International Scholarly Research Network
ISRN Electronics
Volume 2012, Article ID 271836, 11 pages
doi:10.5402/2012/271836

Review Article

A Short Historical Survey of Functional Hardware Languages

Gang Chen

Lingcore Laboratory, 2721 Grand Oaks Loop, Cedar Park, TX 78613, USA

Correspondence should be addressed to Gang Chen, gang.chen@lingcore.com

Received 24 January 2012; Accepted 11 February 2012

Academic Editors: S.-F. Hsiao and A. Mercha

Functional programming languages offer a high degree of abstractions and clean semantics, which are desirable for hardware descriptions. This short historical survey is about functional languages specifically created for hardware design and verification. It also includes those hardware languages or formalisms which are strongly influenced by functional programming style.

1. Introduction

Software programmers have been benefited from functional languages in many aspects. A functional program is typically more concise than an equivalent imperative style program, which results in an improvement of programming productivity. Secondly, functional languages have good mathematical properties and are amenable for program transformations and formal program verifications. Thirdly, modern functional languages have better type systems which provide enhanced reliability. All these nice features may apply to the world of hardware design. In fact, the history of functional hardware languages is almost as old as that of functional software languages, albeit the former receives less attention than the latter.

The history of functional hardware languages (FHLs) might be roughly divided into four (somewhat overlapped) periods. The pioneering work of John Lee [1] in earlier 1970s marked the beginning of formal hardware specification. Experimental functional hardware languages (FHL) appeared around 1980, mostly designed for simulation, formal verification, and netlist generation. Among them the μFP [2] language of Mary Sheeran started a longest continuous research in the FHL field. With the widespread adoption of VHDL and Verilog in industry, many functional languages developed since 1990 were equipped with VHDL/Verilog translators. In particular, the notion of multiple interpretation was invented in Hydra [3] and Lava [4]. Before 2000, with few exceptions, most functional hardware languages were academic works. However, significant changes are happening in recent years, as exemplified by the rule-based language BLUESPEC of Bluespec Inc. and the interface language *reFL*^{ect} [5] in the Forte verification environment in INTEL.

Works in these four periods are briefly described in the next four sections, followed by a summary section.

2. 1970s–1980s: Hardware Formalization

Functional languages can hardly be separated from formal hardware specification and verification. The latter is a big topic and there are already many surveys, for example [6]. This section discusses a few earlier researches on functional style formal hardware descriptions.

John Lee. The idea of using formal notation to describe hardware can be traced as early as John Lee's book *Computer Semantics* [1] published in 1972. In his book, John Lee developed a formal definition system, which is closely related to the Vienna Definition Language [7]. This formal system is intended to describe algorithms, languages, and processors within a single formal framework. As an illustration of the power of the system, formal definitions of a sorting algorithm, BASIC language, and the PDP-8 machine are presented.

The core of the formal system is a data structure called *data set*, which is flexible enough to represent records, lists, and other data structures. This "data set" structure can be viewed as a generalization of association list where the second

element of each pair is itself a "data set." Although the system does not contain a type system in the sense of modern type theory, its notion of "predicate" partially plays the role of types. However, functions in his system are not "typed"; the predicates are essentially "recognizers" of classes of objects. This data set and predicate system combine different data types into a single structure without losing "typability." John Lee's work was perhaps the first formal hardware description in the world, but no software tools were developed to process the descriptions.

Raymond Boute. During the 1980s, Raymond Boute investigated another functional style digital system specification [8]. He used first-order bit-string functions to describe circuits. Each function definition is associated with a set-theoretical type. Types were introduced not only to enhance the readability of function specifications but also to assist the statements of mathematical properties. The focus was the development of formally verified bit-string functions using transformational proofs. To assist the formal proof, commuting diagrams relating bit-string and numeric functions were established through the denotation/representation relationship.

R. E. Frankel and S. W. Smoliar. Frankel and Smoliar [9] used abstract data types to make high-level processor specifications. In their formalization, the functions correspond to basic modules of combinational logic, and the functionals describe the interconnections. In their processor modeling, cpu internal state is represented by the abstract data type PROCESSOR. Individual state components are accessed by functions mapping PROCESSOR to bitstrings. State transitions are realized by an EXECUTE function of type PROCESSOR → PROCESSOR, which interprets the instructions. Semantics of sequential circuits is characterized by a functional which maps state transition function and initial state into state sequence [10].

These works are formal hardware specifications intended to give precise descriptions for hardware. They illustrated how basic circuits as well as prototype microprocessors can be described in functional language style. Another intention is to reason about properties of circuits. These specifications are strongly influenced by functional languages. But they are not "executable," that is, do not equipped with a simulator.

3. 1980s–1990s: Simulation and Layout

Many functional hardware languages were created during 1980s. Most of them support simulation.

Sticks and Stones. Cardelli and Plotkin [11, 12] designed an ML-based language called Sticks&Stones. It is essentially a picture-drawing language and perhaps the earliest applicative language created for hardware layout. Given a textural input written in the language, the implemented tool can draw a picture on a graphical terminal. Expressions in Sticks&Stones denote pictures. More precisely, an expression describes the locations, directions, widths, and colors of all lines in the picture. Each picture has a "sort," which is a list of ports. Pictures are connected by their ports with the help of links (which are pairs of ports). Port names can be renamed through substitutions. Iteration is used to make arrays of cells. Polygons are formed by paths, which are movements of ports. A sample picture is the graph of a n-MOS inverter. Picture expressions have no simulation semantics.

In the early 1980's, two languages with simulation semantics were developed almost at the same time: μFP by Sheeran [13] and Daisy by Johnson [14].

μFP. This is a combinator language based on Backus's FP. Unlike the "typed" first-order function definitions of Lee and Boute, μFP is a pure untyped language with higher-order combinators. It extends FP with streams, allowing it to describe sequential circuits. FP functions are extended to stream functions using semantic equations, with which most algebraic laws of FP are preserved. As a descendant of FP, μFP makes extensive use of combinators rather than recursive function definitions. Combinational circuits are often represented by FP combinator expressions, and sequential circuits are modeled by a recursively defined μ operator. The use of combinators makes μFP convenient for program transformations in circuit optimization, synthesis and verification. A synthesis mapping was graphically presented in [13]. A translation to Functional Geometry [15] was discussed as an intermediate step toward circuit layout. Works started from μFP is the longest continued research in the area of functional hardware languages.

