Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 2, Issue. 7, July 2013, pg.178 – 185

RESEARCH ARTICLE

Pragmatic Reactive Programming on Helpless Records

Kaja Masthan

Assistant Professor, Department of computer science and Engineering, Sphoorthy Engineering College, Hyderabad, Andhra Pradesh, India

Khaja7.skm@gmail.com

Abstract— Pragmatic Reactive Programming (PRP) is an approach to reactive programming where systems are structured as networks of functions operating on signals. PRP is based on the synchronous data-flow paradigm and supports both continuous time and discrete-time signals (hybrid systems). PRP apart from most other languages for similar applications it support for systems with dynamic structure and for higher-order reactive constructs. Statically guaranteeing correctness properties of programs is an attractive proposition. This is true in particular for typical application domains for reactive programming such as embedded systems and animating applications. To that end, many existing reactive languages have type systems or other static checks that guarantee domain-specific properties. We have presented confined types allow this concern to be addressed. Implementation of PRP embedded in the confined typed language Agda, leveraging the type system of the host language to craft a confined type system for PRP. The implementation constitutes a discrete, operational semantics of PRP, and as it passes the Agda type, coverage, and termination checks.

Key Terms: - Operational Semantics; Agda; Hybrid Systems; Signals; Semantics

I. INTRODUCTION

The idea of PRP is to allow the modern Functional Programming to be used for implementing reactive systems: systems that interact with their environment in a timely manner. This is achieved by describing systems in terms of functions mapping signals (time-varying values) to signals, and combining such signal functions into signal processing networks. The nature of the signals depends on the application do-main. Examples include input from sensors in robotics applications video streams in the context of graphical user interfaces. A number of PRP variants exist. However, the synchronous data-flow principle, and support for both continuous and discrete time (hybrid systems), are common to most of them. [1]There are thus close connections to synchronous data-flow languages such as Esterel and Lucid Synchrone Hybrid Automata. PRP goes beyond most of these approaches by supporting dynamism (highly-dynamic system structure), and first-class signal functions.

Dynamism and higher-order data-flow are becoming important aspects of reactive programming as they are essential for implementing reconfigurable systems, including systems that receive software updates whilst running, which are increasingly prevalent. Statically guaranteeing central domain-specific correctness properties is consequently also becoming much more important, as dynamism and higher-order data-flow add levels of system complexity which make it correspondingly harder to test systems sufficiently thoroughly. Moreover, in many reactive application scenarios, the cost of failure is very high thereby making it imperative to statistically

guarantee that the system will not fail. Yampa is an embedding of PRP in Haskell that supports dynamism and first-class signal functions. From the perspective of reactive programming, the Haskell-based type system of Yampa is arguably not safe, as it does not enforce important domain-specific correctness properties [4]. There are reactive languages that do enforce such domain-specific properties but their support for dynamism or higher-order data-flow is limited.

For this problem, we developed a domain-specific type system for PRP that guarantees two central domain-specific correctness properties, Ill-formed feedback loops and proper initialization, while still allowing for dynamism and first-class reactive entities. The type system is safe in that it guarantees that reactive programs are productive under the assumption that the pure functional code Animation in the signal-processing network is total and terminating. Ill-typed term representing the updated configuration. Because the semantic function is total and terminating, it constitutes a proof that the embedded type system guarantees the productivity of Ill-typed signal-function networks, which is the safety property with which we are concerned here. A further benefit of making domain-specific properties manifest in the types of signal functions is that this clarifies their semantics, which, in turn, offers strong guidance as to their proper use. This is in stark contrast to Yampa, where subtle but crucial properties are often implicit; possibly leading to confusion about the exact relation is taken differently named combinatory with the same type.

In this paper, firstly, there is a clear type-level distinction between continuous-time and discrete-time signals. In Yampa, the latter are just continuous-time signals carrying an option type. As a result, certain signal functions, such as the various delays, that in order to guarantee desirable semantical properties would have to treat continuous-time and discrete-time signals differently, actually treat them uniformly. This is another source of subtle bugs that can be eliminated by the more precise type system presented secondly; [3] our development is structured around N-ary signal functions, through the notion of signal vectors. This enables a number of important optimizations, such as change propagation, to an extent that is not possible in Yampa.

