Automatically Restructuring Programs for the Web

Basileios Anastasatos

Universität des Saarlandes
B.Anastasatos@Gmail.com

Abstract. As interactive Web programs become ubiquitous, developers face a serious design constraint. When a script interacts with a user, it must produce a single Web page and terminate. When the user submits a response, then the script is rerun from scratch. Worse, a user can switch back and forth between pages, clone windows, and submit responses repeatedly or simultaneously. To cope with this inversion of control, the Web developer must structure its programs in a most unnatural manner. We overcome this problem by automatically transforming direct style programs into Web programs, by using well-known program transformation techniques from the functional world. Developers can keep using the languages and tools they know.

1 Introduction

1.1 The Importance of Web Programs

Today, most of the content of the World Wide Web is generated on demand by the so-called *"Web scripts"*, which are no more mere scripts, but full-fledged programs of great complexity and importance, often forming the backbone of giant companies like Yahoo!, or Amazon.com.

Thus, programmers are once more called to tame the complexity of growing specifications and implementations, by writing naturally structured, correct, and maintainable code.

1.2 The Problem

Unfortunately, the widely used Web protocols and technologies, namely Common Gateway Interface (CGI), Java servlets, Java Server Pages, etc., impose a serious constraint on the form of Web programs that interact with the user: after generating a single page, the program *must* terminate, instead of waiting for the user response and perfoming with the rest of the computation. Thus, the programs control information is *erased* between interactions with the user and must be ressurected by hand on each invocation of the program. This problem is addressed by the FastCGI protocol, which allows the connection to be kept alive between user interactions.

A further complication stems from the backtrack and clone capabilities of Web browsers, which enable users to answer multiply one and the same question, either by pressing the back button and re-filling a form, or by cloning windows and filling many copies of one and the same form. Thus, the Web client becomes a co-routine, with interaction points that can be resumed arbitrarily often. FastCGI *cant* cope with this complication.

The ad hoc mechanisms to save and restore the program state between interactions are employed by the programmers, resulting in an unnatural program structure, which fails to match the structure of the interaction, is unnecessary complicated, error prone, and difficult to maintain.

Consider a small example written in SML-like typeless pseudocode. The program asks the user via the function `read` for two numbers and prints their sum.

```
fun input msg = print msg; read

fun adder = print (input "1st number?") + (input "2nd number?")
```

The first, direct style version of the program is computation-driven, natural, and therefore easy to understand.

But once we are no more allowed to wait for the function `read` to return, things get weird:

```
fun produce-html msg hidden-mark env = ...

fun adder =
  hidden-mark = extract-hidden-mark
  env = extract-env
  ans = extract-answer
  if hidden-mark = undefined then
    produce-html "1st number?" "step 1" []
  else if hidden-mark = "step 1" then
    produce-html "2nd number?" "step 2" [ans]
  else if hidden-mark = "step 2" then
    produce-html ((hd env) +ans) "done" []
```

In the second version, which is written for the Web, the function for turning a message into HTML-code `produce-html`, also hides a mark and an enviroment in the generated page, encoding the current state of the computation. Now the main function `adder` also acts as a dispatcher, searching in the headers of the HTTP-request for the mark and enviroment, and resuming the process from the corresponding point.

What a mess for such a simple task! Neither the purpose, nor the correctness of the program are clear. How would a script for adding 5, or, generally, n numbers look like?

1.3 Employing Continuations

The way to solve the problem is by storing the current *continuation* and retrieving it when the data the process waited for become available. A continuation can be thought as the rest of the computation that waits for the user to enter some data. The continuations can be stored either on the server, or on the client.

Server-sided storing In the approach developed by Christian Queinnec [2000], the continuations are grabbed by special control mechanisms (function `call/cc`) found in a few languages, and stored in a hash table on the server. A URL corresponding to the current continuation is embedded in the generated HTML-page, and the continuation is reinvoked upon request of the corresponding URL.

Disadvantages But as most languages can't directly manipulate continuations, all existing infrastructure becomes useless, as everything has to be rewritten in another language.

Furthermore there is no way to cope with a most serious distributed garbage collection problem. The server has no way to know whether a continuation might still be invoked. Even browser support, that could report to the server whether the page was bookmarked or not, wouldn't be enough, because the user can still write down the URL on a piece of paper or remember it by heart! The obvious solution of timeouts is far from perfect. Moreover, accidental power outages or server upgrades would delete the continuations for ever.

2 Client store

In a second approach, which is the contribution of the paper, the continuation is stored on the client, by encoding it as a character string and embedding it in the HTML-page sent to the client. The solution does not make use of `call/cc` constructs, thus is suitable for *every* programming language. In this way legacy programs may be reused, and programmers can keep using the old good languages, tools, and programming techniques they know best.

But how can we grab continuations, when the language doesn't expose them as first-class objects? By employing three well-understood, meaning-preserving program transformations, which were initially developed to compile functional languages.

2.1 Continuation Passing Style

The first transformation rewrites the program in Continuation Passing Style (CPS):

```
fun input msg f = print msg; f read
```

```
fun adder =
  input "1st number?"
        λn0 => input "2nd number?"
                     λn1 => print n0 + n1
```

Now the function `input` is passed an additional argument `f`, the continuation. Instead of returning the result of the call to `read`, it calls `f` with the result of `read`.

2.2 Lambda lifting

Although already a progress, CPS is not enough, because the continuation can be an *anonymous* lambda nested inside another function. In the example this is the case with *both* continuations. But then how can we call a function, if it has no name?

The second transformation, called *lambda lifting*, flattens the program structure by making all functions global and endowing them with unique names, so that they can be addressed individually.

```
fun input msg f env = print msg; f env read
fun adder = input "1st number?" f0 []
fun f0[]n0 = input "2nd number?" f1[n0]
fun f1[n0]n1 = print n0 + n1
```

Creating closures Unfortunately, tearing a function out of its context may leave some variables unbound. To cope with this problem, we bind such variables by passing the *context* as an additional argument list. Thus the functions become *closures*.

2.3 Defunctionalization

Now the function `input` is passed uniquely named functions. One last difficulty is that these functions, unlike numbers or strings, are no representable objects, so that they can't be marshalled in an HTML-page. We circumvent this problem by storing the functions in a *vector*, which can be indexed, and passing not the function itself, but the *index* to the function, i.e. an integer number. The special function `apply` looks up a function in the vector and executes it. This step of practically eliminating high-order functions is called *defunctionalization*.

```
fun input msg idx env = print msg; apply idx env read
vector funs = {f0, f1}
fun apply idx env = funs.idx env
fun adder = input "1st number?" 0 []
funf0[]n0 = input "2nd number?" 1[n0]
funf1[n0]n1 = print n0 + n1
```

2.4 Freezing the continuations

Till now all transformations preserved the semantics of the program. As a last step, the function `input` which is the only one that interacts with the user, no more throws the user input to the continuation, but instead of waiting for user input, stores the continuation in an HTML-page, which is now an easy task, as the continuation is represented by a number and a list of values, and *terminates*. The main function `adder` acts as a dispatcher, extracting the continuation together with the user response, and applying the continuation to the response.

```
fun input msg idx env =  produce-html msg idx env

vector funs...
fun apply...
fun f0...
fun f1...

fun adder =
  idx = extract-idx
  env = extract-env
  ans = extract-answer
  apply idx env ans
  handle NoCont =>
    input  "1st number?"  0 []
```

2.5 Miscellaneous

Store in cookies; race conditions The store, i.e. the mutable variables, shouldn't be stored in the continuations, because it is shared between all continuations. A natural place to store it is a cookie on the client side. This simple solution suffers from race conditions that arise when the client sends a request with the cookie attached, and then a second one, *before* the updated cookie from the first request reaches the client. Thus, the changes made to the cookie after the first request are lost.

A simple solution is to introduce *sequence numbers*, that are stored both on the server and in the cookie. If a cookie is outdated, then the server can detect it, because the sequence numbers don't match. The server side storage management problem comes from the window, but this time it is no more a severe problem, because only one integer number per client has to be stored on the server.

Security and efficiency A malicious user could forge the continuation or the cookie, in order to execute arbitrary code with arbitrary arguments. Curious users could inspect the continuation to gather sensitive information. Both problems are easily addressed by encrypting and signing the continuation and the cookie.

The continuations and the store can grow inconvieniently big, in which case we can compress them in order to spare bandwidth.

Debugging The transformed code, which is the one actually executed, bears only a passing structural resemblance to the code written by the programmer, thus making debugging difficult and tiring.

No general solution has been found to this problem. For languages that can directly manipulate continuations, the ad hoc solution is to not transform the original code and to reimplement the function `input` to grab the current continuation and store it on the server instead of embedding it in the HTML-page, i.e. Queinnec's approach is followed.

3 Criticism

This elegant solution seems to solve in a most unexpected way the most serious problem of Web programming.

Storing the continuation and the mutable variables on the client side seems like abandoning sensitive information at the mercy of the user. Although thus the garbage collection problem is solved, possibly a serious security hole is introduced.

Developers in all high-level languages use debuggers to test the original program, although in reality the translated one is executed. Would it be difficult to adapt this solution to the world of the Web?

References

[2001] Graunke P., Krishnamurthi S., Findler R. B., Felleisen M.: Automatically Restructuring Programs for the Web.

[2004] Matthews J., Graunke P., Krishnamurthi S., Findler R. B., Felleisen M.: Automatically Restructuring Programs for the Web.

[2000] Queinnec C.: The influence of browsers on evaluators or, continuations to program Web servers. In ACM SIGPLAN International Conference on Functional Programming.

[1992] Appel A. W.: Compiling with Continuations. Cambridge University Press.

[1985] Johnson T.: Lambda Lifting: Transforming Programs to Recursive Equations. In Proceedings of the Conference on Functional Programming Languages and Computer Architecture. Nancy, France.

[1972] Reynolds J. C.: Definitional Interpreters for Higher-Order Programming Languages. In Proceedings of the 25th ACM National Conference. pp. 717-740.

Written Presentation: JoCaml, a Language for Concurrent Distributed and Mobile Programming

Nicolas Bettenburg[1]

Universitaet des Saarlandes, D-66041 Saarbruecken,
nicbet@studcs.uni-sb.de

Abstract. As traditional programming languages are designed for closed, sequential architectures, concurrent and distributed programming is either hard to achieve or error-prone. JoCaml as a high-level language was designed with dedicated support for concurrency, synchronization as well as the support for distributed execution of programs. JoCaml itself is based on the Join Calculus and extends the OCaml language, keeping all of it's features. This written presentation will show how JoCaml can be used to achieve concurrent and distributed programming, with the main focus on concurrency.

1 Introduction

Distributed, as well as concurrent programs, are usually hard to write and understand - even harder to debug or proved to be correct due to asynchrony and nondeterminism. JoCaml is an attempt to provide a functional high-level language with support for distributed and concurrent programming. Traditional high-level distributed programming languages rely heavily on scripting languages, which are often specialized and lack elements of structure like modules, classes or user-defined types, crucial for flexible programming of large projects.

JoCaml is based on the OCaml programming language and thus inherits all of it's features, which make it a feasible setting for distributed programming.

- *Static typing:* Important for distributed programming, since debugging runtime errors is very hard to cope with.
- *Byte-code compilers:* Providing separate compilation and flexible linking as the key part for JoCaml's implementation of code mobility at runtime.
- *Low-level support:* Good support for low-level system programming.

JoCaml extends OCaml in such way, that OCaml programs and libraries are just a special kind of JoCaml programs and libraries ($OCaml \subseteq JoCaml$).

2 Extending OCaml with support for Concurrent Programming

OCaml is a sequential programming language, so every expression is executed in a deterministic way in call-by-value. Since concurrency is desired, the first

extension of JoCaml to OCaml is the support for lightweight concurrency, message passing and a mechanism for message-based synchronization. JoCaml at this point introduces a new expression called **spawn** with the syntax:

```
spawn process; expression
```

executing process and evaluating expression in parallel. This means, that the operations in process and expression run independently (concurrent) in a non-deterministic way.

Since OCaml only has support for expressions, JoCaml introduces a new syntactic class *process* which is recursively defined with expressions, handing the real mapping of the processes to system threads over to the JoCaml compiler and runtime.

There is another thing, special to processes: they are not meant to return a result; their only means of interaction is sending asynchronous messages on channels. Such an asynchronous message is a process itself. In order to do so, JoCaml introduces *channels* and *local channel definitions* for processes: they are first-class values with a communication type used to form expressions and send messages. Channel definitions bind names with a static scope and attach guarded processes with these names. By passing a message over these names, a copy of the guarded processes is executed.

In order to provide synchronization facilities, JoCaml holds on to the ML paradigm: definition by pattern-matching, to provide a declarative way of specifying inter-process synchronization. This in fact does not export state: it leaves the state inside the process. This is done by allowing the joint definition of a number of channels by matching concurrent message patterns on these channels. This way of synchronization via patterns was first introduced in the join calculus [1] in 1996 [1] and has several advantages for compilation efficiency and efficient implementation of routing.

3 Extending OCaml with support for Distributed Programming

We have previously seen how concurrency was introduced to OCaml. This section will show how this can be used across several machines on an asynchronous network with distributed message passing and even process mobility. The programming model proposed is based on the Join Calculus as stated earlier before and is characterized by an explicit notion of locality: since there is the need to represent a set of runtimes with their local processes on a network, the join calculus defines a basic unit of locality - called *location*.

– Locations have a *nested structure* (so they can contain sub locations); in fact, a whole JoCaml program itself is a location called the *root location*.

[1] The Join calculus itself is based on the π-calculus [2] which was originally proposed by R. Milner as a progress to the original λ-calculus

A configuration of machines, distributed over the network and executing JoCaml programs can thus be seen as a location tree; each location having it's own definitions and processes.
- Locations are *transparent*; channels have a global lexical scope, so any process that has received a channel name can used it independently of the location that defined the channel name.
- Locations form *units of mobility*; At any time, a location (together with it's content) can be migrated from one machine to another. Location names can be passed in messages and the be used as target addresses for such migrations.
- Locations represent *atomic units of failure*; They can be used to detect failure, halt the execution of all the location's content or implement failure recovery mechanisms.

4 Concurrent programming in JoCaml

As addressed in section 2, JoCaml extends the OCaml language with some new language features, which enable concurrent programming. We will now take a closer look on these extensions.

4.1 Channels in JoCaml

Channels (also called *port names*) are the main new primitive values in JoCaml. There are two kinds of channels: synchronous and asynchronous, depending on their usage for communications. In either flavor, a new channel is introduced by a new `let def` binding; the right hand side of the channel definition is the process fired for every message sent via the channel name.

Asynchronous channels are defined by

```
# let def channelname! varname = process
val channelname: <<type>>
```

Syntactically, the presence of a ! in the definition of a channel's name indicates it's asynchrony. The channel has type `<<x>>`, where x is the type of the values the channel carries. Since it is an asynchronous channel, the execution of process is concurrent.

Synchronous channels are defined by

```
# let def channelname varname = process; reply
val channelname: a -> b
```

Syntactically, the absence of a ! in the definition of a channel's name indicates it's synchrony. The channel has type `a -> b` which reminds of a function. The difference is, that a synchronous channel must explicitly send back some values as results using reply - a function implicitly returns the value of their main body.

Message sending on asynchronous channels appears in processes, message sending on synchronous channels appears in expressions (as if they were functions). This partition of channel usage is one possible explanation for the design decision to have two different types for asynchronous and synchronous channels (since the type checker should flag an error whenever a channel is used in the wrong context).

Since channels are first-class values in JoCaml, they can also be sent and received in messages (often referred to as *name mobility*[2]) which adds to the expressiveness of JoCaml. One can write higher-order functions and ports (for example turning a function into an asynchronous channel) or have polymorphic types for channels.

The channels introduced to JoCaml are *uni-directional* channels. However - using a concept called join-patterns, one can also define *bi-directional* channels in JoCaml.

4.2 Expressions in JoCaml

Expressions are the same as they were in OCaml: they are executed in a synchronous, deterministic call-by-value manner and produce a value when they finish. The most basic expression sends a message on a synchronous channel and waits for the result (blocks). For example, the expression

```
# let x=1 in print(x); print (x+1);
=> 12
```

sends two times on a synchronous channel called print and always evaluates to the empty result (since the print channel has type `int->unit` in this case) while outputting `12`.

