Master of Science Thesis

Reactivity: a Road to Incrementality?

Some sub

by

Alessandro Vermeulen

February 6, 2013

Center for Software Technology
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

Daily Supervisor:
prof. dr. S.D. Swierstra

Abstract

...

Dedication

Acknowledgements

CONTENTS

Chapter 1

Introduction

Incremental programs are programs that update their output using the previous output upon changes to their input. Reactive programs are programs that respond directly to changes in input, e.g. a WYSIWYG text editor. If we see each input gesture as a small change to the input, reactive programs by nature have a need for being evaluated incrementally.

In order to write incremental programs we need incremental functions: Incremental functions are functions that re-use previous computations in order to perform subsequent computations. There are two solutions to this problem that have opposite approaches. The first is to simply remember all previous computations and their arguments, and the second is to re-evaluate only the expressions that have changed.

In a pure language, that is a language in which expressions do not have side-effects, we can easily implement the first solution by memoising every function. Memoisation is a technique where return values of a function are stored so that the result of a future function call with the same arguments can be easily looked up instead of recomputing it. This method is easy to implement in languages that lack higher-order functions and which are a strict. The ability to memoise requires the ability to determine the equivalence of two functions, a problem which is undecidable. In addition memoisation also requires some form of memory management as otherwise the stored information consumes to much memory which leads to a performance hit.

The second method to re-evaluate only the expressions that have changed is by definition the most efficient method and can be implemented by updating the state of our programs by performing assignments to existing values. Automatically obtaining the incremental counterpart of a program through this method is, unfortunately, not always possible, and hence are often created by hand. This has the following disadvantages:

- In addition to writing the business logic, the programmer also has to consider implementation details stemming from the desire to achieve incremental evaluation;
- The final code bears little resemblance to the original code, let alone the original algorithm, thus making the program harder to maintain.

We will illustrate these disadvantages with the code in Figure 1.1 where we show how to obtain an incremental counterpart by applying *finite differencing* [12, 17]. To obtain the incremental counterpart we inlined the definition of *fac*, and interleave the code from *fac* with the original code in *main*, and rewrote the original expressions in *main*. It is clear that the code in Figure 1.1b is more convoluted and thus harder to maintain.

Find an Haskell example

```
int fac(int n) {
  int res = 1;
  for (int i = 1; i <= n; i++) {
    res = res * i;
  }
  return res;
}

void main() {
  print( fac( 5 ) )
  print( fac( 10 ))
}
```

(a) Normal program

```
void main() {
  int n = 1;

  for (int i = 1; i <= 5; i++) {
    n = n * i;
  }

  print( fac( 5 ) )

  for (int i = 6; i <= 10; i++) {
    n = n * i;
  }

  print( n )
}
```

(b) Increment counterpart where the definition of *fac* is inlined.

Figure 1.1: The difference between a normal C program and its incremental counterpart

As with memoisation it is not clear how to apply finite differencing in the presence of higher-order functions as the definition of the argument function is not known beforehand which means that the structure of the code is only known at run-time.

An automated or automatic method of obtaining an incremental counterpart of a program would solve the aforementioned problems with handwritten programs. Reps, Teitelbaum, and Demers [14] show how to achieve incremental evaluation of Attribute Grammars (AGs) by combining a *data-dependency graph*s and *change propagation*. A data-dependency graph is a (directed) graph representing the dependencies between values and change propagation is a technique where, given an initial change, the effects stemming from the change are performed until a stable situation, called a fix-point, has been reached. The data-dependency graph is obtained by static analysis of a AG. This static analysis prohibits

the support for higher-order attributes as those are instantiated dynamically, at run-time. In order to add support for higher-order attributes Saraiva, Swierstra, and Kuiper [15] implemented the evaluation of AGs as strict visitor functions, where were automatically memoised.