II. BACKGROUND

PRP programs can be considered to have two levels to them: a functional level and a reactive level. The functional level is a pure, functional language. PRP implementations are usually embedded in a host language, and in these cases the functional level is provided entirely by the host. In the case of Yampa, the host language is Haskell. The reactive level is concerned with time-varying values, which I call signals. At this level, combinators are used to construct synchronous data-flow networks by combining signal functions. The levels are interdependent: the reactive level relies on the functional level for carrying out arbitrary point wise computations on signals, while reactive entities, such as signal functions, are first class entities at the functional level.

2.1 Continuous-Time Signals

The core conceptual idea of PRP is that time is continuous. Signals are modeled as services from time to value, where we take time to be the set of non-negative real numbers:

$$\text{Time} = \{t \in R \mid t > 0\}$$

$$\text{Signal } a \approx \text{Time} \rightarrow a$$

This conceptual model provides the foundation for an ideal PRP semantics. Of course, any digital implementation of PRP will have to execute over a discrete series of time steps and will consequently only approximate the ideal semantics. The advantage of the conceptual model is that it abstracts away from such implementation details. It makes no assumptions as to the rate of sampling, whether this sampling rate is fixed, or how this sampling is performed [2]. It also avoids many of the problems of composing subsystems that have different sampling rates. The ideal semantics is helpful for understanding PRP programs, at least to a first approximation. It is also abstract enough to leave PRP implementers considerable free-dom. That said, implementing PRP completely faithfully to the ideal semantics is challenging. At the very least, a faithful implementation should, for “reasonable programs”, converge to the ideal semantics in the limit as the sampling interval tends to zero. But even then it is hard to know how densely one needs to sample before an answer is acceptably close to the ideal.

2.2 Signal military

Signal services conceptually services from signal to signal:

SS a b ≈ Signal a → Signal b

In Yampa, signal functions, rather than signals, are first class entities. Signals have no independent existence of their own. They exist only indirectly through the signal functions. To make it possible to implement signal services in such a way that output is produced in lock-step with the input arriving, as is required for a system to be reactive.

Causal Signal Function: A signal function is causal if, at any given time, its output can depend upon its past and present inputs, but not its future inputs:

SS a b = {ss : Signal a → Signal b
| ∀ (t : Time) (s_1 s_2 : Signal a) .
(∀ t′ 6 t. s_1 t′ ≡ s_2 t′) ⇒ (ss s_1 t ≡ ss s_2 t)}

In an implementation, signal services that depend upon past inputs need to record past information in an internal state[4]. For this reason, they are often called stateful signal functions. Some signal services are such that their output only depends on their input at the current point in time. I refer to these as stateless signal functions, as they require no internal state to be implemented:

SS $_{\text{STATELESS}}$ a b = {ss: Signal a → Signal b
| ∀ (t: Time) (s_1 s2: Signal a) .
(s_1 t ≡ s_2 t) ⇒ (sf s1 t ≡ sf s_2 t)}

The terms linear and combinatorial are also used for the same notions as stateful and stateless, respectively. This in turn, is one step towards making it easier to implement PRP faithfully and allowing programmers to reason in terms of the ideal semantics with greater confidence.

III. THE NEW CONCEPTUAL MODEL

3.1 Indicator Descriptors and Signal Vectors

To address the limitations of Yampa, the notions of a signal vector, a heterogeneous vector of signals, and redefine the conceptual notion of signal function to be a function on signal vectors. Two distinct kinds of signals: continuous-time signals, defined as before; and discrete-time, or event, signals, which are only defined at countable many points in time. Each point at which an event signal is defined is known as an event occurrence. The crucial point is that we define these notions of different kinds of signals, and vectors of such signals, only as an integral part of the signal function abstraction: they have no independent existence of their own and are thus completely internalized at the reactive level. This means that the PRP implementer has great freedom in choosing representations and exploiting those choices.

A signal descriptor is a type that describes key characteristics of a signal. Signal descriptors only exist at the type-level: there are no values having such types; in particular, a signal descriptor is not the (abstract) type of any signal.