Daisy. In parallel with the development of μFP, Johnson [14, 16] started his applicative digital design project, which included the creation of a functional hardware language Daisy. Like μFP, Daisy is untyped and it also uses streams as the main data structure. While μFP is based on FP, Daisy is based on Scheme. Circuit design in Johnson's approach is divided into specification and realization. A specification is a set of recursive equations; a realization is a system of signal definitions. The main focus is to transform recursive equation specifications to realizations; the later can then be translated to circuit layouts. Both specification and realization are expressed in Daisy language.

νFP/FHDL. In 1985, Meshkinpour and Ercegovac proposed another FP style language FHDL [17–19], based on a related M.S. thesis in 1981. Like μFP, their work uses combinators to describe combinational circuits. Sequential circuits are described via a special structure parameterized by initial value and transition function. A symbolic interpreter is developed to simulate the behavior of circuits and to compute delay time, area estimation, and so forth.

Hydra. The use of higher-order functions to capture structure seems to have been an idea that occurred to many people at once. In similar spirit, O'Donnell developed Hydra [20]. The name comes from the pronunciation of HDRE (Hardware Description with Recursion Equations) [21]. During its evolution, Hydra has been embedded in Daisy,

Scheme, Miranda, LML, Haskell, and Template Haskell [22]. Like Daisy, the initial version of Hydra took streams to represent signals and used recursive equations to represent circuits. While Daisy deals mainly with first-order functions, Hydra includes higher-order combining forms [20]. It is also capable of performing multiple circuit interpretation (simulation, layout, timing, etc.). In the Haskell version of Hydra, multiple semantics is implemented via type classes.

HEVAL/DUAL-EVAL/DE2. Hunt's tour de force in verifying the FM8501 microprocessor in Nqthm showed the power of machine-assisted proof [23]. Initially, circuits were represented as functions in Boyer-Moore logic, which were later found inadequate as inputs to industry circuit design system. To bridge the gap, a structural HDL named HEVAL was introduced and embedded in the Boyer-Moore logic [24, 25], which was evolved to DUAL-EVAL in the FM9001 project [26]. Circuits in DUAL-EVAL are represented as lists of circuit boxes, which are quoted S-expressions specifying subcircuits, connections, and other features. Among other things, each sub-circuit description contains a label, and the list of labels of all state elements are included at the end of the circuit box. In this way, sequential circuits can be specified and simulated.

The well-formedness of a circuit S-expression is checked by a recognizer, which also verifies the absence of combinational loops, checks fanout violations, and derives loading, timing, and other properties. Tools were built for circuit simulation, delay computation, and netlist generation. In this respect, DUAL-EVAL is different from hardware specifications in other theorem provers at that time, where circuits were specified in logic formulas and translators were needed to convert between formulas and circuit structural descriptions. Typical hardware specifications in theorem prover only support verification, while the structural description in DUAL-EVAL allows both simulation and verification. Besides, circuit generators can be defined and proved to be correct.

The latest version of this language is called DE2.

Glass. Boute and his students developed a system description language Glass [27, 28] in the project FORFUN (Formal Description of Arbitrary Systems by means of Functional Languages) which aimed to support both circuit description and software description.

Other works in this period include [29, 30].

4. 1990s–2000s: Translation to VHDL/Verilog

During the 1990s, VHDL and Verilog have obtained wide acceptance in industry. FHL developers began to use them as synthesis targets.

Ruby. Ruby was proposed by Jones and Sheeran [31–36]. Strictly speaking, Ruby is a relational language, but bears a close relationship with functional languages. First, Ruby was evolved from μFP and is the predecessor of the functional hardware language Lava. Second, Ruby deals only with binary relations, often representing functions with domain on the left and codomain on the right. The primary design goal of Ruby is the description of digital signal-processing circuits and the like. As such, Ruby usually deals with regular structures. It has a collection of primitive relations and operations over relations, which enjoy a set of algebraic laws. Relations are used to represent circuits and their connections. Transformation laws are developed for the derivation from specifications to implementations.

Ruby simulators were designed by Hutton in Lazy ML [37] and by McPhee in λPROLOG [38, 39]. A VHDL translator was created by Sandum, Møller, Sharp, and Rasmussen [40, 41]. Sharp and Rasmussen also wrote a transformational rewriting system [42, 43]. The derivation of functional programs from Ruby was investigated by Hutton [37, 44, 45]. The initial version of Ruby was untyped. McPhee implemented a typed variant in λPROLOG [38]. Based on the "pure Ruby" of Rossen [46, 47], Robin Sharp and his group developed T-Ruby, which has both dependent and polymorphic types [41–43, 48–50]. Rasmussen formalized Ruby in Isabelle theorem prover [51, 52]. As part of the Glasgow Ruby compiler project, Block developed a graphical interface to draw Ruby circuits [53].

Lava. Lava [4, 54–63] is the third-generation language of Sheeran's group after μFP and Ruby. This pure functional language is embedded in Haskell. It supports simulation, synthesis, and verification. Due to the powerful Haskell-type system, Lava hardware descriptions are simpler and clearer than its ancestors (μFP and Ruby).

Two methods have been tried to implement multiple interpretations. The first is based on type class and monads. While Hydra uses different types for different interpretations, Lava puts all circuit types into a disjoint sum-type Signal. It is a pure functional approach, but is not so elegant when circuits contain feedback loops. The second method is called "observable sharing" which keeps node sharing information in immutable references. Besides, descriptions in the second approach are the same as ordinary programs and are easy to understand. Observable sharing requires a nonconservative extension to Haskell. Its operational semantics is presented in [64].

Most functional languages support the construction of recursive data types. Although this is a powerful feature, it is restricted to tree-like structures. In contrast, a notable feature of Lava is its ability to describe circuit networks, which are more general than trees.

HML. The HML language was proposed by O'Leary et al. [65] in 1993 and implemented by Li and Leeser [66–68] in 1995. It is a subset of SML extended with some hardware constructs. HML has close relationship with VHDL, permitting both VHDL style structural and behavioral descriptions. What makes it different from VHDL are features like type inference, polymorphism, recursion, and higher-order functions. Hardware circuits can be described more concisely in HML than in VHDL. This is achieved by several means including omitting typing information (which can

be inferred) and removing explicit clock statements. Regular structures can be generated by exploiting polymorphism and high-order functions.