In contrast to determining the data-dependency graph statically at compile-time, Acar, Blelloch, and Harper [1] showed how to create a data- dependency graph at runtime as a side-effect of the evaluation of a program and how to apply change propagation over this graph, achieving incremental evaluation. Instead of calling the resulting program an incremental program they call it an *adaptive* program. The result of this work is a library for ML, which introduced the *adaptive* type. This library was was updated in 2006 with revised algorithms and the addition of memoisation [2]. Chen, Dunfield, and Acar [4] extend SML with extra semantics for the adaptive type annotation, creating LML. They also provide a compiler for LML which transforms standard functions to their incremental counterparts using the library from Acar et al. [2]. The only effort from the programmer necessary for this transformation is the annotation of the type with *adaptive*.

Of this library and the compiler only the first version of the library was ported to Haskell [3]. The use of this port, however, breaks lazy-evaluation of expressions built using the library.

In this thesis we will confine ourself to providing a method for automatic incremental evaluation of specific functions in the the programming language Haskell. Haskell is a modern, lazy evaluated functional language Haskell, which stands out for:

- having a strong type-system that helps preventing run-time errors;
- being expressive enough to embed languages;
- and its functions being first-class citizens.

No in-depth knowledge of Haskell is required for understanding this thesis, except for the description of the performed work where we expect the reader to have working knowledge of Arrows (Section 2.3) and Generalised Abstract Data Types (GADTs) (Section 2.4).

We are not the first to present a solution for incremental evaluation of Haskell programs. Leather, Löh, and Jeuring [8] show how to manually transform catamorphisms (folds) to show incremental behaviour.

Our solution is aimed at providing an (semi-)automatic method of writing incremental Haskell programs that resemble their non-incremental counterparts closely and allow the use of higher-order functions while maintaining lazy- evaluation. It is based on the dynamic generation of the data-dependency graph as shown by Acar, Blelloch, and Harper [1]. Furthermore we use the expressiveness of Haskell to keep this a library-only feature by using the already available

Arrow Syntax [13]. The method of modelling our computation with Arrows is derived from FRP as described by Nilsson, Courtney, and Peterson [11].

Motivation The motivation for our work is simple, there currently is no method for automatically obtaining incremental Haskell programs. Having a method for automatically obtaining incremental counterparts makes it easier to achieve better performing Haskell programs. We opted to keep this work contained to a library because Haskell is expressive enough to keep this at the level of the language without having to resort to hacking the compiler, reducing the amount of work necessary to maintain the feature.

Research question In this thesis we answer the following research question: How can we obtain an incremental counterpart of a normal Haskell program while:

- the incremental-counterpart resembles the original code as much as possible;
- the lazy semantics of Haskell is being maintained;
- the effort required to do so is minimal, preferably of the same order of magnitude as required with the compiler from Chen, Dunfield, and Acar [4];
- and the expressiveness of Haskell is utilized as much as possible, preferably avoiding changing the compiler or performing AST-manipulation with Template Haskell [16].

The remainder of this thesis is structured as follows: Chapter 2 will provide background information for this thesis, starting with an explanation of the terminology used in this thesis (Section 2.1). Which is followed by explanations of Category Theory (**??**), Arrows in Haskell (Section 2.3), and GADTs (Section 2.4). Chapter 3 relates the previous work on incremental evaluation to the work done in this thesis. **??** will describe the contribution of this thesis. We will conclude this thesis in Chapter 5 where we will present a discussion (Section 5.1), briefly visit related work (Section 5.2), suggest future work (Section 5.3) and present final conclusions (Section 5.4).

Chapter 2

Background

In this chapter we will introduce the terminology used in this thesis by giving formal definitions, as well as introduce background information on how concepts from Category Theory (Section 2.2), Arrows (Section 2.3), and GADTs (Section 2.4) are defined in Haskell as the presented work is based upon these concepts.

