times for the optimized basic functional language parser, using `Maybe` parsers.

The programs ran under the same circumstances as those shown in Table 6.1. Each test yields a syntax tree consisting of approximately $300,000$ constructors per iteration. In the optimized Haskell version, this leads to an allocation of 12 bytes per node. Representing a similar syntax tree in an imperative language would require approximately the same number of bytes per node.

6.5.3 Using Continuation-based Parser Combinators

A nice aspect of our approach, is that the polytypic specification of the parser in Sect. 6.3 and the underlying parser combinator library are independent: we are free to choose different combinators, e.g., combinators that produce better error messages, without having to adjust the polytypic definitions. To illustrate this, we replaced the simple `Maybe`-combinators, by a set of continuation-based parser combinators, which collect erroneous parsings. These are similar to the combinators by, e.g., Koopman [KP02] or Leijen and Meijer [LM01]. Although the error reporting technique itself is simple, it appears that the results are already quite accurate. Of course, one can fine-tune these underlying combinators or even switch to an existing set of advanced combinators, e.g., Parsec [LM01], without having to change the polytypic parser definition itself.

We have tested the unoptimized as well as the optimized version of the continuation based parser, see Table 6.3. This time, the figures are more difficult to explain, in particular if you compare them with the execution times from the previous tables. In the literature, continuation passing parsers are often presented as an efficient alternative for the naive combinators. However, our measurements do not confirm this. The polytypic, as well as the optimized versions, are much slower

	Execution time (s)	Garbage collection (s)	Total time (s)
GH+GHC	137.9	10.2	148.2
Clean	77.3	20.0	97.3
Fusion+GHC	18.6	0.41	19.0
Fusion+Clean	55.5	8.74	64.2

Table 6.3: Execution times for the derived and optimized basic functional language parser, using continuation based parsers.

than the corresponding parser from the first test set, up to a factor of ten. One might believe that the additional error information causes this overhead. However, the loss in efficiency is almost the same when this information is not included. Apparently, the gain that is obtained by avoiding explicit constructors and pattern matching is completely undone by the use of continuations and therefore higher-order applications.

6.5.4 A Haskell 98 Parser

As a second test, we have implemented a (nearly) complete Haskell parser, simply by deriving polytypic parser instances for the Haskell syntax specified as a collection of algebraic data types. These data types were obtained by a direct conversion of the Haskell syntax specification as given in section 9.5 of the Haskell 98 Report [Pey03]. Again, we have compared the results for Generic Haskell and Clean for both the `Maybe` and the continuation passing combinators. We also optimized the generic code and compared the performance of all different versions. The results are shown in Table 6.4. The parsers were run on an example input consisting of approximately 500 again manually tokenized lines of Haskell code, 2637 tokens

An optimization that replaces update-frames with indirections was added to the Clean run-time system, reducing both heap and stack usage enough too complete the tests on a 1.5Ghz 512MB Windows PC.

	GH+GHC (s)	Clean (s)	Fusion+GHC (s)	Fusion+Clean (s)
`Maybe`	20.6	17.6	0.03	2.30
CPS	182	15.2	1.12	5.40

Table 6.4: Performance figures for the derived and optimized Haskell 98 parser, using both `Maybe` and continuation bases parsers.

These execution times are quite revealing. We can conclude that Generic Haskell and Clean generate extremely inefficient polytypic code. It is doubtful whether these polytypic language extensions are really useful for building serious applications. However, the optimization tool changes this completely, at least for Haskell. The performance gain for the `Maybe`-parsers is even a factor of 700. This test indicates once more that the continuation passing parsers are less efficient. It is strange to see that for Haskell the difference is much bigger than for Clean: a factor of 35 and 2, respectively. We do not have an explanation for the factor of 75 between GHC and Clean for the optimized `Maybe`-parsers.

We have also compared the efficiency of the optimized parsers with a Haskell parser generated with the *Happy* tool [GM01]. This parser is included in the libraries of the GHC Haskell compiler we used. The result is surprising: its execution time is the same as our Fusion+GHC `Maybe`-parser! To get more significant results we ran both with 100 times the input ($50,000$ lines of Haskell code, using a 4MB heap). Our parser is five percent faster, but does not have a lexer or decent error messages. Nonetheless, we believe that this shows that fusion is really needed and that fusion works for polytypic parsers.

6.6 Related Work

Parsers are standard examples for polytypic programming (see Jansson and Jeuring [JJ99], Hinze [HP01]). However, the common definition gives a parser that can only recognize expressions that can be defined in the corresponding programming language itself. This is very natural because the type definitions in a programming language can be regarded as a kind of grammar defining legal expressions in the corresponding programming language. We have shown that this also works for any context-free grammar.

It has also been shown how a parser for another language can be constructed from a grammar description. Atanassow, Clarke, and Jeuring [FAJ03] construct parsers for XML from the corresponding DTD description. To the best of our knowledge, this chapter is the first that describes the use of algebraic data type definitions as a grammar for deriving polytypic parsers for arbitrary languages.

There exist other (lazy, functional) parser generator tools and combinator libraries [GM01, Hut92, BS04, KP02, LM01], which may generate better parsers than our approach, due to grammar analysis or handwritten optimizations. What makes our approach appealing is that the tool used to generate the parser is part of the language. This removes the need to keep your syntax tree data structures synchronized with an external tool: one can do it within the polytypic functional language, and efficiently too, using extended fusion.

6.7 Conclusions

With this chapter, we have illustrated that polytypic programming techniques, as offered by the Generic Haskell preprocessor and the Clean compiler, can effectively be used for compiler construction. Additionally, we hope to have illustrated that the technique is interesting for programming in general.

Polytypic functions are type driven, it is therefore important to know what can be expressed in a type. In this chapter, we have shown that context-free grammars can be encoded in a straightforward way using algebraic data types. We have defined a polytypic parser using a types–as–grammar approach. Using such a polytypic definition, a parser for an arbitrary context-free language can be derived automatically. The polytypic function is defined in terms of parser combinators, and one can easily switch from one library to another.

Moreover, we have shown how other convenient polytypic post-parsing operations on the resulting rich syntax tree can be defined, even if not all syntax tree operations gain much from the polytypic programming style. It gives you the flexibility of moving data types within larger type structures, mostly by adding polytypic instances without having to change (much of) the existing code.

Finally, we have shown that optimizations that remove the polytypic overhead are absolutely necessary to make polytypic programs usable. Currently, polytypic programming, in either Generic Haskell or Clean, may be suitable for toy examples and rapid prototyping but the derived code is definitely not efficient enough for larger programs. Using the extended fusion optimization technique, the parser's efficiency came close to a parser generated by Happy. We believe that fusion makes polytypic programming for real-world applications possible.

CHAPTER 7

There and Back Again*

Arrows for Invertible Programming

Artem Alimarine
Sjaak Smetsers
Arjen van Weelden
Marko van Eekelen
Rinus Plasmeijer

7.1 Introduction

Arrows [Hug00] are a generalization of monads [Wad93]. Just as monads, arrows provide a set of combinators. They make it possible to combine functions in a very general way. In principle, the combinators assume very little about the functions to combine. In fact, these functions may even comprise side-effects. The main application areas of arrows are in the field of interactive programming and data conversion. More specifically, extensive applications have been made in the areas of user interfaces [CE01], reactive programming [HCNP03], and parser combinators [JJ02a].

For the general area of data conversion, it may be important to prove invertibility of a specified algorithm. This is, for instance, directly the case in encryption, serialization, marshalling, compression, and parsing, but more indirectly in the area of database transactions where rollbacks may have to be performed.

*Title shamelessly stolen from the Lord of the Rings (the book, not the movie).

The goal of our work is to set up an arrow-based framework for the specification of invertible algorithms. We start with extending the monotypic unidirectional framework of arrows to a monotypic bidirectional framework of bidirectional arrows, *bi-arrows*.

In particular, we represent a pair of conversion functions as a single arrow, such that we can specify both conversion functions by one definition. The advantage of such a single definition is that it reduces the amount of code needed for each conversion pair, because more code can be reused from the arrow library. Basically, one specifies the conversion in one direction (usually the more involved case) and one gets the inverse conversion almost for free. For instance, by specifying a parser one also specifies the pretty printer. The price to pay is that specifying the parser becomes a bit more complicated.

The advantages of programming with arrows and inversion are exploited best in a polytypic or generic framework. Therefore, we extend our monotypic bidirectional framework to the polytypic context. In this context, we show how to define several essential combinators and bi-arrow transformers. We give several smaller polytypic examples including invertible (de)serialization. We also discuss how this can be done for the larger example of parsers and pretty-printers.

More specifically, the contributions of this chapter are the following.

- We extend the framework of arrows to support *bidirectional arrows*.
- Our approach explicitly uses *embedding-projection arrows*.
- Our approach is suitable for *monotypic and polytypic* conversion functions.
- We show how to define pairs of conversion functions together in one single definition. We show that specifying one direction of conversion also specifies the other direction. We present several monotypic and polytypic *examples* of such definitions.

We use the pure lazy functional language Haskell [Pey03] in our examples. Polytypic examples use Generic Haskell [LCJ03], the generic programming extension for Haskell. The code can be downloaded from the author's web site[1]. The work can just as easily be expressed in Clean [PvE02] using its built-in generics [AP03]. We assume general knowledge of arrows and polytypic programming, and we will only briefly recall relevant definitions and techniques.

The next section (Sect. 7.2) introduces bidirectional arrow combinators. A small *monotypic* invertible program example is given in Sect. 7.3. This is done by using *embedding-projection arrows*, which are also introduced in that section.

[1]`http://www.cs.ru.nl/A.vanWeelden/bi-arrows/`

In Sect. 7.4 the framework is used in a polytypic context and we introduce invertible arrows with state. We present *polytypic traversals* (mappings) on bi-arrows and *state arrows*. These state arrows are used in Sect. 7.5 to create a somewhat larger example performing (de)serialization of data, based on the structure of a type.

Section 7.6 introduces monadic programming with bi-arrows. Ways to deal with failure in bi-arrows are introduced and a method to lift monads to bi-arrows is given. An application of bi-arrows, consisting of a parser and a pretty-printer, is created in Sect. 7.7. The example uses a combination of state, monadic, and embedding-projection arrows.

Finally, Sect. 7.8 discusses related work and Sect. 7.9 concludes and mentions prospects for future work.

7.2 From arrows to bidirectional arrows

This section introduces a bidirectional framework that consists of a set of reversible arrow combinators. These combinators are based on the arrow combinators defined by Hughes [Hug00].

First, we will recall shortly the standard arrow framework (Sect. 7.2.1). Then we show how these laws have to be adapted for our dyadic bi-arrows framework (Sect. 7.2.2). Finally, we give specific inversion laws for bi-arrows (Sect. 7.2.4). In Sect. 7.3 we show how bidirectional arrows are constructed using a small motivating example.

7.2.1 Arrows

We briefly recall Hughes's definitions expressed in Haskell as a type constructor class.

```
class Arrow arr where
    arr    :: (a → b) → arr a b — pure
    (⋙)    :: arr a b → arr b c → arr a c — infixr 1
    first  :: arr a b → arr (a, c) (b, c)
    second :: arr a b → arr (c, a) (c, b)
    (⁂)    :: arr a c → arr b d → arr (a, c) (b, d) — infixr 3
```

As usual, the definition of ⁂ and second can be expressed in terms of first (corresponding to Haskell's default definition of ⁂ and second):

```
f ⁂ g    = first f ⋙ second g
second f = arr swap ⋙ first f ⋙ arr swap
```

```
swap = snd 'split' fst
split f g = λt → (f t, g t)
```

To allow case distinction Hughes shows that a new combinator is needed. Therefore, he introduces the *choice arrow*:

```
class Arrow arr ⇒ ArrowChoice arr where
    left  :: arr a b → arr (Either a c) (Either b c)
    right :: arr b c → arr (Either d b) (Either d c)
    (+++) :: arr a c → arr b d →
                  arr (Either a c) (Either b d) -- infixr 2
```

As with *** and second, +++ and right can be expressed in terms of left, and Haskell's prelude function either:

```
f +++ g  = left f >>> right g
right f = arr mirror >>> left f >>> arr mirror

mirror = Right 'either' Left
```

By instantiating the arrow class for→we can use ordinary functions as arrows.

```
instance Arrow ( → ) where
    arr f   = f
    f >>> g  = g . f
    first f = f <*> id

instance ArrowChoice (→) where
    left f = f <+> id
```

Here <*> and <+> are the usual product and sum operations for functions:

```
(<*>) :: (a → b) → (c → d) → (a, c) → (b, d)
f <*> g = (f . fst) 'split' (g . snd)

(<+>) :: (a → b) → (c → d) → Either a c → Either b d
f <+> g = (Left . f) 'either' (Right . g)
```

In literature [Hug00, Pat01, Pat03], one can find several other combinators and some derived combinators that make programming with arrows easier, such as:

```
(<<<) :: Arrow arr ⇒ arr c b → arr b a → arr c a  -- infixl 1
f <<< g = g >>> f
```

Here, we refrain from giving an exhaustive overview.

7.2.2 Bidirectional arrows

To support invertibility, we extend the arrows with two new combinators: ↔ (*biarr*/*bipure*) and `inv` (*inverse*).

The first one, ↔, is similar to the standard `arr` but instead of a single function it takes *two* functions and lifts them into a bidirectional arrow (bi-arrow) creating a structure that contains them both. The intention is that these functions are each other's inverse. The second one, `inv`, reverses the direction of computation, yielding the inverse of a bi-arrow, which will boil down to swapping the two comprised functions.

```
class Arrow arr ⇒ BiArrow arr where
    (↔) :: (a → b) → (b → a) → arr a b  — infix 8
    inv  :: arr a b → arr b a
```

We define `BiArrow` on top of the `Arrow` class because conceptually bi-arrows form an extension of the arrow class. Moreover, it allows us to use bi-arrows as normal arrows. Since the derived combinators `second` and `right` use the `arr` constructor to build the adapters `swapA` and `mirrorA` we have to redefine them using ↔ to make these combinators invertible. Therefore, we introduce:

```
secondA f = swapA ⋙ first f ⋙ swapA
                    where swapA   = swap ↔ swap
rightA f  = mirrorA ⋙ left f ⋙ mirrorA
                    where mirrorA = mirror ↔ mirror
arrA f    = f ↔ const (error "arr has no inverse")
```

where `swap` and `mirror` are defined as above.

7.2.3 Arrow laws for bi-arrows

To reason about programs containing arrow combinators we can use properties that are specific to arrows, the so-called *arrow laws*. The collection of arrow laws is not uniquely defined. The laws we have taken are a subset of the ones postulated by Hughes [Hug00].

We need some adaptation of the laws for our framework. The occurrences of `arr f` are replaced with the corresponding dyadic operator for bi-arrows: `f ↔ g` where `g` is intended to be the inverse of `f`.

Definition 1 (Composition Laws)

$$\begin{array}{rcl} f \ggg (g \ggg h) & = & (f \ggg g) \ggg h \\ f_1 \leftrightarrow g_2 \ggg g_1 \leftrightarrow f_2 & = & (f_1 \ggg g_1) \leftrightarrow (f_2 \ggg g_2) \\ idA \ggg f & = f = & f \ggg idA \\ \multicolumn{3}{l}{\text{where}} \\ \multicolumn{3}{l}{\quad idA = id \leftrightarrow id} \end{array}$$

Definition 2 (Pair Laws)

$$\begin{array}{rcl} \mathit{first}\ (f \ggg g) & = & \mathit{first}\ f \ggg \mathit{first}\ g \\ \mathit{first}\ (f \leftrightarrow g) & = & (f \star id) \leftrightarrow (g \star id) \\ \mathit{first}\ h \ggg (id \star f) \leftrightarrow (id \star g) & = & (id \star f) \leftrightarrow (id \star g) \ggg \mathit{first}\ h \\ \mathit{first}\ (\mathit{first}\ f) \ggg assocPA & = & assocPA \ggg \mathit{first}\ f \\ \multicolumn{3}{l}{\text{where}} \\ \quad assocPA & = & assoc \leftrightarrow cossa \\ \quad assoc\ ((x,y),z) & = & (x,(y,z)) \\ \quad cossa\ (x,(y,z)) & = & ((x,y),z) \end{array}$$

In categorial terms, the product type is the dual of the sum type. In general, if a property holds for products, the dual property is valid for sums. The dual is obtained by systematically replacing *split* by *either*, *Left*/*Right* by *fst*/*snd*, *first* by *left*, $\ggg$ by $\lll$, and $f \circ g$ by $g \circ f$. For example, taking the dual of the last product law leads to the following sum law

$$\mathit{left}\ (\mathit{left}\ f) \lll assocSA = assocSA \lll \mathit{left}\ f$$

To obtain the dual *assocSA* of *assocPA* we first express *assoc* and *cossa* in terms of `split`, `fst` and `snd`.

$$\begin{array}{rcl} assoc & = & (fst \circ fst)\ \text{'}split\text{'}\ ((snd \circ fst)\ \text{'}split\text{'}\ snd) \\ cossa & = & (fst\ \text{'}split\text{'}\ (fst \circ snd))\ \text{'}split\text{'}\ (snd \circ snd) \end{array}$$

Now the transformation leads to $assocSA = assocS \leftrightarrow cossaS$, where

$$\begin{array}{l} assocS = (Left \circ Left\ )\ \text{'}either\text{'}\ ((Left \circ Right\ )\ \text{'}either\text{'}\ Right\ ) \\ cossaS = (Left\ \ \text{'}either\text{'}\ (Right \circ Left\ ))\ \text{'}either\text{'}\ (Right \circ Right\ ) \end{array}$$

Note that *right* is also the dual of *second*, since *mirror* is the dual of *swap*.