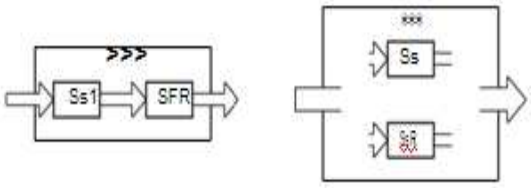

Fig. 1 The Linear (≫) and Parallel (*) Composition Combinators**

DATA Sig Desc : Set WHERE
E : Set → SigDesc -- discrete-time signals (events)
C : Set → SigDesc -- continuous-time signals
SVDesc : Set
SVDesc = List SigDesc

For the purpose of stating the new conceptual definition of signal functions, and for use in semantic definitions later and postulated a function (SVRep) that maps a signal vector descriptor to some suitable type for representing a *sample* of signal vectors of that description, and use this to define signal vectors:

SV Rep: SV Desc → Set
Sig Vec: SV Desc → Set
SigVec as ≈ Time → SVRep as

3.2 Example for Combinators and Primitives

To demonstrate the new conceptual model, we define some common primitive signal service sand combinators from Yampa. These primitives either operate at the reactive level, or mediate between the functional and reactive levels.

3.2.1 Linear and Parallel Composition

Signal services can be composed linearly (**≫**) or in parallel (**⁂**)

_ _ >>: {as bs cs between: SVDesc} →
SSas bs → SSbs cs → SSas cs

_ _ ***: {as bs cs ds : SVDesc } →
SSas cs → SSbs ds → SS(as ++ bs) (cs ++ ds)

Fig. 2 The Feedback Combinator (loop)

_&&& : {as bs cs: SVDesc} →
Ss as bs → Ss as cs → Ss as (bs ++ cs)

3.2.2 Switches

Signal function networks are made dynamic through the use of *switches*. Basic switches have the following type:

switch : ∀ {as bs } → {e : Set } →
SSas (E e :: bs) → (e → SSas bs) → SSas bs

dswitch : ∀ {as bs } → {e : Set } →
SSas (E e :: bs) → (e → Seas bs) → SSas bs

The behavior of a switch is to run the subordinate signal function, emitting all but the head (the event) of the output vector as the overall output. When there is an event occurrence in the event signal, the value of that signal is fed into the function (the second explicit argument) to generate a residual signal function. The entire switch is then removed from the network and replaced with this residual signal function[5]. The difference between a switch and a dswitch (decoupled switch) is whether, at the moment of switching, the overall output is the output from the residual signal function (switch), or the output from the subordinate signal function. A key point regarding switches is that the residual signal function does not start "running" until it is applied to the input signal at the moment of switching. Consequently, rather than having a single global Time, each signal function has its own local time, the time since this signal function was applied to its input signal.

3.2.3 Blocks

The loop primitive provides the means for introducing feedback loops into signal function networks. A loop consists of two signal functions: a subordinate signal function and a feedback signal function. The input of the feedback signal function is a suffix of the output of the subordinate signal function, and the output of the feedback signal function is a suffix of the input to the subordinate signal function:

Loop: ∀ {as bs cs ds } →
SS (as ++ cs) (bs ++ ds) → SSds cs → SSas bs

3.2.4 Ancient Signal Functions

In pure services to the reactive level using the primitives pure and pure E . Such lifted signal services are always stateless:

pure : {a b : Set } → (a → b) → SF [C a] [C b] pureE : {a b : Set } → (a → b) → SF [E a] [E b]

Lift values to the reactive level using the primitive constant. This creates a signal function with a constant, continuous-time, output:

constant : ∀ {as } → {b : Set } → b → SSas [C b]

Events can only be generated and accessed by event processing primitives.
edge : SS[C Bool] [E Unit]
hold : {a : Set } → a → SF [E a] [C a] never : ∀ {as } → {b : Set } → SF as [E b] now : ∀
{as } → SF as [E Unit]

The primitive pre conceptually introduces an infinitesimal delay: pre : ∀ {a } → SF [C a] [C a]

To make this precise, the ideal semantics of pre is that it outputs whatever its input was immediately prior to the current time; that is, the left limit of the input signal at all points:

∀ (t: Time^{+}) (s : Signal a) . pre s

$$t = \lim_{t' \to T} s\, t'$$

Here, Time^{+} denotes positive time. Consequently, at any given point, the output of pre does not depend upon its present input, which is the crucial property. The primitive pre is usually implemented as a delay of one time step. The left limit at any point of a discrete-time signal is undefined Disallowing pre on events thus eliminates a potential source of programming bugs.
Initialize combinator that defines a signal function's output at time$_0$:

initialize : ∀ {as b } → b → SSas [C b] → SSas [C b]