4.3 Processes in JoCaml

Processes are the main new syntactic class in JoCaml. Since only declarations and expressions are allowed at top-level of a JoCaml program, JoCaml provides the *spawn* keyword to turn a process into an expression. The most basic process sends a message on an asynchronous channel.

```
spawn { ... }
```

Processes can be group by using braces "{}" and composited for concurrent execution via "|", for example

```
spawn { ... {... | ...} | ...}
```

Process composition also includes conditionals (if then else), functional matching (match with) and local binding (let in, let def in). Sequences may also appear inside processes, with the general form `expression; process`. This form is due to the fact, that processes do not return values - so the value generated by expression must be discarded. As Processes are executed concurrently, unlike expressions, the code

```
# spawn {echo 1 | echo 2}
```

produces two possible outputs: either `12` or `21` depending on which of the processes ({echo 1} or {echo 2}) was executed first.

4.4 Synchronization Patterns

Join patterns, as introduced with the join calculus[1] extend channel name definitions with synchronization. Such a pattern defines a set of channels at once and specifies a synchronization condition to receive messages on these channels. For example, in

```
#let def channelnameX! var1 | channelnameY! var2 = process
```

there must be messages on channelnameX and channelnameY to trigger the execution of the guarded process on the right-hand side. When multiple message matches are available, which message will be consumed is non-deterministic. Once again, the composition operator "|" is used to form the join patterns.

- Join patterns are the programming paradigm for concurrency in JoCaml. They allow the encoding of many concurrent data-structures.
- Join patterns can mix up synchronous and asynchronous channel definitions.
- If multiple synchronous channels are defined in a join pattern, each `reply` construct must specify the name to which it replies via `reply to name`.

  ```
  # let def channelX var1 | channelY var2
     = reply to channelX | reply to channelY
  ```

- Several join patterns can be co-defined with the keyword `or`.

  ```
  # let def channelX! var1 | channelY! var2 = process1
         or channelX! var1 | channelZ! var3 = process2
  ```

With join patters, many common programming styles, either sequential or concurrent, can be expressed. For example explicit synchronization points in the parallel execution of tasks, a so-called *synchronization barrier* can be expressed with the usage of two synchronous channels in a join pattern:

```
#let def synch1 () | synch2 () = reply to synch1 | reply to synch2
val synch1: unit->unit
val synch2: unit->unit
```

This pattern can now be used for example for process interleaving, by spawning to processes that alternately send an empty message on synch1 and synch2.

The generalization of synchronization barriers is the *Join/Fork Parallelism*, which defines a variant, that performs to computations in parallel and then joins the results.

5 Join-Calculus and other programming languages

The concepts of explicit locality and join patterns are not restricted to JoCaml, or the functional world at all. They can even be transferred to an imperative setting like C# [3]: Processes are here so-called *asynchronous methods*:

```
async myMethod(...) {
    // Method Body
}
```

The execution of the method body is scheduled in a different thread. Again, asynchronous methods (like processes) do not produce a result value, so the `void` keyword is substituted by the `async` keyword.

Join Patterns are here so-called *chords*:

```
class myClass {
    string Get() & async Put(string s) {
        return s;
    }
}
```

In this chord, execution of method Get() blocks until a string is provided via put (if not before). Detailed information on Polyphonic C# can be found in [3].

6 Conclusion

We have seen that the Join Calculus is a nice base for a new type of programming model for concurrent and distributed programming. It is not restricted to the functional world, however, OCaml as the base language with JoCaml as it's extension simplifies concurrent programming (in regard of traditional lock-based approaches). As a difference to another approach to concurrent programming, we have seen in the Seminar: Transactional Memory, the programmer in JoCaml has to keep concurrency in mind while writing the program, where as with the transactional memory approach, he could later on make already existing code safe for concurrency by using `atomic`. Nonetheless, JoCaml also has some draw-backs, which are actually based on the way locations are handled: the programmer has to define suitable portions of code (locations) - suitable in a regard to error handling. Asynchronous processes just print an exception to std-out, synchronous processes terminate with the exception instead of the reply, but the complete failure recovery mechanism is left to the programmer.

References

[1] C. Fournet and G. Gonthier: The join calculus: a language for distributed mobile programming, Applied Semantics. International Summer School, APPSEM 2000

[2] R. Milner: Communication and Mobile Systems: the π-Calculus, Cambridge University Press, 1999

[3] N. Benton, L. Cardelli and C. Fournet: Modern Concurrency Abstractions for C#, FOOL9 workshop, January 2002, Portland, Oregon - Microsoft Research

Dynamic Typing in a Statically Typed Language

[M. Abadi, L. Cardelli, B. Pierce, G. Plotkin]

Matthias Berg
Advisor: Andreas Rossberg

Saarland University, Programming Systems Lab / Software Engineering Chair
Seminar: Advanced Functional Programming

Abstract. Dynamic typing can be useful in statically typed languages. We extend the simply typed λ-calculus with dynamic typing and elaborate additional features like polymorphism and subtyping.

1 Introduction

There are situations, when even statically typed languages need to perform dynamic type checks. Examples are the handling of persistent storage or interprocess communication. If a process receives some data from another process, it cannot rely on this data to be of some expected type. The type has to be checked dynamically.

Another example are heterogeneous data structures. For instance if a language supports lists which can contain values of different types at the same time, then prior to the usage of an element of such a list, its type must be checked. This can only be done dynamically.

The function `eval` takes an expression as argument and evaluates it. The type of the result can only be determined dynamically. This is a further example were a statically typed language needs dynamic type checking.

Most of the work we present here is based on the papers [1] and [2], which propose to use a type called `Dynamic` to allow dynamic type checking. Values of this type are constructed by pairing a value with its type. Since such values contain a type, we can check this type dynamically. This inspection is done by a `typecase` construct.

Dynamic values are of type `Dynamic` and they can contain values of any type. Therefore it is easy to construct heterogeneous data structures in a language which supports `Dynamic`. For example a list, which can contain values of different types, is simply a list of dynamic values.

2 λ-Calculus with `dynamic` and `typecase`

We give now a formal definition of an extension of the simply typed λ-calculus, which knows the type `Dynamic` and a `typecase` construct. The syntax is rather simple:

$\tau \in \mathit{Typ} ::= X \mid \tau \to \tau \mid \texttt{Dynamic}$
$e \in \mathit{Exp} ::= x \mid \lambda x{:}\tau \,.\, e \mid e\ e \mid \texttt{dynamic}(e{:}\tau) \mid \texttt{typecase}\ e\ \texttt{of}\ x{:}P\ .\ e\ \texttt{else}\ e$

The differences to the simply typed λ-calculus are the type `Dynamic` and the new expressions with `dynamic` and `typecase`. Values of type `Dynamic` (upper case) are constructed by `dynamic` (lower case). It is used to pair an expression with its type.

The `typecase` construct takes an expression of type `Dynamic` as argument and checks whether its contained type matches the pattern P. So far a pattern is just a type and the check is a simple equality test. If the match succeeds, the expression following the pattern is evaluated, where the variable x is substituted with the value contained in the dynamic value. So `typecase` does not only dynamic type checking, it also gives us access to the value contained inside a dynamic value. If the type check does not succeed, the expression of the `else` branch is evaluated.

For example, the following function checks whether its argument contains a number (Assuming, the language supports numbers). If the check succeeds, the result is the number increased by one, otherwise the result is zero.

$$\lambda e{:}\texttt{Dynamic}\ .\ \texttt{typecase}\ e\ \texttt{of}\ x{:}\texttt{Nat}\ .\ x + 1\ \texttt{else}\ 0$$

We now give reduction rules for the two new constructs. The others behave as in the simply typed λ-calculus. The rule for `dynamic` is rather simple. It states that the inner expression should be reduces to a value (Values are λ-expressions and `dynamic`-expressions, which contain a value):

$$\frac{e \Rightarrow v}{\texttt{dynamic}(e{:}\tau) \Rightarrow \texttt{dynamic}(v{:}\tau)}$$

The `typecase` construct requires two rules, since the pattern matching can succeed or fail. First, the expression following the `typecase` must reduce to a `dynamic`-value. Its contained type τ is matched with pattern P (This is the side condition). If the match succeeds ($\tau = P$), we reduce e_2, where the variable x is substituted with v_1, the value inside the `dynamic` expression. Otherwise we reduce e_3.

$$\frac{e_1 \Rightarrow \texttt{dynamic}(v_1{:}\tau) \qquad e_2[x := v_1] \Rightarrow v_2}{\texttt{typecase}\ e_1\ \texttt{of}\ x{:}P\ .\ e_2\ \texttt{else}\ e_3 \Rightarrow v_2}\ \tau = P$$

$$\frac{e_1 \Rightarrow \texttt{dynamic}(v_1{:}\tau) \qquad e_3 \Rightarrow v_3}{\texttt{typecase}\ e_1\ \texttt{of}\ x{:}P\ .\ e_2\ \texttt{else}\ e_3 \Rightarrow v_3}\ \tau \neq P$$

Now that we know the behaviour of `dynamic` and `typecase`, their typing rules should be straightforward. For `dynamic` we require that its inner expression really has the claimed type:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{dynamic}(e{:}\tau) : \texttt{Dynamic}}$$

The first expression in the `typecase` construct must have the type `Dynamic`. Furthermore the last two expression must have the same type τ, the type of the whole construct. Additionally the environment of e_2 is extended with the variable x of type P, since x is substituted with something of this type, if the match succeeds during the reduction.

$$\frac{\Gamma \vdash e_1 : \texttt{Dynamic} \quad \Gamma, x{:}P \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{typecase } e_1 \texttt{ of } x{:}P \texttt{ . } e_2 \texttt{ else } e_3 : \tau}$$

As an example we now write a function which, given two dynamic values, tries to apply the first to the second:

```
λdf:Dynamic . λdx:Dynamic .
 typecase df of
   f:Nat → Nat .
     typecase dx of
       x:Nat . f(x)
     else 0
 else 0
```

This function checks whether its first argument contains a function mapping numbers to numbers and whether its second argument contains a number. But how can we write such a function, which applies functions of arbitrary types to their arguments? This problem brings us to the subject of the following section.

3 Pattern Variables

The problem mentioned in the last section arises from the fact, that every pattern only matches a single type. If we allow the patterns to contain pattern variables, we will obtain a more expressive `typecase` construct. With pattern variables we can match parts of types. For example the pattern $U \to V$ with pattern variables U and V matches any functional type. A successful match binds U to the argument type and V to the result type of the function. We can now write a function which applies functions of arbitrary types to their arguments:

```
λdf:Dynamic . λdx:Dynamic .
 typecase df of
   {U,V} f:U → V .
     typecase dx of
       {} x:U . dynamic(f(x):V)
     else dynamic(...)
 else dynamic(...)
```

In front of every pattern we write its patters variables in braces. This is done in order to distinguish them from previously bound variables. For example the first pattern contains the pattern variables U and V. The pattern of the inner `typecase` contains no pattern variables. The U is the variable which was bound in the first pattern. This way we check whether the type of the second argument is equal to the argument type of the function.

Interestingly the result cannot be just $f(x)$. We need to pack it again with a `dynamic` expression. This is because of the requirement, that the `else` branches must have the same type as the matching branch and there is no way to construct something of type V, but we can easily build something of type `Dynamic`.

4 Polymorphism

In the following sections we will discuss some possible extensions of our calculus. It is easy to include polymorphism like in System F [3]. Here polymorphism is modelled by functions which take types as argument. Such functions are written $\lambda X\ .\ e$. For example the polymorphic identity function is written $\lambda X\ .\ \lambda x{:}X\ .\ x$ and has the type $\forall X\ .\ X \to X$. If this function is applied to some type, it reduces to the identity function of this type: $(\lambda X\ .\ \lambda x{:}X\ .\ x)\ [\texttt{Nat}] \Rightarrow \lambda x{:}\texttt{Nat}\ .\ x$

The integration of this scheme into our calculus is straightforward. The following example illustrates the use of a `typecase` which matches a polymorphic function f mapping lists to lists and returns a polymorphic function which, given a type and a list x of values of this type, applies f to the reverse of x:

$\lambda df{:}\texttt{Dynamic}\ .$
 `typecase` df `of`
  $\{\}\ f{:}\forall X\ .\ \texttt{List}\ X \to \texttt{List}\ X\ .$
   $\lambda Y\ .\ \lambda x{:}\texttt{List}\ Y\ .\ f\ [Y]\ (\texttt{reverse}\ [Y]\ x)$
 `else` $\lambda Y\ .\ \lambda x{:}\texttt{List}\ Y\ .\ x$

5 Higher-Order Pattern Variables

Interestingly our first-order pattern variables are not expressive enough in matching against polymorphic types. There is no pattern which matches any polymorphic function in a suitable way. For example the types $\forall X\ .\ X \to X$ and $\forall X\ .\ \texttt{List}\ X \to \texttt{List}\ X$ are incompatible, i. e. there is no non-trivial pattern which matches them both. One might think that the pattern $\forall X\ .\ U \to U$ with pattern variable U does the job, since a successful match can bind U to X or to $\texttt{List}\ X$. But this causes a scoping problem, since U can be used outside the scope of type variable X. This introduces a new free type variable at runtime.

A solution to this problem are higher-order pattern variables. When such a variable is matched, it is not bound to some type, but to a function mapping types to types. The following pattern matches the two types from above by using a second-order pattern variable F: $\forall X\ .\ F\ X \to F\ X$. A successful match binds F to the identity on types, $\Lambda X\ .\ X$, or the following function: $\Lambda X\ .\ \texttt{List}\ X$.

Now X is passed to F as an argument, making F independent of X. This solves the scoping problem from above.

6 Subtyping

If we include subtyping in our language, this has some implications on our pattern matching. A type T should match a pattern P, if $T \leq P$, i. e. if T is a subtype of P. For example, since `Nat` $\leq$ `Int`, the following match should succeed:

```
typecase dynamic(5:Nat) of
  x:Int . ...
else ...
```

Unfortunately, the binding of pattern variables is not an easy task any more. In general there is no unique solution for this problem. For example it is clear, that the type `Int` $\to$ `Nat` matches the pattern $U \to U$. The types `Int` $\to$ `Int` and `Nat` $\to$ `Nat` are both supertypes of `Int` $\to$ `Nat` and valid instances of the pattern. But neither of them is a subtype of the other, so there is no reason to prefer one solution to the other.

This problem can be avoided by only allowing linear patterns, i. e. patterns where a pattern variable occurs at most once. But this approach is a bit too restrictive. Another solution is to add subtyping constraints and to perform exact matching. Here the matching works as in the previous sections, where we did not have subtyping, but additionally we check after the matching whether some specified subtyping constraints are met. So first we do an exact pattern matching and check the constraints afterwards. To match anything which is a subtype of `Int`, we write something like this:

```
typecase dynamic(5:Nat) of
  {U ≤ Int} x:U . ...
else ...
```

Here the match binds U to `Nat` and then it is checked that `Nat` $\leq$ `Int`. The problem from above is avoided, since there is simply no way to exactly match the type `Int` $\to$ `Nat` with pattern $U \to U$.

7 Abstract Data Types

Another problem that arises with dynamic types is that the `typecase` construct destroys parametricity, i. e. now the reduction is not independent of types any more. Unfortunately type abstraction relies on parametricity to hide its representation. Dynamically, abstract types are just their representation types, hence `typecase` can be used to expose them. Solutions to this problem include the dynamic generation of new type names to restore type abstraction [4, 5].

8 Example Languages

There are programming languages which realise some of the presented ideas. For example the logic language Mercury knows the type `univ`, which corresponds to our type `Dynamic`. It also includes the predicates `type_to_univ` and `univ_to_type` to convert any value into something of type `univ` and back [6].

Haskell also includes dynamic typing. The GHC knows the type `Dynamic` which is realised via the type class `Typeable` of types with a known representation [7]. This representation is compared with the expected type's representation during the dynamic checks. This form of dynamic typing only works for monomorphic types.

The language Clean has a quite expressive dynamic typing. It includes pattern matching with first-order pattern variables and also deals with polymorphism [8].

The language Alice ML provides dynamic typing through the concept of packages. The constructs `dynamic` and `typecase` correspond to the operations to create and open packages, `pack` and `unpack` [9]. Alice ML supports subtyping which also applies to the type check performed by `unpack`.