2.1 Terminology

2.1.1 Incremental computation

Given a function $f\ x :: \alpha \to \beta$ we can express incremental evaluation as the following equation:

$$f\ (x\ \oplus\ \delta x) \equiv f\ x\ \otimes\ f'\ x\ \delta x\ (f\ x)$$

Where $\oplus$ is an operator representing an update function for type α, and $\otimes$ is the matching update function for type β and f' is a function that determines the counterpart of δx for $\otimes$. Here f' uses the result of the previous computation $(f\ x)$, the previous input x and the difference (δx) to compute the delta for $\otimes$.

We illustrate this with the code shown in Figure 2.1. In Figure 2.1a the whole map is recomputed, where Figure 2.1b updates the result according to the change of the input. We are allowed to perform this transformation because of lemma defined below. This transformation is an example of *finite differencing*. In the given example the following mapping exists:

$$\begin{aligned} x &= [1\,..\,5] \\ f &= map\ (*\,2) \end{aligned}$$

```
g = let xs  = [1..5]
        ys  = map (*2) xs
        xs' = insert 3 10 xs
        ys' = map (*2) xs'
    in  ys'
```

(a) Non-incremental computation

```
g' = let xs  = [1..5]
         ys  = map (*2) xs
         xs' = insert 3 10 xs
         ys' = insert 3 (10*2) ys
     in  ys'
```

(b) Incrementalized computation

Figure 2.1: Inserting an item in a list

$$
\begin{aligned}
\oplus &= insert\ 3 \\
\delta x &= 10 \\
\otimes\ f'\ x\ \delta x\ (f\ x) &= insert\ 3\ (10 * 2)
\end{aligned}
$$

Lemma 1. *map − insert*

$map\ f\ (insert\ i\ x\ xs) \equiv insert\ i\ (f\ x)\ (map\ f\ xs)$

2.1.2 Reactivity

We will now continue with an explanation of the concepts of Category Theory, Arrows, and GADTs. The knowledgeable reader is invited to skip ahead to the relations between our contribution and previous contributions (Chapter 3).

2.2 Category Theory

Category theory is the theory that describes the relations between objects, arrows between these objects, and the way arrows are composed.

In Haskell a category is captured by the following class, in essence relating input with output.

```
class Category 𝒞 where
  id  :: 𝒞 α α
  (∘) :: 𝒞 β γ → 𝒞 α β → 𝒞 α γ
```

Simply defining an instance declaration for *Category* for a given datatype, however, is not enough. In addition, the instance should have the following properties, *id* should be the identity function under composition, and composition should be associative:

identity $\forall\ p : id \circ p = p$ and $\forall\ p : p = p \circ id$

associativity $\forall\ p\ q\ r : (p \circ q) \circ r = p \circ (q \circ r)$

The most intuitive example of a category is the Haskell function arrow $(\rightarrow)$, where the types correspond to objects and the Haskell functions correspond to the arrows.

```
instance Category (→) where
  id = Prelude.id
  (∘) = (Prelude.∘)
```

Basic explanation of category theory

2.3 Arrows

Arrows in Haskell are an extension of the category class. In addition to composition, they provide a mechanism for capturing additional information on data-flow over categories with *split* and *fanout* which provide insight on how data flows to sub-components.

In contrast with Monads which only capture the result type, arrows (and categories) capture the type of their input as well as their output. This makes arrows exceptionally suited for modelling data-flow.

The Haskell type class is defined as the code below, where *arr* lifts a Haskell function into an arrow. The function *first* sends the first component of the input through the argument arrow, and copies the rest unchanged to the output, or from a different point of view, applies the argument arrow to the first input. The function *second*, is the mirror of *first*. The operator ***, read 'split', splits the input between the two argument arrows and collects the output. The operator &&&, read 'fanout', sends the input to both arrows and combines their output.

```
class Category α ⇒ Arrow α where
  arr     :: (β → γ) →              α β       γ
  first   :: α β γ   →              α (β, d)  (γ, d)
  second  :: α β γ   →              α (d, β)  (d, γ)
  (***)   :: α β γ   → α β′ γ′ →    α (β, β′) (γ, γ′)
  (&&&)   :: α β γ   → α β  γ′ →    α β       (γ, γ′)
```

Instances of *Arrow* also need to have to following properties for them to be real Arrows.