HML makes distinction between software functions and hardware functions. Only hardware functions correspond to hardware modules. Software functions can be used inside hardware functions. Hardware are described by concurrently combined structural statements and behavioral statements. A behavioral statement can be either a combinational assignment or a sequential assignment. HML supports type checking, simulation, and translation to VHDL. Recursive and higher-order functions can be type-checked but not translated.

Hawk. Development of VHDL or Verilog ports is perhaps the main trend in this period, but there are exceptions. Hawk [69] was introduced as a microprocessor modeling language embedded in Haskell. Like Lava and Hydra, Hawk [69–77] is a Haskell library of signal-processing functions. What makes it differ from Lava and Hydra is that it targets at the architecture level specification and verification. For this purpose, it introduces structural signal descriptions, on which a microarchitecture algebra is developed.

The main data structure in Hawk is transaction. A transaction can be viewed as a group of signals or a signal record. In microarchitecture specification, a transaction can be a set of data processed by one instruction. A single-cycle architecture processes one instruction, or one transaction, in one clock cycle. In a pipelined architecture, there are typically more than one transaction under processing in a single clock cycle; on the other hand, the processing of one transaction requires multiple cycles. The main purpose of Hawk is to formally verify microprocessors by means of algebraic laws over transactions. These transformational proofs have been mechanized in Isabelle theorem prover.

5. 2000s–Now: Entering Industry?

The Ella HDL was a pioneering use of ideas from functional programming [78]. A team at Plessey Caswell used both FP and Ella in the design of regular array video picture motion estimator in 1989. But then everything seemed to stop, and one wonders why.

Xilinx Lava. The new millennium sees increased attempts of using HFL in industry. A variant of Lava was developed in Xilinx for FPGA generation by Satnam Singh. It supports layout description [63] and is used in JPEG system specification [55].

Confluence and HDCaml. Circuit design engineer Tom Hawkins started Launchbird Design Systems with his pure functional hardware language Confluence in 1999. Later, it was changed to an open-source project. Compilers are developed to convert Confluence programs to Verilog and VHDL descriptions. It is claimed that Confluence can achieve 2X to 10X code reduction compared to Verilog and VHDL. Recently, the project moved to the website http://www.funhdl.org/. New developments include a new hardware language HDCaml, which is embedded in Ocaml (i.e., used as a library) with side effects, and ATOM, which is embedded in HASKELL to describe synchronous reactive systems.

An HDCaml program describes a circuit building process, which is internally implemented as database creation and incremental modifications. Started with an empty circuit, the circuit database is extended with newly created signals and circuits. Arithmetic operators are interpreted as signal constructors. Internal signal contains a mutable field, which can be updated through assignment to establish signal connection. Sequential circuits are created by using register objects and connecting signals. Regular circuits can be built via recursion.

Bluespec. The theoretical foundation of Bluespec is the term rewriting semantics proposed by Arvind et al. [79–81]. Currently, this language is the product of Bluespec Inc. [82]. The initial version of the Bluespec language was a functional hardware language based on Haskell. Now it is evolved into two languages: one is Bluespec SystemVerilog and another is Bluespec SystemC. However, many Haskell features are still preserved.

In Bluespec, behaviors are described by *rules*. A rule has a condition part and an action part. Whenever a condition is true, the action will be fired. The condition of a rule is a combinational Boolean expression. The most common rule action is register assignment. A rule can have multiple actions executing in parallel. A set of rules can be put together in a module; this is called *rule composition*. Semantically, rules in a module will be executed nondeterministically one at a time when their conditions become true. This semantics determines the functional correctness and is the basis of formal verification. Typically, two rules in the same module will have mutually exclusive conditions. In this case, at most one rule will be executed at anytime. In hardware, each rule is implemented as a condition block and an action block. Outputs from condition blocks of all rules go to a *scheduler* circuit, and outputs from action blocks of all rules go to a *data select* circuit. The scheduler sends control signal to the data select circuit to determine which action outputs will be sent to state registers. A rule executes entirely in one clock cycle. If possible, multiple rules will be executed concurrently in one clock cycle.

FL/reFLect. To combine the power of theorem proving and the efficiency of model checking, Seger et al. developed the VOSS formal verification environment [83–85], which contains an SML style language FL [86] as its interface language. FL was designed to serve multiple purposes. First, it is the script language to control model checking. For this aim, boolean formulas are internally represented as BDD. Second, it is the implementation language of the higher-order classical logic theorem prover ThmTac. Third, it is the user's language for writing proof tactics in the theorem proving environment. Above all, FL has the capability to support hardware specification and transformation. In real

hardware verification, FL has two roles. First, it decomposes input circuits into smaller blocks and invokes the symbolic trajectory evaluation (STE) tool to verify each of them. Second, it performs theorem proving at the high levels of abstraction to validate the correctness of the composed circuit.

FL is both the meta and object language of theorem proving [86]. Model checking tool executes FL expressions, while logic inference tool proves their properties. To achieve both goals, a so-called "lifted" expression is introduced into FL language, which is similar to the quotation mechanism in LISP. Internally, each FL expression has two representations: one is a normal abstract syntax tree used for evaluation as in other ML family languages; another is the "lifted" abstract syntax tree designed for symbolic transformation. This practice eventually turned FL into a new language named *reFL*^{ect} [87–91]; at the same time, the VOSS system evolved to Forte [5]. *reFL*^{ect} not only has quotation and back quotation as in LISP and MetaML [92, 93], but it also supports pattern matching over quoted expressions. However, some type checking must be done at run time [88].

Industrial applications of the system include the verification of an instruction length decoder [94], which is considered as one of the most complex verifications at that time. The specification has 1500 lines and the gates number exceeds 12,000. During the verification, an induction tactic was applied to split the problem into a base case and an induction case, and STE was invoked during the proof of induction case. In the verification of a floating point adder [85, 95], the verification is divided into hundreds of cases, each of them verified by the model checker, while the theorem prover checks that all cases are covered and that the reference model conforms with the IEEE floating point standard.