References

1. Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991.
2. Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, 1995.
3. Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analysis, et son application à l'élimination des coupures dans l'analysis et la théorie des types. In J. E. Fenstad, editor, *Proceedings 2nd Scandinavian Logic Symposium*. North-Holland, 1971.
4. Andreas Rossberg. Generativity and dynamic opacity for abstract types. In Dale Miller, editor, *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, Uppsala, Sweden, August 2003. ACM Press.
5. Matthias Berg. Polymorphic lambda calculus with dynamic types. Bachelor's thesis, Programming Systems Lab, Saarland University, October 2004.
6. F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. *The Mercury language reference manual*. University of Melbourne, http://www.cs.mu.oz.au/mercury, 1996.
7. Simon Marlow, Simon Peyton Jones, and Others. *The Glasgow Haskell Compiler*. University of Glasgow, http://www.haskell.org/ghc/, 2002.
8. Marco Pil. First class file i/o. In *IFL '96: Selected Papers from the 8th International Workshop on Implementation of Functional Languages*, pages 233–246, London, UK, 1997. Springer-Verlag.
9. Alice Team. *The Alice System*. Programming System Lab, Universität des Saarlandes, http://www.ps.uni-sb.de/alice/, 2005.
10. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, February 2002.

A Functional Graph Library

Christian Doczkal

Universität des Saarlandes

Abstract. Algorithms on graphs are of great importance, both in teaching and in the implementation of specific problems. Martin Erwig proposes an inductive view on graphs to achieve a concise notation, persistence and efficiency. I will show that, though meeting each of these goals, the proposed solution fails to archive all three at the same time, especially when extending the term "efficiency" beyond time complexity.

1 Introduction

I will provide an overview and and an evaluation of the inductive view on graphs and its implementation as presented in [Erw01]. Section 2 covers the inductive graph model and presents active patterns which are very convenient when dealing with inductive graphs. In section 3, two possible implementation ideas for the inductive graph model are briefly discussed and compared. In section 4, I will give an example of an inductively defined graph algorithm and comment on its expressiveness, efficiency, and suitability for programming and also for teaching, which is also one of Erwig's goals.

2 Inductive Graph definition

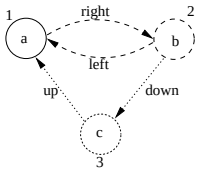
Functions defined over inductive data types like trees are usually very expressive and elegant. By providing an inductive definition for graph data structures, those benefits can also be applied to functional graph algorithms. Erwig proposes such an inductive definition a well as an implementation that allows inductive graph algorithms to be implemented with the same asymptotic complexity as their imperative counterparts. The paper only deals with directed, node and edge labeled multi graphs, since other graph types are merely special cases of this definition. Intuitively, inductive graphs can be seen as an algebraic data type defined as follows, where & is an infix constructor (grouping right).

$$
\begin{aligned}
&\mathit{type}\ \mathit{Node} &&= \mathit{Int}\\
&\mathit{type}\ \mathit{Adj}\ b &&= [(b, \mathit{Node})]\\
&\mathit{type}\ \mathit{Context}\ a\ b &&= (\mathit{Adj}\ b,\ \mathit{Node},\ a,\ \mathit{Adj}\ b)\\
&\mathit{data}\ \mathit{Graph}\ a\ b &&= \mathit{Empty} \mid \mathit{Context}\ a\ b\ \&\ \mathit{Graph}\ a\ b
\end{aligned}
$$

The above definition uses the invariant that upon insertion into the graph, every node that is mentioned in the context of the newly inserted node must already be present in the graph. Although this term representation is insufficient

for implementation purposes, due to the invariant and the need to efficiently retrieve specific nodes, it is a good intuition for writing algorithms. Figure 1 shows a graph and one possible term representation. First the solid then the dashed and finally the dotted elements are inserted.

Fig. 1. Inductive Graph with term representation



$([(\text{“}down\text{“}, 2)], 3, 'c', [\text{“}up\text{“}, 1)])$ &
$([(\text{“}right\ \text{“}, 1)], 2, 'b', [\text{“}left\text{“}, 1)])$&
$([], 1, 'a', [])$ & *Empty*

The above definition also provides two important facts that are important for implementation purposes and for the definition of a new kind of pattern matching called *active patterns*, which provides a very concise notation for graph algorithms.
Fact 1(Completeness) : Each labeled multi-graph can be represented by a graph term.
Fact 2 (Choice of Representation):For each graph g and each node v contained in g there exist p, l, s, and g' such that $(p, v, l, s)\&g'$ denotes g.

2.1 Active Patterns

Fact 2 states that a long as v is present in g, the graph can be separated into vs context and the graph that remains if v and all edges adjacent to v are removed from the graph. We now define the graph primitive $\&^v$-*match* , which conceptually performs this transformation and returns (*Just* c, g') or (*Nothing*, g) if v was not present in the graph. Using $\&^v$-*match* , we define an extension to Haskell's pattern matching feature called active patterns. The pattern $(c\ \&^v\ g)$ is matched if $\&^v$-*match* $v\ g$ returns $Just(c, g)$. This allows a very concise notation for graph algorithms. Although being neat, active patterns are merely a syntactic extension of current Haskell. They can also be expressed using a case construct or *pattern guards* as provided by the current ghc implementation. The function f, which uses our active pattern

$$f\ p\ (c\ \&^v\ g) = e$$
$$f\ p\ g\ \ \ \ \ \ \ \ = e'$$

where v is contained in the pattern p could be reformulated
... using the case construct ... or alternatively using pattern guards

$$f\ p\ g' = \textbf{case}\ match\ v\ g'\ \textbf{of}$$
$$(Just\ c,\ g) \rightarrow e \qquad\qquad f\ p\ g' | (Just\ c, g) \leftarrow match\ v\ g' = e$$
$$(Nothing,\ g) \rightarrow e' \qquad\qquad | g \leftarrow g' \qquad\qquad = e'$$

3 Implementation

The implementation of inductive graphs as an algebraic data type forbids itself, since graph decomposition ($\&^v$-*match*) would require the graph to be transformed. Furthermore the & constructor could be used incorrectly violating the graph invariant. Since the graph data structure is designed for the functional world, the graph implementations should be fully persistent. Additionally we want the graph implementation to allow graph algorithms to be implemented with the same asymptotic complexity as their imperative versions. Erwig provides two different implementations. One uses binary search trees, the second implementation uses a rather complex data structure based on array version trees. I will give a brief overview over both of them. A more detailed description can be found in the original paper. Both give reasonably efficient implementations for the graph primitives from the following table:

Construction	**Decomposition**
& (add context)	&-*match* (extract arbitrary content)
Empty (empty graph)	$\&^v$-*match* (extract specific content)
	Empty-match (test for empty graph)

3.1 Binary Search Tree Representation

In this model, Graphs are implemented as a balanced binary search tree containing all contexts present in the graph in the from
(*node*, (*predecessors*, *label*, *successors*)) The balancing and search tree invariant are based on the node integer. For efficiency reasons, the complete predecessor and successor lists are stored in each node. This leads to a double representation of edges, but allows contexts to be matched without changes and to directly report all contexts that need to be modified when a node is deleted from the graph.

As an optimization of this structure, the predecessor and successor lists are also stored as binary search trees. Furthermore we pair our tree with a single integer containing the highest node present in the graph which is used to report new nodes. Using this representation, *Empty* and *Empty-match* are easily implemented. $c \& g$ works by first inserting the context into the tree and then inserting every successor/predecessor into the corresponding nodes predecessor/successor list. $\&^v$-*match* works the other way around, locating the context to be matched, deleting v from the adjacency lists of all related nodes and returning the context alongside the remaining graph.

Considering the time complexity of the operations above we define n as the number of vertices in the graph, m as the number of edges and c_v as the size of vs context ($|pred\ v| + |suc\ v|$), where *pred* and *suc* denote predecessor and successor functions. Finally c denotes the biggest context relevant to one operation. Using these definitions node insertion and deletion and the *match* functions based on them run in $O(c_v \log n \log c)$

3.2 Array Version Tree Representation

This implementation builds upon the idea of array version trees, an implementation for functional arrays. Instead of a in tree the contexts are stored in an array indexed by the node integer. Any changes to the array are recorded using an inward directed tree of $(index, value)$ pairs, which has the original array as its root. This way the different array versions can be represented as pointers to nodes in the tree, and the nodes along the path to the root mask older versions of the array. New versions can be added in constant time whereas lookup follows the tree structure up to the original array and may thus take $O(u)$ time, where u is the number of previous updates to the array.

Since the goal is to provide a persistent graph implementation that allows all basic operations except & in $O(1)$ time, the structure needs some optimizations. To allow for any operation ob be in $O(1)$, we need to ensure $O(1)$ read access to the graph array, which we do only for single threaded graph usage. To achieve this, we allocate an imperative cache array which contains a copy of the original array whenever we derive a new version from the version tree root and associate it with the new version. This cache array can be used for $O(1)$ access to the latest version of the graph. When the next version is derived from this version, the cache array is updated as well and handed on. On single threaded graph usage, the version tree degenerates to a linear tree where the latest version is always cached.

As with the tree representation, $\&^v$-*match* and &-*match* are costly operations because they require the deletion of v from the context of every adjacent node. To avoid these costs, we equip every node with a positive integer. This node stamp is also put into the adjacency lists of all adjacent nodes. When calculating adjacency of a node only the nodes where the stamp on the edge matches the stamp on the adjacent node are reported, so all that needs to be done when deleting a node is to negate its node stamp. Upon insertion the node stamp is restored and incremented and the new stamp is taken over to adjacent nodes, so any leftover edges are still not reported. To allow for *Empty-match* and &-*match* to be implemented in $O(1)$ we also store the number of nodes currently present in the graph and keep an additional array (cached in the same manner) which carries a partition of nodes currently in the graph and nodes that are deleted. The node partition can also be used to efficiently report free nodes that can be inserted into the graph.

3.3 Comparison

We now have two implementations for functional graphs. The first is easy to implement, fully persistent, fully dynamic, and reasonably efficient for small and medium size graphs, but clearly does not allow algorithms to be implemented so that they meet imperative time complexity. The second implementation allows graph decomposition and testing in constant time and thus allows graph algorithms to be implemented with the same asymptotic complexity as their

imperative counterparts. Unfortunately, it is difficult to implement and the allocation of different cache arrays causes a lot of memory overhead which makes the implementation unsuitable for use in real world applications. Furthermore the structure may be "tricked" by first deriving 2 versions linearly and then continuing with the first version, adding the factor u to the time comlexity for non cached lookup. So already almost single threaded graph usage crashes time bounds.

4 Algorithms and Evaluation

Erwig proposes his graph library for both teaching of graph algorithms and for writing efficient applications. For the purpose of writing practical applications, the library can be of limited use. The binary search tree implementation can be useful since the representation is fully persistent, thus allowing a pure functional stile for working with graphs, while still being fast as log as graphs do not grow to large. This might also be one of the possible reasons to use inductive graphs for teaching graph algorithms. Using the inductive graph implementation described above and the definition for active patterns, depth first search, one of the most important graph algorithms, can be written as follows:

$$
\begin{array}{ll}
\mathit{dfs} & :: [\mathit{Node}] \rightarrow \mathit{Graph}\ a\ b \rightarrow [\mathit{Node}] \\
\mathit{dfs}\ \mathit{vs}\ g \mid \mathit{null}\ \mathit{vs} \mid\mid \mathit{isEmpty}\ g & = [] \\
\mathit{dfs}\ (v : \mathit{vs})\ (c\ \&^{v}\ g) & = v\ :\ \mathit{dfs}\ (\mathit{suc}\ c +\!\!+ \mathit{vs})\ g \\
\mathit{dfs}\ (v : \mathit{vs})\ g & = \mathit{dfs}\ \mathit{vs}\ g
\end{array}
$$

The algorithm works very similar to its imperative counterpart, the biggest difference being the way the single visit constraint is enforced. Imperative implementations tend to use some node marking strategy, either within the graph or in a separate data structure. Since we use an inductive model the single visit constraint is enforced implicitly by decomposing the graph step by step. The algorithm either stops when there are no more nodes to expand or if the whole graph has been traversed and the remaining graph is thus empty. The latter is useful since in dense graphs expanded edges can cause up to $\Omega(n^2)$ nodes to be checked, even after the graph has been fully traversed. The pattern $(c\ \&^{v}\ g)$ is always matched when v is contained in g. In that case v is expanded. Otherwise, if v has already been expanded, the node is simply discarded. Notably this algorithm can be instantly transformed into a breadth first search algorithm merely by exchanging *suc c* and *vs* in the second equation, although a queue implementation is needed to keep the linear time bound. The original paper shows quite a few standard graph algorithms, which can all be elegantly expressed.

So for the simple teaching of graph algorithms the *fgl* might be useful, as long as time complexity is not a concern. If one really wants in-depth teaching of graph algorithms, one hits the limitations of the library. The current implementation of the *fgl* library for Haskell provides only the binary search tree implementation as described above and monadic graphs which are based upon *IOArrays*. They efficiently allow single threaded graph usage. So one has to chose

to either implement algorithms without meeting imperative time bounds or one is left to work within a monad, which completely destroys the functional flavor. Additionally, the documentation of the library has not been updated for the last 4 releases. There is an implementation for SML/NJ which includes both implementations, but it has not been changed since August 1999, and so far I have not been successful in building the library with the current SML/NJ release.

On the other hand, in an impure functional language such as ML, functional graph algorithms can be implemented easily and efficiently using imperative style book keeping. The extra book keeping allows most graph algorithms to be implemented to only read from the graph so a static/functional array representation is entirely sufficient. Furthermore algorithms written in a "read only" style can be implemented in a way so that they appear purely functional to the outside. This approach has been taken in [KL93]. Here, the monad of state transformers is used to efficiently implement the depth first forest algorithm in Haskell. Although notationally considerably less concise, the state of the log is threaded through the different function calls roughly in the same manner as one threads the remaining graph though the recursive function calls when writing inductive style algorithms. The result of this approach is an efficient *dff* algorithm for Haskell that appears completely functional to the outside.

So altogether, the idea to view graphs as inductive data types is a nice idea that theoretically allows for a very concise and elegant notation of functional graph algorithms that do have the same time complexity as their imperative counterparts. Unfortunately one either has to cope with a considerable memory overhead, with unmet imperative time bounds or with a loss of expressiveness and persistence if one decides to use monadic graphs thus destroying the functional flavor. For now, there seems to be no way to achieve persistence, efficiency, and concise and alegant notation all at the same time.

Bibliography

[Erw01] M. Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

[KL93] David J. King and John Launchbury. Lazy Depth-First Search and Linear Graph Algorithms in Haskell. In John T. O' Donnell and Kevin Hammond, editors, *GLA*, pages 145–155, Ayr, Scotland, 93. Springer-Verlag.

Pickler Combinators – Explained

Benedikt Grundmann
benedikt-grundmann@web.de

[1] Software Engineering Chair (Prof. Zeller)
Saarland University
[2] Programming Systems Lab (Prof. Smolka)
Saarland University

Abstract. This paper summarizes the paper "Pickler Combinators" by Andrew J. Kennedy. Kennedy presents an purely library based approach to pickling similar in spirit to the well-known parser combinators. This approach to pickling is also compared to builtin pickling services as presented in the paper "Generic Pickling and Minimization" by Guido Tack et al.

1 Motivation

It is frequently necessary to externalize data in order to store it on disk or transmit it over the network. This process is also known as serialization or pickling. The reverse process is called deserialization or unpickling.

As long as the data is atomic such as a number or a simple sequence of atomic values of the same type serializing it is rather easy. But as soon as more complex heterogeneous data structures have to pickled doing so by hand easily gets very error prone.

One reason for that is that three different definitions have to be synchronized. These are the definitions of the datatype to be pickled, the definition of the pickling function and the definition of the unpickling function. And in most cases there is more than one datatype involved!

2 A pickler library: Kennedy's Pickler Combinators

In [3] Kennedy describes an solution to pickling purely based on an combinator library and therefore embeddable in any programming language which offers higher order functions.

A combinator library is based on the idea of combining higher order functions of very uniform type. The combinators are carefully designed higher order functions which act as the glue; they provide a variety of ways of composing functions together into more powerful functions.

In the case of Kennedy's Parser Combinators both the pickling and the unpickling function are composed at the same time. It is therefore impossible to create an inconsistent pair of pickling/unpickling functions. In the library such

pairs are provided for the built-in types of the programming language (e.g unit, booleans, characters, non-negative integers and integers between 0 and some upper bound). The pairs are given the type `PU` α, where α is the type of the value to be pickled. Kennedy refers to such a type as a "pickler for α". The definition of the type

```
PU a = PU { appP :: (a, [Char]) -> [Char]
          , appU :: [Char] -> (a, [Char]) }
```

is not made accessible outside the implementation of the library further ensuring that the construction of an inconsistent pickler is not possible.