Using the laws above several properties can be proven easily. For example, $\mathit{first}\ idA = idA = \mathit{second}\ idA$ is proven by substituting the definitions for `first` and `second` taken from Sect. 7.2.1 and applying the appropriate laws for *first* and $\ggg$.

7.2.4 Inversion Laws

Most importantly, implementations of bi-arrows are proper if they satisfy some additional inversion laws.

Definition 3 (Inversion Laws)

$$\begin{aligned}
\mathit{inv}\ (\mathit{inv}\ f) &= f \\
\mathit{inv}\ (f \ggg g) &= \mathit{inv}\ g \ggg \mathit{inv}\ f \\
\mathit{inv}\ (f \leftrightarrow g) &= g \leftrightarrow f \\
\mathit{inv}\ (\mathit{first}\ f) &= \mathit{first}\ (\mathit{inv}\ f) \\
\mathit{inv}\ (\mathit{left}\ f) &= \mathit{left}\ (\mathit{inv}\ f)
\end{aligned}$$

The last two rules are only appropriate for arrows that are pure functions. In a more general case, where arrows can have side-effects (e.g., when monads with internal side effects are lifted to bi-arrows), it is required that, instead of *first* and *left*, *cofirst* and *coleft* respectively are used. These 'inverse combinators' are the categorical duals of *first* and *left*. They are needed to revert possible side-effects of *first* and *left*. Throughout the rest of this chapter all arrows will be pure. Hence, we will use the rules above since they are sufficient for this chapter. Nevertheless, for the rest of the framework no assumptions will be made on the absence of side-effects.

Of course, when introducing a new instance for one of the arrow classes defined above we have to guarantee that all the corresponding laws hold. We say that f is a *bi-arrow* if the composition, pair and inverse laws hold. Let f be a bi-arrow. Then f is *invertible* if

$$\mathit{inv}\ f \ggg f = \mathit{idA} = f \ggg \mathit{inv}\ f$$

The essence of our framework is that invertibility is preserved by our (bi-)arrow combinators. We are working on finishing the details of the formal proof of this property, using the various bi-arrow laws. It will be presented in a separate paper. The emphasis of this chapter will be on introducing the framework and on its applications.

7.3 Monotypic programming with bi-arrows

The idea of using bi-arrows is that after specifying an operation in one direction one gets the inverse of this operation (in the opposite direction).

In this section, we first discuss how to create an invertible definition using the bi-arrow definitions (Sect. 7.3.1). Then, we discuss the inherent differences between functions and bi-arrows (Sect. 7.3.2). This motivates why we introduce a structure that contains both functions (Sect. 7.3.3). Finally, we discuss some problems with the use of Paterson notation for bi-arrows (Sect. 7.3.4).

7.3.1 A motivating example

How easy or difficult is it to define functions by means of the arrow constructors? In this section, we will give an example. Of course, one has to keep in mind that some functions are not easily invertible. Take, for instance, a simple function like ++ (append), which concatenates two lists. It is clear that the inverse cannot be a function with the same type, since in general there are many ways to split a list into two parts.

An example of a function that does have an (obvious) inverse is `reverse`. We take the standard definition as starting point to get an arrow-based version. We could have lifted `reverse` to a bi-arrow using `reverse ↔ reverse`, but this does not illustrate the concerns of bidirectional programming.

```
reverse :: [a] → [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Case distinction, using arrows, is done by using `left` and `right`, which means that we first have to tag the input with `Left` or `Right`, indicating the empty and non-empty list respectively. Tagging and untagging are done by applying the following bi-arrow, which forms an isomorphic mapping from lists to `Either`s.

```
list2EitherA :: BiArrow arr ⇒ arr [a] (Either () (a, [a]))
list2EitherA = list_either ↔ either_list
    where
        list_either  []     = Left  ()
        list_either  (x:xs) = Right (x, xs)

        either_list (Left ())       = []
        either_list (Right (x, xs)) = x:xs
```

Now we can give the arrow version of `reverse`: `reverseA`.

```
reverseA :: (ArrowChoice arr, BiArrow arr) ⇒ arr [a] [a]
reverseA = list2EitherA
         ⋙ right (second reverseA ⋙ appElemA)
         ⋙ inv list2EitherA
```

Here `appElemA` is an adjusted version of `append` that takes one element and attaches it to the end of a list. If one specifies invertible arrows it appears to convenient to use 'symmetrical' versions, i.e., arrows that handle the argument and the result symmetrically. This leads to the following definition of `appElemA`. We will give an example of its usage later in this section.

```
appElemA :: (ArrowChoice arr,BiArrow arr) ⇒ arr (a,[a]) (a,[a])
appElemA = second list2EitherA ⋙ liftRSA
        ⋙ right (swapXYA ⋙ second appElemA)
        ⋙ inv (second list2EitherA ⋙ liftRSA)
```

The auxiliary arrow `liftRSA` converts a product–of–sum into a sum–of–product, and `swapXYA` exchanges the *x* and *y* field of a nested pair. The last one is defined in terms of `assocPA` and `swapA` introduced in Sect. 7.2.

```
liftRSA :: BiArrow arr ⇒ arr (a, Either b c) (Either (a,b) (a,c))
liftRSA = liftr ↔ rtfil
    where
        liftr (x, Left y)  = Left (x, y)
        liftr (x, Right y) = Right (x, y)

        rtfil (Left (x, y))  = (x, Left y)
        rtfil (Right (x, y)) = (x, Right y)

swapXYA :: BiArrow arr ⇒ arr (a, (b, c)) (b, (a, c))
swapXYA = inv assocPA ⋙ first swapA ⋙ assocPA
```

7.3.2 Functions are not bi-arrows

Although `reverseA` is constructed to be invertible, we cannot use the inverse of reverse using the → instance for arrows. This means that the following will not work:

```
(inv reverseA) [1, 2, 3] — this is a compile time error
```

This is caused by an absence of an instance of `BiArrow` for →. Since `ReverseA` itself depends on the `BiArrow` class, we even cannot write

```
reverseA [1, 2, 3]          — this is also a compile time error
```

There is no sensible way to define an instance of `BiArrow` for →. Of course, one *could* define ↔ for functions by dropping the second argument, however, this instance only works in one direction. For the last two examples, this would mean that we would not get a compile-time error anymore. Instead, we would get the correct result for the latter expression, but evaluation of the first one would result in a run-time error.

7.3.3 The embedding-projection bi-arrow transformer

We can circumvent this problem by handling inversion explicitly via *embedding-projection* (*EP*) *pairs*. See, for instance, [HP01]. We generalize embedding-projections from pairs of functions to pairs of arrows. This makes `EpT` an *arrow transformer*, i.e., it enables us to construct bi-arrows on top of existing arrows (particularly functions). Therefore, our type for embedding projections is parameterized with an arrow:

```
data EpT arr a b = Ep {toEp :: arr a b, fromEp :: arr b a}
```

The instances of the (bi-)arrow classes can be defined straightforwardly.

```
instance Arrow arr ⇒ Arrow (EpT arr) where
    arr     = arrA
    f ⋙ g   = Ep (toEp f ⋙ toEp g) (fromEp g ⋙ fromEp f)
    first f = Ep (first (toEp f)) (first (fromEp f))
    second  = secondA

instance ArrowChoice arr ⇒ ArrowChoice (EpT arr) where
    left f = Ep (left (toEp f)) (left (fromEp f))
    right  = rightA

instance Arrow arr ⇒ BiArrow (EpT arr) where
    f ↔ g = Ep (arr f) (arr g)
    inv f  = Ep (fromEp f) (toEp f)
```

To ensure the invertibility preserving property of the `EpT` bi-arrow transformer, one should not use the `arr` because an arrow constructed with `arr` has no inverse. We still define the `arr` function for `EpT`, in terms of the ↔ and `error` (using `arrA` from the previous section) to give a more informative run-time error and to support normal arrow operations.

By adding `toEp` to the example, we can force the use of the instance for the (`EpT` →) arrow:

```
toEp reverseA [1, 2, 3]         — yields [3, 2, 1]
toEp (inv reverseA) [1, 2, 3]   — yields [3, 2, 1]
```

In the same way, we can show an example of `appElemA`.

```
toEp appElemA (4, [1, 2, 3])    — yields (1, [2, 3, 4])
```

7.3.4 Paterson notation

The example from the previous section clearly shows that, without any support, programming with arrow combinators can be quite complicated.

The notation for arrows as proposed by Paterson [Pat01] can be helpful because it relieves the programmer from defining many small adaptor arrows. For example, the definition of appElemA using this arrow notion becomes:

```
appElemA = proc (e, xs) → case xs of
            []     → returnA −≺ (e, [])
            (x:xs) → do
                         (h, t) ← appElemA −≺ (e, xs)
                         returnA −≺ (x, h:t)
  where returnA = arr id
```

Unfortunately, this syntactic sugar for arrows does not support invertibility. The translation scheme, as described in [Pat01], uses unidirectional adaptors that cannot easily be made bidirectional. The (internal) adaptors are unidirectional, since they are defined using arr instead of $\leftrightarrow$. This is similar to the problem we encountered defining bi-arrows as an extension of the original arrow class (the default second also uses arr, hence the introduction of secondA and the like).

7.4 Polytypic programming with bi-arrows

In the following sections our framework is used in a polytypic context. First, in Sect. 7.4.1 we present *polytypic traversals* (generalized mappings). We show how to define the right–to–left traversals in terms of the left–to–right using duality. Secondly (Sect. 7.4.2), we introduce a *state arrow transformer*, i.e., an arrow implementation with which arbitrary arrows can be lifted to an arrow supporting invertible computations on states.

7.4.1 Polytypic traversals

Polytypic traversals are generalizations of polytypic mappings. They are introduced in Jansson and Jeuring [JJ02a]. Polytypic mappings operate on functions, whereas polytypic traversals operate on abstract arrows. Thus, mapping is just a special case of traversal.

However, unlike for mapping, the order of traversal of a data structure now becomes important, due to possible side effects within the arrow.

We specify the traversal operation using the polytypic programming extension of Haskell: Generic Haskell [LCJ03]. Every type, except certain predefined/basic types as Int, has a generic representation using only sums, products, and units. The Generic Haskell preprocessor can derivethe code for a polytypic function, as long as we define the polytypic function for the base instances: Sum, Prod, and Unit.

```
mapl{|a, b|arr|} :: (ArrowChoice arr, BiArrow arr, mapl{|a, b|arr|})
                 ⇒ arr a b

mapl{|Unit|}     = idA
mapl{|Prod a b|} = inv prodA ⋙ mapl{|a|} *** mapl{|b|} ⋙ prodA
mapl{|Sum a b|}  = inv sumA ⋙ mapl{|a|} +++ mapl{|b|} ⋙ sumA

prodA :: BiArrow arr ⇒ arr (a, b) (Prod a b)
prodA = fst `splt` snd ↔ exl `split` exr

sumA :: BiArrow arr ⇒ arr (Either a b) (Sum a b)
sumA = Inl `either` Inr ↔ Left `junc` Right
```

Some remarks about `mapl`:

- There is a context restriction on the monotypic type variable `arr`. Generic Haskell expects such type variables to be declared after the polytypic type variables, separated by a `|`.
- Besides the usual context restrictions on `arr`, there is also a context restriction over `mapl` itself. This is because the `mapl` is polytypic. Usually, these are derived automatically by Generic Haskell and can be omitted.
- The adaptors `prodA` and `sumA` would be superfluous if the definitions of `Prod` and `Sum` would coincide with `(,)` and `Either`. The `splt` and `junc` functions are the `Prod` and `Sum` counterparts of `split` and `either` for tuples and `Either`s, respectively.
- For clarity reasons we have omitted the cases for constructor information (i.e., instances for `Con` and `Label`) as they are not essential for the examples in this chapter.

Generic Haskell can derive a specific traversal function for any data type using the schematic representation of that type. In the present chapter, we will not need derived instances other than for types of kind $\star \to \star$. Unfortunately, Generic Haskell does not yet support the use of generic functions in the context restrictions of type classes and instances. We simulate this by introducing a dummy class, for which define the necessary instances in the obvious way. For types of kind $\star \to \star$ this leads to the class `Gmapl`.

```
class Gmapl t where
    gmapl :: (ArrowChoice arr, BiArrow arr) ⇒
              arr a b → arr (t a) (t b)
```

For instance, we can use polytypic traversal to map the increment function to a tree of integers, using the following data type definition for `Tree`, and instance definition of `Gmapl`

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

instance Gmapl Tree where
    gmapl = mapl{|Tree|}
```

Now we can write, again forcing the use of the `(EpT ⇀)` bi-arrow:

```
toEp (gmapl ((λx → x + 1) ↔ (λx → x - 1)))
                    (Leaf 1 `Node` Leaf 2 `Node` Leaf 3)
-- yields Leaf 2 `Node` Leaf 3 `Node` Leaf 4
```

The way the ⁂ and ⧺ are defined determines the traversal order. Basically, the order is left–to–right because ⁂ and ⧺ give preference to `first` end `left` respectively. Analogously, one can define the traversals using right–to–left variants of our basic combinators.

Jansson and Jeuring [JJ02a] show that such left–to–right and right–to–left traversals (e.g., `mapl` and `mapr`) form a pair of data conversion functions, which are each other's inverse. We want to show here that instead of defining both traversals separately, we can define one of them as the inverse of the other, using bi-arrows. We define the `mapr` (the right–to–left traversal) as the dual of the left–to–right traversal.

```
mapr :: (Gmapl t, ArrowChoice arr, BiArrow arr) ⇒
         arr a b → arr (t a) (t b)
mapr f = inv (gmapl (inv f))

toEp (gmapr ((λx → x + 1) ↔ (λx → x - 1)))
                    (Leaf 1 `Node` Leaf 2 `Node` Leaf 3)
-- also yields Leaf 2 `Node` Leaf 3 `Node` Leaf 4,
-- because the order does not matter in this example
```

7.4.2 The state bi-arrow transformer

Like monads, arrows can be used to specify computations with side effects on a state. We will show how to define a state arrow in our bi-arrow framework. This state arrow will be used later in an example to define an invertible pair of conversion functions that: *separate* a functor into its shape and its contents and *combine* the shape and the contents back.

Consider the following *arrow transformer*, which adds a state to a given arrow:

```
newtype StT s arr a b = St {unSt :: arr (a,s) (b,s)}
```

The corresponding instances of `Arrow` and `BiArrow` are defined below. This arrow transformer also occurs in [Hug00]. The instances below can be obtained directly from [Hug00] by replacing the unidirectional adapters (defined by means of *arr*) by bidirectional adapters using ↔.

```
instance BiArrow arr ⇒ Arrow (StT s arr) where
    arr      = arrA
    f ⋙ g   = St (unSt f ⋙ unSt g)
    first f = St (swapYZA ⋙ first (unSt f) ⋙ swapYZA)
    second   = secondA

instance (ArrowChoice arr, BiArrow arr) ⇒
         ArrowChoice (StT s arr) where
    left f = St (liftLSA ⋙ left (unSt f) ⋙ inv liftLSA)
    right  = rightA

instance BiArrow arr ⇒ BiArrow (StT s arr) where
    f ↔ g = St (first (f ↔ g))
    inv f  = St (inv (unSt f))

liftLSA :: (ArrowChoice arr, BiArrow arr) ⇒
           arr (Either a b, c) (Either (a, c) (b, c))
liftLSA = swapA ⋙ liftRSA ⋙ swapA +++ swapA

swapYZA :: BiArrow arr ⇒ arr ((a, b), c) ((a, c), b)
swapYZA = assocPA ⋙ second swapA ⋙ inv assocPA
```

The method ↔ of the state arrow is implemented using `first` and ↔ of the underlying arrow. The composition of state arrows just composes the underlying arrows.