Lookup the real names of these rules

distributivity-*arr* $\forall\ f\ g : (arr\ f) \circ (arr\ g) = arr\ (f \circ g)$

$\forall\ f : first\ (arr\ f) = arr\ (first\ f)$

$\forall\ f : second\ (arr\ f) = arr\ (second\ f)$

- $\forall\ f\ g : arr\ f \ast\!\!\ast\!\!\ast\ arr\ g = arr\ (f \ast\!\!\ast\!\!\ast\ g)$
- $\forall\ f\ g : arr\ f\ \&\&\&\ arr\ g = arr\ (f\ \&\&\&\ g)$
- $\forall\ f\ g : (first\ f) \circ (first\ g) = first\ (f \circ g)$
- $\forall\ f\ g : (second\ f) \circ (second\ g) = second\ (f \circ g)$

Add Arrow syntax examples

Add Arrow background links

2.4 GADTs

Chapter 3

Previous Work

In the remainder of this thesis we will use three examples to relate the results of previous approaches with our contribution.

We will consider the following properties:

- Does the solution support lazy-evalution?
- Does the solution support higher-order functions?
- For which language is the solution available? (Haskell, Attribute Grammars, ML or the LRC);
- Is the solution a library, a compiler, or a combination?
- Does the solution offer support for static/dynamic optimisation?
- Does the solution support deployment on another platform, such as JavaScript?;

3.1 Running Examples

3.1.1 Example 1: A simple expression

When looking at the expression: $x + y * z$, we see that the sub-expression $y * z$ doesn't depend on the value of x. Consequently, when x changes we ideally should only have to recompute the $(+)$ with the new value of x. When either y or z changes we need to perform the whole computation again: both the multiplication and the addition.

3.1.2 Example 2: The length of a list

3.1.3 Example 3: Control-flow, higher-order

3.2 Finite Differencing and program transformation

3.3 Solutions in an imperative setting

When using an imperative, object-oriented language, such as Java, JavaScript, or C#, it is relatively straightforward to achieve incremental evaluation as local assignments make it possible to directly update variables.

One of the standard ways to achieve incremental behaviour is to make use of the so-called observer pattern [5]; each value which is to be recomputed is represented by an object which subscribes to the changes of the objects it depends on. Whenever one of these objects changes value, the depending object is notified and re-evaluates itself in due time. Implementations of this idea differ in the way the programmer can or has to specify the order of re-evaluation, on whether the dependency structure can change as a result of re-evaluation, and how these changes to the structure are scheduled for evaluation themselves.

A solution that uses change propagation is called Reactive extensions[10] and is available for C#. This library does not provide support for automatic incrementalisation. However, the reactive functions are composable giving the possibility of an incrementalisationtal computation.

Aangeven dat dit makkelijk een rotzooitje wordt: dit kan snel verzanden in het 'slim' moeten kiezen van welke methoden aangeroepen worden, om de state soms wel aan te passen maar om verdere observers nog niet te triggeren etc.

3.4 Incremental Attribute Grammar evaluation

3.5 Incrementalisation in ML

Due to no laziness all changes are *pushed*, and thus potentially to much work is done.

