Efficient Data-Types Analysis for a Functional Concurrent Model of Programming

Mohamed A. El-Zawawy
College of Computer and Information Sciences, Al Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh, Kingdom of Saudi Arabia;
Department of Mathematics, Faculty of Science, Cairo University, Giza 12613, Egypt

Mohammad N. Alanazi
College of Computer and Information Sciences, Al Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh, Kingdom of Saudi Arabia

Summary
Asynchronous tasks in programming are those tasks executed free of context of the main task. Therefore asynchronous tasks are methods implemented in a non-blocking style, permitting the main method to continue running. Functional programming is a programming style to express the hierarchy and components of computer code. In this style calculations are treated similarly to treating computations of functions in mathematics.
Hence memory-states and modifiable data structures are not needed. Functional programming can be introduced as a declarative style of coding in the sense that expressions replace programs.
On a functional object-oriented model, this paper presents an accurate type system for asynchronous operations. The job of the type system is to stop undefined functions from execution and hence from aborting programs. In other words, the type system ensures soundness of data types and hence avoiding static errors like field-not-defined and method-not-defined from occurring at execution time. The paper introduces as well a programming example for the proposed system.
Key words:
Data-Types Analysis, Type System, Functional Concurrent Models, Syntax, Asynchronous Programming.

1. Introduction

Combination of event-based interactions and threads is necessary for most of concurrent programs implementing critical applications. Such programs include various threads which do interactions via posting jobs to each other. This posting has the form of asynchronous call of functions. Function items are used to execute the asynchronous calls to functions. Each function item is mainly a container of a reference to the method that is to be executed on a specific thread using the convenient inputs. Typically, the inputs as well include function items which act as callbacks. Reasoning about complicated parallel structures including function references and callback methods is part of the verification of these programs. This makes the verification process a very tricky one, notably in the existence of recursion.
Programming languages that are functional use no assignment commands, no variables, and no iterative structures. This architecture is inspired by the view of mathematical functions which use segregation of different cases in their definitions. Typically each of these cases is defined separately by using (recursive) applications of functions. In programming languages that are functional, these definitions are expressed almost straightly into the language syntax. Therefore, the complete program is just a function. This function, in turn is defined using more functions.
Models of asynchronous programming languages that are functional are quite important as they combine the advantages of asynchronous programming and that of functional programming. However verifying programs produced by such models is not an easy job as they are very involved. One of the most important verification issues for programming languages in general is that of type compatibility. This verification aims at verifying the correctness of type uses in the programs. On a powerful model for asynchronous programming that are functional and object-oriented programming model, this paper studies the problem of types verifications. The paper presents an accurate type system to verify programs produced by the studied model. The system consists of set of types (defined in the language) and a set of inference rules (built using the language constructs). The paper also shows in details a motivating example of the research behind the paper.

Contributions
Contributions of the paper are the following.
1) A new type system for asynchronous programming using functional and object-oriented model of programming.
2) A detailed motivation example of research presented in this paper.

Paper Outline
The content in the remaining sections is as follows. Section II shows in details a motivating program-example of the research. Section III presents the proposed type system. The related work is reviewed in Section IV that also suggests future-work directions. Section V (the final section) summaries the paper.

Manuscript received February 5, 2016
Manuscript revised February 20, 2016

```
1  class c: Post class{
2          int f1,
3          int f2,
4          int f3;
5          int proj1   (int x1, intx2, intx3)
6                      {
7                      return x1;
8                      }
9          int proj2   (int x1, intx2, intx3)
10                     {
11                     return x2;
12                     }
13         int proj3   (int x1, intx2, intx3)
14                     {
15                     return x3;
16                     }
17         }
18 (c, {}, Post(new c(1,2,void).proj1)).
```

Fig 1: A motivating Example.