IV. DECOUPLED INDICATION SERVICES

The loop combinator allows feedback to be introduced into a network. This is an essential capability, as feedback is widely used in reactive programming. However, feedback must not cause deadlock due to a signal service depending on its own output in an unproductive manner. Signal services that can be used safely in feedback loops, and index the type of a signal service by whether or not it is decoupled.
Decoupled Signal Service: A signal service is decoupled if, at any given time, its output can depend upon its past inputs, but not its present and future inputs:

SF $_{DEC}$ as bs = {ss: SSas bs
| ∀ (t : Time) (sv $_1$ vs. $_2$: SigVec as) . (∀ t ‹ sv. $_1$ t ≡ sv $_2$ t)
□ (sssv $_1$ t ≡ sssv $_2$ t)}

Decoupled Cycle: A cycle is decoupled if it passes through a decoupled signal service.
Instantaneous Cycle (Algebraic Loop): A cycle is instantaneous if it does not pass through a decoupled signal service.

Many reactive languages deal with this problem by requiring a specific decoupling construct to appear syntactically within the definition of any feedback loops. This works in a first order setting, but becomes very restrictive in a higher order setting as decoupled signal services cannot be taken as parameters and used to decouple loops. Our solution is to encode decoupledness information in the types of signal services. This allows us to statically ensure that a Ill-typed program does not contain any instantaneous cycles. Furthermore, the decoupledness of a signal service will be visible in its type signature, providing guidance to an FRP programmer.

4.1 Decoupledness Descriptors

A records of decoupledness descriptors is given below:

DATA Dec: Set WHERE
DEC: **Dec** -- decoupled signal services CAU: **Dec** -- causal signal services

Index **Ss** with a decoupledness descriptor:

$$\frac{(SSs,\ as) \xrightarrow{\delta t} (SSs\ ,\ Event\ e\ ::\ bss) \qquad f\ 7 \to SSr \qquad (SSr\ ,\ as) \Rightarrow^{0} (SSr\ ,\ bsr)}{(dswitch\ SSs\ f,\ as) \xrightarrow{\delta t} (SSr\ ,\ bss)}\ \text{DSW-EV}$$

SVDesc → SVDesc → Dec → Set

enforce that the feedback signal service within a **loop** is decoupled:

```
loop : ∀ {as bs cs ds } → {d : Dec } →
       SS(as ++ cs) (bs ++ ds) d → SSds cs DEC →
       SSas bs d
```

The primitive signal services now need to be retyped to include appropriate decoupledness descriptors:

```
pure : ∀ {a b } → (a → b) → SS[C a ] [C b ] CAU
pureE : ∀ {a b } → (a → b) → SS[E a ] [E b ] CAU
constant : ∀ {as b } → b → SSas [C b ] DEC
edge : SS[C Bool ] [E Unit ] CAU
hold : ∀ {a } → a → SS[E a ] [C a ] CAU
never : ∀ {as b } → SSas [E b ] DEC
now : ∀ {as } → SSas [E Unit ] DEC
pre : ∀ {a } → SS[C a ] [C a ] DEC
initialise : ∀ {as b } → {d : Dec }
             → b → SSas [C b ] d → SSas [C b ] d
```

The primitive combinators compute the decoupledness descriptor of their composite signal service from the descriptors of their components. To do this, I use the join (**∨**) and meet (**∧**) of the decoupledness descriptors (with respect to subtyping):

```
_≫ _ : ∀ {as bs cs d1 d2 } →
       SSas bs d1 → SSbs cs d2 → SSas cs (d1 ∧ d2)
_***_ : ∀ {as bs cs ds d1 d2 } →
       SSas cs d1 → SSbs ds d2 →
       SS(as ++ bs) (cs ++ ds) (d1 ∨ d2)
_&&&_ : ∀ {as bs cs d1 d2 } →
       SSas bs d1 → SSas cs d2 →
       SSas (bs ++ cs) (d1 ∨ d2)
switch : ∀ {as bs e d1 d2 } →
         SSas (E e :: bs) d1 → (e → SSas bs d2) →
         SSas bs (d1 ∨ d2)
dswitch : ∀ {as bs e d1 d2 } →
         SSas (E e :: bs) d1 → (e → SSas bs d2) →
         SSas bs (d1 ∨ d2)
```

Without indexing signal services by their decoupledness, an Agda implementation using this type system would not pass Agda's termination checker.