The instance of `StT` for the choice arrow is defined with help of distributivity of the product type over the sum type. As usual, such a property is specified by constructing an appropriate bi-arrow, in this case `liftLSA`, a transformation of `liftRSA` from Sect. 7.3. Again, only minor modifications of the instance declarations given in [Hug00] were necessary.

7.4.3 Polytypic shape

We use the state arrow of the previous section to define polytypically an invertible pair of conversion functions that separate a functor into its shape and its contents and combine the shape and the contents back. Expressed as ordinary functions the type signatures of these two functions are:

```
separate :: Functor f ⇒ f a → [a] → (f (), [a])
combine  :: Functor f ⇒ f () → [a] → (f a, [a])
```

Instead of defining these functions as primitives, we will use the invertible state arrow. The data stored in/retrieved from the functor is passed as a state. For list states, we introduce the `getputA` arrow. The `getputA` arrow operates on this state and it is used to get an input element from or to add an element to the state.

```
getputA :: BiArrow arr ⇒ StT [a] arr () a
getputA = St (get ↔ put)
    where
        get ((), x:xs) = (x, xs)
        put (x, xs)    = ((), x:xs)
```

Since our shape operations are each other's inverse, we only have to specify one of them explicitly. We choose to define the `combine` function by using the polytypic traversals introduced in Sect. 7.4.1.

```
combine :: (Gmapl t, ArrowChoice arr, BiArrow arr) ⇒
           StT [a] arr (t ()) (t a)
combine = gmapl getputA

separate :: (Gmapl t, ArrowChoice arr, BiArrow arr)⇒
            StT [a] arr (t a) (t ())
separate = inv combine
```

The following example illustrates how we can use `combine` to fill an empty tree structure with integers.

```
(toEp . unSt) combine
         (Leaf () `Node` Leaf () `Node` Leaf (), [3, 4, 5])
–– yields Leaf 3 `Node` Leaf 4 `Node` Leaf 5

(toEp . unSt) separate (Leaf 3 `Node` Leaf 4 `Node` Leaf 5)
–– yields (Leaf () `Node` Leaf () `Node` Leaf (),
–– [3, 4, 5])
```

7.5 Polytypic (de)serialization

In this section, we present an example of encode-decode pair of functions that implement structure-based encoding and decoding of data.

The packing function takes data and converts it into a list of bits (Booleans), whereas the unpacking function recovers data from a list of bits. The bit representation directly represents the structure of data using only *static* information (the type of the data), not *dynamic* information (the value stored in a data structure), like some other compression methods do.

The choice which conversion should be specified is again arbitrary. We pick the decoder, which reads the bits from the input, and produces the original data structure. To obtain such a decoder for any data type, we will give a polytypic specification.

Basic types, like Char and Int, are encoded with a fixed number of bits. Although we could specify this primitive operation by means of arrow combinators, it appears to be easier to define it as a pure function, and to lift it to an arrow.

```
int2KBitsA :: BiArrow arr ⇒ Int → arr Int [Bool]
int2KBitsA k = int2bits k ↔ bits2int k
   where
       int2bits 0 n = []
       int2bits k n = odd n:int2bits (k-1) (n ‘div‘ 2)

       bits2int 0 bs          = 0
       bits2int k (True:bs)  = 1+bits2int (k-1) bs*2
       bits2int k (False:bs) = bits2int (k-1) bs*2
```

Now, the decoder for integers can be defined. It expects a list of bits, which has to be taken from the state. This is done by first producing the shape of the list and then by filling this list using the combine arrow of the previous section.

```
decodeInt :: (ArrowChoice arr, BiArrow arr) ⇒
             Int → StT [Bool] arr () Int
decodeInt k = createShapeA k ⋙ combine ⋙ inv (int2KBitsA k)

createShapeA :: BiArrow arr ⇒ Int → arr () [()]
createShapeA size = create ↔ etaerc
   where
       create () = replicate size ()
       etaerc l | length l == size = ()
```

The encoder for integers is the dual of the decoder for integers:

```
encodeInt :: (ArrowChoice arr, BiArrow arr) ⇒
             Int → StT [Bool] arr Int ()
encodeInt k = inv (decodeInt k)
```

The decoder defined as a polytypic function is:

```
decode{|t|arr|} :: (ArrowChoice arr, BiArrow arr,
                   decode{|t|arr|}) ⇒ StT [Bool] arr () t

decode{|Unit|}      = voidUnitA
decode{|Int|}       = decodeInt 32
decode{|Char|}      = decodeInt 8 ⋙ toEnum ↔ fromEnum
decode{|Bool|}      = getputA
decode{|Prod a b|} = dupVoidA
                      ⋙ decode{|a|} *** decode{|b|}
                      ⋙ prodA
decode{|Sum a b|}  = getputA ⋙ bool2EitherA
                      ⋙ decode{|a|} +++ decode{|b|}
                      ⋙ sumA
```

voidUnitA is the conversion between () and Unit, dupVoidA duplicates the input (), and bool2eitherA is the isomorphism between the boolean type and the co-product of voids.

```
voidUnitA :: BiArrow arr ⇒ arr () Unit
voidUnitA = (λ() → Unit) ↔ (λUnit → ())

dupVoidA :: BiArrow arr ⇒ arr () ((), ())
dupVoidA = (λ() → ((), ())) ↔ (λ((), ()) → ())

bool2EitherA :: BiArrow arr ⇒ arr Bool (Either () ())
bool2EitherA = bool2either ↔ either2bool
    where
        bool2either b = if b then Right () else Left ()
        either2bool (Left ()) = False
        either2bool (Right ()) = True
```

Since Unit can be encoded with zero bits, the decoder for Units just returns Unit. Booleans require just one bit and hence a single get. 32-bit integers are decoded with help of the integer decoder defined before. For characters, the decoder reads an 8-bit integer and converts into a character. Pairs are decoded by first makes two units out of one, and then applying the decoding componentwise. Finally, the case for the sum type first reads one bit to determine whether the left of the right branch should be decoded next.

Using duality, we get the encoder for free from the definition of the decoder.

```
encode{|t|arr|} :: (ArrowChoice arr, BiArrow arr,
                   decode{|t|arr|}) ⇒ StT [Bool] arr t ()
encode{|t|} = inv decode{|t|}
```

For example, to encode a tree containing the integers 1, 2, and 3 we simply write:

```
(toEp . unSt) encode{|Tree Int|}
            (Leaf 1 `Node` Leaf 2 `Node` Leaf 3, [])
```

The output consists of 101 bits: 96 for the integers and 5 bits for the nodes and leaves of the tree structure.

7.6 Monadic programming with bi-arrows

Up to now, our examples did not have to deal with failure. Of course, the decoding algorithm will not terminate properly if the input data does not correspond to a value, e.g., if some of the bits are missing. For expressing the algorithm, this was not essential, but in a real application, such an decoding function is not acceptable because it might lead to uncontrolled termination. On the other hand, it is much harder to preserve invertibility if functions are able to fail.

In this section, we present appropriate techniques to handle failure without losing invertibility completely. We first introduce bi-arrow definitions for polytypic zipping/unzipping (Sect. 7.6.1). Then, we define the class `ArrowZero` (Sect. 7.6.2) and show how in certain cases it can be used for the zipping example. To obtain a useful implementation of this new class, Sect. 7.6.3 adds a monadic arrow transformer to our arsenal. As a short example, this monadic bi-arrow is applied to the `Maybe` monad, which adds support for graceful failure to the polytypic zip function. In Sect. 7.7 we will extend our collection of arrow classes further with a combinator that, when applied to two arrows, will choose the second one if the first one fails.

7.6.1 Partial polytypic zipping

First, we introduce a polytypic function that is closely related to the polytypic traversals of Sect. 7.4.1: polytypic zipping/unzipping. It cannot deal with failure, which we will fix later on.

A binary zipping takes two structures of the same shape and combines them into a single structure. Unzipping does the opposite. In our bidirectional framework, we get unzipping for free if we define zipping as a bi-arrow. This can be done as follows:

```
zip{|a, b, c|arr|} :: (ArrowChoice arr, BiArrow arr,
                       zip{|a, b, c|arr|}) ⇒ arr (a, b) c

zip{|Unit|}     = inv dupUnitA
zip{|Prod a b|} = unprod2A ⋙ zip{|a|} *** zip{|b|} ⋙ prodA
```

```
zip{|Sum a b|}  = unsum2A ⋙ zip{|a|} +++ zip{|b|} ⋙ sumA

dupUnitA :: BiArrow arr ⇒ arr Unit (Unit, Unit)
dupUnitA = (λUnit → (Unit, Unit)) ↔ (λ(Unit, Unit) → Unit)

unprod2A :: BiArrow arr ⇒
            arr (Prod a b, Prod c d) ((a, c), (b, d))
unprod2A = dorp ↔ prod
    where
        dorp (x1:*:x2, y1:*:y2) = ((x1, y1), (x2, y2))
        prod ((x1, y1), (x2, y2)) = (x1:*:x2, y1:*:y2)

unsum2A :: BiArrow arr ⇒
           arr (Sum a b, Sum c d) (Either (a, c) (b, d))
unsum2A = mus ↔ sum
    where
        mus (Inl l1, Inl l2) = Left (l1, l2)
        mus (Inr r1, Inr r2) = Right (r1, r2)

        sum (Left (l1, l2))  = (Inl l1, Inl l2)
        sum (Right (r1, r2)) = (Inr r1, Inr r2)
```

Just as encode is the inverse of decode, we define unzip as the inverse of zip.

```
unzip{|t|arr|} :: (ArrowChoice arr, BiArrow arr, zip{|t|})
               ⇒ arr c (a, b) → arr (t c) (t a, t b)
unzip{|t|} f = inv (zip{|t|} (inv f))
```

Note that this definition for zip is partial: when two structures do not have the same shape the result of zipping these structures is undefined. Obviously, the inverse of zipping is a total function.

```
toEp (unzip{|Tree|} idA) (Leaf (1, 'a') `Node` Leaf (2, 'b'))
-- yields
-- Leaf 1 `Node Leaf 2, Leaf 'a' `Node` Leaf 'b'
```

Sometimes it is necessary that zipping itself is total, i.e., it should check whether the input structures match and handle it gracefully if not. This is usually done by returning a Maybe value in which Nothing indicates that the structures were not of the same shape/size.

However, in this case the inverse, unzipping, becomes partial: if zipping returns Nothing it is in general impossible to reconstruct the non-matching argument structures.

7.6.2 Bi-arrows with zero

To deal with operations that can fail we use the ArrowZero class.

```
class Arrow arr ⇒ ArrowZero arr where
    zeroArrow :: arr a b
```

The arrow zeroArrow is the multiplicative zero for composition with pure (bi-)arrows, i.e.,

$$f \ggg zeroArrow = zeroArrow = zeroArrow \ggg f$$

Clearly, this law excludes that zeroArrow has an inverse. However, this does not imply that we completely lose invertibility when zeroArrow is used: in many cases the *left* inverse of a failing operation still exists. More formally, an arrow f if *left-invertible* if $inv\ f \ggg f = idA$

The following derived combinator ||▷ (left-fanin), which is a bidirectional variant of the ||| (fanin) arrow combinator, appears to be convenient in combination with zeroA.

```
(||▷) :: (ArrowChoice arr, BiArrow arr) — infixr 4
        ⇒ arr a c → arr b c → arr (Either a b) c
f ||▷ g = f +++ g ⋙ untagRA

untagRA :: BiArrow arr ⇒ arr (Either a a) a
untagRA = id `either` id ↔ Right
```

From this definition we cannot conclude directly that it is invertible, because id `either` id is not the inverse of Right and, therefore, the occurrence of ↔ in untagRA is not invertible. We call this combinator *right-biased* because, in the reverse direction, it always yields Right. Nevertheless, we can show that the ||▷ combinator preserves left-invertibility. More specifically, it can be shown that the arrow f ||▷ g is left-invertible if g is left-invertible. Analogously, it follows that left-biased combinators preserve right-invertibility.

We can use the new combinator ||▷ with zeroA to extend zip with explicit failure. In fact, the only polytypic instance that changes is the one for Sum, see below. Additionally, we must add the ArrowZero class as a context restriction to the type of zip.

```
zip{|a, b, c|arr|} :: (ArrowZero arr, ArrowChoice arr,
                       BiArrow arr, zip{|a, b, c|arr|}) ⇒
                       arr (a,b) c

zip{|Sum a b|} = unsum2FA
            ⋙ zeroArrow ||▷ (zip{|a|} +++ zip{|b|})
```

```
           ⋙ sumA
unsum2FA = mus ↔ sum
where
    mus (Inl l1, Inl l2) = Right (Left  (l1, l2))
    mus (Inr r1, Inr r2) = Right (Right (r1, r2))
    mus (s1, s2)         = Left (s1, s2)

    sum (Right (Left  (l1, l2))) = (Inl l1, Inl l2)
    sum (Right (Right (r1, r2))) = (Inr r1, Inr r2)
    sum (Left  (s1, s2))         = (s1, s2)
```

Now the adaptor `unsum2FA` tags the result with an additional sum constructor to indicate whether the constructors matched. In particular, it uses `Right` in case both constructors were identical, and `Left` if they were different. In the latter case the `zeroArrow` branch of ||▷ is chosen, whereas in the first case the 'normal' `zip{|a|} +++ zip{|b|}` is performed.

7.6.3 Lifting monads to bi-arrows

To be able to apply `zip` to concrete data structures we need appropriate instances for our arrow classes, including `ArrowZero`.

A convenient and flexible way to manage failures, but also to implement other concepts such as non-determinism and states, is obtained by using monads. Monadic arrows are arrows that represent monadic computations.

The goal of this section is twofold: to show how we deal with monadic arrows in the bidirectional arrow framework and to provide the basis for handling failures.

We use the same classes for monads that can be found in Haskell [HPF99]. The basic monad is defined with the *return* and *bind* operations:

```
class Monad m
where
    return :: a → m a
    (>>=)  :: m a → (a → m b) → m b
```

The plus monad will be used to support failures of monadic arrows, and to implement *choices* as well.

```
class Monad m ⇒ MonadPlus m where
    mzero :: m a
    mplus :: m a → m a → m a
```

Usually, the Kleisli arrow transformer is used to represent monadic computations [Hug00, JJ02a], which is defined on a monad *m* as follows:

```
newtype K m arr a b = K {unK :: arr a (m b)}
```

However, this arrow is not suitable for our purposes, because it is not possible to define an instance of `inv` on it: it handles the argument and result asymmetrically. As symmetrical version of the Kleisli transformer can be obtained by adjusting the argument type in the definition of `K` as follows:

```
newtype MoT m arr a b = Mo {unMo :: arr (m a) (m b)}
```

The instances of `Arrow`, `BiArrow` and `ArrowChoice` on `MoT` require that we are able to traverse the underlying monad. This will be done by using the polytypic mapping `Gmapl` from Sect. 7.4.1.