$$
\begin{aligned}
&m \in Me = \textit{The set of all method names.}\\
&l \in L = \textit{The set of all memory locations.}\\
&c \in C = \textit{The set of all class names.}\\
&f \in F = \textit{The set of all names of class fields.}\\
&v \in V = L \cup \textit{Integers.}\\
&t \in Types ::= C \cup L \cup \{int, void\}.\\
&i \in Inst ::= v \mid x \mid e.f \mid new\ c(e^*) \mid skip \mid new\ e \mid if\ e\ then\ e_t else\ e_f\\
&\qquad \mid while\ e\ do\ S \mid call\ e.m(e^*) \mid return\ e.\\
&mc \in PostInst ::= Post\ e.m(e^*) \mid RemovePosted\ e.m(e^*)\\
&\qquad \mid RunPostedNow\ e.m(e^*) \mid DelayPosted\ e.m(e^*).\\
&e \in Exp ::= i \mid mc.\\
&M \in Methods ::= t\ m((tx)^*)\{return\ e;\}.\\
&PostClass ::= class\ Post.app.activity\{c\ root; int\ result; int\ fin; \ldots; M^*\}.\\
&Post\ activity \in Classes ::= classc: Post\ class\{(t\ f)^*: M^*\}.\\
&P \in Programs ::= (Post\ activity\ T, Method\ T, e).
\end{aligned}
$$

Fig 2: Asynch-OP: A Robust Framework for asynchronies Operations on a Functional Object-Oriented Model.

2. Motivating Example

Figure 1 presents a motivating example of our research. The program is built using the syntax of Figure 2. The program consists of a class that is defined in the lines 1 17. The class consists of three integer variables ”x1, x2, and x3” and three projection methods ”proj1, proj2, and proj3”. The main program is in line 18 which consists of an activity table of defined classes, a method table (empty in this example), and a main expression, ”Post (new c(1,2,void).proj1)”. This last expression is the main program as our model is functional.

We note that the main expression posts a method of an object that is created with three inputs. However the type of the third input is not correct. A type system to detect such errors in our model is necessary. Building such system is the main motivation for the research of this paper.

3. Syntax and Type system

The studied model for asynchronous programming in an object-oriented and functional style is presented in Figure 2. The syntax of the figure uses the asterisks to express sequences. Therefore for example the potentially empty sequence $e_1;\ldots;e_n$ is denote by e^* and also the potentially empty sequence $t_1\ f_1;\ldots;\ t_n\ f_n$ is denoted by $(t\ f)^*$. The syntax uses semicolons and commas to express concatenations. In the context of the syntax, it is assumed that names of parameters, arrangements of variables declarations, definitions of methods, and method names do not include repeated names. The syntax is based on a main class named ”Post class” which hosts required information about each posting action. These information includes:

1) the name of the method that posted the concerned method: root,

2) the result of the posted method: result, and

3) an indicator of if the posted method is finished: finished.

$$
t \prec t \quad (s_1^t) \qquad\qquad c \prec Post\ class \quad (s_2^t)
$$

$$
\frac{c_1 \prec c_2 \quad c_2 \prec c_3}{c_1 \prec c_3} \quad (s_3^t)
$$

$$
\frac{\begin{array}{c} class\ c_1: Post\ class\{(t_1\ \ f_1)^*: M_1^*\} \quad class\ c_1: Post\ class\{(t_2\ \ f_2)^*: M_2^*\} \\ (t_1\ \ f_1)^* \subseteq (t_2\ \ f_2)^* \qquad M_1^* \subseteq M_2^* \end{array}}{c_1 \prec c_2} \quad (s_4^t)
$$

Fig 3. Sub-typing relation for Asynch-OP.

$$
\frac{class\ c_1: Post\ class\{(t_1\ \ f_1)^*: M_1^*\} \quad \{t\ \ m((t\ \ x)^*)\{return\ e;\}\} \subseteq M^*}{\vDash m: (t^* \to t)} \quad (m_1^t)
$$

$$
\frac{class\ Post.app.activity\ \{c\ root; int\ fin; \ldots; M^*\} \quad \{t\ \ m((t\ \ x)^*)\{return\ e;\}\} \subseteq M^*}{\vDash m: (t^* \to t)} \quad (m_2^t)
$$

Fig 4. Methods Typing rules for Asynch-OP.

In the proposed syntax, every class is derived from the ”Post class”. A class, *c*, definition in the syntax is composed of the class name, the name ”Post class” of its superclass, field presentations $(t\ f)^*$, and a group of presentations for the class methods M^*.