As you can see in the definition above the type of the pickling and unpickling functions had to be extended to enable composition. The semantics of `appP` are defined like this `appP (v, s)` prepends a serialized representation of `v` to an existing stream of serialized values `s`. Whereas `appU s` returns the deserialized value and the remaining stream. Ignoring sharing and minimization (see section 2.1) the following equation holds for all cycle free values `v` and byte sequences `s`: `(v, s) = appU (appP (v, s))`.

As set of combinators – functions from and to picklers – are provided to generate picklers for composite types. Given experience with a combinator library they have the expected types mirroring the corresponding type constructor:

- The pickler combinators for tuples `pair :: PU a -> PU b -> PU (a, b)`, `triple :: PU a -> PU b -> PU c -> PU (a, b, c)`, ...
- The pickler for lists `list :: PU a -> PU [a]`
- Optional values `pMaybe :: PU a -> PU (Maybe a)`

All these combinators are defined by means of the two combinators `lift` and `sequ`. Pickling of fixed values is done by the `lift :: a -> PU a` combinator. Its implementation is simple as no value has to (de)serialized.

```
lift x = PU (\ (_, s) -> s) (\ s -> (x, s))
```

The combinator `sequ :: (b->a) -> PU a -> (a -> PU b) -> PU b` is more interesting. It encodes sequential composition of picklers, in particular sequential dependencies are allowed. Assuming two values `A :: a` and `B :: b`, a pickler `pa :: PU a` and the two functions `f :: (b->a)` and `k:: a -> PU b` the pickler `p = sequ f pa k` has the following semantics. The call `appP p (B, s)` precedes the encoding of `B` by the encoding of `A = f B`. Most notably this encoding can depend on the value `A` as the pickler for `B` is generated by the call `k A`. The call `appU p s` in turn decodes `A`, generates the pickler for `b` by calling `k A` and decodes and returns `B`.

This looks quite complicated but as mentioned above it allows for simple definitions of pickler combinators such as `pair`

```
pair pa pb = sequ fst pa (\ a ->
             sequ snd pb (\ b ->
             lift (a, b) ))
```

This definition is easy to understand if read in reverse order. In line three the values `a` and `b` are fixed and to pickle a pair of fixed values we can simple use lift. Now we precede this empty encoding of the fixed pair by the encodings of `b` and `a`.

Another combinator called `wrap :: (b->a, b->a) -> PU a -> PU b` is also implemented in terms of `sequ` and `lift` to provide mapping on picklers. Given an implementation of a fixed range cardinal number pickler `zeroTo :: Int -> PU Int` all ranged ordinal type picklers can be defined in terms of `wrap` and `zeroTo`. For example the definition of a pickler for boolean values looks like this
`bool = wrap (toEnum, fromEnum) (zeroTo 1)`.

A number of combinators make use of recursion. A good example is the previously mentioned `zeroTo :: Int -> PU Int`. `zeroTo n` creates a pickler for integers in the range $[0, n]$. It separates the representation into $\lceil \log_{256} n \rceil$ digits. Each digit has 256 possible values thereby making maximum use of the available storage.

Pickling of custom datatypes is done by combining the combinator `alt` with the `wrap` combinator. The `alt` combinator is used to combine several distinct picklers for values of the same type. Each pickler handles a disjunct set of possible values contained in the type. The user must also specify a tagging function which is used to determine which pickler to use.

Therefore for each constructor a separate pickler is defined by either lifting the constructor into a pickler or by wrapping a pickler for the argument of the constructor. These are then combined using `alt`, as seen in the example below.

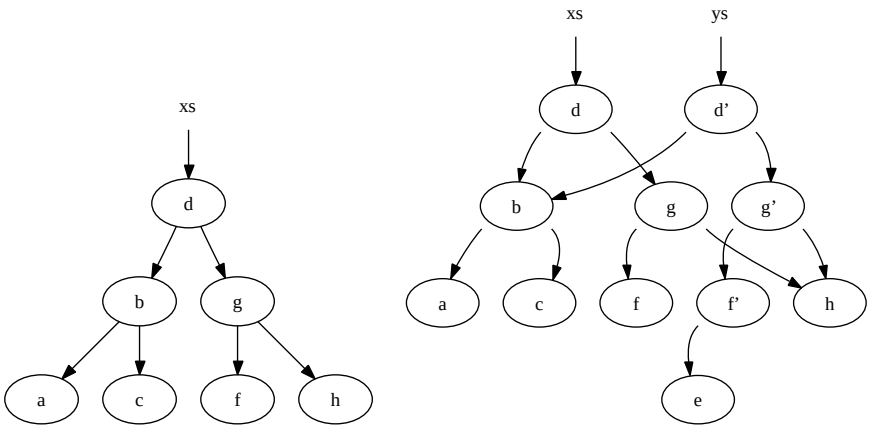
```
data Tree
  = Node (String, Tree, Tree)
  | Empty

tree  :: PU Tree
tree  = alt tag  [
          wrap (Node, \\(Node d) -> d)
                 (triple string tree tree)
        , lift Empty
        ]
      where tag (Node _)  = 0
            tag Empty     = 1
```

2.1 Sharing and Minimization

Thanks to persistence programs written a functional programming language usually make extensive use of sharing. A popular example are binary search trees. After inserting an element into the tree shown in figure 1 (a) we do not end up with two separate trees `xs` and `ys = insert (e, xs)`, but rather two trees which share a large number of nodes (see figure 1 (b)). Asides from memory consumption this difference is normally not observable from within the programming language. But there are two points which make sharing so important. One if we

had copied the elements upon insertion the runtime cost of insert would have been a lot worse. Second and even more important with the increased memory consumption it would be impossible to keep several versions of the tree in memory.



(a) before (b) after

Fig. 1. tree(s) before and after insertion

The library as described so far does not preserve sharing. In the example mentioned above this would matter as soon as more than one tree were to be pickled. In order to support sharing Kennedy implements the following idea. Sharing is detected by memorizing which values have already been pickled. If a value has not been pickled yet an numerical id is generated and added to the encoding. If the value is already part of the overall encoding just the id – which is part of the dictionary – will be included in the encoding.

Users of the combinator library must indicate shared datatypes by using the `share` combinator on its pickler. This combinator extends the normal pickler by the algorithm outlined above. To do so the datatype must support the equality operation to detect whether it is already part of the dictionary or not. Also the definition of the pickler datatype has been changed into

```
PU a s = PU { appP :: (a, (s, [Char])) -> (s, [Char])
            , appU :: (s, [Char]) -> (a, (s, [Char])) }
```

As you can see a pickler now also threads the dictionary. As we can not know what the type of the dictionary is – it depends on which type the `share` combinator is applied on – this type is a new type parameter of the pickler. This also

implies that sharing of more than one type of value at the same time requires rewriting the `share` combinator.

Still as long as the value pickled is not cyclic (see section 2.2) this library can be used minimize the heap representation of a value by maximizing sharing with respect to one component. As an example one could use the `share` combinator to either share the nodes of the tree, or the values of the keys but not both at the same time.

2.2 Cyclic values

Pure functional programming languages such as Haskell[2] normally use a non-eager evaluation model and can therefore express infinite (cyclic) data structures. The algorithm outlined above could in principle be used to serialize cyclic values. But the implementation given in the paper can not do so as the equality test used would diverge. Some low level pointer based comparison, which does not diverge on cyclic values has to be used instead.

In a non pure functional programming language such as SML[4] cyclic data structure are introduced explicitly by using references. Martin Elsman[1] presents an SML variant of Kennedy's library which uses an adopted variant of the algorithm outlined above to serialize references.

3 Builtin pickling and the abstract store

A different approach to pickling was defined by Guido Tack et al [5]. They defined an language independent memory model – the so called abstract store – and introduced pickling as a runtime system service similar in spirit to the garbage collector. Which they implemented as part of the virtual machine of the programming language Alice, a variant of SML. They also defined and included a generic minimization algorithm based on graph minimization. In particular as this minimization algorithm works on the representation level, values of different types can be shared and true minimization is achieved.

4 Comparison and Conclusions

If we compare the two different approaches we have to realize that both have different strength and weaknesses. One the one hand the combinator based approach does not require any kind of runtime support and is easily extensible and adaptable by the programmer. The approach by Guido Tack et al is essentially a runtime service and not extensible by the programmer in any way. But it supports both arbitrary sharing and full minimization in the presence of cyclic values. In the standard case it is a even simpler to use than the combinator based library.

It is also interesting to compare the different ways used to embed a dynamically typed value into a statically typed language. In Alice the types of the

pickled values are included in the pickled representation and the language was extended by a dynamic typecheck facility, similar to the well known `typecase` instruction. Therefore it is not necessary to known exactly what type of value was pickled. In the pickler combinator library instead one has to specify the toplevel pickler and there are no checks at all whether the pickled representation was actually generated using the same pickler.

One advantage of the combinator based library not mentioned by Kennedy is that it could be extended to support different backends such as binary versus textual by just changing a small number of the combinators. Still the solution to sharing presented by Kennedy does not scale as the number and types of shared values have to be known in advance. Even worse using a standard equality test used by the sharing/minimization algorithm actually results in an quadratic runtime behavior. I am therefore not sure whether the library as described is ready for use in non toy programs.

Still the principles presented are interesting. I do not know any other combinator library which creates more than one function at the same time and found that idea inspiring.

References

1. Martin Elsman. Type-specialized serialization with sharing. In *Sixth Symposium on Trends in Functional Programming (TFP'05)*, September 2005.
2. S. Peyton Jones and et al, editors. *Haskell 98 Language and Libraries, the Revised Report.* CUP, April 2003.
3. Andrew Kennedy. Pickler combinators. *J. Funct. Program.*, 14(6):727–739, 2004.
4. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* August 1990.
5. Guido Tack, Leif Kornstaedt, and Gert Smolka. Generic pickling and minimization. *Electronic Notes in Theoretical Computer Science*, 148(2):79–103, March 2006.

Imprecise Exceptions - Exceptions in Haskell*

Christopher Krauß

Universität des Saarlandes
Informatik

Abstract. Ausnahmen geben uns die Möglichkeit, gravierende Fehler, die normalerweise zum Programmabsturz führen würden, aufzuzeigen, um dann an einer anderen Stelle im Programm mit einer alternativen Berechnung fortfahren zu können. Dieses Paper präsentiert ein Modell, welches Ausnahmen in die *pure* und *lazy* Sprache `Haskell` einbettet, ohne dabei wichtige Eigenschaften wie *referential transparency*[2] und *laziness* zu verletzen.

1 Introduction

Die Idee von *Imprecise exceptions* kommt aus der Hardware. Moderne super scalar Microprozessoren führen zur Verbesserung der Performance viele Instruktionen parallel aus. Eine unmittelbare Folge daraus ist eine Ungenauigkeit beim Werfen von Ausnahmen: Die Ausnahme, die zuerst geworfen wird, muss jetzt nicht mehr die Ausnahme sein, die in einem sequentiellen Lauf geworfen wird.

In diesem Paper findet dieselbe Idee auf der Programmebene Anwendung. Auch hier kann die Veränderung der Auswertungsreihenfolge[3] eine Verbesserung der Performance bewirken. Dabei tritt das selbe Problem im Bezug auf Ausnahmen wie auf Hardwareebene auf. Das Paper präsentiert eine Lösung, die *Präzision für Performance verkauft*: Um die Transformationen weiterhin durchführen zu können, bleibt das Modell unpräzise darüber, welche Ausnahme geworfen wird. Das in Haskell präsentierte Modell macht von den Eigenschaften des `IO` mondas gebrauch, um somit die Erhaltung von purity und laziness zu gewährleisten. Im letzten Teil des Papers werden kurz die Semantik und einige Erweiterungsmöglichkeiten des Modells aufgezeigt.

2 Ausnahmen

Viele Programmiersprachen stellen Ausnahmen zur Verfügung. Ausnahmen können im Fall des Auftretens eines Fehlers geworfen und an einer anderen Stelle im Programm wieder gefangen werden. So muss das Auftreten von Fehlern nicht unmittelbar den Absturz des Programms zur Folge haben.

---

* Nach "'Imprecise exceptions"' von S.P.Jones [3]

[2] Das Auswerten eines Ausdrucks liefert immer das selbe Ergebnis

[3] z.B. durchgeführt vom Compiler bei der Optimierung

2.1 Verwendung von Ausnahmen

Disaster recovery: Hier werden Ausnahmen verwendet, um das Auftreten eines Fehlers, wie Division durch null, zu signalisieren. Die Ausnahme kann dann an einer anderen Stelle im Programm wieder gefangen werden, um dort die Berechnung fortzusetzen.
Alternative result: Ausnahmen werden auch häufig eingesetzt, um ein alternatives Ergebnis zurückzugeben. Existiert z.B. beim Nachschlagen in einer Map der gesuchte key nicht, wird eine Ausnahme geworfen, um dies anzuzeigen.
Short circuit control flow: Eine weitere Möglichkeit besteht in der Abkürzung des Kontrollflusses. Wird beim Durchsuchen einer Liste das gesuchte Element gefunden, kann anstatt die Liste bis zum Ende zu traversieren eine Ausnahme geworfen werden.
Asynchronous events: Die vierte Stelle, an der Ausnahmen verwendet werden können, sind asynchrone Ereignisse. Dies sind Fehler, die nicht direkt im Programm auftreten, sondern externe Ereignisse, wie das Drücken von Ctrl^c oder ein stack-overflow.

Dabei lassen sich zwei Arten von Ausnahmen unterscheiden: *Synchronous exceptions*, Ausnahmen, die synchron zum Programm auftreten, wie disaster recovery, alternative result und short circuit control flow und *Asynchronous exceptions*, Ausnahmen die jederzeit auftreten können, die nicht vorhersehbar oder reproduzierbar sind. Zu ihnen zählen die asynchronous events.[2, 1]

2.2 Ausnahmen in lazy Sprachen

Aus dem vorherigen Abschnitt ergibt sich unweigerlich die Frage, warum Ausnahmen in pure und lazy Sprachen nicht zur Verfügung stehen. Dies hat mehrere Gründe. Zum einen haben Programme bei lazy evaluation keinen direkt vorhersehbaren Kontrollfluss, so dass es nicht möglich ist, zu folgern, wann welche Ausnahme geworfen wird. Zum anderen wird die purity verletzt, wenn Ausnahmen auf die übliche Art verwendet werden. Weiterhin gibt es die Möglichkeit, Ausnahmen als Werte (*exceptions as values*) zu repräsentieren. Einen ähnlichen Ansatz gibt es im IEEE floating-point Standard, wo bestimmte bit-patterns exceptional Werte codieren.

2.3 Exceptions as values

Dieser Abschnitt beschäftigt sich mit dem Ansatz, Ausnahmen als Werte zu modellieren: `data ExVal a = OK a | Bad Exception`. Es ist also nicht nötig, die Sprache zu erweitern. Weiterhin kann anhand des Typs der Funktion erkannt werden, ob eine Ausnahme geworfen werden kann, wodurch es unmöglich wird, eine Ausnahme nicht zu fangen. Außerdem lässt sich dieses Modell komfortabel verwenden, da `ExVal` einen *monad* bildet.

Allerdings kann beim Fangen von Ausnahmen leicht die strictness erhöht werden. Weiterhin muss bei jedem Funktionsaufruf getestet werden, ob es sich um eine Ausnahme (`Bad Exception`) oder einen normalen Wert (`OK a`) handelt.

Damit propagieren die Ausnahmen also auch nicht implizit und der Ansatz ist ineffizient. Modularität und die Wiederverwendbarkeit von Code gehen verloren. Und zuletzt sind viele Transformationen nicht mehr möglich.

3 Ein neues Design

In diesem Abschnitt wird ein neues Design präsentiert, welches die Probleme, die durch die Kombination von pure und lazy Sprachen mit Ausnahmen auftreten, auf zufrieden stellende Art und Weise löst. Dabei erhält dieses Model lazyness und referential transparency. Damit bleiben alle sinnvollen Transformationen[4] erhalten. Weiterhin ist es möglich, über die Ausnahmen, die in einem Programm geworfen werden können, zu argumentieren. Außerdem bleiben Semantik und Laufzeit von Programmen, die keine Ausnahmen beinhalten, unverändert. Das Design wird in der Programmiersprache `Haskell` vorgestellt.