However, this limits the choice for `m` to data types, because it is impossible to instantiate `Gmapl` for function types. In the instance definitions, we use the auxiliary arrows `firstMA` and `leftMA` based on the monadic `join` and `return` operations.

```
instance (Gmapl m, Monad m, ArrowChoice arr,
         BiArrow arr) ⇒ Arrow (MoT m arr) where
    arr     = arrA
    f ⋙ g   = Mo (unMo f ⋙ unMo g)
    first f = Mo (inv firstMA ⋙
                  gmapl (first (unMo f))
                  ⋙ firstMA)
    second  = secondA

instance (Monad m, ArrowChoice arr, BiArrow arr,
         Gmapl m) ⇒ ArrowChoice (MoT m arr) where
    left f = Mo (inv leftMA ⋙
                 gmapl (left (unMo f))
                 ⋙ leftMA)
    right  = rightA

instance (Gmapl m, Monad m, ArrowChoice arr,
         BiArrow arr) ⇒ BiArrow (MoT m arr) where
    f ↔ g = Mo (liftM f ↔ liftM g)
    inv f  = Mo (inv (unMo f))
```

with

```
firstMA :: (Monad m, BiArrow arr) ⇒ arr (m (m a, b)) (m (a, b))
firstMA = joinP ↔ splitP
    where
        joinP  = (=<<) (λ(mx, y) → mx >>= λx → return (x, y))
        splitP = (=<<) (λ(x, y) → return (return x, y))
```

```
leftMA :: (Monad m, BiArrow arr) ⇒
          arr (m (Either (m a) b)) (m (Either a b))
leftMA = joinS ↔ splitS
    where
        joinS  = (=<<) ((=<<) (return . Left)
                           `either` (return . Right))
        splitS = (=<<) ((return . Left . return)
                           `either` (return . Right))
```

```
liftM :: Monad m ⇒ (a → b) → m a → m b
liftM f m = m >>= λx → return (f x)
```

Here we should mention that invertibility of `firstMA` and `leftMA` depends on the underlying monad. E.g., for the `Maybe` monad it can be shown that both `firstMA` and `leftMA` are invertible; for the list monad this does not hold.

One of the purposes of the monadic arrows is to handle failures. The zero monadic arrow is defined with help of `mzero`.

```
instance (Gmapl m, MonadPlus m, ArrowChoice arr,
         BiArrow arr) ⇒ ArrowZero (MoT m arr) where
    zeroArrow = Mo (const mzero ↔ const mzero)
```

To illustrate the use of the monadic arrow we return to our generic zipping function. For example, combining the information of two trees is successful:

```
(toEp . unM) (zip{|Tree|} idA)
  (Just (Leaf 1 `Node` Leaf 3, Leaf 2 `Node` Leaf 4))
 — yields Just (Leaf (1,2) `Node` Leaf (3,4))
```

And if we try to combine two trees with different shape, it yields the `mzero`:

```
(toEp . unMo) (zip{|Tree|} idA)
                (Just (Leaf 1 `Node` Leaf 3, Leaf 2))
— yields Nothing
```

7.7 Parsing and pretty-printing

In this section, we show how to define a parser based on our reversible arrow combinators. Again, we will get the inverse, a pretty-printer, for free.

We give an example of a parser for a very simple functional language, specified by the following grammar in *BNF* notation.

```
<Expression>  ::= <Expression> <Expression>
                |  "(" <Expression> ")"
                | "λ" <Variable> "→" <Expression>
                | <Variable>
                | <Constructor>

<Variable>    ::= <String>
<Constructor> ::= <String>
```

The main difference between the decoder of Sect. 7.5 and a parser is that the decoder does not have to choose between alternatives, since its action for the sum type is solely depends on the next input bit. The parser presented in this section will try alternatives to see, which of them succeeds.

Another difference is that the parser is not completely determined by the type of the term it parses. It is because it needs to parse extra spaces, parentheses etc. Consequently, we cannot expect that the resulting parser is (left *and* right) invertible, because different input sentences, may lead to the same result.

Analogously to encode-decode, we define the parser and derive the corresponding pretty-printer. Therefore, the programmer does not need to write the complete pretty-printer code.

7.7.1 The plus arrow

Failure of parsers is handled by the `ArrowZero`. What we still need is a combinator that, when applied to two parsers, will choose the second in case the first one fails.

We therefore introduce one further arrow class, comparable to the `MonadPlus` class of monadic parser combinators.

```
class ArrowZero arr ⇒ ArrowPlus arr where
    (<|>) :: arr a b → arr a c → arr a (Either b c)
```

In contrast to the Haskell's arrow plus combinator `<+>`, our combinator tags its result so we can still see which parser has been chosen.

As said before, if possible the $\lhd\!\!\!\rhd$ chooses a non-failing computation. This is expressed by the law

$$zeroArrow \lhd\!\!\!\rhd f = f = f \lhd\!\!\!\rhd zeroArrow$$

The implementations of `ArrowZero` and `ArrowPlus` for the state arrow are straightforward (`liftLSA` has been defined in Sect. 7.4.2).

```
instance (ArrowZero arr, BiArrow arr) ⇒
         ArrowZero (StT s arr) where
    zeroArrow = St (first zeroArrow)
```

```
instance (ArrowPlus arr, ArrowChoice arr,
         BiArrow arr) ⇒ ArrowPlus (StT s arr) where
    f <+> g = St (unSt f <+> unSt g >>> inv liftLSA)
```

Instantiating `ArrowPlus` for the monadic arrow is much more complex. We defer its definition until the end of this section.

7.7.2 A concrete parser

As in the previous sections, we will use a combination of the state and monadic arrows to build a concrete example parser. The resulting syntax tree is represented by the data structure.

```
data Expression = App Expression Expression
                | Nested Expression
                | Lambda String Expression
                | Variable String
                | Constructor String
```

Observe that the syntax tree explicitly stores whether an expression was enclosed by brackets. This is done to ensure that, when printing a parsed expression, brackets are displayed correctly.

To abstract from the parsing issues at the lexical level, we assume a separated scanner/lexer and that the parser will work on a list of tokens. This leads to:

```
data Token = Id_T String | Lambda_T | Open_T
           | Close_T | Arrow_T | EOF_T deriving Eq

type Parser arr t  = StT [Token] arr () t
type Printer arr t = StT [Token] arr t ()
```

7.7.3 Parsing keywords

Before defining a parser for expressions, we introduce two auxiliary parsers to examine the input tokens.

The first one, `parseKeyword`, tries to read a given token from the input stream. If it succeeds, this token is delivered as result; if not, the parser fails. As with the zip example of Sect. 7.6.3 we use ||> in combination with `zeroArrow` to handle failure.

```
parseKeyword token = getputA >>> tagTokenA
                   >>> zeroArrow ||> idA
    where
```

```
        tagTokenA = test ↔ id `either` id
        test t = if t == token then Right t
                                else Left t
```

The second one examines the input list to see whether the next token is an identifier. Moreover, to distinguish variables (starting with a lower case `char`) from constructors (starting with a upper case `char`) this parser is parameterized with a predicate. The parser succeeds in case of an identifier token fulfilling the predicate. Then the identifier itself is returned, otherwise the parser fails.

```
parseIdentifier p = getputA ⋙ tagIDA p
                ⋙ zeroArrow ||> idA
    where
        tagIDA p = tagID p  ↔ id `either` Id_T

        tagID p (Id_T name) | p name = Right name
        tagID _ token                = Left token
```

7.7.4 Parsing expressions

The grammar of our input language is left-recursive, and hence cannot be directly translated into a parser. We introduce an intermediate function for parsing expressions (called *terms*) which are no applications.

```
parseTerm = parseNested
          <|> parseLambda
          <|> parseVariable
          <|> parseConstructor
          ⋙ toExp ↔ fromExp
    where
        toExp = Nested `either` (uncurry Lambda
                    `either` (Variable `either` Constructor))

        fromExp (Lambda var exp) = Right (Left (var, exp))
        fromExp (Variable var)   = Right (Right (Left var))
        fromExp (Constructor c)  = Right (Right (Right c))
        fromExp (Nested nested)  = Left nested
```

`parseTerm` combines parsers for all expression kinds by using the arrow plus combinator. The result, tagged with various `Left`s and `Right`s, is converted by the adapter `to_expr ↔ from_expr` into the corresponding part of the syntax tree.

For parsing consecutive elements, we use a helper combinator based on ⁂ and the `dupVoidA` arrow defined in Sect. 7.5.

```
(<&>) :: BiArrow arr ⇒                          — infixl 6
         arr () a → arr () b → arr () (a, b)
f <&> g = dupVoidA ⋙ f ⁂ g

parseLambda = parseKeyword Lambda_T
            <&> parseVariable
            <&> parseKeyword Arrow_T
            <&> parseExpression
            ⋙ toLambda ↔ fromLambda
    where
        toLambda (((_, v), _), e) = (v, e)
        fromLambda = const Lambda_T `split` fst
                    `split` const Arrow_T `split` snd

parseNested = parseKeyword Open_T
            <&> parseExpression
            <&> parseKeyword Close_T
            ⋙ toExp ↔ fromExp
    where
        toExp ((_, e), _) = e
        fromExp e = ((Open_T, e), Close_T)

parseVariable    = parseIdentifier (isLower . head)
parseConstructor = parseIdentifier (isUpper . head)
```

The parser for applications takes some more doing. It first reads a list of terms and converts this into a tree of binary applications.

We introduce a function `parseOneOrMore` to parse a list of elements that, when applied to a parser *p*, tries to parse one or more *p*-elements.

```
parseOneOrMore p = p <&> parseOneOrMore p <|> p
                 ⋙ untag ↔ tag
    where
        untag (Left  (x, (y, l))) = (x, y:l)
        untag (Right  x)          = (x, [])

        tag (x, y:l) = Left (x, (y, l))
        tag (x, [])  = Right x
```

Note that this `parseOneOrMore` will try to find the longest list. The parser for expressions can now be expressed easily.

```
parseExpression = parseOneOrMore parseTerm
                ⋙ uncurry to_apply ↔ from_apply []
    where
        to_apply app []     = app
        to_apply app (x:xs) = to_apply (App app x) xs

        from_apply l (App f a) = from_apply (a:l) f
        from_apply l t         = (t, l)
```

Finally, the pretty-printer for expressions is obtained by taking the inverse of the parser.

```
parse :: (ArrowPlus arr, ArrowChoice arr,
         BiArrow arr) ⇒ Parser arr Expression
parse = parseExpression <&> parseKeyword EOF_T ⋙ eofA
    where
        eofA = fst ↔ (λx → (x, EOF_T))

print :: (ArrowPlus arr, ArrowChoice arr, BiArrow arr)
         ⇒ Printer arr Expression
print = inv parse
```

7.7.5 A monadic plus arrow

Before we can really use our parser we have to provide an appropriate implementation of the plus arrow.

More specifically, we need an instance of `ArrowPlus` for the monadic arrow transformer `M`. Of course, this instance will be based on the `mplus` of the underlying monad.

```
instance (Gmapl m, MonadPlus m, ArrowChoice arr, BiArrow arr)
         ⇒ ArrowPlus (MoT m arr) where
    l <|> r = Mo (dupMA ⋙
              (unMo l ⋙ inlMA) *** (unMo r ⋙ inrMA)
             ⋙ inv dupMA)
```

The adapter arrows `dupMA`, `inlMA` and `inrMA` are defined as follows.

```
dupMA :: (MonadPlus m, BiArrow arr) ⇒ arr (m a) (m a, m a)
dupMA = (λx → (x, x)) ↔ uncurry mplus
```

```
inlMA :: (MonadPlus m, BiArrow arr) ⇒ arr (m a) (m (Either a b))
inlMA = inlM ↔ uninlM
    where
        inlM   = (=<<) (return . Left)
        uninlM = (=<<) (return `either` const mzero)

inrMA :: (MonadPlus m, BiArrow arr) ⇒ arr (m a) (m (Either b a))
inrMA = inrM ↔ uninrM
    where
        inrM   = (=<<) (return . Right)
        uninrM = (=<<) (const mzero `either` return)
```

The adapter dupMA is in general *not* invertible, because the arguments of ↔ are obviously not each other's inverse. This means that the instance of ⊲|⊳ is also not invertible, because it defined in terms of dupMA and inv dupMA.

Consequently, when defining an operation using this instance of ⊲|⊳ one does not get invertibility for free, i.e., it is no longer sufficient to prove that all pairs of pure functions lifted with ↔ are each other's inverse. To show correctness, global reasoning is required.

In practice, this may imply that the inverse of the operation needs to be fine-tuned in order to produce the expected result. In particular, this holds for our parser example. The Nested constructor was added to the syntax tree to be able to reconstruct the brackets that were used to disambiguate expressions.

7.7.6 Parser/printer examples

Suppose we have the following list of input tokens:

```
tokens = [Open_T, Lambda_T, Id_T "x", Arrow_T,
          Id_T "x", Close_T, Lambda_T, Id_T "y",
          Arrow_T, Id_T "y", EOF_T]
```

To parse this and convert it into an expression, we write:

```
(toEp . unMo . unSt) parse (return ((), tokens))
                       :: Maybe (Expression, [Token])
```

And if we want to print the expression:

```
expr = App (Nested (Lambda "x" (Variable "x")))
                          (Lambda "y" (Variable "y"))
```

we simply write:

```
(toEp . unMo . unSt) print (return (expr, []))
                       :: Maybe ((), [Token])
```

The `Maybe`-monad does not reveal that the expression parser is ambiguous.

Suppose we leave out the `Nested` constructor in the last example expression. Printing this expression will lead to a list of tokens not containing the open and close brackets anymore. Our parser will still be able to parse this list but it will not produce the same expression we have started with: the `App` will occur inside the first lambda expression. The reason is that our parser only delivers one successful parse.

However, in our framework it is very easy to change the parser in such a way that it delvers all successful parses, namely, by using the list monad instead of the maybe monad. This list monad is a standard implementation of the monad class. Therefore, the only thing we have to change for our example is the type!

```
(toEp . unMo . unSt) parse (return ((), tokens))
                       :: [(Expression, [Token])]
```

Running this expression with the following list of tokens

```
tokens = [Lambda_T, Id_T "x", Arrow_T, Id_T "x",
          Lambda_T, Id_T "y", Arrow_T, Id_T "y",
          EOF_T]
```

will now yield two expressions:

```
App (Lambda "x" (Variable "x")) (Lambda "y" (Variable "y"))
```

and

```
Lambda "x" (App (Variable "x") (Lambda "y" (Variable "y")))
```

7.8 Related Work

This work is inspired by Jansson and Jeuring [JJ02a, JJ99] who define polytypic functions for parsing and pretty-printing and then prove invertibility. They maintain invertibility using pairs of separate definitions, leading to many proof obligations for the programmer. In contrast, we use one single definition for both conversion directions using invertibility-preserving combinators. As a result, we only have to prove invertibility for the primitives that are used. Furthermore, our approach is neither limited to the example of parsing nor to the use of polytypic functions.

Invertibility is an important practical property used in many algorithms. For instance, it plays an important role in the database world where one has to ensure that any change in a view domain leads to a corresponding change in the underlying data domain.

To ensure this property, Foster et al. [FGM$^+$05] present a domain-specific programming language in which all expressions denote bi-directional transformations on trees. They use two functions, a get function for extracting an abstract view from a concrete one, and a put function that creates an updated concrete view given the original concrete view and the updated abstract view. Using the proper get and put functions, invertibility is guaranteed.

For similar purposes, Mu et al. [MT04] define a programming language in which only injective functions can be defined, thus guaranteeing invertibility. Again put and get functions are defined, but the crux here is to do some bookkeeping when doing a get such that a put can always be made invertible.

A different approach is taken by Robert Glück and Masahiko Kawabe [GK04, GK05]. They try to construct the inverse function from the original one automatically. They use a symmetrical representation for functions such that the inverse function can be constructed by interpreting the original function backwards. Our arrow combinators have a representation with this same property. The main difference with our work is we obtain the inverse function by construction while they try to automatically generate an inverse function from the original one. They use LR-parsing techniques and administrative bookkeeping to invert choices made by conditional branches in the original function.

There is much literature about inverting *existing* programs, both functional and imperative, see for example: Dijkstra [Dij79], Chen [CU90], and Ross [Ros97]. Our approach is more hands-on and focusses on constructing (parts of) programs in an invertible way.

7.9 Conclusions and Future Work

We feel that we have provided an interesting framework in the area of invertible programming.

We have extended arrows to bidirectional arrows, bi-arrows, which preserve invertibility properties. We have presented several invertible bi-arrow transformers. Bi-arrows were used in a monotypic and in a polytypic context. We introduced ways to deal with state and with monads. A concrete parser/pretty printer example was presented with a discussion of its properties.

For future work we want to provide full formal proof that the framework preserves invertibility properly. Furthermore, we will investigate whether the approach scales up to real world practical examples where invertibility properties are a requirement. Among other things, this will require creating a translation scheme similar to Paterson notation in such a way that the required properties are preserved, and programs are easier to read and write.