SAFL/SAFL+. At the theoretical side, Sharp and Mycroft proposed SAFL and SAFL+ [96–103]. SAFL (Statically Allocated Functional Language) is an SML style hardware language. It is a first-order monomorphic call-by-value functional language with extensions for hardware descriptions. The hardware-specific properties include concurrency, static allocation, and resource awareness. The term "statically allocatable" means that the program memory is allocated at compile time; therefore, dynamic data structures such as lists are not allowed. Functions are either nonrecursive or tail recursive. SAFL+ extends SAFL with channels and arrays. The former are abstractions of bus controls; the later are abstractions of memories or registers. Channels can be read, written, passed as parameters to functions, and declared as local identifier.

The *FLaSH* (Functional Languages for Synthesizing Hardware) Silicon Compiler transforms SAFL+ programs to Verilog programs. This compiler is designed to be *resource aware*, meaning that each function is translated to a single hardware block and multiple calls to a function share the same hardware. When necessary, multiple accesses to the shared hardware will be resolved dynamically by arbiters generated by the compiler. To avoid unnecessary arbiter generation, *Parallel Conflict Analysis* (PCA) is performed to check if there are parallel calls to the same function (a situation called *conflicting*). After PCA analysis, duplicated function calls may either be statically rearranged or be controlled by an arbiter.

In SAFL/SAFL+, source programs can be transformed and optimized. It has been showed that unfolding can be applied to increase performance (usually with more gates), folding and abstraction can be applied to reduce resource duplication, and tail-recursive mutual recursion can be transformed to single recursive function.

Resource Aware Programming (RAP). RAP languages [104] deal with problems with limited resources. The group of Taha investigated combinational circuit generation within the framework of RAP [105]. RAP is closely related to multistage programming [106], which provides type safe program generation. RAP ensures that generated programs are both well typed and resource bounded, a feature desirable for circuit generation. An FFT circuit generator is created as an illustration of the power of this approach [105, 107]. The idea is to first define a general algorithm parameterized with a natural number then generate a specialized program through abstract interpretation. The generator is implemented in MetaOcaml.

Wired. The group of Shareen proposed a low level language Wired [108] for estimating nonfunctional circuit properties.

Domain Specific Languages. The Cryptal language [109] developed in Galois Connections Inc. (led by Launchbury) is a functional language for cryptographic applications. It is aimed at hardware/software codesign. A useful feature of this language is that its type system is capable of describing bit vectors of fixed length.

6. FHL'07

The Workshop of Hardware Design and Functional Language FHL'07 was held on the 24th and 25th of March 2007 [110]. At the time of this writing, the papers in this workshop represent the state of the art in functional hardware languages. They can be roughly classified into two groups: new researches and improvements to existing works.

6.1. Improvements to Existing Works. Avind, Dave, and Pellauer [111] report an extension of BLUESPEC with *sequential connective* and *scheduling primitives*. The intention is to overcome the limitations in the specification of concurrent execution of guarded atomic actions as well as the resolution of nondeterminism among competing rules in previous BLUESPEC implementation. Sequential connective is used to build bigger atomic actions. The new language is called BS1.

Singh [112] extends his combinator style FPGA circuit description to GPU programming description. Two implementations have been presented: one is in a variant of ML, and another is in a C# style language. A parallel sorter is used as a demonstration example.

Sheeran [113] presents a parallel prefix network generator using combinators.

Hunt's hardware language DUAL-EVAL/DE2 has evolved to The E Language [114]. It allows functional, property, timing, and power specifications. An interesting aspect of his work is the description and verification of circuit generators.

Seger [115] discusses the Integrated Design and Verification system (IDV) of INTEL. This system intends to support the transformations from high-level models down to physical implementations while keeping each step formally verified. One example is to derive the detailed implementation of a superscalar processing unit. The transformations include retiming, duplication, merging, and read after write.

Claessen and Pace [116] made an interesting comparison about alternative design decisions among existing embedded functional hardware languages.

Naylor et al. [117] are developing a Haskell library for Wired. In their work, the relational operators are once again put into practice. It is argued that the use of relation can reduce the numbers of combinators and can provide support for *bidirectional evaluation* as well as *layout inference*. An example is the definition of an encoder in terms of a decoder. The power of this relational version of Wired is demonstrated by the layout and delay computation of a prefix network, using its notion of tiles.

6.2. Emerging Researches. Martin and Gheith [118] are developing "System ML" language embedded in Ocaml in IBM Austin Lab. The language supports both simulation and synthesis. Circuit description is based on a stream type. Reflexion mechanism is introduced to the language to synthesize combinational circuits. The language is used for the high-level description of a multithreaded microprocessor.

Chong and Ishtiaq [119] from ARM Ltd. discuss their ongoing work of formal RTL level description and verification of ARM V7 in Coq. The main focus is on the exception and memory models.

Schmidt-Schauß and Sabel [120] proposes a call-by-need λ-calculus L_{por} in the spirit of Lava. In circuit description, concurrent signal assignments are modeled by **letrec**, and registers are modeled by the *delay* function. The novel feature of this calculus is the introduction of a parallel-or operator, which models the parallel execution of OR gate with a nondeterministic reduction semantics. This calculus establishes a sequential circuit equivalence relation, which is the foundation of the circuit transformations such as retiming, sharing introduction, partial evaluation, constant folding, and constant introduction. An implementation of parallel-or is demonstrated in Concurrent Haskell.

Baptiste Note and Vuillemin [121] describe a hardware/software codesign system, in which a part of the original design is compiled into hardware for acceleration. The hardware compiler proceeds with a series of steps including code analysis (e.g., SSA transformation), partial place and route, retiming, and asynchronous communication interface generation. The design language DSL is a synchronous, dataflow-oriented, functional language with higher-order types and operator overloading in the style of Lustre.

Sheard [122] attempts to make a very expressive dependent-type approach named Omega system. For example, the function "add3bits," which adds three bits and returns a two bit binary, has the type

```
add3bits: (Bit i) → (Bit j) → (Bit k)
          → Binary Bit #2
{plus {plus j k} i}.
```

It means that each argument n of the function is of the singleton type (Bit n) and that the return object should be a 2 bit binary whose value is $j + k + i$. That is to say, the type of this function contains its semantics. A further example shows that the type of a ripple carry adder is a semantic specification of the adder. Type checking is illustrated by a running example. Apart from the type system, it is proposed to do synthesis through symbolic evaluation and simulation via staged programming.