3.1 Grundidee des neuen Designs

Da hier lazy evaluation beibehalten werden soll, werden Ausnahmen weiterhin als Werte und nicht als control flow modelliert. Allerdings wird dieses Modell um die Idee erweitert, dass Werte eines beliebigen Typs entweder **normal** oder **exceptional** sind. Zu diesem Zweck werden ein Datentyp `Exception`, der alle möglichen Ausnahmen beinhaltet, eine `raise` Funktion, die eine Ausnahme in einen beliebigen Typ einbettet, und eine Funktion `catch`, die aus dem Wert eines beliebigen Typs gegebenenfalls die Ausnahme extrahiert, eingeführt.

```
data Exception = DivideByZero | Overflow | UserError String | ...
raise :: Exception -> a
catch :: a -> ExVal
```

3.2 Propagierung

Die Propagierung von Ausnahmen läuft, da es sich um Werte handelt, offensichtlich automatisch. Im Hinblick auf lazy evaluation ist es allerdings nötig, sich Gedanken über die Bedeutung von Propagierung zu machen. Man betrachte die folgende Funktion, die zwei Listen mit der Funktion f zippt:

```
zipWith f [] [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f xs ys = raise UserError "Uneq lists"
```

Es gibt hier drei mögliche Verhaltensweisen bei einem Aufruf dieser Funktion:

- Sind beide Listen gleich lang, wird ein normaler Wert zurückgeliefert.
- Ist genau eine der Listen leer, wird unmittelbar eine Ausnahme geworfen.
- Interessant ist der Fall, in dem beide Listen nicht leer aber von unterschiedlicher Länge sind: Hier ist die Ausnahme in dem nicht vollständig ausgewerteten Teil versteckt.

[4] z.B. β-Reduktion

Das bedeutet also: Propagierung ist nur dann gewährleistet, wenn eine Auswertung erzwungen wird.

3.3 Fangen von Ausnahmen

Für das Fangen von Ausnahmen haben wir die Funktion `catch :: a -> ExVal a` eingeführt. Aber was passiert beim Aufruf `catch ((1/0)+(raise Overflow))`? Je nach Auswertungsreihenfolge wird eine andere Ausnahme geworfen. Für dieses Problem gibt es drei Lösungsansätze:

- Das Festlegen der Auswertungsreihenfolge: Dies verletzt laziness und verbietet außerdem sinnvolle Transformationen.
- Nicht-deterministisches Auswählen einer Ausnahme: Dies verletzt purity und referential transparency, wodurch Transformationen wie β-reduction nicht mehr möglich sind.
- Zurückgeben beider Ausnahmen: D.h. ein exceptional Wert beinhaltet jetzt eine Menge von Ausnahmen. Dieser Ansatz hat zur Folge, dass bei der Implementierung die gesamte Menge aufrecht erhalten werden muss, was eine schlechte Performance zur Folge hat. Weiterhin ist die Propagierung höchstgradig unautomatisiert: Wenn eine Ausnahme gefunden wird, muss trotzdem der gesamte Code ausgewertet werden. Damit verletzt dieser Ansatz außerdem lazyness.

Alle drei Ansätze lösen das Problem nicht auf eine zufrieden stellende Art und Weise. Die tatsächliche Lösung liegt in einem kleinen Trick: Denotational verhält man sich so, als ob die gesammte Menge von Ausnahmen aufrecht erhalten wird. Operational sucht man sich eine Ausnahme aus der Menge und gibt nur diese zurück. Die Lösung des Problems besteht also aus einer Kombination der zweiten und dritten Alternative.

3.4 Reparieren der `catch` Funktion

Wählt man sich zufällig eine Ausnahme aus der Menge der möglichen Ausnahmen, die bei der Auswertung eines Terms auftreten können, aus, ergibt sich wieder das Problem des Nicht-determinismus. Um diese Problem zu umgehen, bedient man sich des `IO` monads[5]. `catch` erhält nun den Typ `catch :: a -> IO ExVal a`. `catch` darf also jetzt bei jedem Aufruf ein anderes Ergebnis liefern. Damit ist das Problem in den impuren Teil der Sprache verschoben. Purity und referential transparency bleiben erhalten. Durch diesen Trick wurde der Nicht-determinismus in den Ausnahmen vom Nicht-determinismus in den Werten getrennt.

Die Lösung basiert also auf Unterschieden in der denotationalen und operationalen Semantik. Die denotationale Semantik beschäftigt sich ausschließlich

[5] Im `IO` monad werden alle Berechnungen ausgeführt, die nach außen hin einen Effekt haben. `IO t` ist dabei eine Berechnung, die ohne Seiteneffekte ausgewertet wird, und erst einen Effekt hat, wenn sie ausgeführt wird.[4]

mit dem reinen Teil der Sprache. Da Ausnahmen aber nun im `IO` monad behandelt werden, dessen Ausführung in der denotationalen Semantik nicht betrachtet wird, kann der Nicht-determinismus dort nicht beobachtet werden.

4 Semantics

In diesem Abschnitt soll ein kurzer Einblick in die Semantik von `imprecise exceptions` gegeben werden. Im Folgenden wird die Semantik der Additionsoperation vorgestellt.

$$[e_1 + e_2]\rho = \begin{cases} v_1 + v_2 & \text{if} \quad OK\ v_1 = [e_1]\rho \quad \text{and} \quad OK\ v_2 = [e_2]\rho \\ Bad\ (\mathcal{S}([e_1]\rho)\ \cup\ \mathcal{S}([e_2]\rho)) & \text{otherwise} \end{cases}$$

Falls sowohl e_1 als auch e_2 zu normalen Werten ausweten, wird die Addition der beiden zurückgegeben. Falls einer der Ausdrücke zu einem exceptional Wert auswertet, werden die Ausnahmen aus diesem Wert mit denen aus dem anderen Wert vereinigt. $\mathcal{S}$ liefert zu $Bad\ s\ s$ und zu OKv die leere Menge.

Aber wie wird der folgende Fall gehandhabt: `loop + raise Overflow`? Soll dieser Ausdruck divergieren oder zu Bad {`Overflow`} auswerten? Dieses Problem löst man, indem $\perp$ als Vereinigung der Menge aller Ausnahmen mit `NonTermination` modelliert: $\perp\ =\ \mathcal{E}\ \cup$ {`NonTermination`}.

Die Regeln für Konstanten, Variablen, `raise`, Abstraktionen, Applikationen, Konstruktoren und `fix` sind ähnlich. Die Regel für `case` ist ein wenig komplizierter, da darauf geachtet werden muss, keine Transformationen zu verletzen.

4.1 Semantics von `catch`

Da `catch` auf dem `IO` monad arbeitet, muss diese Funktion in der operationalen Semantik betrachtet werden.

$$\texttt{catch}\ (OK\ v) \rightarrow \texttt{return}\ (\texttt{OK}\ v)$$
$$\texttt{catch}\ (Bad\ s) \rightarrow \texttt{return}\ (\texttt{Bad}\ x) \quad \text{if} \quad x \in s$$
$$\texttt{catch}\ (Bad\ s) \rightarrow \texttt{catch}\ (Bad\ s) \quad \text{if} \quad \texttt{NonTermination} \in s$$

Wird `catch` also auf einen normalen Wert angewendet, wird dieser einfach zurückgegeben. Bei der Anwendung auf einen exceptional Wert wählt `catch` eine Ausnahme aus der Menge s aus und gibt diese zurück. Falls sich außerdem `NonTermination` in der Menge befindet, kann `catch` bei der Auswertung auch zum selben Term zurückkehren. Damit kann der Aufruf von `catch` auf unserem Beispiel `loop + raise Overflow` sowohl `Overflow` werfen oder divergieren. Tatsächlich ist es durch die Modellierung von $\perp$ nach der Semantik sogar erlaubt, dass `catch` eine beliebige Ausnahme wirft.

5 Implementation

Dieser Ansatz lässt sich mit standard exception handling Mechanismen implementieren. `catch` forciert die Auswertung seines Arguments zu head-normal

Form. Die Auswertung von `raise ex` trimmt den Auswertungsstack auf den nächsten `catch` Aufruf und gibt `Bad ex` zurück. Falls kein Fehler auftritt, gibt `catch` einfach den Wert zurück. Durch eine solche Implementierung bleibt die Effizienz von Programmen ohne Ausnahmen unverändert. Exceptional Werte verhalten sich wie first class Werte.

6 Erweiterungen

Im Folgenden werden noch einige Erweiterungsmöglichkeiten angesprochen.
Asynchronous exception: Das präsentierte Modell kann leicht um asynchrone Ausnahmen erweitert werden: Jeder Auswertungschritt kann eine Ausnahme zur Folge haben.
Detectable bottoms: Es gibt Divergenz, die durch das Anwenden Graphalgorithmen auf Transitionssysteme erkannt werden kann. Anstatt einfach zu divergieren könnte an solchen Stellen eine Ausnahme geworfen werden.
Pure Funktionen auf exceptional Werten: Es ist möglich, auf exceptional Werten zu rechnen. Funktionen wie `mapException :: (Exception -> Exception) -> a -> a` sind implementierbar wenngleich auch nicht unbedingt sinnvoll. Ein Zurückkehren in normale Werte ist allerdings nicht möglich. Schon eine Implementierung der Funktion `isException :: a -> Bool` stellt uns vor unlösbare Probleme: Es ist nicht möglich den Aufruf `isException ((1/0) + loop)` zufrieden stellend zu modellieren, da hier der Nicht-determinismus in Bezug auf Divergenz und Werfen einer Ausnahme auftritt.

7 Zusammenfassung

Im Vergleich zu anderen Sprachen ist das präsentierte Design weniger ausdrucksstark. In ML können Ausnahmen lokal deklariert, geworfen und gefangen werden, ohne dass dies nach außen hin sichbar ist. Ein weiterer Nachteil ist, dass sich der `IO` monad wie eine Falltür verhält. Allerdings gehen in diesem Modell keine sinnvollen Transformationen verloren! Des weiteren lässt sich dieses Modell problemlos mit Dingen wie Nebenläufigkeit erweitern. Tatsächlich kommt es bereits im Glasgow Haskell Compiler (4.0 und später) zum Einsatz.

References

[1] Jones, S. P.: Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: *Microsoft Research, Cambridge* (2005)

[2] S. P. Jones, A. M. J.: Asynchronous exceptions. In: *Microsoft Research, Cambridge* (2005)

[3] S. P. Jones, T. H. S. F. H.: A semantics for imprecise exceptions. In: *PLDI Atlanta* (1999)

[4] Thompson, S. : *International Computer Science Series*. Bd. -: *Haskell: The Craft of Functional Programming*. Bonn - Amsterdam - Tokyo : Addison-Wesley, 1996

This article was processed using the LaTeX macro package with LLNCS style

Quick Check
Advanced Functional Programming

Sebastian Meiser

Universität des Saarlandes

1 Testen vs. Beweisen

Sicherlich stellt sich für einen Programmierer stets die Frage, ob die von ihm geschriebenen Programme korrekt sind. Eine eindeutige Antwort darauf wäre zwar schön, ist aber nur durch einen Beweis möglich. Um zu beweisen, dass eine Funktion korrekt ist, muss sie vollständig verifiziert werden. Der Beweis, dass eine Funktion nicht korrekt ist wirkt dagegen deutlich leichter, es muss schliesslich nur ein Gegenbeispiel gefunden werden, also eine Parameterkonfiguration, die zu einem falschen Ergebnis führt. Eine Verifikation ist meist entweder aufgrund fehlender Spezifikationen gar nicht möglich, oder aber sie zieht einen Aufwand mit sich, der in keinem sinnvollen Verhältnis zum Nutzen steht. Aus diesem Grund ist es nur verständlich, dass sich ein Programmierer in den meisten Fällen mit einer Auswahl an Tests begnügt. Findet er mit einem der Tests einen Fehler, so kann er diesen beseitigen, findet er keinen, so kann er zumindest davon ausgehen, dass keine größeren Fehler mehr vorhanden sind, ohne sich dessen aber sicher sein zu können.

Auch wenn die Möglichkeiten der Verifikation in den vergangenen Jahren einen großen Vortschritt durchlaufen haben, ist das Testen noch immer gängige Praxis, da es mit vergleichweise geringem Aufwand jederzeit eingesetzt werden kann. Gerade bei funktionalen Programmiersprachen, deren Funktionen üblicherweise keine Seiteneffekte haben lassen sich besonders gut testen, da man davon ausgehen kann, dass eine Funktion entweder richtig oder falsch ist, nicht aber ihre Richtigkeit von ihrer Position im Programm oder irgendwelchen Seiteneffekten abhängt. Da es aber trotz dieser Einfachheit noch einen nicht zu vernachlässigenden Aufwand bedeutet und dieser bei jeder Veränderung der Funktionen erneut betrieben werden muss, um ein gewisses Maß der Sicherheit zu erlangen, sind teilweise- oder vollautomatische Tests wünschenswert.

2 Automatisches Testen mit QuickCheck

Für einen automatischen Test ist es von Nöten, zu definieren, ob das Ergebnis der Funktion in Abhängigkeit ihrer Parameter zufriedenstellend ist. Beispielsweise kann man definieren, dass das Ergebnis einer Sortierprozedur für Listen eine sortierte Liste sein sollte, deren Länge gleich der Einghabeliste ist. Alternativ kann, falls eine schon bekannte und als richtig vorrausgesetzte naive Implementierung der selben Funktion vorhanden ist, getestet werden, ob das Ergebnis der neuen Funktion, dem der womöglich ineffizienteren alten Funktion entpricht.

Um zu definieren, ob eine Funktion ein zufriedenstellendes Ergebnis geliefert hat, werden sogenannte *Propperties* verwendet, welche Funktionen vom Typ $\alpha \rightarrow Bool$ sind. α ist dabei der Parametertyp der Funktion.

Ein einfaches Beisiel:

$$
\begin{aligned}
&fac\_naive \;\; n \\
&\qquad\quad | \; n < 2 \;=\; 1 \\
&\qquad\quad | otherwise \;=\; n \;*\; fac\_naive\; (n-1)
\end{aligned}
$$

$$fac\; n \;=\; foldr\; (*)\; 1\; [0..n]$$

$$
\begin{aligned}
&prop\_fac \;\; :: \; Int \;\rightarrow\; Bool \\
&prop\_fac\; x \;=\; fac\; x \;==\; fac\_naive\; x
\end{aligned}
$$

Hierbei ist fac die Funktion, die getestet werden soll und fac_naive eine als korrekt vorrausgesetzte Funktion. Ruft man QuickCheck mit der Property prop_fac auf, so ergibt sich:

$$
\begin{aligned}
&Main > \; quickCheck\; prop\_fac \\
&Falsifiable,\; after\; 1\; tests: \\
&1
\end{aligned}
$$

Die getestete Funktion ist also falsch. QuickCheck hat herausgefunden, dass bei dem Argument 1 die Bedingung nicht erfüllt ist. Nach Beheben des Fehlers in der Funktion

$$facn = foldr\; (*)\; 1\; [1..n]$$

liefert QuickCheck beim erneuten Testen:

$$
\begin{aligned}
&Main > \; quickCheck\; prop\_fac \\
&OK,\; passed\; 100\; tests.
\end{aligned}
$$

3 Das Generieren von Test Daten

Es zeigt sich, dass durch das Testen in der Tat Fehler gefunden werden können. Die Effizienz des Testens hängt allerdings verständlicherweise hauptsächlich davon ab, welche Werte zum Testen verwendet wurden. Die Aussage “OK, passed 100 tests.” ist natürlich nichts wert, wenn 100x mit der Zahl 1 getestet wurde. Das automatische Generieren von Testdaten ist nun nicht ganz trivial. Die Entwickler von QuickCheck haben sich entschlossen, die einfachste Form der Generierung, die zufällige Erzeugung von Werten, zu verwenden. Eine Alternative wäre es gewesen, ein spezielles Schema zu definieren, nach dem die Testdaten generiert werden. Dies ist jedoch ebenfalls recht aufwendig, weniger flexibel aber nicht lohnend besser als das zufällige Testen.

Um Daten von beinahe jedem Typ generieren zu können wurde Gebrauch von Typklassen gemacht. Für alle von der Typklasse *Arbitrary* abgeleiteten Typen α muss ein Generator in Form der überladenen Funktion *arbitrary* vom Typ $Gen\; \alpha$ zur Verfügung gestellt werden. Werte einfacher Typen wie Int oder $Bool$ lassen sich denkbar einfach erzeugen. Im Falle eines Bool Wertes

wird Zufallsgeneriert entweder $True$ oder $False$ erzeugt, im Falle eines Integers eine zufällige Zahl.