4.2 Example: Switching Integration Methods

To demonstrate the usefulness of decoupledness descriptors, a small example of how they can be used to allow dynamic switching between several integration signal services is provided. This is inspired by an example from Lucid Synchrone.

Initialisation Descriptors: It allows the type checker to reject any programs where uninitialised signals could cause a run-time error. Event signals are, by definition, only defined at discrete points in time, and thus there is no need to initialise them if they are not defined at $time_0$.

The primitive signal functions that mention continuous-time signals in their types now need to be retyped:

```
pure : ∀ {a b } → {i : Init } →
       (a → b) → SS [C i a ] [C i b ] CAU constant : ∀ {as b } → b → SS as [C INI b ] DEC edge : SS [C INI Bool ] [E Unit ] CAU
hold : ∀ {a } → a → SS [E a ] [C INI a ] CAU
pre : ∀ {a } → SS [C INI a ] [C UNI a ] DEC
initialise : ∀ {as b d } → {i : Init } →
             b → SS as [C i b ] d → SS as [C INI b ] d
```

Initialized signals are subtypes of uninitialized signals (INI **<:** UNI), as they can be coerced by forgetting the value at $time_0$. We extend the weaken primitive to reflect this:

```
weaken : ∀ {as as' bs bs' d d' } →
         as <: as' → bs <: bs' → d <: d' → SS as bs d → SS as' bs' d'
```

4.3 Switching into Uninitialized Signals

For example, consider switch. When the residual signal function is switched in, it could produce uninitialized output at its (local) $time_0$. But this uninitialized signal then escapes, potentially causing a run-time error. In fact, switches are the only place that this can occur, as it is only switches that create sub-networks at a different local time. We resolve this by requiring that all output signals from the residual signal function are

initialized, enforcing this at the type level:

```
initc : SVDesc → SVDesc
initc = map initcAux
  WHERE initcAux : SigDesc → SigDesc
        initcAux (E a)   = E a
        initcAux (C _ a) = C INI a
switch : ∀ {as bs e d1 d2 } →
         SS as (E e :: bs) d1 →
         (e → SS as (initc bs) d2) →
         SS as bs (d1 ∨ d2)
```

We do not require this constraint for decoupled switches, as their output at $time_0$ is defined as being the output from the subordinate signal function. The ($time_0$) output from the residual signal function is discarded, so it does not matter if it is uninitialized. Furthermore, this means that the initialization of the residual signal function's output does not affect the overall initialization of the switch construct. We redefine dswitch to reflect this flexibility:

```
dswitch : ∀ {as bs bs' e d1 d2 } →
          SS as (E e :: bs) d1 →
          (e → SS as bs' d2) → initc bs <: bs →
          SS as bs (d1 ∨ d2)
```

Our example differs in that the integration signal service is being used to decouple a loop, allowing the decision of whether to switch integration services to depend upon the current output.

4.3.1 Recurring Switches

The behavior of a recurring switch is to apply its subordinate signal service to the tail of its input, producing the overall output[3]. Whenever an event (the head of the input) occurs, the signal service carried by that event replaces the subordinate signal function. Recurring switches come in two varieties: like basic switches, they differ in whether the output at the instant of switching is from the new (rswitch) or old (drswitch) subordinate signal service.

```
rswitch  : ∀ {as bs d1 d2 } → SSas bs d1 →
           SS(E (SSas bs d2) :: as) bs CAU
drswitch : ∀ {as bs d1 d2 } → SSas bs d1 →
           SS(E (SSas bs d2) :: as) bs (d1 ∨ d2)
```

4.3.2 Dynamic Integrator

An integrator (Intgr) is a causal signal service that integrates a signal, and a decoupled integrator (dIntgr) is one that is decoupled. I assume integral is an existing decoupled integrator.

```
Input    = Float
Output   = Float
Intgr    = SS[C Input ] [C Output ] CAU
dIntgr   = SS[C Input ] [C Output ] DEC
integral : dIntgr
```

The intgrDecider also receives as input new integrators, along with some decision rules that allow it to determine when they should be used. Now define a dynamic integrator (dynIntgr) that uses a loop to connect intgrDecider with a drswitch containing an initial decoupled integrator:

```
DecisionRules : Set
NewIntgr = dIntgr × DecisionRules
intgrDecider : SF (E NewIntgr :: C Input :: C Output :: [ ]) (C Output :: E dIntgr :: C Input :: [ ])
dynIntgr : SF (E NewIntgr :: C Input :: [ ]) [C Output ] CAU dynIntgr = loop intgrDecider (drswitch integral)
```