Taha et al. [123, 124] formalize a core Verilog in a language with a two-level static type system. The motivation is to type-check a synthesizable subset of Verilog, especially the loops and parameterized modules which are essential to generic designs. For this purpose, a sophisticated type system is formally defined and it is proved that well-typed core Verilog programs are synthesizable.

7. Concluding Remarks

Due to the complexity of hardware design, functional hardware languages are highly diversified.

FHLs may differ at the levels of abstraction: system level (e.g., Glass), architecture level (e.g., Hawk), algorithmic level (e.g., Bluespec), RTL level (e.g., Lava), netlist level (e.g., Wire), and layout level (e.g., Sticks&Stones). They may address one or more aspects in hardware development: specification (e.g., John Lee), simulation (e.g., Hawk), verification (e.g., $reFL^{ect}$), optimization (e.g., Ruby), timing and cost analysis (e.g., Hydra), and synthesis (e.g., Bluespec). The description can be structural (e.g., DUAL-EVAL) or behavioral (e.g., SAFL+). A tricky issue in the circuit modeling is the representation of sequential circuit; a number of approaches have been proposed: recursive equation (e.g., Daisy), addition of special operator (e.g., μFP), use of special label (e.g., Hunt), recursive function (e.g., SAFL), and module of rules (e.g., Bluespec).

An FHL can be implemented as an embedded language (e.g., HDCaml), or as an independent language (e.g Confluence); it can be embedded in a theorem prover (e.g., DUAL-EVAL) or as an implementation language of a theorem prover (e.g., $reFL^{ect}$). An FHL can be a general purpose programming language (e.g., $reFL^{ect}$), a general purpose hardware language (e.g., Bluespec), or a domain-specific language (e.g., Cryptal). Most FHL's are based on an existing software functional language: FP (e.g., μFP), Haskell (e.g., Hydra), ML (e.g., HML), Scheme (e.g., Daisy), Common Lisp (e.g., DUAL-EVAL), and MetaOcaml (RAP circuit generation). The language can be first order (e.g., SAFL), or higher order (e.g., Lava); it may be untyped (e.g., μFP), monomorphically typed (e.g., SAFL) or polymorphically typed (e.g., HML). Traditionally, FDL's are single staged; a

new trend is to use multistaged language (MetaOcaml) or reflection language ($reFL^{ect}$).

More than half of HDL's are designed to support formal verification and/or circuit transformation. The most popular formal verification technique is equational reasoning (e.g., Ruby); other methods include induction (e.g., Daisy), term rewriting (e.g., Bluespec), combination of theorem proving, and model checking (e.g., $reFL^{ect}$).

Despite these significant progresses, functional hardware languages are still facing many challenging problems. (1) There are only few published circuit designs developped using FHLs. This indicates that the functional language community has much less experiences in hardware development than in software programming and that there is a lack of functional circuit libraries to support large-scale hardware designs. (2) The quality of a design is difficult to measure. Unlike a software program whose performance can be evaluated through execution of a set of benchmark samples, the "execution speed" of a circuit needs to be analysed via timing analysis, which depends not only on the critical path length, but also on factors like wire length, wire loads, and fanouts. Besides, a real design is often a tradeoff between timing, power, area, and costs. (3) The main data structures in functional languages are tree-like structures, but hardware structures are often graphs. A nice advancement in this direction is the languages Ruby and Lava, which demonstrated functional style graph composition using specially designed combinators. Still, more functional style netlist oriented structure manipulations are expected. (4) The description method for some special circuits, for example, sequential circuits, has not reached an agreement in the FHL community yet. (5) There is no simple hardware model yet. Many hardware features are not easily captured in a language. For example, a ripple carry adder is slow when all inputs arrive at the same time. However, if signal arrival time is unevenly distributed, a ripple carry adder could be the best. (6) An ideal hardware language needs to support both parameterised circuit generation and the circuit simulation. Therefore, the execution of a "hardware program" could be either an instantiated hardware structural description or a circuit simulation result. To address this problem, the notion of multiple interpretation has been proposed in Lava. A recent trend is to use two-level languages or reflexive language as demonstrated in $reFL^{ect}$.

Acknowledgment

The author grateful to his wife Ping Hu for her support and patience. Thanks are due to Mary Sheeran for his insightful remarks.

References

[1] J. A. N. Lee, *Computer Semantics*, Van Nostrand Reinhold Company, 1972.

[2] M. Sheeran, *μFP, an algebraic VLSI design language*, Ph.D. thesis, Oxford University, 1983.

[3] J. O'Donnell, "Generating netlists from executable circuit specifications in a pure functional language," in *Proceedings of the Functional Programming Workshops in Computing, Glasgow 1992*, Springer, 1993.

[4] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in Haskell," in *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pp. 174–184, September 1998.

[5] C.-J. H. Seger, R. B. Jones, J. W. O'Leary et al., "An industrially effective environment for formal hardware verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1381–1405, 2005.

[6] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: a survey," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 2, pp. 123–193, 1999.

[7] P. Wegner, "The Vienna definition language," *ACM Computing Surveys*, vol. 4, no. 1, pp. 5–63, 1972.

[8] R. T. Boute, "Representational and denotational semantics of digital systems," *IEEE Transactions on Computers*, vol. 38, no. 7, pp. 986–999, 1989.

[9] R. E. Frankel and S. W. Smoliar, "Beyond register transfer: an algebraic approach for architectural description," in *Proceedings of the 4th International Conference on Computer Hardware Description Languages*, pp. 1–5, 1979.

[10] R. E. Frankel and S. W. Smoliar, "Digital systems as mathematical expressions," in *Proceedings of the International Conference on COMPCON (COMPCON '81)*, pp. 414–416, Spring, 1981.

[11] L. Cardelli and G. D. Plotkin, "An algebraic approach to VLSI design," in *Proceedings of the 1st International Conference on Very Large Scale Integration (VLSI '81)*, J. P. Gray, Ed., pp. 173–182, University of Edinburgh, Academic Press, August 1981.

[12] L. Cardelli, "Sticks&stones: an applicative VLSI design language," Internal Report CSR-85-81, University of Edinburgh, Department of Computer Science, 1981.