In this section we will introduce Acar's work on incremental computation. In two iterations [1, 2] Acar et al. have developed and perfected a library for incremental programs in ML. In a third iteration [4] writing incremental ML programs is made simple. We will step through the iterations of the library, using the aforementioned examples.

```
sig plus = int mod → int mod → int mod
fun lift2 f l r =
  mod ( ≡ ) (fn d ⇒ read (l,
                          fn l' ⇒ read (r,
                                        fn r' ⇒ write (d,
                                                       f (l', r'))))
fun plus l r = lift2 ( + ) l r
fun mul l r = lift2 ( * ) l r
fun newElt (v) = mod ( ≡ ) (fn d ⇒ write (d, v))
fun test (lst, v) =
  let
    val α = init ()
    val x = newElt 4
    val y = newElt 8
    val z = newElt 9
    val r = x + y * z
  in
    (change (x, 2);
    propagate ();
    r)
  end
```

Figure 3.1: ...

The first iteration of the library [1] provides three functions *mod*, *read*, and *write* for declaring dependencies between modifiable references, and *change* and *propagate* for the actual mutation of values and propagation of changes, respectively.

Rewrite this

Example 1 We again use the example $x + y * z$. Remember that we do not want to recompute the multiplication when only x changes. The variables x, y, and z are modifiable references of type *Int*. Additionally for each $+$ and $*$ node an extra modifiable of type *Int* is created as shown in **??**. When we perform a change to x the result of the addition is invalidated and has to be recalculated. We are able to do so efficiently because the result of the multiplication is still accessible in the result variable of the multiplication node. Without this, we could not efficiently recompute the expression.

Example 2

```
datatype 'a list' =
    NIL
  | CONS of ('a * ' α list' mod)
fun modl f = mod (fn (NIL, NIL) ⇒ true
                   | x          ⇒ false) f
```

Figure 3.2: Adaptive lists as defined in [1], only adaptive in the structure of the list.

Example 3

3.6 Incremental evaluation in Haskell

There exists a port[3] of Acar's first work[1] in Haskell. It adds type-safety to the readers and writers by using the class-system. Rewrite this

memoising catamorphisms in Haskell

3.7 (Arrowized) Functional Reactive Programming

3.7.1 YAMPA

3.7.2 reactive-banana

3.8 Recap current state

Chapter 4

Incrementality in Haskell

In this chapter we will introduce our contribution, a library for achieving incremental evaluation in Haskell. The approach taken with this library is based upon the dynamic construction of data-dependency graphs as done by Acar, Blelloch, and Harper [1]. The lazy evaluation that Haskell offers is maintained.

The code is structured as follows: there are *reactive variables* that go by the type *RVar* α. These *RVar*s contain the value α and meta-data. This meta-data consists of debugging information as well as a list of *RVar*s that have the given *RVar* as a source.

There are basic functions that *link* *RVar*s together, we will describe them in the following section. Furthermore we describe an embedded Domain-Specific-Language (DSL) (eDSL) that uses the arrow syntax as supported by the GHC that is used to write *models* of functions that actually resemble normal Haskell functions [7].

4.1 Basic combinator functions

4.2 Monad combinator functions

4.3 Arrow combinators

4.3.1 The GADT

My GADT is defined as follows:

The R datatype is a GADT that provides deep-embedding of the *Category*, *Arrow*, *ArrowChoice*, *ArrowLoop* and *ArrowApply* classes, this means that we have a constructor for each function in the mentioned classes:

```
data R α β where
    -- Category
  Id     :: R α α
  Comp   :: R β γ → R α β  → R α γ
    -- Arrow
  Arr    :: (α → β) → R α β
  Split  :: R β γ → R β′ γ′ → R (β, β′) (γ, γ′)
  Cache  :: (α → α → Bool) → R α α


  -- ArrowChoice
Choice :: R β γ → R β′ γ′ → R (Either β β′) (Either γ γ′)
  -- ArrowLoop
Loop   :: R (β, d) (γ, d) → R β γ
  -- ArrowApply
Apply  :: R (R β γ, β) γ


  -- Recursion
Fix :: (R α α → R α α) → R α α
Rec :: R α β → R α β
deriving Typeable


data Label α β = Label {unLabel :: StableName (R α β)}
  -- Why a single f? The recursive call should go to the same
  -- type.
data RFix f = RIn {out :: R (f (RFix f)) (f (RFix f))}
```