Werte mit komplexeren Typen können rekursiv definiert werden: Existieren bereits ein $Gen\ \alpha$ und ein $Gen\ \beta$, so kann rekursiv auch ein $Gen\ (\alpha, \beta)$ definiert werden. Ähnlich kann eine zufällige Liste vom Typ $[\alpha]$ erzeugt werden, indem zuerst entschieden wird, wie lang die Liste sein soll und dann soviele Werte vom Typ α erzeugt werden.

4 Generator Kombinatoren

Während die bislang genannten Generatoren von QuickCheck mitgeliefert werden, ist es auch möglich selbst neue Generatoren zu erzeugen, oder bestehende Generatoren zu kombinieren. Hierzu werden einige Kombinator-Funktionen zur Verfügung gestellt: Die einfachste davon ist

$$return :: \alpha \rightarrow Gen\ \alpha,$$

die einen Generator erzeugt, welcher "zufällig" immer den als Parameter übergebenen Wert liefert.

$$elements :: [\alpha] \rightarrow Gen\ \alpha$$

gibt ein zufällig ausgewähltes Elemen der Eingabeliste aus.

$$choose :: (Int, Int) \rightarrow Gen\ Int$$

erzeugt eine Zufallszahl im Angegebenen Zahlenbereich

$$oneof :: [Gen\ \alpha] \rightarrow Gen\ \alpha$$

wählt zufällig einen Generator aus der Liste und lässt ihn einen Wert erzeugen

$$frequency :: [(Int, Gen\ \alpha)] \rightarrow Gen\ \alpha$$

funktioniert sehr ähnlich wie $oneof$, lässt jedoch eine Gewichtung der Generatoren untereinander zu.

$$sized :: (Int \rightarrow Gen\ \alpha) \rightarrow Gen\ \alpha$$

letztendlich benötigt eine Funktion, die eine Größenbeschränkung für Werte realisiert, indem für kleinere Integers stärkere Schranken gesetzt werden als für größere Integers. Ein Beispiel im folgenden Kapitel wird ein wenig Licht in die Funktionsweise werfen.

5 Das Generieren von benutzerdefinierten Daten

Benutzerdefinierte Datentypen müssen, wenn auch Funktionen, die diese Typen als Parameter haben getestet werden sollen, natürlich ebenfalls von *Arbitrary* abgeleitet werden. Mit Hilfe der Kombinatoren kann eine passende *arbitrary* Funktion, und damit ein Generator für den Typ erzeugt werden.

Betrachtet man beispielsweise den Datentyp $data\ Colour = Red|Blue|Green$, so lässt sich eine passende *arbitrary* Funktion beispielsweise wie folgt erzeugen:

```
instance Arbitrary Colour where
  arbitrary = oneof [return Red, return Blue, return Green]
```

Alternativ hätte man hier natürlich auch *elements* verwenden können, um sich die drei einzelnen *return*-Generatoren zu sparen.

Auch rekursiv definierte Datentypen können erzeugt werden:

```
data Tree a = L a | T (Tree a) (Tree a)

instance Arbitrary a => instance Arbitrary Tree a where
  arbitrary = oneof [liftM L arbitrary, liftM2 T arbitrary arbitrary]
```

Ein solcher Baum, bei dem nur die Blätter markiert sind, wird hier erzeugt, indem zufällig entweder ein Blatt erzeugt wird, dessen Markierung von der *arbitrary*-Funktion für den Typ a erzeugt und mit Hilfe der Monad-Funktion $liftM$ an den Konstuktor des Blattes weitergegeben wird, oder ein innerer Knoten, dessen beide Kinder auf die gleiche Weise mit Hilfe der *arbitrary*-Funktion des Baumes und, da es sich um zwei Werte handelt, unter Verwendung von $liftM2$ erzeugt werden.

Es wird schnell ersichtlich, dass die Hälfte der auf diese Weise erzeugten Bäume nur aus einem Blatt besteht. Um dies zu verhindern ist man womöglich versucht, *oneof* durch $frequency$ zu ersetzen:

```
instance Arbitrary a => instance Arbitrary Tree a where
  arbitrary = frequency [(1, liftM L arbitrary),
      (2, liftM2 T arbitrary arbitrary)]
```

Nun werden doppelt so häufig innere Knoten, wie Blätter gewählt und damit nicht mehr so viele zu einfache Testdaten. Es stellt sich jedoch heraus, dass eine gewisse, nicht allzu geringe Wahrscheinlichkeit besteht, dass der Generator beim Erzeugen eines Testwertes divergiert. Für jeden inneren Knoten wird abermals entschieden welche Kinder er haben soll, und mit $\frac{2}{3}$ Wahrscheinlichkeit werden es wiederum innere Knoten sein. Strenggenommen existierte dieses Problem bereits in der Variante mit *oneof*. Um es zu lösen, kann der ohne Beispiel nicht ganz intuitive Kombinator *sized* verwendet werden:

```
instance Arbitrary a => instance Arbitrary Tree a where
  arbitrary = sized arbTree
```

Die Magie liegt in der Funktion $arbTree$, welche eine (zufällig ausgewürfelte) Maximalgröße erhalten und abhängig davon einen Baum generieren soll:

$$
\begin{aligned}
&\mathit{arbTree} \quad :: \mathit{Int} \rightarrow \mathit{Gen}\ a \\
&\mathit{arbTree}\ 0 = \mathit{liftM}\ L\ \mathit{arbitrary} \\
&\mathit{arbTree}\ n = \mathit{frequency}\ [(1,\ \mathit{liftM}\ L\ \mathit{arbitrary}), \\
&\quad (2,\ \mathit{liftM2}\ T\ (\mathit{arbTree}\ (n\ `\mathit{div}`\ 2))\ (\mathit{arbTree}\ (n\ `\mathit{div}`\ 2)))\ ]
\end{aligned}
$$

Für Bäume der Größe 0 wird einfach ein Blatt generiert, Bäume der Größe n hingegen können ebenfalls Blätter sein, oder aber weitere innere Knoten, deren Kinder jedoch nur noch die halbe Größe haben dürfen. Hierdurch wird eine Divergenz verhindert. Die Möglichkeit, auf bei einem erlaubten größeren Baum ein Blatt zu erzeugen ist nötig, um die zufälligkeit der Bäume nicht zu stören. Ohne diese Möglichkeit würden nur balancierte Bäume erzeugt werden.

6 Generierung zufälliger Funktionen

Es können mittlerweile also Werte sowohl von Standard-Typen, alsauch von Listen, Paaren und benutzerdefinierten Datentypen erzeugt werden. Wie sieht es jedoch mit Funktionen aus?

Der Gedanke, zufällige Funktionen zu erzeugen erscheint auf den ersten Blick abwegig, wenn nicht gar unmöglich. Schrittweise kann man sich jedoch an den möglichen Weg herantasten. Der erste Schritt ist die nähere Betrachtung der Generatoren. Wenn man einen Generator nicht mehr als Black Box ansieht, sondern als die Funktion, die er ist, so ist insbesondere der Typ von Bedeutung:

$$\mathit{newtype}\ \mathit{Gen}\ \alpha = \mathit{Int} \rightarrow \mathit{Rand} \rightarrow \alpha$$

Der Integerwert wird hierbei nur für die Erzeugung von größenabhängigen Generatoren benötigt. Vernachlässigt man ihn an dieser Stelle, so sieht man, dass ein Generator nichts anderes ist, als eine Funktion, die eine Zufallszahl in einen Wert vom Typ α umwandelt. Ein Funktionsgenerator für eine Funktion vom Typ $\alpha \rightarrow \beta$ hat damit den Typ:

$$\mathit{Gen}\ (\alpha \rightarrow \beta) = \mathit{Int} \rightarrow \mathit{Rand} \rightarrow \alpha \rightarrow \beta$$

Eine Umordnung der Parameter dieser Funktion ergibt eine andere Funktion vom Typ

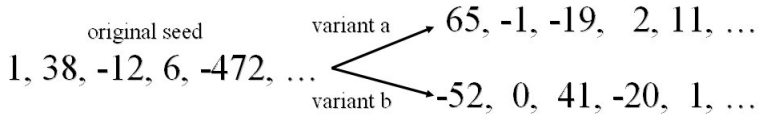
$$\alpha \rightarrow \mathit{Int} \rightarrow \mathit{Rand} \rightarrow \alpha = \alpha \rightarrow \mathit{Gen}\ \beta$$

Dieses Umordnen kann mit Hilfe der Funktion

$$\mathit{promote} :: (\alpha \rightarrow \mathit{Gen}\ \beta) \rightarrow (\mathit{Gen}\ (\alpha \rightarrow \beta))$$

bewerkstelligt werden. Nun mag dies noch immer wenig verständlich und seh abstrakt wirken, jedoch ist der wichtigste Schritt bereits getan. Die Erkenntnis, dass wir zum erzeugen eines Generators für zufällige Funktionen $(\alpha \rightarrow \beta)$ nur eine Funktion benötigen, die abhängig von ihrem Eingabewert vom Typ α einen Generator für Werte vom Typ β erzeugt ist hierbei der Schlüssel.

Um eine solche Funktion zu erhalten, muss man sich zuerst mit der Veränderung der Zufallszahlenfolge beschäftigen. Hier ermöglicht die *variant* :: $Int \rightarrow Gen\ \beta \rightarrow Gen\ \beta$ Funktion, eine Zufallszahlenfolge in Abhängigkeit eines Integerswertes zu verändern:



Ein weiterer Schritt ist die Typklasse *Coarbitrary* deren abgeleitete Typen α eine Instanz der Funktion *coarbitrary* :: $\alpha \rightarrow Gen\ \beta \rightarrow Gen\ \beta$, die im Grunde nichts anderes tut, als den Wert des Types α auf einen zugehörigen Integerwert abzubilden und damit Variant aufzurufen.

Gegeben ein Typ α, abgeleitet von *Coarbitrary* und ein Typ β abgeleitet von *Arbitrary* existiert sowohl ein *coarbitrary* :: $\alpha \rightarrow Gen\ \gamma \rightarrow Gen\ \gamma$ alsauch ein *arbitrary* :: $Gen\ \beta$. Die Funktion $\backslash x \rightarrow coarbitrary\ x\ arbitrary$ vom Typ $\alpha \rightarrow Gen\ \beta$ erzeugt damit in Abhängigkeit eines Wertes vom Typ α einen $Gen\ \beta$ indem der Standard-Generator für β verändert wird. Übergibt man nun diese Funktion der *promote* Funktion, so erzeugt diese daraus einen Funktionsgenerator.

```
instance (Coarbitrary a, Arbitrary b) => Arbitrary (a -> b) where
  arbitrary = promote (\ x -> coarbitrary x arbitrary)
```

7 Fazit

Das Tool stellt sicherlch keine perfekte Lösung für alle Probleme dar. Bei mangelnder Aufmerksamkeit können Fehler, wie divergente Generatoren, oder Testdurchläufe, die nur Trivialfälle testen auftreten. Auf der anderen Seite darf man jedoch nicht vergessen, dass es sich lediglich um ein Hilfsmittel zum Testen handelt, das niemals eine vollkommene Korrektheit beweisen kann.

Es ist jedoch QuickCheck möglich, ohne großen Aufwand eine vielzahl umfassender Tests durchzuführen. Für die genauen Beobachtungs- und Ausgabemöglichkeiten, sowie weitere Konfigurationsoptionen verweise ich auf das Originalpaper.

Sogar alleine die im Quellcode befindlichen *Properties* führen schon dazu, dass dieser etwas besser dokumentiert ist und auch besser verstanden werden kann. So treten in der Praxis durchaus Missverständnisse über die Erfüllung bestimmter Eigenschaften auf, die trotz fehlerfreiem Programm nicht erfüllt werden können und den Programmierer zum Überdenken seiner Spezifikationen bewegen.

Literaturverzeichnis

[1] Claessen, Koen and Hughes, John. 2000. *Quick Check - A Lightweight Tool for Random Testing of Haskell Programs. In ACM SIGPLAN Notices, pages 268–279.*

A Model for Browser/Server Interaction

Andi Scharfstein, 2006

Saarland University, 66041 Saarbruecken, Germany

Abstract. Interactions between web servers and clients are still not well-understood, since no formal descriptions exist for them. A paper by Krishnamurti et al. [1], which gives a first attempt at a formal model for these interactions, is presented and discussed.

1 Introduction

"Programming for the Web is essentially a solved problem."

Well, is it? It is true that we have come a long way since the conception of the web. Scripting languages used to construct dynamic web sites have seen a great surge in interest, large stores like Amazon or eBay have been successfully erected, and most users surf these sites without ever encountering problems.
Now, why should the last item be treated like an achievement? Could it be the case that somehow, users are expected to run into problems in a way they never would in a regular store, and that preventing them from doing so is a cause for celebration and admiration of this technical (or sociological) feat? As a matter of fact, it is, and the rest of this paper will deal with an attempt to clarify and remedy this situation – that is, explain what bugs the user is supposed to encounter, why these bugs exist at all, and how to fix them. Let's begin with a typical problem a user might run into!

2 The Orbitz Bug

Suppose our hypothetical user – let's call him Hans – wants to book a flight online. Hans opens orbitz.com, a popular site offering flights from multiple airlines, and chooses his flight destination. Since Hans is well-versed in the use of browsing techniques, he employs tabs (or *multiple windows*) to compare several flights at once, each offering different benefits for different prices. After carefully choosing the one that suits his needs best, Hans closes all windows but this one, concentrating on his selected flight. However, when he goes on to book this flight, he discovers to his surprise and dismay that the confirmation page doesn't display his chosen flight at all! Instead, it tries to sell him another flight – invariably the one from the most recently opened details window, even though it was subsequently closed. Hans discovers that "going back" from a choice is impossible: whenever he opens a window to look at a flight's details, this flight is set as the one to be booked later on, regardless whether the booking process was initiated from this or any other flight's details page. Disgruntled, Hans leaves the site and books his flight at a competitor's web page.

3 Modelling the Web

Fixing any bug necessitates its complete understanding. The Orbitz Bug was first identified by Krishnamurti et al. (see [1], terminology also from that paper) in 2003, but still persists as of this writing, three years later. One could argue that this signifies a lack of proper understanding of the issues involved in building such an application, so in order to gain this understanding, Krishnamurti et al. developed a formal model describing web interactions.
Some reservations have to be made: The model does not deal with multiple clients accessing a single server, nor with any "concurrency" issues (deadlocks on shared resources, etc.). It also neglects static web pages, instead focusing on the (more interesting) case of dynamic sites. However, each of these concerns can be addressed: Multiple clients can be distinguished via sessions, effectively allowing to model them as single entities (which are covered by the model). Concurrency, while a valid research interest in its own right, is orthogonal to the problem at hand and so can be ignored for the moment. As for static pages, they can easily be modelled as special cases of dynamic pages, so this is really not an issue. Starting from a very abstract view and going into details later on (waterfall-style), a *web configuration* W is just a pair consisting of a single web server and a single client: $W = S \times C$. We shall now look at each component of the pair in detail.

3.1 The Server

Obviously, the web server needs some kind of internal storage to hold user data and the like. It will be modelled by a function $\sigma \in \Sigma$, where Σ is the set of all functions with the type $Id \to V_b$. Id and V_b in turn designate identifiers and values, $V_b = \text{Int} \mid \text{String}$. The σ in use can be thought of as the current server state, since it captures all mutable entities accessible by the server.
The other necessary server component is required by what we "normally" think of as the web server: A dispatcher that deals with the process of looking up the pages and delivering them. In particular, since the web pages are dynamic, the dispatcher has to evaluate a looked-up program with respect to a yet to be defined language, and return the results of this computation. So, aside from a lookup table that assigns programs to URLs, some evaluation function is needed. Formally: The lookup table $P = \text{Url} \to M^\circ$ is a function from URLs to valid programs in some language, denoted by M°. A server is a tuple that consists of storage state and lookup table ($S = \Sigma \times P$), and has an associated dispatch operator d_p that will be properly defined later on in terms of the reductions it allows, which define program evaluation.

3.2 The Client

Taking a formal definition for web pages F for granted (it is given in the very next section), the client can be modelled quite easily as a tuple consisting of the currently shown page (as in, displayed on the screen in the browser window) and

a collection of all pages formerly visited during this session: $C = (F \times \overrightarrow{F})$. The latter can be thought of as the browser cache, although strictly speaking this is not the exact truth (as shall be seen later on).