[13] M. Sheeran, "mμFP, a language for VLSI design," in *Proceedings of the Conference Record of the ACM Symposium on LISP and Functional Programming (LISP '84)*, pp. 104–112, Austin, Tex, USA, 1984.

[14] S. D. Johnson, *Synthesis of Digital Designs from Recursion Equations*, The MIT Press, Cambridge, Mass, USA, 1983.

[15] P. Henderson, "Functional geometry," in *Proceedings of the ACM Symposium on LISP and Functional Programming (LISP '82)*, p. 179, 1982.

[16] S. D. Johnson, "Applicative programming and digital design," in *Proceedings of the Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages (POPL '84)*, pp. 218–227, 1984.

[17] F. Meshkinpour and M. D. Ercegovac, "A functional language for description and design of digital systems: sequential constructs," in *Proceedings of the 22nd ACM/IEEE Conference on Design Automation*, pp. 238–244, ACM Press, 1985.

[18] M. D. Ercegovac and T. Lang, "A high-level language approach to custom chip layout design," Technical Report MICRO Project Reports 1982-83, University of California, Berkeley, Calif, USA, 1982.

[19] D. R. Patel, M. Schlag, and M. D. Ercegovac, "An environment for the multi-level specification, analysis, and synthesis of hardware algorithms," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, vol. 201 of *Lecture Notes in Computer Science*, pp. 238–255, Springer, 1985.

[20] J. O'Donnell, "Hydra: hardware description in a functional language using recursion equations and high order combining forms," in *Proceedings of the Fusion of Hardware*

Design and Verification, G. J. Milne, Ed., pp. 309–328, North-Holland, Amsterdam, The Netherlands, 1988.

[21] J. O'Donnell, "Hardware description with recursion equations," in *Proceedings of the 8th International Symposium on Computer Hardware Description Languages and Their Applications (CHDL '87)*, M. R. Barbacci and C. J. Koomen, Eds., IFIP WG 10.2, pp. 363–382, North Holland, 1987.

[22] J. O'Donnell, "Embedding a hardware description language in template Haskell," in *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, German*, C. Lengauer, D. Batory, C. Consel, and M. Odersky, Eds., vol. 3016 of *Lecture Notes in Computer Science*, Springer, 2004.

[23] W. A. Hunt Jr., *FM8501: a verified microprocessor*, Ph.D. thesis, 1985.

[24] B. Brock and W. A. Hunt Jr., "The formalization of a simple hardware description language," in *Applied Formal Methods For Correct VLSI Design*, L. Claessen, Ed., pp. 778–792, Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1989.

[25] B. Brock, W. A. Hunt Jr., and W. D. Young, "Introduction to a formally defined hardware description language," in *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, Nijmegen (TPCD '92)*, V. Stavridou, T. F. Melham, and R. T. Boute, Eds., vol. 10 of *IFIP Transactions*, pp. 3–35, North-Holland, Amsterdam, The Netherlands, June 1992.

[26] B. Brock and W. A. Hunt Jr., "The dual-eval hardware description language and its use in the formal specification and verification of the FM9001 microprocessor," *Formal Methods in System Design*, vol. 11, no. 1, pp. 71–104, 1997.

[27] J. De Man, "The description of digital systems by means of a functional language," Internal Report, Bell Telephone Mfg. Cy, Antwerp, Belgium, 1986.

[28] C. van Reeuwijk, *The implementation of a systems description language and its semantic functions*, Ph.D. thesis, Delft University, 1991.

[29] D. Lahti, *Applications of a functional programming language to hardware synthesis*, M.S. thesis, 1980.

[30] D. Lahti, "Application of a functional programming language," Tech. Rep. CSD-810403, University of California, Los Angeles, Department of Computer Science, Los Angeles, Calif, USA, 1981.

[31] M. Sheeran, "Describing hardware algorithms in Ruby," in *Proceedings of the Declarative Systems*, North-Holland, 1990.

[32] G. Jones and M. Sheeran, "Circuit design in Ruby," in *Formal Methods for VLSI Design*, J. Staunstrup, Ed., Lecture Notes on Ruby from a Summer School in Lyngby, Denmark, North Holland, 1990.

[33] G. Jones, "Designing circuits by calculation," Tech. Rep. PRG-TR10-90, Oxford University Press, New York, NY, USA, 1990.

[34] G. Jones and M. Sheeran, "A certain loss of identity," in *Proceedings of the Functional Programming, Workshops in Computing*, J. Launchbury and P. M. Sansom, Eds., pp. 113–121, Springer, London, UK, 1992.

[35] G. Jones and M. Sheeran, "Designing arithmetic circuits by refinement in Ruby," *Science of Computer Programming*, vol. 22, no. 1-2, pp. 107–135, 1994.

[36] G. Jones and M. Sheeran, "Deriving bit-serial circuits in Ruby," in *Proceedings of the IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration (VLSI '91)*, pp. 71–80, 1991.

[37] G. Hutton, "The Ruby Interpreter," Research Report 72, Chalmers University of Technology, 1993.

[38] R. McPhee, "Implementing Ruby in a higher-order logic programming language," Tech. Rep., Oxford University Computing Laboratory, 1995.

[39] R. McPhee, *Towards a relational programming language*, Qualifying Dissertation Submitted in Application for Transfer to DPhil Status, 1995.

[40] O. Sandum, *Translation of Ruby into VHDL*, M.S. thesis, Department of Computer Science, Technical University of Denmark, 1994, 2.

[41] R. Sharp, "T-Ruby: a tool for handling Ruby expressions," Tech. Rep. ID-TR: 1994-154, Departement of Computer Science, Technical University of Denmark, 1994.

[42] R. Sharp and O. Rasmussen, "Transformational rewriting with Ruby," in *Proceedings of the Computer Hardware Description Languages and their Applications*, D. Agnew, L. Claesen, and R. Camposano, Eds., pp. 231–248, Elsevier Science B.V., Ottawa, Canada, 1993.

[43] R. Sharp and O. Rasmussen, "An introduction to Ruby," Tech. Rep. ID–U: 1995-80, Department of Computer Science, Technical University of Denmark, 1994.

[44] G. Hutton, "A relational derivation of a functional program," in *Proceedings of the STOP Summer School on Constructive Algorithmics*, Ameland, The Netherlands, September 1992.