The language syntax has two sorts of expressions: typical and posting expressions . The command ”Post” represents the classical posting command. The syntax provides a set of advanced posting commands that are:

- DelayPosted $e{:}m(e^*)$: the command delaying a method that is posted but not done yet.
- RunPostedNow $e{:}m(e^*)$: the command rushing up the run of a method that is posted but not executed yet.
- RemovePosted $e{:}m(e^*)$: the command removes a previously posted method that is not required any more.

Figures 3 and 4 present sub-typing relation of the types presented in the language syntax and an algorithm to determine types of syntax methods. Figure 5 presents the typing rules of our propped type system. Typing judgments have the following form.

$$P, \Delta \vDash e: t$$

In this form P denoted the set of posted methods and Δ denotes the typing context (which assigns a type to each variable in the program) of assigning the type t to the expression e.

4. Related and Future Work

This section reviews most related work to the work presented in this paper and presents directions for future work. An active area of research is the study of sequential verifications for asynchronous program analysis. The verification presented in [1] was among the first tries in this direction. In [1], a source-code-to-source-code translation from parallel programs into sequential and equivalent ones was presented. This technique approximates the original-program behaviors. A better approach was presented in [2] where the source-code to source-code translation calculated a bound approximation of context switches for any arbitrary context-bound. Moreover, [2] provided a technique for predicting context-switches values with constrains for later use at a convenient program point during the program run. However, a main drawback of [2] is that unreachable control configurations in the original asynchronous program may be considered in the equivalent sequential form.

The work in [3] treated this drawback via repeated execution to the configurations at which predicted values are supposed to be used. The last two techniques were compared in [4] using the testing-specification-verification style, rather than in the model-checking style. In the earlier style, advantages of a lazy approach are not obvious because the technique using it overcomes the lazy approach. The work in [5] presented a transformation obtaining sequential programs for synchronous ones equipped with schedulers that are priority-preemptive. However the transformation is based on a bound for the tasks count. All the techniques of sequentializations reviewed above do not consider runtime taskinitiation. However, [6] proposed a sequentialization technique for a parameterized design-testing approach [7] with no bound on tasks number.

One of the related research direction is the compositional transformations from asynchronous programs to sequential ones [8]. This process is called sequentialization. An example of sequentialization is in [1] that studied multi-threaded programs with at most a single context-switch among threads. This work was later extended to treat a parameterized number of context-switches among a fixed (statically) group of threads that run in a specific order (round-robin) [2]. Later [3] presented another transformation that focused on model-testing sequential programs resulted from transformation. This technique was later generalized to treat parameterized programs with statically-fixed threads of an unconstrained number [7].

$$P, \Delta \vDash t \ (v^t) \qquad \frac{\vDash m: (t^* \to t) \quad P, \Delta \vDash e: l \quad \forall i.\ P, \Delta \vDash e_i: t_i}{P, \Delta \vDash Post\ e.m(e^*): void} \ (Post_1^t)$$

$$\frac{\begin{array}{c}\vDash m: (t^* \to t) \quad P, \Delta \vDash v: l \quad \forall i.\ P, \Delta \vDash v_i: t_i \\ T(v, m) = e \quad P' = [P \mid (m, e.v^*)] \quad P', \Delta \vDash e: t\end{array}}{P', \Delta \vDash Post\ v.m(v^*): void} \ (Post_2^t)$$

$$\frac{P, \Delta \vDash e: void \quad P = [P' \mid (m, e.v^*)] \quad P', \Delta \vDash call\ e.m(v^*): t}{P', \Delta \vDash Post\ v.m(v^*): void} \ (Pick^t)$$

$$\frac{\vDash m: (t^* \to t) \quad P, \Delta \vDash e: l \quad \forall i\ P, \Delta \vDash e_i: t_i}{P, \Delta \vDash RemovePosted\ e.m(e^*): void} (RPost_1^t)$$

$$\frac{\begin{array}{c}\vDash m: (t^* \to t) \quad P, \Delta \vDash v: l \quad \forall i.\ P, \Delta \vDash v_i: t_i \\ T(v, m) = e \quad P' = [P \mid (m, e.v^*)] \quad P', \Delta \vDash e: t\end{array}}{P', \Delta \vDash RemovePosted\ v.m(v^*): void} (RPost_2^t)$$