3.3 Web Pages/Forms

Since the only opportunity for true client/server interaction arises when the client sends information to the server (as the other way round is deterministic, if state-dependent), it is sufficient to only consider pages where the client has the opportunity to do so. This is the case with HTML forms, and nothing else.[1] Hence, for our purposes, we identify web pages with HTML forms, and model only the elements needed to describe a form. To this end, we employ a constructor **form** that takes some URL and a collection of key/value pairs, and constructs their respective HTML representation: $F = (\textbf{form}\ \text{Url}\ \overrightarrow{(Id\ V_b)})$. The URL denotes the location where the information from the key/value pairs is sent and is called *submit URL*. The key/value pairs model text fields where user input can occur (in HTML, `<input type="text">`), and their respective content. So, for any pair (k_0, v_0), an HTML tag `<input type="text" name="`k_0`" value="`v_0`">` will be generated and inserted into the according HTML form construct `<form action="Url">`$\ldots$`</form>` at evaluation time. Typically, all values will start out empty and be filled in by the user later on.

4 Web Interactions

Now that we have the necessary definitions down, let's consider all possible actions a user could perform in such a setting. Surprisingly, three distinct rewriting rules suffice to model the whole range of these possibilities. Entering data into a form input field is the first one. The other two options concern changing the page shown in the current browser window: The user may use the browser's back button or switch between tabs to display any previously visited page at any time, or he may load a new page by submitting the form data on the currently active one. We'll discuss each of these options in detail:

4.1 Filling Out Forms

The first rule is called `fill-form`. It is stated as follows:

`fill-form`: $W \to W$
$\langle \text{s}, \langle(\textbf{form}\ \text{u}\ \overrightarrow{(k\ v_0)}),\ \overrightarrow{f}\rangle\rangle \hookrightarrow \langle \text{s}, \langle(\textbf{form}\ \text{u}\ \overrightarrow{(k\ v_1)}), \{(\textbf{form}\ \text{u}\ \overrightarrow{(k\ v_1)})\} \cup \overrightarrow{f}\rangle\rangle$

In essence, form values can be modified as desired. Since the form is added

[1] With modern web programming techniques such as AJAX, where other interaction paradigms are introduced, this no longer holds true. The discussed paper completely fails to address this issue.

to the "cache" at once after the modification (even before a submit), it is in fact not technically accurate to call it a cache (at least not in the sense used in today's browsers, where only submits indicate cache updates). This doesn't impair the model's functionality for the use cases we are interested in, however, so we'll ignore the issue from now on.

4.2 Switching to Cached Pages

If at first it's not obvious why the model includes the browser cache at all, a look at the side condition to the second rule should clarify this concern: Switching to any page without loading it can only be done if it was previously seen by the client, so naturally the client has to keep track of its visited pages. This is done in the cache. The rule is quite easy to grasp:

`switch`: $W \to W$
$\langle \mathrm{s},\, \langle f_0,\, \overrightarrow{f} \rangle \rangle \hookrightarrow \langle \mathrm{s},\, \langle f_1,\, \overrightarrow{f} \rangle \rangle$, **where** $f_1 \in \overrightarrow{f}$.

It merely states that users may switch to any previously visited page, including the one that is currently shown.[2]

4.3 Submitting Forms

The third and most involved rule captures the notion of submitting form data, including the server's reaction to this.

`submit`: $W \to W$
$\langle \langle \sigma_0, p \rangle,\, \langle f_0,\, \overrightarrow{f} \rangle \rangle \hookrightarrow \langle \langle \sigma_1, p \rangle,\, \langle f_1, \{f_1\} \cup \overrightarrow{f} \rangle \rangle$, **where** $\langle \sigma_1,\ f_1 \rangle \ = \ d_p \langle \sigma_0,\ f_0 \rangle$

The already mentioned lookup function d_p is used to compute a new server state and the next form that will be sent to the browser, depending on the old state and current client form. The server is assigned this new state, and the new form is delivered to the client. On the client side, the currently active page is updated to the new form, which is simultaneously added to the cache. Note that the previous form is already in the cache, because it was either modified (and automatically cached) by `fill-form`, or left unchanged, in which case it was added to the cache during `submit`.

4.4 The Scripting Language

A formal definition of the scripting language is omitted at this point, since it is not particularly enlightening with respect to the problem at hand. Basically, it behaves like the λ-calculus extended with records – see [1] if you're interested in details. However, we will cover its capabilities in a short summary: Besides the

[2] This means that the reduction relation is not terminating. Also note that in `fill-form`, v_0 and v_1 are not required to be distinct.

basics (function application, abstractions, constants, variables), it can handle forms by creating them (with the **form** constructor seen before) and by taking them apart (i.e., getting the value designated by some key). Besides β-reduction, this is the only semantic action defined for the basic language.
The dispatcher d_p works as follows: when a form is submitted, its "successor" is fetched from the form's submit URL. This successor is an abstraction (the only valid program type, M^o) that takes as input the data from the old form, and returns the new one, so all that's left to do for the dispatcher is to apply it accordingly. The new form is then delivered to the client.
The basic language can be extended with the notion of server storage. This is done by adding **read** and **write** directives, which modify the state accordingly, for instance $\langle \sigma,\ E[(\textbf{write}\ Id\ v_b)]\rangle \longrightarrow \langle \sigma[Id \backslash v_b],\ E[v_b]\rangle$, where E is a reduction context, $Id \in dom(\sigma),\ v_b \in V_b$. Note that the storage is server-global.

5 Dissecting the Bug

A careful look at the three rewriting rules should already reveal an interesting fact: only one of them actually modifies the server state, and of the changes that the other two perform, only one can be noticed by the server at some later point in time (i.e., when submitting user data). The problem at the heart of the matter is that the server cannot know if the user has multiple windows opened, since the HTTP protocol is inherently stateless. It doesn't support the "Observer" design pattern [3] (as there is no way to implement a *push* method to get a notification from client to server on a page `switch`), so Krishnamurti et al. call this the *observer problem*. Basically, modern browsers afford the users previously unknown degrees of freedom, while at the same time making web programmers despair of the complexity introduced by not knowing what the user did, or more to the point: where he came from. Certain invariants that programmers implicitly assume while developing the application ("The user will only look at one given flight at a time", "The user only has the opportunity to click on 'Cash cheque' once") no longer have to hold true. A regular store can always rely on the fact that the customer will not buy his products on multiple lanes at once; in an online store, you can't be so sure.
The Orbitz Bug is introduced by a violated assumption, namely that the customer only ever will book the flight he was last interested in (which holds in a sequential model, but not in this setting). It is fixed easily enough by making explicit the distinction between local and global storage (where local refers to the environment defined by a form), and placing the information about which flight should be booked in the environment, that is, the form. The key/value pairs already supported by forms are all that's needed to model local storage: it suffices to make the relevant input fields constant, e.g. by changing the HTML type to `hidden`.

6 Preventing Further Bugs

Now that we have gained a thorough understanding of the issues involved, we have found that fixing the bug isn't really all that hard. Maybe it can even be done automatically?
Using a slightly extended model, the answer is a reasonably qualified yes – while true bug fixes are beyond the scope of any automated computational process, at least warnings can be given when something's gone wrong (in this case, when client and server have run out of sync because of outdated information on the client side). For this to work, first of all the server needs a notion of time. The total number of submits during one session suits this need well. Next, the server needs to know which information can become outdated, so a registry is added to record all fields accessed (i.e., **read** or **written**) during the evaluation of every program. The registry is called that program's *carrier set*. Making use of these facilities, every form is timestamped during its creation, and its carrier set added to its internal storage.
Now, all that's left to do is to keep track of changes to the server state (which only occur during **writes** to some Id). Whenever these changes affect items from any carrier set of an opened form, the form associated with that set is considered outdated, and a warning can be emitted at its submission. Keeping track of **writes** works by modifying the definition of Σ to describe functions of the type $Id \to Time \times V_b$, so every Id now has an associated "last write" timestamp as well as its familiar stored value. Checking works by comparing this timestamp with a submitted form's timestamp for every item of the form's carrier set. If in any of these comparisons the timestamp from the server storage is larger, the submit is potentially outdated and should be treated accordingly.

7 Conclusion

A formal model for a particular kind of web interactions has been presented and discussed, the Orbitz Bug has been explained and fixed, and it has been shown how the type of bugs represented by it can be detected automatically using an updated version of the basic model.
By this, the usefulness of the model as well as the need for further work in this area have been demonstrated. Let's hope that these efforts will enable customers one day to browse **any** online shop with the same ease as a normal store!

References

1. Krishnamurti, S., Findler, R.B., Graunke, P., Felleisen, M.: Modeling Web Interactions and Errors. *Proc. of 12th European Symp. on Programming*, LNCS **2618** (2003) 238–252
2. Licata, D., Krishnamurti, S.: Verifying Interactive Web Programs. *IEEE International Symposium on Automated Software Engineering* (2004)
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. *Addison-Wesley* (1994)

Functional Programming in Hardware Design

Tomasz Wegrzanowski

Saarland University
Tomasz.Wegrzanowski@gmail.com

1 Introduction

According to the Moore's law, hardware complexity grows exponentially, doubling every 18-24 months. While the 1993's Intel Pentium processor had only 3.1 million transistors, the 2004's Intel Itanium 2 has over 592 million, and the number is already getting into billions.

It does not mean that Itanium 2 is 190 times more complex than Pentium, as bigger caches and wider data paths are responsible for most of the transistor count increase, however without doubt the complexity of the ICs is growing very quickly and the hardware designers are trying to find some ways of managing that complexity.

Moore's law is not the only reason why the hardware design is so difficult. Unlike the software, the hardware must be completely correct on the release date, as it is not possible to fix the bugs by publishing a patch later. The highly competitive environment does not permit any delays, or the hardware would be obsolete before even hitting the market. The problem is not limited to the few big firms producing CPUs, memories and other general-purpose chips. Hardware is also designed by a lot of smaller manufacturers, often in ASIC or FPGA technology.

Hardware design usually proceeds in a few well defined stages. The early stages are informal and performed with little computer assistance. The design begins as an idea, which is captured into a vague specification. Such specification is then converted to an abstract algorithm, that precisely defines the designed hardware's functionality, but is not too specific about the implementation details. This algorithm is then refined to the word level, and then to the bit level.

The bit level algorithm usually requires little further human assistance, as the computer software is able to find a good layout of the chip, convert the bit operations to logical gates and connections between them, the gates to individual transistors, and then create the fabrication masks corresponding to those transistors and wires between them. The most time consuming part of the process are the stages where the design is precisely specified, and then refined to word and bit level.

The design serves more functions than being an input to the fabrication process. It is simulated, to explore and debug, it serves as a specification, it is also increasingly often used as an input to a theorem prover, for formal verification.

2 The mainstream of hardware design

The current mainstream of the hardware design are hardware description languages Verilog and VHDL. They are similar in expressive power, with the main difference being the syntax, with Verilog being based on C, and VHDL being based on Ada and slightly more verbose.

The languages were originally meant for precise specification and simulation of computer hardware. Because of that focus, they support various levels of abstraction, from very low level that deals with signal strengths and time bounds on gate delay, to a significantly higher level that treats word level operation like addition and multiplication as primitive. They also provide some support for analog and mixed-signal circuits.

Limited subsets of Verilog and VHDL are considered "synthesizable". If the design uses only synthesizable elements, it can be compiled to a form that is accepted by the hardware manufacturing process. Different technologies consider different constructs "synthesizable".

Verilog and VHDL are lacking in two aspects – they provide limited support for designing at a level higher than word transfer, and they have very complex semantics that makes formal verification difficult.

Many more powerful systems have been recently developed and some are presented here. Most of them operate on a higher level, describing only purely digital systems with a single clock, and for synthesis support compilation to either Verilog or VHDL.

3 Examples of functional hardware design

3.1 Lava

Lava is a Haskell library that exists in a few versions that significantly differ in their architecture, the 1998 version being based on monads, and the 2000 version on explicit circuit representation. There also exists a special version for Xilinx FPGA synthesis.

The 1998 version of Lava [1] is based on monads and type classes. The circuit is a function `a -> m b`, where `a` are the input signals, `b` are the output signals, and `m` is an appropriate monad belonging to `Circuit` type class or one of its subclasses (`Arithmetic`, `Sequential` etc.). The circuits are composed from monadic operations and basic logic gates. Many convenient functions for composing circuits are provided and it is easy to write new ones. However, the `Bit` datatype is abstract, and while it is possible to set the circuit layout using any Haskell code, it is not possible to make arbitrary code operate inside the circuit.

The 2000 version [2] has completely different design. Instead of monads, `Signal t` family of types is used, which provides no way of inserting arbitrary types into it. Values of `Signal t` types can only be constructed using functions provided by the library. Internally, the signals are represented by abstract gates and references to other signals. This "impurity" is necessary to avoid recomputing the signal that is used multiple times, avoiding exponential or in case of

circular definitions even non-terminating behaviour (it is the so called "observable sharing" issue). Specifying the circuits is more convenient than in the 1998 version, as normal function composition can be used instead of monads.
For comparison, here is a half adder circuit in Lava 1998 and Lava 2000.

```
-- Half adder in Lava 1998
halfAdd :: Circuit m => (Bit,Bit) -> m (Bit,Bit)
halfAdd (a,b)=
  do carry <- and2 (a,b)
     sum   <- xor2 (a,b)
     return (carry,sum)

-- Half adder in Lava 2000
halfAdd :: (Signal Bool,Signal Bool) -> (Signal Bool,Signal Bool)
halfAdd(a,b)=(sum,carry)
where sum   = xor2 (a,b)
      carry = and2 (a,b)
```

3.2 Hawk

Hawk [3] is attempting to solve a problem of high-level design of a modern superscalar microprocessor. Conceptually, a processor is executing instructions one at a time. Physically, it has not been the case for a very long time, and at any given moment of time different parts of the processor execute different instructions. The instructions can be executed in parallel, reordered, and even executed speculatively (the processor does not know yet whether their results will be committed or rejected). Many instructions can raise exceptions that break normal execution stream, invalidating not only further instructions but also those that are currently being executed or even those that have already finished and are just waiting for the commit. Instructions that interact with the external world (the memory, the bus, various devices) may also interfere with each other.

These effects cannot be dealt with separately, and it is the interaction between them what causes microprocessor design to take so much time and effort. Even the biggest CPU manufacturers make serious mistakes here. Both the Pentium f00f bug [4] and the Cyrix coma bug [5] were result of mishandled corner cases that allowed unprivileged code to hang the CPU or make it enter an infinite loop.

Hawk is a library built on top of Haskell that provides facilities for dealing with superscalar CPU design. The most important concept is a "transaction" that encapsulates all aspects of an instruction being executed.

The signals in Hawk are values of type `Signal t`. Unlike with both versions of Lava, in Hawk it is possible to apply any function to a signal by using `lift :: (a -> b) -> (Signal a) -> (Signal b)` function.

This design makes it possible to pass arbitrarily complex through the wires, including the aforementioned transactions. Of course, it is not possible to synthesize or automatically prove such circuits.

3.3 HDCaml

Haskell is not the only functional language used in hardware design. An OCaml library HDCaml [6] provides hardware design facilities roughly comparable to Lava 2000.

The architecture of HDCaml is very straightforward – all signals have type `signal`, and there are no type classes, monads, lazy lists or other features. Like in Lava 2000, signal is represented internally by abstract gates and references to other signals. This representation is then used for simulation, generation of Verilog code and verification. Type safety is reduced, as signals of all types have the same type.

4 Common issues

4.1 Deep vs. shallow embedding

The approaches fall into two categories. Hawk and to smaller extend Lava 1998 represent the so called "shallow embedding", where the objects of the embedded language are represented as analogous objects of the host language. For example an adder circuit in Hawk is just a lifted Haskell addition function. This approach allows for easily extending and easier cooperation withe the rest of the host language and other libraries.

The other approach of so called "deep embedding", taken by Lava 2000 and HDCaml, represents the objects of the embedded language explicitly. It makes it more difficult to extend the circuits by new constructs, especially to use other libraries, but on the other hand it makes it much easier to code new circuit transformations (optimization, compilation to Verilog/VHDL) and analyzes (proving, timing computations etc.).

4.2 Observable sharing

A common theme in circuit libraries coded in Haskell is the "observable sharing" problem. The circuit is a small finite graph. It is natural to represent such graph as a set of nodes, each of them linked to others. Such graph contains a lot of sharing, and fully expanded form of it would be exponentially bigger or even infinite. Unfortunately Haskell does not provide us with means of taking advantage of the sharing. Computations on such graphs are as slow as computation of fully expanded graph, that is exponentially slow or even non-terminating.