[45] G. Hutton, *Between functions and relations in calculating programs*, Ph.D. thesis, University of Glasgow, 1992, Research Report FP-93-5.

[46] L. Rossen and R. Algebra, "Designing correct circuits," in *Proceedings of the Workshops in Computing*, G. Jones and M. Sheeran, Eds., pp. 297–312, Springer, 1990.

[47] L. Rossen and R. Sharp, "Sequence semantics of Ruby," in *Proceedings of the Designing Correct Circuits*, J. Staunstrup and R. Sharp, Eds., vol. A-5, pp. 159–171, North-Holland, 1992.

[48] R. Sharp and O. Rasmussen, "Using a language of functions and relations for VLSI specification," in *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*, pp. 45–54, ACM Press, NewYork, NY, USA, 1995.

[49] R. Sharp, "The Ruby framework," Tech. Rep. ID-TR: 1993-119, Departement of Computer Science, Technical University of Denmark, 1993.

[50] R. Sharp and O. Rasmussen, "The T-Ruby design system," *Formal Methods in System Design*, vol. 11, no. 3, pp. 239–264, 1997.

[51] O. Rasmussen, "An embedding of Ruby in Isabelle," in *Proceedings of the 13th International Conference on Automated Deduction*, M. A. McRobbie and J. K. Slaney, Eds., vol. 1104 of *Lecture Notes in Artificial Intelligence*, pp. 186–200, Springer, New Brunswick, NJ, USA, 1996.

[52] O. Rasmussen, "Formalising ruby in isabelle," in *Proceedings of the 1st Isabelle Users Work Shop*, L. C. Paulson, Ed., University of Cambridge, Computer Laboratory, Cambridge, UK, 1995.

[53] C. J. Block, *A graphical interface for ruby*, M.S. thesis, Datavetenskap, Chalmers Tekniska Hgskola, 1996.

[54] K. Claessen and M. Sheeran, *A Tutorial on Lava: A Hardware Description and Verification System*, 2000.

[55] S. Singh, "System level specification in lava," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '03)*, pp. 370–375, IEEE Computer Society, 2003.

[56] K. Claessen, M. Sheeran, and S. Singh, "Using lava to design and verify recursive and periodic sorters," *International*

Journal on Software Tools for Technology Transfer, vol. 4, no. 3, pp. 349–358, 2003.

[57] K. Claessen, *Embedded languages for describing and verifying hardware*, Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2001.

[58] K. Claessen, M. Sheeran, and S. Singh, "The design and verification of a sorter core," in *Proceedings of the Conference on Correct Hardware Design and Verification Methods (CHARME '01)*, Lecture Notes in Computer Science, Springer, New York, NY, USA, 2001.

[59] K. Claessen, M. Sheeran, and S. Singh, "Functional hardware description in Lava," in *The Fun of Programming, Cornerstones of Computing*, pp. 151–176, Palgrave, 2003.

[60] S. Singh, The lava hardware description language, http://raintown.org/lava/.

[61] S. Singh and P. James-Roxby, "Lava and JBits: from HDL to bitstream in seconds," in *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, pp. 91–100, IEEE Computer Society, Washington, DC, USA, 2001.

[62] S. Singh, "Designing reconfigurable systems in Lava," in *Proceedings of 17th International Conference on VLSI Design, Concurrently with the 3rd International Conference on Embedded Systems Design (VLSI '04)*, vol. 17, pp. 299–306, IEEE Computer Society, 2004.

[63] S. Singh and M. Sheeran, "Designing FPGA circuits in lava," In press.

[64] K. Claessen and D. Sands, "Observable sharing for functional circuit description," in *Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science (ASIAN '99)*, pp. 62–73, Springer, London, UK, 1999.

[65] J. O'Leary, M. H. Linderman, M. Leeser, and M. Aagaard, "HML: a hardware description language based on standard ML," in *Proceedings of the IFIP Conference on Hardare Description Languages and Their Applications (CHDL '93)*, no. A-32, pp. 327–334, 1993.

[66] Y. Li and M. Leeser, "HML: an innovative hardware description language and its translation to VHDL," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '95)*, pp. 691–696, 1995.

[67] Y. Li, *HML: an innovative hardware description language and its translation to VHDL*, M.S. thesis, Graduate School of Cornell University, 1995.

[68] Y. Li and M. Leeser, "HML, a novel hardware description language and its translation to VHDL," *IEEE Transactions on Very Large Scale Integration Systems (VLSI '00)*, vol. 8, no. 1, pp. 1–8, 2000.

[69] Matthews, Cook, and Launchbury, "Microprocessor specification in hawk," *Proceedings of the IEEE International Conference on Computer Languages (ICCL '98)*, pp. 90–101, 1998.

[70] B. Cook, J. Launchbury, and J. Matthews, "Microprocessor specification in hawk," in *Proceedings of the International Conference on Computer Languages (FTH '98)*, pp. 90–101, May 1998.

[71] J. Matthews, J. Launchbury, and B. Cook, "Microprocessor specification in hawk," in *Proceedings of the IEEE International Conference on Computer Languages (ICCL '98)*, pp. 90–101, 1998.

[72] J. Launchbury, J. R. Lewis, and B. Cook, "On embedding a microarchitectural design language within Haskell.," in *Proceedings of the International Conference on Functional Programming (ICFP '99)*, pp. 60–69, 1999.

[73] J. Matthews, *Algebraic specification and verification of processor microarchitectures*, Ph.D. thesis, Oregon Graduate Institute, 2000.

[74] N. A. Day, J. R. Lewis, and B. Cook, "Symbolic simulation of microprocessor models using type classes 18 in Haskell," in *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '99)*, Lecture Notes in Computer Science, pp. 346–349, Springer, London, UK, 1999.

[75] N. A. Day, J. R. Lewis, and B. Cook, "Symbolic simulation of microprocessor models using type classes in Haskell," Tech. Rep. CSE-99-005, Department of Computer Science, Oregon Graduate Institute, 1999.

[76] S. Krstic, B. Cook, J. Launchbury, and J. Matthews, "Top-level refinement in processor verification," Tech. Rep., 1998.

[77] S. Krstic, B. Cook, J. Launchbury, and J. Matthews, *A Correctness Proof of a Speculative, Superscalar, Out-of-Order, Renaming Microarchitecture*, 1998.