CHAPTER 8

On–the–Fly Formal Testing of a Smart Card Applet

ARJEN VAN WEELDEN
MARTIJN OOSTDIJK
LARS FRANTZEN
PIETER KOOPMAN
JAN TRETMANS

8.1 Introduction

Smart devices are often used in critical application domains, such as electronic banking and identity determination. This implies that their quality, such as their safety, security, and interoperability, is very important. Such devices commonly implement a Java Card virtual machine, which is able to execute Java Card applets. Each application is then implemented as a separate applet.

One way to increase the quality of applets is the use of formal methods. Such applets are sufficiently small to make a complete formal treatment with current day formal technology feasible. Systematic testing is another method, predominantly used to check the quality of smart devices in an experimental way.

In this chapter, we combine testing and formal methods: we test a Java Card applet, in a black-box setting, based on a formal specification of its required behavior. Compared with formal verification, testing has the advantage that it examines the real, complete system consisting of applet, platform and hardware together, whereas formal verification is usually restricted to a model of the applet only. Compared with traditional, manual testing, formal testing has as first advantage that the formality reduces the ambiguities and misinterpretations in the specification so that it is clearer what should be tested. Secondly, formal specifications allow completely automation of the testing process: test cases are algorithmically generated from the formal specification, and test results can automatically be analyzed. This makes it possible to generate and execute large quantities of large tests in a short time. It is mainly this second advantage that we will pursue in this chapter.

Our investigation of formal testing is conducted using a case study. The applet that we test is a simple electronic purse application, implemented as a Java Card applet, with a limited set of methods like asking the value on the card, debiting, crediting, etc. The formal specification of the electronic purse applet is given as a State-chart [Gro]. The case and its formal specification are described in Sect. 8.2.

Automatic test generation, test execution, and test result analysis are performed in an *on–the–fly* fashion, using the test tool GAST [KP04]. The test tool GAST is described in Sect. 8.3; Section 8.4 describes how the State-chart specification is transformed into Clean [PvE02], a functional programming language used as the input language for GAST. How GAST, the applet under test, and the platform are connected is described in the test architecture, which is given in Sect. 8.5.

We constructed one (assumed to be) correct applet implementation. From this implementation, we derived 22 mutants by inserting subtle bugs to see whether such bugs would be detected by our automated, formal testing method. A summary of the performed tests is given in Sect. 8.6. Finally, Sect. 8.7 and 8.8 discuss related work, conclusions, and possible future extensions.

8.2 Case Study

We demonstrate our testing methodology by applying it to a simple electronic purse application as a case study. The basic events, which the electronic purse can receive, are:

- set an initial value `n` via `setValue(n)`
- query the actual value via `getValue()`
- pay an amount of `n` via `debit(n)`
- authenticate with a `pin` (personal identification number) via `authenticate(pin)` before charging the card
- charge the card with an amount of `n` via `credit(n)`
- reset the card using a `puk` (personal unlocking key) via `reset(puk)`

All these events are input events for the card, because they are sent from the Card Accepting Device (CAD, also called *terminal*) to the card. To every input event, the card answers with a corresponding output event:

- acknowledge an operation via `ackOK` or `ack(n)`
- report an error via `error(n)`

Figure 8.1 shows the specification of the purse, modelled as a State-chart.

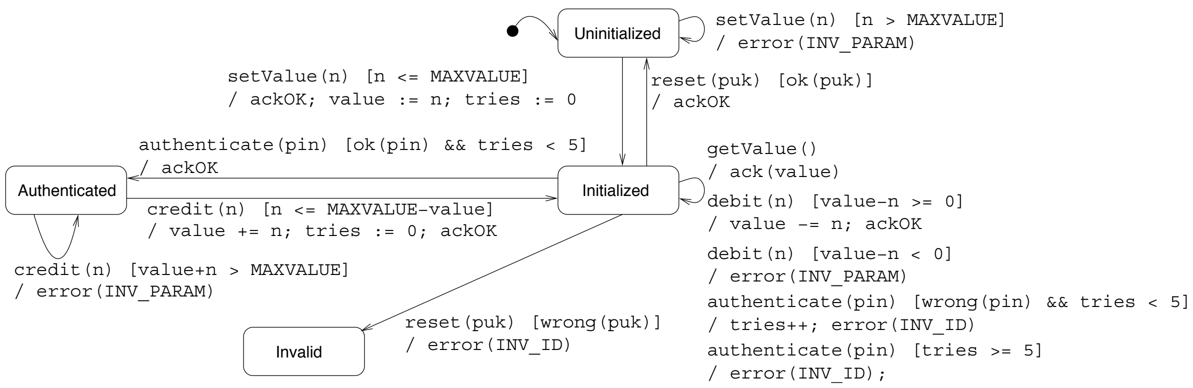

Uninitialized
setValue(n) [n > MAXVALUE] / error(INV_PARAM)
setValue(n) [n <= MAXVALUE] / ackOK; value := n; tries := 0
reset(puk) [ok(puk)] / ackOK
authenticate(pin) [ok(pin) && tries < 5] / ackOK
Authenticated
Initialized
credit(n) [n <= MAXVALUE-value] / value += n; tries := 0; ackOK
getValue() / ack(value)
debit(n) [value-n >= 0] / value -= n; ackOK
debit(n) [value-n < 0] / error(INV_PARAM)
credit(n) [value+n > MAXVALUE] / error(INV_PARAM)
authenticate(pin) [wrong(pin) && tries < 5] / tries++; error(INV_ID)
Invalid
reset(puk) [wrong(puk)] / error(INV_ID)
authenticate(pin) [tries >= 5] / error(INV_ID);

Figure 8.1: State-chart model of the purse applet.

The transition labels between two states s_1 and s_2 are of the form:

$$s_1 \xrightarrow{i\ [g]\ /\ act} s_2$$

with *i* being an input event, *g* being a guard, and *act* representing a sequence of actions. We exemplify the semantics with this transition:

$$\text{Authenticated} \xrightarrow{\text{credit(n) [n} \leq \text{MAXVALUE} - \text{value] / value} += \text{n; tries:=0; ackOK}} \text{Initialized}$$

The input event (*i*) is `credit(n)`. The argument `n` represents the amount of money to be added to the card. The applet uses signed 16-bit `shorts` and it gives an `error(INV_PARAM)` on negative values. We abstract from that in the State-chart to keep it concise. The actual value of the card is saved in the variable `value`. A transition can only fire when the corresponding guard *g* holds. In this example, one can only increase the value of the card by `n`, when `n` does not exceed the `MAXVALUE`-value. If the transition is taken, the actions (*act*) are performed. In this case, the variable `value` is incremented by `n`, the `tries` variable is reset to zero, and the acknowledgment `ackOK` is sent to the terminal.

Intuitively, the purse works as follows. First, the card is in the `Uninitialized` state. It then is initialized by the credit institution, which issues the card to the customer by putting a certain amount `n` of money on it via the `setValue(n)` event.

In the `Initialized` state the customer can query the actual value via the `getValue()` event, or pay with the card via the `debit(n)` event. To increase the value, one must first authenticate at a terminal with a card-specific `pin`, leading to the `Authenticated` state. Being in that state, one can add money via the `credit(n)` event, leading back to the `Initialized` state. The card checks that its value does not exceed the `MAXVALUE`.

Furthermore, there is a maximum of five tries to enter the `pin`. After the fifth wrong attempt, one can no longer credit the card. If the credit institution enters the reset code (called `puk`) correctly, the card goes back to the `Uninitialized` state and can be re-initialized via the `setValue(n)` event. If the `puk` is entered wrongly, the card goes to the `Invalid` state and cannot be used anymore.

Two kinds of erroneous events can be sent to the card. Firstly, a syntactically correct input event that is not specified for the actual state may occur, e.g., a `credit(n)` when the card is in the `Initialized` state. Such an unspecified input event is called an *inopportune event*, and the response of the applet should be an error message `error(INV_CMD)`, whereas the applet remains in its actual state. Secondly, a syntactically incorrect event may occur, e.g., a command-APDU with a non-existing event-code. This is also implicitly assumed to lead to an error message, while the card stays in its current state.

8.3 The Test Tool GAST in a Nutshell

The test tool GAST is designed to be open and extendable. For this reason, it is implemented as a library rather than a stand-alone tool. The functional programming language Clean is chosen as host language due to its expressiveness.

GAST can handle two kind of properties. It can test properties stated in logic about (combinations of) functions. GAST can also test the behavior of reactive systems based on an Extended (Finite) State Machines (E(F)SM). Here we will only discuss the ability to test reactive systems.

An ESM as used by GAST comes quite close to the State-chart of Fig. 8.1. It consists of states with labelled transitions between them. A transition is of the form $s \xrightarrow{i/o} t$, where s,t are states, i is an input which triggers the transition, and o is a, possibly empty, list of outputs. The domains of the inputs, outputs, and states can be given by arbitrarily complex recursive Algebraic Data Types (ADT). This constitutes the main difference with traditional testing with FSM's where the testing algorithms can only handle finite domains and deterministic systems [LY96].

A transition $s \xrightarrow{i/o} t$ is represented by the tuple (s,i,t,o). A relation based specification δ_r is a set of these tuples: $\delta_r \subseteq S \times I \times S \times O^*$. The transition function

δ_f is defined by $\delta_f(s,i) = \{(t,o)|(s,i,t,o) \in \delta_r\}$. Hence, $s \xrightarrow{i/o} t$ is equivalent to $(t,o) \in \delta_f(s,i)$. A specification is *partial* if for some state s and input i we have $\delta_f(s,i) = \emptyset$. A specification is *deterministic* if for all states and inputs the size of the set of targets contains at most one element: $\#\delta_f(s,i) \leq 1$. A *trace* σ is a sequence of inputs and associated outputs from a given state. Traces are defined inductively: the empty trace connects a state to itself: $s \overset{\varepsilon}{\Rightarrow} s$. We can combine a trace $s \overset{\sigma}{\Rightarrow} t$ and a transition $t \xrightarrow{i/o} u$ form the target state t, to trace $s \overset{\sigma;i/o}{\Longrightarrow} u$. We define $s \xrightarrow{i/o} \equiv \exists t. s \xrightarrow{i/o} t$ and $s \overset{\sigma}{\Rightarrow} \equiv \exists t. s \overset{\sigma}{\Rightarrow} t$. All traces from state s are: $traces(s) \equiv \{\sigma|s \overset{\sigma}{\Rightarrow}\}$. The inputs allowed in a state s are given by $init(s) = \{i|\exists o. s \overset{i/o}{\Longrightarrow}\}$. The states after trace σ in state s are given by $s\,\mathsf{after}\,\sigma \equiv \{t|s \overset{\sigma}{\Rightarrow} t\}$. We overload *traces*, *init*, and after for sets of states instead of a single state by taking the union of the individual results.. When the transition function, δ_f, to be used is not clear from the context, we will add it as subscript.

The basic assumption for our formal testing is that the Implementation Under Test (IUT), iut, is also a state machine. Since we do black box testing, the state of the iut is invisible. The iut is assumed to be *total*: any input can be applied in any state. Conformance of the iut to the specification spec is defined as (s_0 is the initial state of spec, and t_0 of iut):

$$\begin{aligned} \mathsf{iut}\ \mathit{conf}\ \mathsf{spec} \quad &\equiv \quad \forall \sigma \in traces_{\mathsf{spec}}(s_0), \forall i \in init(s_0\,\mathsf{after}_{\mathsf{spec}}\,\sigma), \forall o \in O^*. \\ &\qquad (t_0\,\mathsf{after}_{\mathsf{iut}}\,\sigma) \xrightarrow{i/o} \Rightarrow (s_0\,\mathsf{after}_{\mathsf{spec}}\,\sigma) \xrightarrow{i/o} \end{aligned}$$

Intuitively: if the specification allows input i after trace σ, the observed output of the iut should be allowed by the specification. If spec does not specify a transition for the current state and input, anything is allowed. This notion of conformance is very similar to the *ioco* relation [Tre96] for Labelled Transition Systems (LTS). In an LTS each input and output is modelled by a separate transition. In our approach, an input and all induced outputs up to *quiescence* are modelled by a single transition. Quiescence characterizes a state of the iut that will not produce any output before a new input is provided, i.e., a quiescent system waits for input and cannot do anything else.

In order to test conformance, a collection of input sequences is needed. At the beginning of each input sequence GAST resets the iut and the spec their initial state. By applying the inputs of a sequence one by one, GAST investigates if it can be transformed to a trace of spec. The previous inputs and observed responses are remembered in trace σ. If $\delta_{\mathsf{spec}}(s,i) \neq \emptyset$ for the current input i and some state s reachable from the initial state, s_0, by trace σ (i.e., $s_0 \overset{\sigma}{\Rightarrow} s$), the input is applied to the iut, and the observed output is checked by spec.

GAST has several algorithms for input generation, e.g.:

- Systematic generation of sequences based on the input type.
- Sequences that cover all transitions in a *finite* state machine.
- Pseudo random walk through the transitions of a specification.
- User-defined sequences.

In this chapter, we will only use the third algorithm to generate input sequences. Testing is *on–the–fly*, which means that input generation, execution, and result analysis are performed in lockstep, so that only the inputs actually needed will be generated. The lazy evaluation of Clean used for the implementation of GAST makes this easy.

Within the test tool GAST, the mathematical state transition function, δ_f, specifying the desired behavior is represented by a function in the functional programming language Clean. Functional languages allow very concise specifications of functions and have well understood semantics. Using an existing language as notation for the specification prevents the need to design, implement and learn a new language. The rich data types and available libraries enable compact and elegant specifications. The advanced type system of functional languages enforces consistency constraints on the specification, and hence prevents inconsistencies in the specification. Since the specification is a function in a functional programming language, it can be executed. This is convenient when one wants to validate the specification by observation of its behavior. Any Clean type can be used to model the state, the input and the output of the function specifying δ_f, including user-defined data types. GAST uses generic programming techniques for generating, comparing, and printing of these types. This implies that default implementation of these operations can be derived without any effort for the test engineer. Whenever desired, these operations can be tailored using the full power of the functional programming language.

8.4 The Purse Specification for GAST

The specification given in the State-chart is transformed to the functional language Clean in order to let GAST execute and manipulate it. This section gives some details about the representation in Clean of the electronic purse from Fig. 8.1. Due to space limitations, we will show only snapshots of the (executable) specification. A parameterized enumeration type represents the state of the purse.

```
:: PurseState  = Uninitialized | Initialized Short Short
               | Authenticated Short | Invalid
```

We use one constructor for each state from the State-chart in Fig. 8.1. The arguments of the constructor `Initialized` represent the tries counter and the value.

The type Short represents signed 16-bit integers. This implies that there are actually $2 \times 2^{16} = 2^{17}$ different initialized states, of which some are not reachable. There are similar types for input and output.

A transition function purse similar to δ_f in Sect. 8.3 models the transitions. The only difference with the mathematical specification is that the result is a list of tuples instead of a set of tuples. Some function alternatives specifying characteristic transitions are:

```
purse :: PurseState PurseInput → [(PurseState, [PurseOutput])]
purse Uninitialized (SetValue n)
    = if (n ≥ 0 && n ≤ MAXVALUE)
        /*then*/ [(Initialized 0 n, [AckOK])]
        /*else*/ [(Uninitialized, [Error INV_PARAM])]
purse (Initialized tries value) Reset
    = [(Uninitialized, [AckOK])]
purse (Initialized tries value) GetValue
    = [(Initialized tries value, [Ack value])]
...
purse state any = [(state, [Error INV_CMD])]
```

The first alternative models both transitions for the input SetValue n from the state Uninitialized. The second and third alternative show two transitions form the state Initialized. The last alternative captures the informal requirement that inopportune events should cause no state transition and an error message as output. Since state and input any are variables, this alternative covers any combination not listed above. Since exactly one transition is defined for each combination of state and input, the specification is total and deterministic.

This specification is an ordinary definition in the functional programming language Clean. It is checked by the compiler before it is used by GAST. This guarantees well-defined identifiers and type correctness.

8.5 Testing Java Cards with GAST

The tests, which will be described in Sect. 8.6, have been executed using the test architecture of Fig. 8.2.