V. SAFETY AND SEMANTICS

Agda has completeness and termination checkers, ensuring that Agda programs are total and terminating. Thus our FRP embedding within Agda has these assurances as well[3]. To run signal functions, our prototype implementation operates by running the network iteratively over a discrete sequence of time steps. At each time step, the input is sampled and fed into the network, along with the time delta since the preceding time step. The network then updates any internal state, and produces an output sample. This (one time step) evaluation function is accepted by Agda; therefore the evaluation of each time step is guaranteed to terminate,


```
DATA SS  : SVDesc → SVDesc → Dec → Set WHERE
  prim   : ∀ {as bs State } → (Δt → State → SVRep as → State × SVRep bs)        → State → SS as bs CAU
  dprim  : ∀ {as bs State } → (Δt → State → (SVRep as → State) × SVRep bs) → State → SS as bs DEC
  ≫      : ∀ {as bs cs d1 d2 } → SS as bs d1 → SS bs cs d2                      → SS as cs (d1 ∧ d2)
  ***    : ∀ {as bs cs ds d1 d2 } → SS as cs d1 → SS bs ds d2                   → SS (as ++ bs) (cs ++ ds) (d1 ∨ d2)
  loop   : ∀ {as bs cs ds d } → SS (as ++ cs) (bs ++ ds) d → SS ds cs DEC       → SS as bs d
  switch : ∀ {as bs e d1 d2 } → SS as (E e :: bs) d1 → (e → SS as bs d2)        → SS as bs (d1 ∨ d2)
  dswitch : ∀ {as bs e d1 d2 } → SS as (E e :: bs) d1 → (e → SS as bs d2)       → SS as bs (d1 ∨ d2)
```

producing output. Although execution of an FRP program is usu-ally modelled as non-terminating (an infinite sequence of steps), we can nevertheless guarantee that they are *productive* because each individual step is productive.

5.1 Model Implementation

Signal functions are represented internally as records, of which the constructors are five primitive combinators and two primitive signal functions. We will call these the core primitives. The two core signal functions are prim and dprim. The internal structures of these two primitives reflect the properties we require them to have. A causal signal function must be able to produce output at the current time, provided it has access to all past and present inputs. We realize this by giving prim an internal state, and a function that maps the time delta, state and input to an updated state and output[4]. A decoupled signal function must be able to produce output at the current time, provided it has access to all past inputs. Thus its internal function requires only the time delta and state to produce an output. In order to manage updating the state, it also produces a function mapping input to an updated state. The key point here is that while the input will be required to fully evaluate the signal function at the current time step, the output can be produced before that input is provided.

5.2 Semantics of Uninitialized Signals

The semantics given here omit any mention of signal initialization. This is because these semantics only apply to time steps after the initialization step (at $time_0$). when our evaluation rules are used with a zero time delta (as in SW-EVENT), we should use the $time_0$ semantics.

VI. CONCLUSION

A domain specific type system for PRP is a type of system ensures that signal function networks with feedback loops and uninitialized signal will be productive without support for high order dataflow and structural dynamism. This type of system also makes differentiation between the Pragmatic and reactive levels of an PRP program. It implemented a prototype interpreter for our program in Agda by providing simplified version of the operational semantics of the interpreter.

REFERENCES

[1] The Agda mailing list, 2012. https://lists.chalmers.se/mailman/listinfo/agda.

[2] The Agda wiki, 2012. http://www.cs.chalmers.se/~ulfn/Agda.

[3] P. Martin-L¨of. Intuitionistic Type Theory. Bibliopolis Napoli, 1984.

[4] C. McBride and J. McKinna. The view from the left. Journal of Functional Programming, 14(1):69–111, January 2004.

[5] U. Norell and J. Chapman. Dependently Typed Programming in Agda (sourcecode), 2012. http://www.cse.chalmers.se/~ulfn/darcs/AFP08/LectureNotes

Author Bibliography

Mr. Kaja Masthan, working as Assistant Professor in the Department of computer science and Engineering and having 7years of teaching experience. He has done M.C.A and M.Tech in computer science and Engineering