Makin R an instance of the the aforementioned classes is straightforward and is a matter of calling the right constructors. For reasons of simplicity we choose to express *first* and *second* in terms of $***$. The functions *left* and *right* are expressed in terms of $+++$ for the same reason. We can choose to add extra constructors for performance reasons in a later stage.

```
instance Category R where
  id = Id
  (∘) = Comp
instance Arrow R where
  arr    = Arr
  first  = (*** Cat.id)
  second = (Cat.id ***)
  (***) = Split
```

```
instance ArrowChoice R where
  left      = (+++ Cat.id)
  right     = (Cat.id +++)
  (+++) = Choice
instance ArrowLoop R where
  loop      = Loop
instance ArrowApply R where
  app = Apply
```

4.3.2 Installing the model

Tidy this up for inclusion in Thesis

Cache the value before providing it to the function if the input is the same the output should be the same as well. This propagates throughout the model of course.

```
cache :: Eq α ⇒ R α β → R α β
cache = (Cache (≡) ⋙)
```

```
installCache :: (Eq α, Eq β) ⇒ R α β → IO (RVar α, RVar β)
installCache = install' ∘ (λx → Cache (≡) ⋙ x ⋙ Cache (≡))
install' :: R α β → IO (RVar α, RVar β)
install' model = do
  input ← newNamedRVal "input" (error "Uninitialized install'")
  output ← install model input
  return (input, output)
install :: R α β → RVar α → IO (RVar β)
install (Id) inp = return inp
```

The implementation of the cache, sadly, relies on the handling of errors thrown by evaluation of $\bot$. An attempt to wrap α in *RVal* with *Maybe* lead to pattern matches elsewhere which broke lazy-evaluation.

```
install (Cache eq) inp = do
  out ← newRVal (error "Uninitialized install")
  let setter n = trace "Set!" $ void $ setRVal out n
  let set n o = handle (λ(_ :: ErrorCall) → setter n) $
    unless (eq n o) (setter n)
  linkRVal out inp set
  trace "install.Cache" $ return out
```

```
install (Comp g f) inp = do
  outF ← install f inp
  outG ← install g outF
  trace "install.Comp" $ return outG
install (Arr f)     inp =
  trace "install.Arr" $ rValFMap f inp
install (Split f g) inp = trace "install.Split" $ installSplit f g inp
  -- ArrowChoice
install (Choice l r) inp = trace "install.Choice" $ installChoice l r inp
install (Loop f)    inp = trace "install.Loop"   $ installLoop f inp
install Apply       inp = trace "install.Apply" $ installApply inp

installSplit :: R β γ → R β' γ' → RVar (β, β') → IO (RVar (γ, γ'))
installSplit f g inp = do
  projL ← rValFMap fst inp
  projR ← rValFMap snd inp

  outL ← install f projL
  outR ← install g projR

    -- TODO: Should we initialize here?
  out  ← newRVal ((error "installSplit.undefined...", error "installSplit.undefined...2")) ::
    --
  linkRVal out outL (λx _ → void $ modifyRVar out (first (const x)))
  linkRVal out outR (λx _ → void $ modifyRVar out (second (const x)))

  return out

installChoice :: R β γ → R β' γ' → RVar (Either β β') → IO (RVar (Either γ γ'))
installChoice l r inp = do
  (inL, outL) ← install' l
  (inR, outR) ← install' r

  out ← newRVal (error "installChoice.newRVal") :: IO (RVar (Either γ γ'))

  let setIn α _ = case α of
                    Left x → void $ setRVal inL x
                    Right x → void $ setRVal inR x

  let setOutL α _ = void $ setRVal out (Left α)
  let setOutR α _ = void $ setRVal out (Right α)

  linkRVal inL inp setIn
  linkRVal inR inp setIn

  linkRVal out outL setOutL
  linkRVal out outR setOutR

  return out

installLoop :: R (β, d) (γ, d) → RVar β → IO (RVar γ)
installLoop f inp = do
    -- loop f b = let (c,d) = f (b,d) in c
  let update x = do rec (y, out) ← do (inR, outR) ← install' f
```

```
    setRVal inR (x, out)
    getRVal outR
                    return (y, out)
  r ← rValFMapIO update inp
  rValFMap fst r
installApply :: RVar (R α β, α) → IO (RVar β)
installApply inp = do
  out ← newRVal (error "installApply.newRVal")
    -- TODO: This is not compatible with parallel execution of setHandlers. As it breaks
    -- the idea of responding to sets. Perhaps the getRVal reads an
    -- older version of the value.
  let onSet (r, v) _ = do (inp', out') ← install' r
                          setRVal inp' v
                          v' ← getRVal out'
                          setRVal out v'
                          return ()
  link ← linkRVal out inp onSet
  return out
fix = Fix

runR :: R α β → α → β
runR Id           = id
runR (Cache _)    = id
runR (Comp g f)   = runR g ∘ runR f
runR (Arr f)      = f
runR (Split f g)  = runR f *** runR g
runR (Choice l r) = runR l +++ runR r
runR (Loop f)     = loop (runR f)
runR Apply        = λ(f, x) → (runR f) x
```