The IUT is the Java Card applet implementing our simple electronic purse. To make testing easier and more flexible, we used a simulation platform to execute the applet. The simulation environment is the *C-language Java Card Runtime Environment* (CREF), which comes with the *Java Card Development Kit*. CREF simulates a Java Card technology-compliant smart card in a card reader. It further

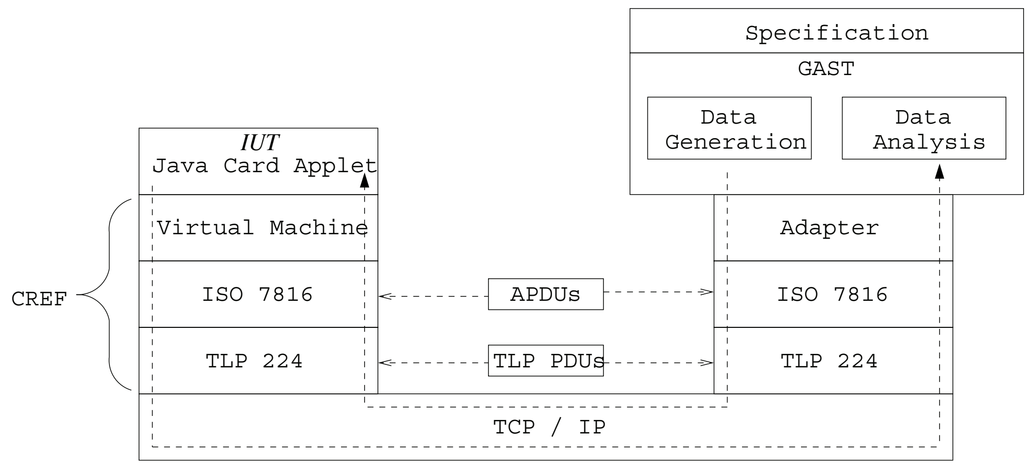


Figure 8.2: The general testing framework.

consists of a Java Card Virtual Machine, and communication protocol entities to allow communication between the applet and the outside world.

To communicate with the applet under test, GAST was enhanced to be able to deal with these typical smart card communication protocols *ISO–7816–4* and *TLP–224* over TCP/IP. On top of these protocol entities an *adapter* (glue code), was implemented. The adapter transforms the high level inputs, generated by GAST, and represented as Clean data values, into the low-level APDUs, coded as appropriate byte codes, and then sent according to the ISO–7816–4 protocol. Vice versa, the adapter decodes the APDUs received from the applet under test to Clean data values, which are then analyzed and checked by GAST.

For data generation and analysis, GAST uses the Clean EFSM specification, which was presented in Sect. 8.4. Except for the access to TCP/IP, the right-hand side of Fig. 8.2 was implemented in Clean.

The use of a simulation platform for testing is not a restriction with respect to testing of real smart cards. Since only standardized protocols are used, GAST cannot see the difference between testing on a simulator, or testing a real smart card. The test architecture could easily be adapted to test real cards by swapping CREF with a real card and its reader. The use of a simulation platform does facilitate easy switching between different applets, and saving and restoring applet state.

8.6 Results

The State-chart in Fig. 8.1 and its implementation as an applet were developed in an incremental way. GAST appears to spot differences between the specified and actual behavior very rapidly. Once the specification and implementation were finished, the testing power of the GAST system was determined in a systematic way using *mutants*. Starting from the ideal (assumed to be correct) applet we injected typical programming errors into the applet, and analyzed how long it took for GAST to find errors by generating, executing, and analyzing tests.

no.	paths	trans-itions	time (sec)	counter exam.	comments
1	1	25	0.49	25	6 tries allowed in this mutant
2	2	66	0.09	31	incorrect overflow during `credit`
3	1	9	0.47	9	negative balance allowed in mutant
4	5	247	0.71	41	tries not reset after `authenticate`
5	8	406	0.38	51	tries not reset after `reset`
6	1	1	0.05	1	`credit` allowed without `authenticate`
7	1	1	0.52	1	`setvalue(0)` not allowed
8	1	4	0.06	4	`credit` with negative amount
9	1	2	0.50	2	`debit` with negative amount
10	11	542	0.48	23	no check for locked flag
11	7	327	0.80	26	not locked after 5 attempts
12	1	13	0.06	13	stays authenticated
13	21	1020	1.28	21	not unlocked after `reset`
14	1	16	0.09	16	`MAXVALUE` too low
15	1	24	0.07	24	`authenticate` does not authenticate
16	1	33	0.52	33	`reset` does not make it uninitialized
17	94	4757	3.82	29	tries ≤ 5 instead of tries < 5
18	4	207	0.26	23	fresh card has nonzero balance
19	1	6	0.30	6	`setValue` allowed in initialized state
20	3	145	0.18	44	`setValue` does not initialized/unlock
21	1	4	0.50	4	`MAXVALUE` too high
22	5	206	0.67	2	`MAXVALUE` balance not allowed
	7.8	366.4	0.56	19.5	averages
	100	5081	4.20	n/a	original applet, no counter example

Table 8.1: Overview of test results.

The mutants are obtained by subtle changes like omitting checks or updates to the state of the applet. The test result of the 22 mutants used are listed in

Table 8.1. For instance, mutant 17 differs from the ideal applet by testing whether the number of remaining authentication tries is *less than or equal to* five rather than *less than* five before setting a flag indicating that the applet should no longer accept authentication attempts. This mutant was found after executing 94 paths, within 3.82 seconds, containing 4757 transitions in total. This mutant showed an invalid output after an input sequence of length 29 in path 94. To identify the error, the trace of inputs and associated responses are written to a file.

GAST was able to identify the 22 incorrect implementations without any help, using a minimum path length of 50 transitions and a maximum of 100 paths. It took an average of 0.56 seconds to generate and execute, on average, 366 transitions on a 1.4GHz Windows computer. This shows that GAST is an efficient test tool.

8.7 Related Work

Two approaches are closely related to ours because both rely on tools which implement variants of the *ioco* testing relation [Tre96]. Du Bousquet and Martin [Md01] use UML specifications, which are translated into Labelled Transition Systems to serve as input for the TGV tool [JJ02b]. Instead of an on–the–fly execution, TGV uses additional test purposes to generate test cases. The authors created a tool to automate the generation of test purposes based on common testing strategies. The generated test cases are finally translated into Java code that communicates with the applet and executes the test. TGV does not treat data symbolically, which can easily lead to a state space explosion when dealing with large data domains. Because we generate test cases on–the–fly based on the (symbolic) EFSM, this problem does not occur.

To support symbolic treatment of data, Clarck et al. [CJRZ01] use Input/Output Symbolic Transition Systems. The basic approach is similar to TGV, hence also here test purposes are needed. The test automation is done via a translation to C++ code that is linked with the implementation. This restricts the IUT to be a C++ class with a compatible interface.

Rather than *testing* properties of the IUT, its implementation (i.e., the Java Card applet) can also be formally *verified*. Testing and verification are complementary techniques to check the correctness of systems, as explained in Sect. 8.1. A common technique used for verifying Java Card applets is to prove their correctness with respect to a specification in the Java Modelling Language (JML). State-based specifications similar to the one in Fig. 8.1 can uniformly be translated to JML specifications as shown by Hubbers, Oostdijk, and Poll [HOP03]. The resulting annotated Java Card applet can then be verified using one of the many JML tools [BCC$^+$03], for instance, the ESCJava2 static analyzer [CK05]. Most

Java Card applets are small enough to even attempt a formal correctness-proof using the Loop tool, as demonstrated by Jacobs, Oostdijk, and Warnier [JOW04].

8.8 Conclusions and Future Work

We have presented an approach to automate the testing of Java Card applets using the test tool GAST. The test case derivation is based on a State-chart specification of the applet under test. The specification can directly be translated into a corresponding GAST specification. Tests were completely automatically derived, executed, and analyzed. Discrepancies between the formal specification and its Java Card implementation were successfully detected, which shows the feasibility of this approach.

The direct translation from the State-chart model to the GAST specification, and the on–the–fly execution of the test cases enable the developer to start with automatic testing of the applet in the early stages of development. The simultaneous development of the formal model and the implementation, and the facility to do automatic tests, has shown to be very useful. Both the code and the specification have evolved simultaneously, vastly improving the quality of the applet, and leading to a complete and reliable specification. Such a specification delivers insight on how to specify similar cases, and can serve as a pattern for these.

The tested mutants, representing typical programming errors, have increased our confidence in the error detecting power of the GAST algorithm. We are planning to compare this with other test tools, e.g., the ioco-based tool TorX [TB03], to test more complex applets, testing applets on real cards, and testing advanced aspects like the integration, interference, and feature interaction between different applets on one card.

Finally, we will compare the testing approach with the formal verification approach, e.g., using JML, to see how far we can get in unifying verification and testing techniques into one common framework, and to investigate the precise shape of their complementarity.

CHAPTER 9

General Conclusions

This thesis presented a functional (language) approach to several software issues, such as communicating processes, command-line interfaces, automatic parser generation, reversible program construction, and testing. We made use of the strengths of functional programming languages such as abstraction, explicit state, and strong types. We also used, and explored the limits of, two emerging techniques to prevent errors in software. We catch errors using the advanced type system provided by functional languages to warn the programmer early in the development of the software. Where we could not statically apply type-checking, we applied type-checking at run-time. Another way to prevent errors is to automatically generate large parts of programs.

The first technique is Clean's hybrid static/dynamic type system, which we use to provide type safe communication of any value between processes and a type-checking command-line shell that uses a typed file system. The second technique is polytypic function definitions, provided by both Clean and (Generic) Haskell. We show that the type-safe and type-directed generation of functions is applicable to more than simple problems and that it decreases the amount of code that needs to be written and maintained.

We will now mention some related work and some new work that spawned from this research. This is presented per chapter in Sect. 9.1. Furthermore, we look at what 'future work' from each chapter remains. We conclude this thesis in Sect. 9.2 after reflecting on the conclusions of each chapter and their contributions to improving software quality during construction.

9.1 Old Future Work and New Related Work

Each chapter of this thesis has been peer-reviewed and published. Besides informing others about our research results, it has provided us with feedback on our ideas and the incentive to further develop this line of research. Because this work spans a relative long time in computer science, new related work has appeared and we can investigate which parts of this research have inspired others.

Chapters 2 and 3:

Other people have investigated the use of functional programming techniques related to operating systems. House [HJLT05] is a small operating system written almost completely in Haskell. It builds on the hOp [CB04] project. The main difference between our proof–of–concept with type-safe communication and House, is that House has been implemented on bare hardware. In that sense, House is more of a real operating system and it is used as a research vehicle to try functional language programming approaches to implementing the core of an operating system. In contrast to our shell, which provides type safety for the user, House uses advanced functional techniques to provide type safety for developers of the operating system and other components, such as hardware drivers.

As the developments towards a mobile Haskell [RTL05] shows, there is an interest in using pure and lazy functional languages for distributed computing. Until the development of hs-plugins [PSSC04], Clean was the only system that supported moving functional expressions from one program to another (in a type-safe way). It makes perfect sense to require plug-ins (or other untrusted code) to be written in a pure and lazy language, since dangerous operations, such as I/O, must show up in the static type of expressions. Of course, one would need trusted compilers and/or (byte)code verification to prevent attackers from by-passing the type checker.

Chapter 4:

It is still not exactly proven that the Esther shell correctly infers all types. The proof given in Sect. 4.5 is limited to the lambda-calculus with letrec-sharing to combinators part of the translation of a command line. To be sure, we would need to extend the proof to incorporate all parts of the translation including syntactic sugar, case expressions, etc. We chose not to model this in Chap. 4 to keep the model and proof manageable. Proving such a relatively small program as the entire shell is already practically intractable.

By proving type preservation of Diller–algorithm–C, we tried to keep the results interesting for a more general audience. We see no serious problems in extending the proof to encompass the entire translation; it is just a lot of work. Recently, we completed the proof of the conjecture used in this chapter. The completeness proof builds on a principle types proof done by Smetsers, which we intend to publish separately.

Chapter 5:

The functional Graphical Editor Component system has been redesigned, extended, re-implemented for the web, and renamed iData [PA06]. Polytypic techniques are used for (de)serialization of state on either client- or server-side, HTML code generation, storage of (typed) information (of any structure) in relational databases. There is also ongoing work on a semantical framework for type directed automatically generated graphical editors.

Chapter 6:

Since our publication about automatically generating parsers from (rich) syntax trees, new parser combinators have appeared that could be very useful for implementing the types–as–grammar approach. In Sect. 6.3 we mentioned parser combinators by Baars and Swierstra that can, in principle, be used in combination with our technique to eliminate left-recursion. Recently, Baars [BS07] has presented yet unpublished work on automatic left-recursion removal with Generalized Algebraic Data Types (GADTs), which can do grammar transformations while preserving the original types. Especially this last feature has our interest, since we use types to derive our parsers. Unfortunately, polytypic programming and GADTs cannot be used together at the moment.

There have been little further developments on the fusion technique to improve the efficiency of the automatically generated code for polytypic instances. Although we have shown that Clean's fusion prototype works very well for parser generators, fusion is not yet commonplace in languages that provide polytypic functions. Therefore, polytypic functions will continue to choke on large inputs until this situation improves.

However, we have seen that recent tuning of the Clean compiler and the runtime system can improve the performance of polytypic programs by generating better code for unknown higher-order function calls. Although these improvements do nothing to remove the actual polytypic overhead as fusion does, they can greatly reduce the penalty for using polytypic functions.

Chapter 7:

We have not gotten around to complete and publishing the proofs about the invertibility preserving bi-arrow combinators. Besides sparking theoretical works about arrows by Heunen and Jacobs [HJ06] and by Jacobs and Hasuo [JH06], this approach has not been used in practice. This is likely due to the inability to use the special arrow syntax, which makes arrow programs readable. Furthermore, most common computations are, in their usual form, simply not invertible. Nonetheless, it was interesting to see how far we could take the ideas of abstraction over function application and type directed automatic derivation, and to apply them to invertible program construction.

Chapter 8:

We have extended the generic test system described in Chap. 8 with a Java Card adaptor with the intention of testing the new Dutch electronic passport. Unfortunately, we were unable to finish the formal specification in Clean while we had access to the specimens. Nonetheless, we feel that we could contribute to the testing of the passport in formalizing an executable specification as well as automatic testing of the applet. Such an executable specification in Clean would allow us to test properties on that specification using the polytypic test system. This is possible because the GAST approach is very flexible. The specification can also function as an implementation because it is once again executable. These layers can be stacked upon each other until any desired level of abstraction is reached. The producer of the passport applet has shown an interest in a formal model of the specification in combination with automated tests.

9.2 Final Conclusions

We applied hybrid static/dynamic typing in a proof–of–concept operating system and command-line shell. In this thesis, we presented and implemented a very minimal operating system kernel (Chap. 2) and shell (Chap. 3). We successfully applied typing at run-time to make interprocess communications type safe. We found it very useful and elegant that we could leave (de)serialization of functions and data with preservation of sharing to Clean's run-time system. The implementation (and the formal proofs of Chap. 4) of the shell shows that Clean's Dynamics are expressive enough to do type inference on a functional language. As an example, we provided the command-line user with early warnings in case of type errors. Besides the hybrid static/dynamic typing, the hybrid interpretation/compilation is interesting. Clean's Dynamics are not only used for type-checking but also to compose new functions from existing code interactively, which can be used (in a type safe way) by any other compiled Clean program. We think that the vision of a strongly typed operating and file system is still attractive to anyone who is fond of strong typing.

Both polytypic programming and hybrid static/dynamic typing are used in Chap. 5. The shell from Chap. 3 is reused as a library to parse and type-check text from automatically generated graphical user interfaces. This line of research, deriving GUIs in a type directed manner, is still ongoing.

By applying polytypic programming to the area of syntax tree operations, we showed that it could be used for more than just the usual generic operations, such as equality and data structure traversal. This is important because type directed program generation is limited by the expressiveness of types in functional languages. The types–as–grammar approach presented in Chap. 6 shows that context free grammars could easily be encoded in algebraic data types. Moreover, we believe to have shown that this approach yields concise, elegant, and easy to maintain programs, even with large collections of (mutually recursive) types that change during the development of software. The side-effect freeness and automatic generation of functions reduced the complexity of making changes to (the choice of) the data structures of a program during its development. Using recent advanced fusion techniques, we were even able to optimize the generated code to a level were it was actually usable for larger texts. Another benefit therefore, is that we no longer need external parser generators, which protects us from forgetting to synchronize the grammar and syntax-tree data types with other tools.