$$\frac{\vDash m: (t^* \to t) \quad P, \Delta \vDash e: l \quad \forall i\ P, \Delta \vDash e_i: t_i}{P, \Delta \vDash RunPostedNow\ e.m(e^*): void} (RunPostN_1^t)$$

$$\frac{\begin{array}{c}\vDash m: (t^* \to t) \quad P, \Delta \vDash v: l \quad \forall i.\ P, \Delta \vDash v_i: t_i \\ T(v, m) = e \quad P' = [P \mid (m, e.v^*)] \quad P', \Delta \vDash e: t\end{array}}{P', \Delta \vDash RunPostedNow\ v.m(v^*): t} (RunPostN_2^t)$$

$$\frac{\vDash m: (t^* \to t) \quad P, \Delta \vDash e: l \quad \forall i\ P, \Delta \vDash e_i: t_i}{P, \Delta \vDash DelayPosted\ e.m(e^*): void} (DelayPost_1^t)$$

$$\frac{\begin{array}{c}\vDash m: (t^* \to t) \quad P, \Delta \vDash v: l \quad \forall i.\ P, \Delta \vDash v_i: t_i \\ T(v, m) = e \quad P' = [P \mid (m, e.v^*)] \quad P', \Delta \vDash e: t\end{array}}{P', \Delta \vDash DelayPosted\ v.m(v^*): void} (DelayPost_2^t)$$

$$\frac{(x, t) \in \Delta}{P, \Delta \vDash x: t} (v^t) \quad \frac{P, \Delta \vDash e: int \quad P, \Delta \vDash e_t: t}{P, \Delta \vDash while\ e\ do\ e_t: t} \ (while^t)$$

$$\frac{P, \Delta \vDash e: t}{P, \Delta \vDash return\ e: t} (return^t) \quad \frac{P, \Delta \vDash e: c \quad (t\ f)^* \in c}{P, \Delta \vDash e.f_i: t_i} \ (f^t)$$

$$P, \Delta \vDash skip: void \ (skip^t) \quad \frac{P, \Delta \vDash e: int \quad P, \Delta \vDash e_t, e_f: t}{P, \Delta \vDash if\ e\ then\ e_t else\ e_f: t} \ (if^t)$$

$$\frac{\vDash m: (t^* \to t) \quad P, \Delta \vDash e: l \quad \forall i\ P, \Delta \vDash e_i: t_i}{P, \Delta \vDash call\ e.m(e^*): void} (call_1^t)$$

$$\frac{\begin{array}{c}\vDash m: (t^* \to t) \quad P, \Delta \vDash v: l \quad \forall i.\ P, \Delta \vDash v_i: t_i \\ T(v, m) = e \quad P, \Delta \vDash e: t\end{array}}{P', \Delta \vDash call\ v.m(v^*): t} (call_2^t)$$

$$\frac{P, \Delta \vDash e: t}{P, \Delta \vDash new\ e: t} (new_1^t) \quad \frac{P, \Delta \vDash e: c \quad (t\ f)^* \in c \quad \forall i.\ P, \Delta \vDash e_i: t_i}{P, \Delta \vDash new\ c(e^*): c} \ (new_2^t)$$

Fig 5. Expression Typing rules for Asynch-OP.

This last result was even extended more in [9] to treat unconstrained number of tasks dynamically-established. This made the technique applicable both to multi-threaded and event-based asynchronous programs [10], [11], [12]. Yet in the same direction is [13] that presented a sequentialization which studied as many properties as possible according to a given set of constraints. The last reviewed two techniques of sequentializations can be applied to asynchronous programs with dynamic establishment of an unconstrained number of tasks. However they do not consider priorities of task executions nor many buffers of tasks. In [5] priority-style sequentialization was presented. However reduction in [5] is based on a fixed number of tasks (statically-fixed) and does not consider many buffers of tasks.