4.3.3 Optimising the model

4.3.4 Translation to JavaScript

This needs a lot of work. We would actually need to paramerise the model with the type of the function that should be lifted in *arr* for example.

4.4 $solution

Decide on solution, `SKI` or Observeable Sharing?

Chapter 5

Concluding

5.1 Discussion

5.2 Related Work

Incremental programs are programs that maintain the invariant of the relation between input and output by updating their output. The "Views" system introduced by maintaining invariants that are inferred from constraints, this is called constraint satisfaction [9]. Ganzevoort [6] was the first to introduce the name *Structure Watching* for the technique of maintaining invariants in the setting of imperative languages.

Apparently some effort has been made to achieve reactive programs in an imperative setting as well: "Reactive extensions" is a framework for C# that introduces concepts for creating reactive C# programs.

5.3 Future Work

5.4 Conclusion

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. "Adaptive functional programming". In: *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '02. Portland, Oregon: ACM, 2002, pp. 247–259. ISBN: 1-58113-450-9. DOI: 10.1145/503272.503296. URL: http://doi.acm.org/10.1145/503272.503296.

[2] Umut Acar et al. "A Library for Self-Adjusting Computation". In: *Electron. Notes Theor. Comput. Sci.* 148.2 (Mar. 2006), pp. 127–154. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.11.043. URL: http://dx.doi.org/10.1016/j.entcs.2005.11.043.

[3] Magnus Carlsson. "Monads for incremental computing". In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. ICFP '02. Pittsburgh, PA, USA: ACM, 2002, pp. 26–35. ISBN: 1-58113-487-8. DOI: 10.1145/581478.581482. URL: http://doi.acm.org/10.1145/581478.581482.

[4] Yan Chen, Joshua Dunfield, and Umut A. Acar. "Type-directed automatic incrementalization". In: *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. PLDI '12. Beijing, China: ACM, 2012, pp. 299–310. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254100. URL: http://doi.acm.org/10.1145/2254064.2254100.

[5] E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. ISBN: 9780201633610. URL: http://books.google.nl/books?id=iyIvGGp2550C.

[6] Job Ganzevoort. *Maintaining presentation invariants in the Views system*. Tech. rep. Amsterdam, The Netherlands, The Netherlands, 1992.

[7] GHC. *7.15. Arrow notation*. [Online, accessed 4-february-2013]. 2013. URL: http://www.haskell.org/ghc/docs/latest/html/users_guide/arrow-notation.html.