Polytypic programming can significantly reduce the amount of code that needs to be written by the programmer. We applied it to the field of invertible programming using powerful abstraction mechanisms such as arrows. In combination with bi-arrow combinators, which construct the inverse of a computation that is only specified in one direction, we tried to minimize the required manually written code in Chap. 7 in an extreme way. Not only the inverse is constructed for free, it is also generalized to all types. Unfortunately, it turns out to be hard and clumsy to write larger programs in this style, which is also because many basic computations, such as the logical and and or, are not invertible in their common form. We believe that the polytypic approach works very well in Chap. 7 (except for the occasional bug we encountered, which shows that polytypic language extensions are still in development). Still, programming with such a high abstraction as arrows is not easy, not even for experienced functional programmers. The good part of arrows, or other well thought-out combinator sets, is that they form a nice abstraction layer for proving program correctness, which we did both formally and informally, and defining operational properties of the underlying implementation. Apparently, the arrow abstraction, which is a very general abstraction over application, is a double-edged sword and should be wielded carefully.

Deriving (complicated) test cases automatically using polytypic programming, worked like a charm in Chap. 8. Writing the specification of a Java Card applet in a functional language gave us the benefit of a concise, clearly correct, and executable specification. After the weary part of interfacing the GAST test system with the Java Card simulator, we have the pleasure of writing both the specification and parts of the implementation/adaptor using the full expressiveness of a functional programming language. This allows us to use higher-order abstraction and composition, for example. The effectiveness of the GAST test system allowed for a tight program–test–adjust development cycle.

From the research presented in this thesis, we conclude that Clean's Dynamics provides us with additional protection against run-time errors by tagging values with their static types. Furthermore, it provides us with a way to store and retrieve any functional expression, even between different programs. Polytypic programming also keeps us from making mistakes in (tediously) writing similar code for many data types. Because the automatic derivation/instantiating of polytypic functions is both type directed and type safe (it cannot introduce type errors), there is less room for programming errors.

We conclude this thesis with the idea that we put types to good use: both at compile-time for type-directed program generation and at run-time for type-safe communications.

Bibliography

[Ach96] Peter Achten. *Interactive Functional Programs - Models, Methods, and Implementations*. PhD thesis, University of Nijmegen, 1996.

[ACP$^+$92] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, Gordon Plotkin, and Didier Rèmy. Dynamic typing in polymorphic languages. In *Proceedings of the ACM SIGPLAN Workshop on MAL and its Applications*, 1992.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *TOPLAS'91: ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[AP98] Peter Achten and Rinus Plasmeijer. Interactive Functional Objects in Clean. In Chris Clack, Kevin Hammond, and Antony J.T. Davie, editors, *IFL'97: Selected papers of the ninth International Workshop on the Implementation of Functional Languages*, volume 1467 of *Lecture Notes in Computer Science*, pages 304–321. Springer, 1998.

[AP01] Peter Achten and Simon Peyton Jones. Porting the Clean Object I/O library to Haskell. In Markus Mohnen and Pieter Koopman, editors, *IFL'00: Selected papers of the twelfth International Workshop on the Implementation of Functional Languages*, volume 2011 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2001.

[AP03] Artem Alimarine and Rinus Plasmeijer. A generic programming extension for Clean. In Thomas Arts and Markus Mohnen, editors, *IFL'02: Selected Papers of the thirteenth International Workshop on Implementation of Functional Languages*, volume 2312 of *Lecture Notes In Computer Science*, pages 168–185. Springer, 2003.

[AS05] Artem Alimarine and Sjaak Smetsers. Improved fusion for optimizing generics. In Manuel V. Hermenegildo and Daniel Cabeza, editors, *PADL'05: Proceedings of the seventh International Symposium on Practical Aspects of Declarative Languages*, volume 3350 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 2005.

[AvEP04a] Peter Achten, Marko van Eekelen, and Rinus Plasmeijer. Compositional model-views with generic graphical user interfaces. In *PADL'04: Practical Aspects of Declarative Programming*, volume 3057 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2004.

[AvEP04b] Peter Achten, Marko van Eekelen, and Rinus Plasmeijer. Generic graphical user interfaces. In Greg Michaelson and Phil Trinder, editors, *IFL'03: Selected Papers of the fifteenth International Workshop on the Implementation of Functional Languages*, volume 3145 of *Lecture Notes in Computer Science*. Springer, 2004.

[AvEPvW04a] Peter Achten, Marko van Eekelen, Rinus Plasmeijer, and Arjen van Weelden. Arrows for generic graphical editor components. Technical Report NIII-R0416, Radboud University Nijmegen, 2004.

[AvEPvW04b] Peter Achten, Marko van Eekelen, Rinus Plasmeijer, and Arjen van Weelden. Automatic generation of editors for higher-order data structures. In Wei-Ngan Chin, editor, *APLAS'04: Proceedings of the second Asian symposium on Programming Languages and Systems*, volume 3302 of *Lecture Notes in Computer Science*, pages 262–279. Springer, 2004.

[AvEPvW04c] Peter Achten, Marko van Eekelen, Rinus Plasmeijer, and Arjen van Weelden. GEC: a toolkit for generic rapid prototyping of type safe interactive applications. In Varmo Vene and Tarmo Uustalu, editors, *AFP'04: Revised Lectures of the fifth International Summer School on Advanced Functional Programming*, volume

3622 of *Lecture Notes in Computer Science*, pages 210–244. Springer, 2004.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[BCC$^+$03] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *FMICS: Proceedings of the eighth International Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[Blu91] B.I. Blum. Some very famous statistics. *The Software Practitioner*, 1991.

[BS99] Erik Barendsen and Sjaak Smetsers. *Graph Rewriting Aspects of Functional Programming*, chapter 2, pages 63–102. World scientific, 1999.

[BS04] Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self inspecting code. In *Haskell'04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 69–79. ACM Press, 2004.

[BS07] Arthur I. Baars and S. Doaitse Swierstra. Removing left-recursion using left-corner transformation. Presented at the Dutch functional programming day[1], 2007.

[BTS$^+$98] Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. Java operating systems: design and implementation. Technical Report UUCS-98-015, University of Utah, 1998.

[Bun90] Martin W. Bunder. Some improvements to Turner's algorithm for bracket abstraction. *Journal of Symbolic Logic*, 55(2):656–669, 1990.

[CB04] S. Carlier and J. Bobbio. hOp. The hOp website[2], 2004.

[1] `http://fpdag2007.hypernation.net/publications/FPDag2007-Baars.pdf`
[2] `http://www.macs.hw.ac.uk/~sebc/hOp/`

[CE01] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In Ralf Hinze, editor, *Haskell'01: Proceedings of the ACM SIGPLAN 2001 Haskell Workshop*, pages 41–69. ACM Press, 2001.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*. North Holland, 1958.

[CH93] Magnus Carlsson and Thomas Hallgren. Fudgets – a graphical user interface in a lazy functional language. In *FPCA'93: Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, 1993.

[CJRZ01] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Automated test and oracle generation for smart-card applications. In Isabelle Attali and Thomas Jensen, editors, *E-SMART'01: Proceedings of the International Conference on Research in Smart Cards*, volume 2140 of *Lecture Notes In Computer Science*, pages 58–70. Springer, 2001.

[CK05] David Cok and Joe Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS'04: Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2005.

[CL03] Dave Clarke and Andres Löh. Generic Haskell, Specifically. In Jeremy Gibbons and Johan Jeuring, editors, *Proceedings of the IFIP TC2 Working Conference on Generic Programming*, pages 21–48. Kluwer Academic Publishers, 2003.

[Cla99] Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999.

[CM90] Erik Cooper and J. Greg Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, Carnegie Mellon University, 1990.

[CU90] Wei Chen and Jan Tijmen Udding. Program inversion: more than fun! *Science of Computer Programming*, 15(1):1–13, 1990.

[Cur69] Haskell B. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92, 1969.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[Dij79] Edsger W. Dijkstra. Program inversion. In Friedrich L. Bauer and Manfred Broy, editors, *Proceedings of the Marktoberdorf International Summer School on Program Construction*, volume 69 of *Lecture Notes in Computer Science*, pages 54–57. Springer, 1979.

[Dil88] Antoni Diller. *Compiling Functional Languages*. John Wiley & Sons, 1988.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL'82: Proceedings of the ninth ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[DM99] Catherine Dubois and Valérie Ménissier-Morain. Certification of a type inference tool for ML: Damas–Milner within Coq. *Journal of Automated Reasoning*, 23:319–346, 1999.

[Doc06] Robert Dockins. The GHC type-checker is turing-complete. On the Haskell mailing-list[3], 2006.

[FAJ03] Dave Clarke Frank Atanassow and Johan Jeuring. Scripting XML with Generic Haskell. Technical Report CS-2003-023, University of Utrecht, 2003.

[FGM$^+$05] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: a linguistic approach to the view update problem. In *POPL'05: Proceedings of the thirty-second ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–246. ACM Press, 2005.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL'98: The twenty-fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, 1998.

[3] `http://article.gmane.org/gmane.comp.lang.haskell.general/14088`

[GK04] Robert Glück and Masahiko Kawabe. Derivation of deterministic inverse programs based on lr parsing. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Proceedings of Fuji International Symposium on Functional and Logic Programming (FLOPS)*, volume 2998 of *Lecture Notes in Computer Science*, pages 291–306. Springer, 2004.

[GK05] Robert Glück and Masahiko Kawabe. Revisiting an automatic program inverter for LISP. *TOPPS SIGPLAN Notices*, 40(5):8–17, 2005.

[Gla98] Robert L. Glass. Is there really a software crisis? *Computing Trends (Inaugural column)*, 15(1):104–105, 1998.

[Gla00] Robert L. Glass. Talk about a software crisis — not! *Journal of Systems and Software (Editor's corner)*, 55(1):1–2, 2000.

[Gla06] Robert L. Glass. The Standish report: does it really describe a software crisis? *Communications of the ACM (Practical Programmer)*, 49(8):15–16, 2006.

[GM01] Andy Gill and Simon Marlow. Happy: The parser generator for Haskell. Online web site, 2001. The Happy web site[4].

[Gro] Object Management Group. Unified modeling language. UML Resource Page. The UML web site[5].

[Han02] Keith Hanna. Interactive visual functional programming. In Simon Peyton Jones, editor, *Proceedings of the International Conference on Functional Programming*, pages 100–112. ACM Press, 2002.

[HCNP03] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon Peyton Jones, editors, *AFP'02: Revised Lectures of the fourth International School on Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer, 2003.

[Hin69] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transaction of the American Mathematical Society*, 146:29–60, 1969.

[4]`http://www.haskell.org/happy/`

[5]`http://www.uml.org/`

[Hin97] J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1997.

[Hin00a] Ralf Hinze. *Generic Programs and Proofs*. Habilitationsschrift, Universität Bonn, 2000.

[Hin00b] Ralf Hinze. A new approach to generic functional programming. In *The twenty-seventh Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132, 2000.

[HJ06] Chris Heunen and Bart Jacobs. Arrows, like monads, are monoids. In *MFPS XXVI: Proceedings of the twenty-second Annual Conference on Mathematical Foundations of Programming Semantics*, volume 158 of *Electronic Notes in Theoretical Computer Science*, pages 219–236, 2006.

[HJLT05] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *ICFP'05: The 10th ACM SIGPLAN International Conference on Functional Programming*, 2005. The HOUSE website[6].

[HJSA02] Bastiaan Heeren, Johan Jeuring, S. Doaitse Swierstra, and Pablo Azero Alcocer. Improving type-error messages in functional languages. Technical Report UU-CS-2002-009, Utrecht University, 2002.

[HN01] Frank Huch and Ulrich Norbisrath. Distributed programming in Haskell with ports. In Markus Mohnen and Pieter Koopman, editors, *IFL'00: Selected Papers of the twenty-fourth International Workshop on Implementation of Functional Languages*, volume 2011 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2001.

[HOP03] Engelbert Hubbers, Martijn Oostdijk, and Erik Poll. From finite state machines to provably correct java card applets. In Dimitris Gritzalis, Sabrina De Capitani di Vimercati, Pierangela Samarati, and Sokratis K. Katsikas, editors, *Security and Privacy in the Age of Uncertainty, IFIP TC11 eighteenth International*

[6]`http://programatica.cs.pdx.edu/House/`

Conference on Information Security (SEC2003), volume 250 of *IFIP Conference Proceedings*, pages 465–470. Kluwer, 2003.

[HP01] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Haskell'00: Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[HPF99] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell 98. Online tutorial, 1999. The Haskell 98 tutorial[7].

[HR93] Paul Haahr and Byron Rakitzis. Es: a shell with higher-order functions. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 51–60, 1993.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus*. London Mathematical Society Student Texts. Cambridge University Press, 1986.

[Hug00] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000.

[Hut92] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.

[JH06] Bart Jacobs and Ichiro Hasuo. Freyd is Kleisli, for Arrows. In Conor McBride and Tarmo Uustalu, editors, *MSFP'06: Proceedings of the Workshop on Mathematically Structured Functional Programming*, eWiC. BCS, 2006.

[JJ97] Patrik Jansson and Johan Jeuring. PolyP – a polytypic programming language extension. In *POPL'97: The twenty-fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[JJ99] Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In S. Doaitse Swierstra, editor, *ESOP'99: Proceedings of the eighth European Symposium on Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 1999.

[7]`http://www.haskell.org/tutorial/`

[JJ02a] Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

[JJ02b] Claude Jard and Thierry Jéron. Tgv: Theory, principles and algorithms. In *IDPT'02: The Sixth World Conference on Integrated Design & Process Technology*. Society for Design and Process Science, 2002.

[JOW04] Bart Jacobs, Martijn Oostdijk, and Martijn Warnier. Source code verification of a secure payment applet. *Journal of Logic and Algebraic Programming*, 58(1-2):107–120, 2004.

[JR02] Mark P. Jones and Alastair Reid. *The Hugs 98 User Manual*. The Yale Haskell Group and the OGI School of Science and Engineering at OHSU, 2002. The Hugs web site[8].

[JRB85] Michael S. Joy, Vic J. Rayward-Smith, and F. Warren Burton. Efficient combinator code. *Journal of Computer Languages*, 10(3/4):211–224, 1985.

[Kam76] S. Kamal Abdali. An abstraction algorithm for combinatory logic. *Journal of Symbolic Logic*, 41:222–224, 1976.

[KP02] Pieter Koopman and Rinus Plasmeijer. Layered combinator parsers with a unique state. In Thomas Arts and Markus Mohnen, editors, *IFL'01: Proceedings of the thirteenth International Workshop on the Implementation of Functional Languages*, volume 2312 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2002.

[KP04] Pieter Koopman and Rinus Plasmeijer. Testing reactive systems with gast. In Stephen Gilmore, editor, *TFP'03: Proceedings of the Fourth Symposium on Trends in Functional Programming*, pages 111–129, 2004.

[LCJ03] Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In *ICFP'03: Proceedings of the eighth ACM SIGPLAN International Conference on Functional programming*, pages 141–152. ACM Press, 2003. The Generic Haskell web site[9].

[8]`http://www.haskell.org/hugs/`

[9]`http://www.generic-haskell.org/`

[Lin98] Albert Lin. Implementing concurrency for an ML-based operating system. Master's thesis, Massachusetts Institute of Technology, 1998.

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS 2001-35, University of Utrecht, 2001. The Parsec web site[10].

[LY96] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines – a survey. *Proceedings of the IEEE*, 84(8):1090–1126, 1996.

[Mat] Jim Mattson. The Haskell shell. Online source code. Ralf Hinze's web site[11].

[Md01] Hugues Martin and Lydie du Bousquet. Automatic test generation for java-card applets. In Isabelle Attali and Thomas Jensen, editors, *JavaCard'00: Revised Papers of the First International Workshop on Java on Smart Cards: Programming and Security*, volume 2041 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 2001.

[MPMR01] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. In *PLDI'01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 274–285. ACM Press, 2001.

[MT04] Shin-Cheng Mu and Masato Takeichi. An algebraic approach to bi-directional updating. In Wei-Ngan Chin, editor, *APLAS'04: The Second Asian Symposium on Programming Language and Systems*, volume 3302 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2004.

[Nie] Pat Niemeyer. Beanshell. Online web site. The Beanshell web site[12].

[NN96] Dieter Nazareth and Tobias Nipkow. Formal verification of algorithm $\mathcal{W}$: The monomorphic case. In J. von Wright, J. Grundy, and J. Harrison, editors, *TPHOL'96: Theorem Proving in Higher*

[10]`http://www.cs.uu.nl/~daan/parsec.html`

[11]`http://www.informatik.uni-bonn.de/~ralf/software/examples/Hsh.html`

[12]`http://www.beanshell.org/`

Order Logics, volume 1125 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 1996.