The solution used by Lava 2000 is to make sharing observable by using some "unsafe" operations. If we can observe the sharing, it is possible to do computations on the circuit graphs much more efficiently.

This problem affects only libraries written in "deep embedding" style. In "shallow embedding" we can represent the signals as infinite streams, and Haskell will take full advantage of any sharing present, not computing the signals more than once.

References

1. Bjesse, P., Claessen, K., Sheeran, M., Singh, S., 1998: Lava: Hardware Design in Haskell. http://www.cs.chalmers.se/ koen/Papers/lava.ps
2. Claessen, K., Sheeran, M., 2000: A Lava Tutorial. http://www.cs.chalmers.se/ koen/Lava/tutorial.ps
3. Matthews, J., Launchbury, J., and Cook, B.: Microprocessor Specification in Hawk. http://www.cse.ogi.edu/PacSoft/projects/Hawk/papers/hawkIntro.ps
4. http://en.wikipedia.org/wiki/F00f
5. http://en.wikipedia.org/wiki/Cyrix_coma_bug
6. http://www.confluent.org/wiki/doku.php

Review - 'Functional Differentiation of Computer Programs' by Jerzy Karczmarczuk

Henning Lars Zimmer (henning@xantippe.cs.uni-sb.de)

Department of Computer Science - Saarland University

Abstract. In the paper of Karczmarczuk, we are introduced to the usage of the functional programming paradigms *lazy evaluation, overloading, type classes* and *co-recursive data structures* to realize algebraic manipulations in *Haskell* programs. The main contribution lies in the *implicit truncation* of the used data structures by lazy evaluation.
One striking example is the functional differentiation of computer programs, yielding point-wise derivatives of *any order* and of *any function* definable in Haskell, with machine precision and without symbolic computations, relying purely on *numerics*.

1 Introduction

'Why do we want to compute derivatives?' The answer is relatively easy. Derivatives are useful for solving *optimization problems*, in *image processing* for object recognition (optic flow computations, segmentation of images) and feature extraction (edges, corners). The field of *3-D-Modelling* uses them for describing geometric properties of curves and surfaces and in many fields of scientific computing like physics, engineering, biology,..., they are indispensable, and therefore we want to have an automatic differentiation, which is fast and accurate.
There are a few *basic ideas* behind our *functional differentiation approach*: One heavily used feature of functional programming languages is *lazy evaluation*. Recall that this strategy postpones the evaluation of function arguments until they are actually needed during the evaluation of the function body, allowing to use *co-recursive data structures* $R\ \alpha = C\ \alpha \mid T\ \alpha\ (R\ \alpha)$ for coding recursive, *infinite* data structures or, to be more exact, of *a priori* unknown, unbounded length. One important aspect is that co-recursion actually *creates* data, whereas strict recursion just traverses or transforms existing data. These two features plus the possibility to *overload* operators acting on these data structures (which is also possible in other languages) offer a powerful team for efficient implementation of algebraic manipulations. The implementation becomes even more elegant if we use Haskell's *type classes*, which allow to build an *algebraic style library*, defining type classes for algebraic structures like groups, rings and fields. The overloaded operators are then implemented as methods of instances of the appropriate type class (e.g. the method multiplication `(*)` should be a method of an instance of the ring class, the one offering division `(/)` an instance of the field class,...).
A huge benefit comes from the *implicit truncation* done by the lazy evaluation strategy. One can define complex co-recursive functions (equations) and the unhappy truncation due to limited memory in a computer is done by the lazy evaluation of the underlying programming language. The presented idea yields a very elegant, clear and semantically powerful *coding tool* for algebra packages.

In [1] and [2] certain applications of co-recursive, lazy data structures are given. Examples are the manipulation (integration and differentiation) of *power series* $U(x) = u_0 + u_1x + u_2x^2 + \ldots$ Furthermore one can iteratively approximate roots by finding zero crossings of functions like $x^2 - 5$, leading to $x_n \approx \sqrt{5}$, using the *Newton method* $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$, with initial guess x_0. But we can also to leave the field of pure mathematics and can –for example– implement methods for generating 'infinite' graph structures.
One further, big field of application is the computation of derivatives, using the *functional differentiation method*, which is the main topic of paper [1] and described in more detail next.

2 Functional Differentiation

The presented approach shows a purely functional (Haskell) implementation of the method, whereas existing packages use low-level languages like C or C++ (see [3]). It uses co-recursive data structures with *lazy evaluation* and is only based on *numerics*. Also *overloading* of arithmetic operators in combination with *type classes* is used. The result are (point-wise) derivatives $f'(\xi), \forall \xi \in dom(f)$ of *any* order (using *co-recursive* data structures) and *any* mathematical function definable in Haskell code.
Basically we have three different ways to compute derivatives with a computer system:
First, *finite differences approximation*: $f'(x) \approx \frac{f(x+\Delta x)-f(x)}{\Delta x}$
The problem hereby is that the method is either *inaccurate* if Δx is too big, or *cancellation errors* occur, if Δx is too small (we get too much undesirable 0 values of our derivatives).
A second approach is *symbolic differentiation*. This is the equivalent to the manual, formal method (paper & pencil). It yields symbolic results, is exact, but *quite costly* because control structures, like branching, loops, etc. have to be 'unfolded', which leads to a symbolic interpretation of the whole program.
Last, but not least, there is the idea of *computational differentiation - CD*, which is the approach of the paper! It purely relies on *numeric* algorithms, is based on standard arithmetic operations, where we (from school) know the differential properties and is as *exact* as numerical evaluation of symbolic derivatives. One basic concept is the *overloading* of arithmetic operators, and therefore there already exist (more efficient) implementations in C++, but here we have to confine ourselves to computing an a priori known number of derivatives. This leads us to our first approach.

2.1 First approach without laziness

This first approach does not make use of laziness, by actually only computing *first derivatives*. In this way we re-gain a bit of efficiency. We use an *extended numerical structure* `Dx`, that groups the numerical value (the *main value*) of an expression e with its first derivative at the same point e', so Dx consists of pairs (e, e'):

```
type Dx = (Double, Double)
```

We note that the type used in `Dx` should be a ring $(R, +, \times)$, or a field $(F, +, \times, /)$ if division is needed.
What we get is `(c, 0.0)` for constants c and `(x, 1.0)` for variables x. It is important

to notice that we perform no symbolic calculations, so our variables don't need to have explicit (symbolic) names, like x. What we obtain are results like `(3.141, 0.0)` for a constant (here: a coarse approximation of π) or `(2.523, 1.0)` for a variable at point 2.523.
The next step is to define *overloaded arithmetic operators* for the type `Dx` (e.g.: `(+) :: Dx -> Dx -> Dx`), implementing the basic derivation laws, like *sum-, product-, quotient-rule, ...*.

```
(x,a)+(y,b) = (x+y, a+b)
(x,a)*(y,b) = (x*y, x*b+a*y)
recip (x,a) = (w,(negate a)*w*w) where w=recip x
{- ... -}
```

We left out the concrete implementation of the operators `(-)`, `(/)` and `negate`, for details see [1].
We also use auxiliary functions to construct constants and variables, and a conversion function `dCst :: Double -> Dx` and `dVar :: Double -> Dx`:

```
dCst z = (z, 0.0) {;} dVar z = (z, 1.0) {;} fromDouble z = dCst z
```

But haven't we forgot a rule?
Yes, we have, the infamous *chain rule* $\frac{d}{dx}(f(g(x))) = f'(g(x)) \cdot \frac{d}{dx}(g(x))$. It is important for computing derivatives of elementary functions like $\exp, \sin, \cos, \log, \sqrt{x}$. These functions `f` are *lifted* to the `Dx` domain, given we know their *derivative form* `f'` (e.g. $sin'(x) = cos(x)$):

```
dlift f f' (x,a) = (f x , a * f' x)
   exp = dlift exp exp
   sin = dlift sin cos
   {- ... -}
```

Note that this leads for example to `exp :: Dx -> Dx`, as expected.
This is all we need and now we can define arbitrary complex functions. An example may be the function $f(x) = x^2 \cdot \cos(x)$ coded as `f x = x*x * cos(x)`. Here we obtain: `f 6.5` $\rightsquigarrow$ `(41.260827, 3.606820)` $\equiv (f(6.5), f'(6.5))$, where *lifting* of `f :: Dx -> Dx` yields `f 6.5` $\rightsquigarrow$ `f (dVar 6.5)` $\rightsquigarrow$ `f (6.5, 1.0)`.
You should have noticed that we omitted the definition of the algebraic style library, using Haskell's *type classes* concept. We will need it for the upcoming, final approach and therefore we define, according to the mathematical hierarchy:
The type class `AddGroup` for addition and subtraction, `Monoid` for multiplication and `Group` for division. We use `Ring` for *structures* supporting addition and multiplication and `Field` adding division to a ring. `Module` abstracts over a multiplication of a complex object by elements of basic domain (e.g.: $\lambda \cdot \boldsymbol{v}$) and `Number` uses `fromInt, fromDouble` to convert standard numbers in our `Dx` domain.

2.2 Final approach

Finally, we want to compute *all (= a priori unknown number)* derivatives of a function. We use a co-recursive structure, representing an expression e of an infinite domain: $e = [e_0, e_1, e_2, ..]$, where e_0 is the main value of e, and e_i, the i-th derivative $e^{(i)}$.
For this purpose, we need a little background in *Differential Algebra*: Consider a field

$(F, +, \times, /)$ with derivation operator $a \mapsto a'$. The case $F = \mathbb{R}$ is trivial, as we get $a \mapsto 0,\ \forall a \in F$. So, we extend the field to $F(x)$ by adjoining an algebraic indeterminate x. But, as the mathematical structure of our expressions is known, we can discard the x, e.g. represent polynomial as (lazy) list of its coefficients.
A last *assumption* is the independence of e and e', yielding the assignment of all derivatives of e by (repeated) derivation $e_n \mapsto e_{n+1}$.
So, we can define the mentioned, co-recursive, parameterized, list-like structure as

```
data Dif a = C a | D a (Dif a)
```

with constants `C a` and `D x (D a (D b ...))` coding the numerical value of the expression (x), combined with the *tower of derivatives* $(a = x', b = x'', \ldots)$. We note that expressions of type `Dif a` are actually created by the co-recursion and they are *not explicitly* truncated. This is done by lazy evaluation! Another point is that we actually do not need `C a`, this is just used for efficiency reasons as the tower of derivatives would be anyway $[0, 0, \ldots]$ for any constant.
The implementation is similar to the first approach, we also use conversion and auxiliary functions, like
`dCst x = C x` and `dVar x = D x 1.0`, where Haskell's *lifting* yields `D x 1.0` $\rightsquigarrow$ `D x (C 1.0)`.
The class `Diff a` encompasses the derivation operator
`df :: a -> a` with `df (C _) = C 0.0` and `df (D _ p) = p`, where the latter just selects the tower of derivatives, acting like the `tail` function on the list-like structure `Dif`. The basic derivation laws are then implemented more or less straight-forwardly but a bit lengthy, as we have to distinguish between the two constructors `C a` and `D a (Dif a)` of our type `Dif a`.
One example is the *sum-rule*:

```
C x + C y = C (x+y)
C x + D y y' = D (x+y) y'    {- and symmetrically D x x' + C y -}
D x x' + D y y' = D (x+y) (x'+y')
```

The *product-rule* encompasses the rule for *unaltered constants* and uses
`x*>s = fmap (x*) s`, where `fmap` is a generic map for the list-like structure `Dif`

```
C x * C y = C (x*y)  {;}  C x * p = x*>p
p@(D x x')*q@(D y y') = D (x*y)(x'*q+p*y')
```

Note that for $D\ (x +_1 y)\ (x' +_2 y')$ holds $(+_1) :: \alpha \to \alpha \to \alpha$,
whereas $(+_2) :: Dif\ \alpha \to Dif\ \alpha \to Dif\ \alpha$
The *reciprocal* $(\frac{1}{u(x)})' = \frac{-u'(x)}{(u(x))^2}$ heavily uses lazy evaluation (see use of `ip`):

```
recip (C x) = C (recip x)
recip (D x x') = ip where ip = D (recip x) (neg x'*ip*ip)
```

Division may be a problem if we face $0/0$. Here, the paper uses the *L'Hopital* rule:

```
p@(D x x') / q@(D y y')
     | x==0.0 \&\& y==0.0 =  x'/y'
     | otherwise          =  D (x/y) (x'*q - p*y'/(q*q))
```

This will work out, but may be not the totally mathematical correct solution. For $e/0$ with $e \neq 0$ we cannot do anything and have to throw an exception.
But we also must not forget the chain rule. Functions `f`, like $\exp, \sin, \cos, \log, \sqrt{x}$ need *lifting* to the `Dif` domain. Here, we got to code their *lists of formal derivatives* `fq`:

```
dlift (f:fq) p@(D x x') =  D (f x) (x' * dlift fq p)
   exp (D x x') = r where r = D (exp x) (x'*r)
   sin = dlift (cycle[sin,cos,(neg . sin),(neg . cos)])
   {- ... -}
```

Basically, this is all we need. What we obtain is for example: `df (df (df (f 6.5)))` $\rightsquigarrow$ `-30.288818` $\equiv f'''(6.5)$, for `f :: Dif a -> Dif a` and lifting yields `f 6.5` $\rightsquigarrow$ `f (D 6.5 (C 1.0))`. Now, we can have a look at concrete applications.

2.3 Applications

The field of CD encompasses a wide spread, huge application domain, 'ranging from reactor diagnostic, meteorology, oceanography, up to biostatistics' ([1]) and quantum theory. Our approach can be used to compute formal solutions of differential equations via iterated differentiation or for the Stirling approximation of the factorial, using asymptotic expansion.
One nice example is the elegant coding of differential recurrences, like the *Hermite function* $H_n(x)$:

$$H_0(x) = exp(\frac{-x^2}{2}) \ , \quad H_n(x) = \frac{1}{\sqrt{2n}}(x \cdot H_{n-1}(x) - \frac{d}{dx}(H_{n-1}(x)))$$

The almost literal, straight-forward implementation is given by:

```
herm n x = cc where
  D cc _ = hr n (dVar x)
  hr 0 x = exp(neg x * x / fromDouble 2.0)
  hr n x = (x*z - df z)/(sqrt(fromInteger (2*n))) where z=hr (n-1) x
```

We should note the very clear and elegant coding compared with other computer algebra packages, which partially use a quite intricate syntax.

3 Evaluation and Summary

Drawbacks of this method are, that thunks of lazy evaluation may screw up the memory, and therefore the efficiency of the approach is not the best, at least a good memory management is crucial! A possible remedy comes from using a strict method, like the first approach, if we a priori know the number of derivatives to compute. Nonetheless, our approach is still useable and really a nice *coding tool*, because of its clearness and compactness. Another unpleasant result is that discontinuous or non-differentiable functions, e.g. `abs x`, also yield a result for their derivatives.
As a *future outlook*, one could think of using an *exact type* for rational numbers [4] instead of types with built-in numerics, like `double`. The question hereby may be if the tradeoff between the (totally) exact computations and the loss of efficiency will pay off. To get a more efficient implementation, we could use a *non-lazy* language like ML,

where we pay the price of explicit truncating the infinite towers of derivatives, which may be quite error-prone. But we could win some efficiency and maintain the mentioned elegance of the coding style using the package.
In conclusion, one can say that the paper shows a rewarding application of modern functional programming paradigms to scientific computing, which is usually the domain of low-level languages. The main *contribution* lies in the usage of lazy evaluation, in connection with co-recursive data structures to code *recurrent equations* elegantly and compact, without having to worry about explicit *truncation*! An example is the *derivation operator*, which is applicable an *a priori* unknown number of times. Type inference and overloading are used for constructing the *overloaded arithmetic operators* and declare *differentiation variables*. Type classes and lifting can be used to extended the arithmetics to any *basic domain*, e.g. complex numbers, polynomials,.... Even a generalization to vector or tensor objects may be possible.

References

1. Karczmarczuk, Jerzy, Functional Differentiation of Computer Programs, *Journal of Higher-Order and Symbolic Computation*, 14:35–37, 2001.
2. Karczmarczuk, Jerzy, Generating power of lazy semantics, *Journal of Theoretical Computer Science*, 187:203–219, 1997.
3. Griewank Andreas et al., ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++ *ACM Transactions on Mathematical Software*, 22(2):131–167, Algorithm 755, 1996.
4. The Haskell Wiki - ExactRealArithmetic, available under `http://www.haskell.org/hawiki/ExactRealArithmetic`