Many tries have been attempted to augment C and Java with synchronous concurrency structures. Reactive C [14] is one of such augmentations that uses the definitions of preemptions and ticks. However it does not enable real concurrency. FairThreads [15] is another augmentation that uses native threads. Synchronous C [16] and Precision Timed C (PRET-C) [17] are equipped with libraries for expressing threads of synchronous concurrent. Synchronous C as well allows runtime scheduling for threads which makes it convenient many synchronous program analyses. Using CCSscheduling communication [18] and and exception handlers, SHIM [19], one more C-augmentation, implements concurrent Kahn network systems. SHIM followed spirit of synchronous languages. However it does not employe the classical models synchronous programming, Rater than that it is based on using synchronisation channels for communications. All languages including signals are black boxes that do not disjoin updates and initialisations.

Signal methods can be included in Elm [20] gathering the advantages of deterministic AFRP [21], [22], [23], [24], [25] to Elm users permitting programs to install, on the run, graphical structures, instead of using signals on signals. Elms deterministic meanings make using concurrency and asynchrony an obvious process. This was believed to be very complex in a running AFRP. Processing of concurrent signals are possible using parallel FRP [26], such as Elm [20].

In Parallel FRP, events of a signal are ordered in such a way that permitting events to be executed randomly. In an extreme case, this amounts to the order of processing requests is not the order of their arrival. Therefore, parallel computing is possible and results in responses to be returned immediately. It is not convenient to achieve this intra-signal asynchrony in a GUI setting because it would main tasks to be executed out of order. Alternatively, Elm [20] allows asynchrony of inter signal via removing the events order among various signals. It is believed that that asynchrony of inter- and intra-signal are consistent. However in GUI programming, it is more proper to focus on asynchrony of inter-signal.

Trying to eliminate repeated computations motivated selfadapting computations [27], [28], [29]. The benefits of removing unnecessary repeated computations, as clear in FElms signal evaluation using pipelines, improved performance and guaranteed correctness. FElm 10 prevented some unnecessary recompilation. However it permits propagation of various messages through graph of the signal. Ideas from self adapting computation are usable to eliminate such messages and to boost the performance. Of course, it is likely that improved accesses [27] – employed in self-adapting calculations to express values that may adapt and thus initiate recalculations are usable to encrypt signals, and to represent asynchronous signals [30]. For future work, it is interesting to study different and important static analyses (like pointer analysis) of classical programming models on the model studied in this article.

5. Summary

This paper presented a precise type system for asynchronous operations, on a functional object-oriented model. Stopping undefined functions from execution (hence from aborting programs) is the main job of the type system. Therefore, the type system guarantees correctness of data types. Hence the type system as well prevents static errors like field-not-defined and method-not-defined from occurring at execution time. The paper introduced also a programming example for the importance of the proposed system.

Acknowledgement

The first author is affiliated with Department of Mathematics, Faculty of Science, Cairo University, Giza 12613, Egypt. He is now on a sabbatical leave to College of Computer and Information Sciences, Al Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh, Kingdom of Saudi Arabia. The authors acknowledge the support of Al Imam University.

References

[1] S. Qadeer and D. Wu, "KISS: keep it simple and sequential," in Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004, 2004, pp. 14–24.

[2] A. Lal and T. W. Reps, "Reducing concurrent analysis under a context bound to sequential analysis," Formal Methods in System Design, vol. 35, no. 1, pp. 73–97, 2009.

[3] S. La Torre, P. Madhusudan, and G. Parlato, "Reducing contextbounded concurrent reachability to sequential reachability," in Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings, 2009, pp. 477–492.

[4] N. Ghafari, A. J. Hu, and Z. Rakamaric, "Context-bounded translations for concurrent software: An empirical evaluation," in Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings, 2010, pp. 227–244.

[5] N. Kidd, S. Jagannathan, and J. Vitek, "One stack to run them all - reducing concurrent analysis to sequential analysis under priority scheduling," in Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings, 2010, pp. 245–261.