[NN99] Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm $\mathcal{W}$ in Isabelle/HOL. *Journal of Automated Reasoning*, 23:299–318, 1999.

[NSvEP91] Eric Nocker, Sjaak Smetsers, Marko van Eekelen, and Rinus Plasmeijer. Concurrent Clean. In Aarts, Leeuwen, and Rem, editors, *PARLE'91: Proceedings of the Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*, pages 202–219. Springer, 1991.

[OSRS01] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. PVS language reference (version 2.4). Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 2001.

[Ous90] John K. Ousterhout. Tcl: An embeddable command language. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 133–146. USENIX Association, 1990.

[PA06] Rinus Plasmeijer and Peter Achten. iData for the world wide web – programming interconnected web forms. In Ralf Hinze and Andres Löh, editors, *FLOPS'06: Proceedings of the eighth International Symposium on Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 242–258. Springer, 2006.

[Pat01] Ross Paterson. A new notation for arrows. In *ICFP'01: Proceedings of the sixth ACM SIGPLAN International Conference on Functional programming*, volume 36 of *SIGPLAN Notices*, pages 229–240. ACM Press, 2001.

[Pat03] Ross Paterson. Arrows and computation. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 201–222. Palgrave, 2003.

[Pey87] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* International Series in Computer Science. Prentice-Hall, 1987.

[Pey03] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003. The Haskell web site[13].

[PGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL'96: Proceedings of the twenty-third ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308. ACM Press, 1996.

[Pil97] Marco Pil. First class file I/O. In Werner Kluge, editor, *IFL'96: Selected Papers of the eighth International Workshop on Implementation of Functional Languages*, volume 1268 of *Lecture Notes in Computer Science*, pages 233–246. Springer, 1997.

[Pil99] Marco Pil. Dynamic types and type dependent functions. In Kevin Hammond, Antony J.T. Davie, and Chris Clack, editors, *IFL'98: Selected Papers of the tenth International Workshop on Implementation of Functional Languages*, volume 1595 of *Lecture Notes in Computer Science*, pages 169–185. Springer, 1999.

[PRH$^+$99] Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. A semantics for imprecise exceptions. *SIGPLAN Conference on Programming Language Design and Implementation*, 34(5):25–36, 1999.

[PSSC04] André Pang, Don Stewart, Sean Seefried, and Manuel M.T. Chakravarty. Plugging Haskell in. In *Haskell'04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 10–21. ACM Press, 2004.

[PvE02] Rinus Plasmeijer and Marko van Eekelen. *Concurrent Clean Language Report (version 2.1)*. Radboud University Nijmegen, 2002. The Clean web site[14].

[PvW04] Rinus Plasmeijer and Arjen van Weelden. A functional shell that operates on typed and compiled applications. In Varmo Vene and Tarmo Uustalu, editors, *AFP'04: Revised Lectures of the fifth International Summer School on Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 245–272. Springer, 2004.

[13]`http://www.haskell.org/`

[14]`http://clean.cs.ru.nl/`

[Ros97] Brian J. Ross. Running programs backwards: the logical inversion of imperative computation. *Formal Aspects of Computing*, 9(3):331–348, 1997.

[Roy91] Winston Royce. Current problems. In Christine Anderson and Merlin Dorfman, editors, *Aerospace Software Engineering: A Collection of Concepts*. American Institute of Aeronautics, Inc., 1991.

[RTL05] André Rauber Du Bois, Phil Trinder, and Hans-Wolfgang Loidl. mHaskell: Mobile computation in a purely functional language. *Journal of Universal Computer Science*, 11(7):1234–1254, 2005.

[Sch24] Moses Schönfinkel. Über die bausteine der mathematischen logik. In *Mathematische Annalen*, volume 92, pages 305–316. Springer, 1924.

[Sch04] M. Schrage. *Proxima, a Presentation-Oriented Editor for Structured Documents*. PhD thesis, University of Utrecht, 2004.

[SH01] Volker Stolz and Frank Huch. Implementation of port-based distributed Haskell. The Distributed Haskell website[15], 2001.

[Shi94] Olin Shivers. A Scheme shell - the design paper on the Scheme shell scsh. Technical Report MIT/LCS/TR-635, MIT Laboratory for Computer Science, 1994. The Scheme Shell FAQ[16].

[Sta94] Standish Group International. The CHAOS report, 1994. The on-line CHAOS report[17].

[SvW06] Sjaak Smetsers and Arjen van Weelden. Bracket-abstraction preserves typability: A formal proof of Diller-algorithm-C in PVS. In Jordi Levy, editor, *UNIF'06: Preliminary proceedings of the twentieth International Workshop on Unification*, pages 29–43, 2006.

[TB03] Jan Tretmans and Ed Brinksma. Torx: Automated model-based tesing. In A. Hartman and K. Dussa-Zieger, editors, *First European Conference on Model-Driven Software Engineering*. Imbuss, 2003.

[15] `http://www-i2.informatik.rwth-aachen.de/Research/distributedHaskell/`

[16] `http://www.faqs.org/faqs/unix-faq/shell/scsh-faq/`

[17] `http://www.standishgroup.com/sample_research/chaos_1994_1.php`

[Tre96] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[Tur76] David A. Turner. *SASL Language Manual*. St. Andrews University, 1976.

[Tur79] David A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44(2):267–270, 1979.

[vdBvDH$^+$01] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The Asf+Sdf meta-environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction 2001*, pages 365–370. Springer, 2001.

[VP03] Martijn Vervoort and Rinus Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In Ricardo Peña and Thomas Arts, editors, *IFL'02: Selected Papers of the fourteenth International Workshop on Implementation of Functional Languages*, volume 2670 of *Lecture Notes in Computer Science*, pages 101–117. Springer, 2003.

[vW65] Adriaan van Wijngaarden. Orthogonal design and description of a formal language. Technical Report MR 76, Mathematisch Centrum, Amsterdam, 1965.

[vWOF$^+$05] Arjen van Weelden, Martijn Oostdijk, Lars Frantzen, Pieter Koopman, and Jan Tretmans. On–the–fly formal testing of a smart card applet. In Ryoichi Sasaki, Sihan Qing, Eiji Okamoto, and Hiroshi Yoshiura, editors, *SEC'05: Proceedings of the twentieth IFIP TC11 International Information Security Conference*, pages 564–576. Springer, 2005.

[vWP02] Arjen van Weelden and Rinus Plasmeijer. Towards a strongly typed functional operating system. In Thomas Arts and Ricardo Pena, editors, *IFL'02: Selected papers of the fourteenth International Workshop on Implementation of Functional Languages*, volume 2670 of *Lecture Notes in Computer Science*, pages 215–231. Springer, 2002.

[vWP03] Arjen van Weelden and Rinus Plasmeijer. A functional shell that dynamically combines compiled code. In Philip Trinder, Greg Michaelson, and Ricardo Peña, editors, *IFL'03: Selected Papers of the 15th International Workshop on Implementation of Functional Languages*, volume 3145 of *Lecture Notes in Computer Science*, pages 36–52. Springer, 2003.

[vWSP05] Arjen van Weelden, Sjaak Smetsers, and Rinus Plasmeijer. Polytypic syntax tree operations. In *IFL'05: Selected papers of the seventeenth International Workshop on Implementation and Application of Functional Languages*, volume 4015 of *Lecture Notes in Computer Science*. Springer, 2005.

[Wad93] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Proceedings of the Marktoberdorf International Summer School on Program Design Calculi*. Springer, 1993.

[Wan80] Mitchell Wand. Continuation-based multiprocessing. In *LFP'80: Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 19–28. ACM Press, 1980.

[Wan98] Keith Wansbrough. Instance declarations are universal. Unpublished note[18], 1998.

[18]`http://www.lochan.org/keith/publications/undec.html`

Summary

Writing correct software is still difficult due to its increasing complexity, which requires many abstraction layers. We believe that functional programming languages might alleviate some of those difficulties at the source of software construction. Pure and lazy languages, like Clean and Haskell, provide very good support for abstraction, composition, and equational reasoning. They also feature advanced type systems, which can be used to warn programmers about a certain class of mistakes early.

This thesis investigates the usefulness of hybrid static/dynamic typing and polytypic programming by applying these techniques to some common software or programming problems. We show those two techniques "in action" to assess applicability, performance, and expressiveness of the programming techniques themselves, as well as their implementations in both Clean and (Generic) Haskell. The hybrid static/dynamic typing is exciting in the way it extends the type safety of a strongly typed functional language into the run-time world. The polytypic programming approach to writing software is attractive in its promise to reduce the amount of code that needs to be written, to enable better maintenance by adding code instead of rewriting, and to elegantly describe abstract algorithms that work for any data structure.

The first part of this thesis applies hybrid static/dynamic typing to software issues on the design of an operating system and command-line shell. Using strong dynamic type-checking at run-time, we provide type-safe communications and type-safe execution of software components. Clean's dynamic linker gives us the ability to store and retrieve any (functional) expression to and from disk. These typed 'expressions on disk' can be seen as a typed file system and we implemented an interactive command-line shell that infers and checks the type of the entered expression, using the types of files. Bracket abstraction has been used

for implementing a functional language; we use it for type inference/checking. We formally prove that bracket abstraction preserves typability in order to increase our confidence in our type inference using Clean's hybrid static/dynamic typing. We conclude this part by showing an application of the type-checking shell (used as a library) by constructing type safe graphical editors automatically in the polytypic GEC/iData system.

The second part of this thesis applies the polytypic/generic programming technique to a few software construction issues. We present the so-called types–as–grammar approach, which uses parameterized algebraic data types to represent context independent grammars. This is used to generate parsers automatically that construct a rich syntax tree and other generic syntax tree operations. We elegantly abstract over different kinds of parser combinators. Applying an advanced optimization technique called fusion to the generated code yielded great speed-ups, which makes the approach practically usable. We also use the high-level functional abstraction technique called 'arrows' in combination with generating programs in a polytypic way. We attempt to ease the construction of invertible programs by automatically generating the inverse, and generalizing both directions of computation to all data types. We wrap this part up by presenting a case study in testing Java Cards using a polytypic test system that can derive test cases from a functional specification automatically.

We conclude, from the research presented, that Clean's hybrid static/dynamic typing provides us with additional protection against run-time errors by tagging values with their static types. Furthermore, it provides us with a way to store and retrieve any functional expression, even between different programs. Polytypic programming also keeps us from making mistakes in (tediously) writing similar code for many data types. Because the automatic derivation/instantiating of polytypic functions is both type directed and type safe (it cannot introduce type errors), there is less room for programming errors.

Samenvatting

Het blijkt nog steeds moeilijk te zijn om correcte programmatuur te ontwikkelen. Een van de problemen hierbij is de stijgende complexiteit die vele abstractielagen vereist. Wij zijn ervan overtuigd dat functionele programmeertalen sommige van die problemen al bij het bouwen van programmatuur zouden kunnen verminderen. Met name de zuivere en luie (niet strikte) talen, zoals Clean en Haskell, bieden zeer goede ondersteuning voor het samenstellen van programma's uit eenvoudigere onderdelen en het abstraheren en redeneren over programma's. Dergelijke talen bevatten ook geavanceerde typesystemen, die programmeurs vroegtijdig attenderen op bepaalde programmeerfouten.

Dit proefschrift beschrijft een onderzoek naar het gebruik van hybride statische/dynamische typering en de generieke (polytypische) programmeertechniek. Dit wordt gedaan aan de hand van een aantal bekende programmeerproblemen. We bekijken de twee technieken "in actie" om de toepasbaarheid, prestatie en expressiviteit van de programmeertechnieken zelf te beoordelen, alsmede hun implementaties in zowel Clean als (Generic) Haskell. Het hybride typesysteem is interessant omdat het de statische typeveiligheid van een sterk getypeerde functionele taal uitbreidt naar de dynamische wereld tijdens het uitvoeren van programma's. De generieke programmeertechniek is aantrekkelijk omdat het in principe de hoeveelheid code die moet worden geschreven vermindert, de code beter te onderhouden zou moeten zijn door code toe te voegen in plaats van te herschrijven en het elegante abstracte algoritmen ondersteunt die werken op iedere datastructuur.

In het eerste deel van dit proefschrift wordt de hybride typering toegepast bij het ontwerp van besturingssystemen en regelgebaseerde interactieve gebruikersomgevingen (een shell). We laten zien dat sterke dynamische typecontrole typeveilige communicatie en typeveilige uitvoering van programmacomponenten oplevert. Clean's dynamische linker geeft ons de mogelijkheid om willekeurige (functionele) expressies permanent te bewaren en weer in te lezen. Het geheel van deze getypeerde ‘expressies op schijf’ kan gezien worden als een getypeerd bestandssysteem. Wij presenteren een shell die het type van de ingegeven commando’s controleert aan de hand van de types van de bestanden waarna gerefereerd wordt. Bracket-abstractie, een vertaal techniek, is in het verleden gebruikt voor de implementatie van een functionele taal na het controleren van types. Wij gebruiken het echter vóór het controleren en afleiden van types. Om ons ervan te overtuigen dat dit correct is, bewijzen we formeel dat bracket-abstractie typeerbaarheid behoudt. Dit deel eindigt met het gebruik van de typecontrolerende shell (als een bibliotheek) voor het construeren van typeveilige grafische invoervelden in het polytypische GEC/iData-systeem.

Het tweede deel van dit proefschrift past de polytypische taaluitbeiding toe op een aantal programeerproblemen. De zogenaamde types–als–grammatica benadering wordt geïntroduceerd die contextvrije grammatica’s kan beschrijven door middel van algebraïsche datastructuren. Dit wordt gebruikt om automatisch syntactische vertalers (parsers) te produceren die rijke (niet abstracte) syntaxbomen oplevert. Ook worden er andere generieke operaties op syntaxbomen geconstrueerd. Hierbij abstraheren we op een elegante manier over verschillende soorten van parsercombinatoren. Het toepassen van een optimalisatietechniek genaamd fusie op de automatisch gegenereerde code levert een grote snelheidwinst op zodat deze aanpak ook praktisch bruikbaar is. Daarnaast proberen we de bouw van omkeerbare programma’s te vereenvoudigen door de omgekeerde berekening automatisch te laten construeren én beide richtingen van de berekening te veralgemeniseren voor alle datastructuren. Hiervoor gebruiken we een zeer algemene functionele abstractietechniek gebaseerd op zogenaamde “arrows” (pijlen), gecombineerd met het genereren van programma’s op een polytypische manier. Dit deel wordt afgerond met een praktijkvoorbeeld over het testen van chipkaarten (Java Cards) door middel van een polytypisch test– en rapportagesysteem, dat aan de hand van een functionele specificatie automatisch steekproeven uitvoert.

Uit het onderzoek beschreven in dit proefschrift concluderen we dat Clean’s hybride statische/dynamische typesysteem ons extra bescherming geeft tegen fouten tijdens het uitvoeren van programma’s door actuele dynamische waarden te voorzien van hun statische types. Verder hebben we op deze manier de mogelijkheid om praktisch iedere functionele expressie te bewaren of uit te wisselen, zelfs tussen verschillende programma’s. Polytypisch programmeren helpt ook bij het voorkomen van fouten in het (tot vervelends toe) schrijven van vergelijkbare code voor verschillende datastructuren. Ook deze techniek kan dus het aantal fouten in programmatuur verminderen. De automatische afleiding van polytypische functies is namelijk zowel typegeleid als typeveilig.

Curriculum Vitae

1978	Born on October 21, Arnhem, the Netherlands
1990–1996	Highschool (VWO), van Lingen College, Arnhem
1996–2000	Part-time IT-supporter, SAB adviseurs voor ruimtelijke ordening, Arnhem
1996–2001	Master's computer science (doctoraal informatica), Katholieke Universiteit Nijmegen*
2001–2002	Additional researcher on STW project number NWI.4411: *Clean: A Software Development System for Safety Critical Systems*, Katholieke Universiteit Nijmegen*
2002–2003	Researcher sponsored by InterNLnet, Katholieke Universiteit Nijmegen*
2003–2007	Junior researcher, Radboud Universiteit, Nijmegen*

*On September 1st, 2004, the Katholieke Universiteit Nijmegen changed its name to Radboud Universiteit, Nijmegen.

Titles in the IPA Dissertation Series since 2002

Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java - Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of