[8] Sean Leather, Andres Löh, and Johan Jeuring. "Pull-ups, push-downs, and passing it around". In: *Proceedings of the 21st international conference on Implementation and application of functional languages*. IFL'09. South Orange, NJ, USA: Springer-Verlag, 2010, pp. 159–178. ISBN: 3-642-16477-

3, 978-3-642-16477-4. URL: http://dl.acm.org/citation.cfm?id=1929087.1929097.

[9] Lambert G.L.T. Meertens and Steven Pemberton. *The ergonomics of computer interfaces – Designing a system for human use.* Tech. rep. Amsterdam, The Netherlands, The Netherlands, 1992.

[10] Erik Meijer. "Reactive extensions (Rx): curing your asynchronous programming blues". In: *ACM SIGPLAN Commercial Users of Functional Programming*. CUFP '10. Baltimore, Maryland: ACM, 2010, 11:1–11:1. ISBN: 978-1-4503-0516-7. DOI: 10.1145/1900160.1900173. URL: http://doi.acm.org/10.1145/1900160.1900173.

[11] Henrik Nilsson, Antony Courtney, and John Peterson. "Functional reactive programming, continued". In: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. Haskell '02. Pittsburgh, Pennsylvania: ACM, 2002, pp. 51–64. ISBN: 1-58113-605-6. DOI: 10.1145/581690.581695. URL: http://doi.acm.org/10.1145/581690.581695.

[12] Robert Paige and Shaye Koenig. "Finite Differencing of Computable Expressions". In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), pp. 402–454. ISSN: 0164-0925. DOI: 10.1145/357172.357177. URL: http://doi.acm.org/10.1145/357172.357177.

[13] Ross Paterson. "A new notation for arrows". In: *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*. ICFP '01. Florence, Italy: ACM, 2001, pp. 229–240. ISBN: 1-58113-415-0. DOI: 10.1145/507635.507664. URL: http://doi.acm.org/10.1145/507635.507664.

[14] Thomas Reps, Tim Teitelbaum, and Alan Demers. "Incremental Context-Dependent Analysis for Language-Based Editors". In: *ACM Trans. Program. Lang. Syst.* 5.3 (July 1983), pp. 449–477. ISSN: 0164-0925. DOI: 10.1145/2166.357218. URL: http://doi.acm.org/10.1145/2166.357218.

[15] João Saraiva, S. Doaitse Swierstra, and Matthijs F. Kuiper. "Functional Incremental Attribute Evaluation". In: *Proceedings of the 9th International Conference on Compiler Construction*. CC '00. London, UK, UK: Springer-Verlag, 2000, pp. 279–294. ISBN: 3-540-67263-X. URL: http://dl.acm.org/citation.cfm?id=647476.727632.

[16] Tim Sheard and Simon Peyton Jones. "Template meta-programming for Haskell". In: *SIGPLAN Not.* 37.12 (Dec. 2002), pp. 60–75. ISSN: 0362-1340. DOI: 10.1145/636517.636528. URL: http://doi.acm.org/10.1145/636517.636528.

[17] Doron Swade and Charles Babbage. *Difference Engine: Charles Babbage and the Quest to Build the First Computer*. Viking Penguin, 2001. ISBN: 064153440X.

AG Attribute Grammar

CPS Continuation Passing Style

DSL Domain-Specific-Language

eDSL embedded DSL

GADT Generalised Abstract Data Type

Todo list

Woordenlijst

- incrementalization
- recursion
- recursive datastructures
- recursive algorithms

- applicative
- monads
- arrows
- sharing
- overloaded application
- observable binding
- observable structure
- mutable variables / references
- efficiency
- optimisation
- lists
- folds
- deep embedding
- side-effects
- Fix
- Mu
- Nu
- Manual mutations
- Analyseable
- Optimisation
- Pattern matching (Conditions / etc.)
- Modularity and composibility
- Modelling input
- Garbage Collection
- Granularity for changeable