[6] S. La Torre, P. Madhusudan, and G. Parlato, "Sequentializing parameterized programs," in Proceedings Fourth Workshop on Foundations of Interface Technologies,

FIT 2012, Tallinn, Estonia, 25th March 2012., 2012, pp. 34–47.
[7] ——, “Model-checking parameterized concurrent programs using linear interfaces,” in Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings, 2010, pp. 629–644.
[8] M. Emmi, A. Lal, and S. Qadeer, “Asynchronous programs with prioritized task-buffers,” in 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE’12, Cary, NC, USA - November 11 - 16, 2012, 2012, p. 48.
[9] M. Emmi, S. Qadeer, and Z. Rakamaric, “Delay-bounded scheduling,” in Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, 2011, pp. 411–422.
[10] P. Ganty and R. Majumdar, “Algorithmic verification of asynchronous programs,” ACM Trans. Program. Lang. Syst., vol. 34, no. 1, p. 6, 2012.
[11] R. Jhala and R. Majumdar, “Interprocedural analysis of asynchronous programs,” in Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007, 2007, pp. 339–350.
[12] K. Sen and M. Viswanathan, “Model checking multithreaded programs with asynchronous atomic methods,” in Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings, 2006, pp. 300–314.
[13] A. Bouajjani, M. Emmi, and G. Parlato, “On sequentializing concurrent programs,” in Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings, 2011, pp. 129–145.
[14] F. Boussinot, “Reactive C: an extension of C to program reactive systems,” Softw., Pract. Exper., vol. 21, no. 4, pp. 401–428, 1991.
[15] ——, “Fairthreads: mixing cooperative and preemptive threads in C,” Concurrency and Computation: Practice and Experience, vol. 18, no. 5, pp. 445–469, 2006.
[16] R. von Hanxleden, “Synccharts in C: a proposal for light-weight, deterministic concurrency,” in Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, 2009, 2009, pp. 225–234.
[17] S. Andalam, P. S. Roop, and A. Girault, “Deterministic, predictable and light-weight multithreading using PRET-C,” in Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010, 2010, pp. 1653–1656.
[18] P. Noce, “Noninterference security in communicating sequential processes,” Archive of Formal Proofs, vol. 2014, 2014.
[19] O. Tardieu and S. A. Edwards, “Scheduling-independent threads and exceptions in SHIM,” in Proceedings of the 6th ACM & IEEE International conference on Embedded software, EMSOFT 2006, October 22-25, 2006, Seoul, Korea, 2006, pp. 142–151.
[20] E. Czaplicki and S. Chong, “Asynchronous functional reactive programming for guis,” in ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, June 16-19, 2013, 2013, pp. 411–422.
[21] H. Liu and P. Hudak, “Plugging a space leak with an arrow,” Electr. Notes Theor. Comput. Sci., vol. 193, pp. 29–45, 2007.
[22] H. Liu, E. Cheng, and P. Hudak, “Causal commutative arrows and their optimization,” in Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009, 2009, pp. 35–46.
[23] H. Nilsson, “Dynamic optimization for functional reactive programming using generalized algebraic data types,” in Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP2005, Tallinn, Estonia, September 26-28, 2005, 2005, pp. 54–65.
[24] W. Jeltsch, “Categorical semantics for functional reactive programming with temporal recursion and corecursion,” in Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP 2014, Grenoble, France, 12 April 2014., 2014, pp. 127–142.
[25] N. Sculthorpe and H. Nilsson, “Safe functional reactive programming through dependent types,” in Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009, 2009, pp. 23–34.
[26] J. Peterson, V. Trifonov, and A. Serjantov, “Parallel functional reactive programming,” in Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000, Boston, MA, USA, January 2000, Proceedings, 2000, pp. 16–31.
[27] U. A. Acar, G. E. Blelloch, and R. Harper, “Adaptive functional programming,” in Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002, 2002, pp. 247–259.
[28] U. A. Acar, A. Ahmed, and M. Blume, “Imperative self-adjusting computation,” in Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008, 2008, pp. 309–322.
[29] A. J. Demers, T. W. Reps, and T. Teitelbaum, “Incremental evaluation for attribute grammars with application to syntax-directed editors,” in Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, Williamsburg, Virginia, USA, January 1981, 1981, pp. 105–116.
[30] U. A. Acar, G. E. Blelloch, K. Tangwongsan, and D. T¨urkoglu, “Robust kinetic convex hulls in 3d,” in Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings, 2008, pp. 29–40.