[78] J. D. Morison, N. E. Peeling, and T. L. Thorp, "The design rationale of ella, a hardware design and description language," in *Proceedings of the 7th International Symposium on Computer Hardware Description Languages and their Applications (CHDL '85)*, pp. 303–320, North-Holland, 1985.

[79] Arvind and X. Shen, "Using term rewriting systems to design and verify processors," *IEEE Micro*, vol. 19, no. 3, pp. 36–46, 1999.

[80] J. C. Hoe and Arvind, "Hardware synthesis from term rewriting systems," in *Proceedings of the IFIP TC10/WG10.5 10th International Conference on Very Large Scale Integration (VLSI '99)*, L. M. Silveira, S. Devadas, and R. A. da Luz Reis, Eds., vol. 162, pp. 595–619, Kluwer, Lisbon, Portugal, December 1999.

[81] J. C. Hoe and Arvind, "Synthesis of operation-centric hardware descriptions," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '00)*, pp. 511–518, 2000.

[82] Blucspec Inc., http://bluespec.com.

[83] C. Seger, "Voss—a formal hardware verification system user's guide," Tech. Rep., UBC Press, Vancouver, Canada, 1993.

[84] M. Aagaard, R. B. Jones, T. F. Melham, J. W. O'Leary, and C.-J. H. Seger, "A methodology for large-scale hardware verification," in *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD '00)*, pp. 263–282, Springer, London, UK, 2000.

[85] R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham, "Practical formal verification in microprocessor design," *IEEE Design and Test of Computers*, vol. 18, no. 4, pp. 16–25, 2001.

[86] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "A pragmatic implementation of combined model checking and theorem proving," in *Proceedings of the 12th International Conference (TPHOLs '99)*, July 1999.

[87] J. Grundy, T. Melham, and J. O'Leary, "A reflective functional language for hardware design and theorm proving," Tech. Rep. PRG-RR-03-16, Oxford Univerity, Computing Laboratory, 2003.

[88] J. Grundy, T. Melham, and J. O'Leary, "A reflective functional language for hardware design and theorem proving," *Journal of Functional Programming*, vol. 15, no. 2, pp. 157–196, 2006.

[89] S. Krstic and J. Matthews, "Semantics of the eflect language," in *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '04)*, pp. 32–42, ACM Press, New York, NY, USA, 2004.

[90] S. Krstic and J. Matthews, "Subject reduction and confluence for the reFLect language," Tech. Rep. CSE-03-014, OGI, 2003.

[91] http://web.comlab.ox.ac.uk/oucl/work/tom.melham/res/reflect.html.

[92] W. Taha and T. Sheard, "Metamland multi-stage programming with explicit annotations," *Theoretical Computer Science*, vol. 248, no. 1-2, pp. 211–242, 2000.

[93] T. Sheard, "Accomplishments and research challenges in meta-programming," in *Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG '01)*, pp. 2–44, Springer, London, UK, August 2000.

[94] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Combining theorem proving and trajectory evaluation in an industrial environment," in *Proceedings of the 35th Design Automation Conference*, pp. 538–541, ACM/IEEE, June 1998.

[95] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating-point hardware," *Intel Technology Journal*, vol. 3, no. 1, article 10, 1999.

[96] A. Mycroft and R. Sharp, "A statically allocated parallel functional language," in *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP '00)*, Montanari et al., Ed., vol. 1853 of *Lecture Notes in Computer Science*, pp. 37–48, Springer, 2000.

[97] R. Sharp and A. Mycroft, "Soft scheduling for hardware," in *Proceedings of the 8th International Symposium on Static Analysis (SAS '01)*, P. Cousot, Ed., vol. 2126 of *Lecture Notes in Computer Science*, pp. 57–72, Springer, London, UK, 2001.

[98] A. Mycroft and R. Sharp, "Hardware/software co-design using functional languages," in *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, T. Margaria and W. Yi, Eds., pp. 236–251, 2001.

[99] A. Mycroft and R. Sharp, "Hardware synthesis using safl and application to processor design," in *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '01)*, T. Margaria and Melham, Eds., 2001.

[100] R. Sharp, "Functional design using behavioural and structural components," in *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD '02)*, M. Aagaard and J. W. O'Leary, Eds., pp. 324–341, Springer, Portland, Ore, USA, 2002.

[101] A. Mycroft and R. Sharp, "Higher-level techniques for hardware description and synthesis," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 3, pp. 271–297, 2003.

[102] R. Sharp and A. Mycroft, "A higher-level language for hardware synthesis," in *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '01)*, vol. 2144 of *Lecture Notes in Computer Science*, p. 228, 2001.

[103] R. Sharp, *Higher-Level Hardware Synthesis*, Springer, New York, NY, USA, 2005.

[104] W. Taha, "Resource-aware programming," in *Proceedings of the 1st International Conference of the Embedded Software and Systems (ICESS '04)*, Z. Wu, C. Chen, M. Guo, and J. Bu, Eds., vol. 3605 of *Lecture Notes in Computer Science*, pp. 38–43, Springer, 2004.

[105] O. Kiselyov, K. N. Swadi, and W. Taha, "A methodology for generating verified combinatorial circuits," in *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT '04)*, G. C. Buttazzo, Ed., pp. 249–258, ACM Press, 2004.

[106] W. Taha, "A gentle introduction to multi-stage programming," in *Domain-Specific Program Generation*, C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, Eds., vol. 3016 of *Lecture Notes in Computer Science*, pp. 30–50, Springer, 2003.

[107] A. Megacz, "Multi-stage programming languagemeta-hdl," In press.

[108] E. Axelsson, K. Ciaessen, and M. Sheeran, "Wired: wire-aware circuit design," *Proceedings of the Conference on Correct Hardware Design and Verification Methods (CHARME '05)*, vol. 3725, pp. 5–19, 2005.

[109] Galois Inc., http://www.galois.com/cryptol.

[110] "Introducing scheduling primitives and derived interfaces in bluespec," in *Proceedings of the International Workshop on Hardware Design and Functional Languages*, T. Margaria and W. Yi, Eds.A. Martin, C.-J. Seger, and M. Sheeran, Eds., March 2007.

[111] Arvind, N. Dave, and M. Pellauer, "Introducing scheduling primitives and derived interfaces in bluespec," in *Proceedings of the International Workshop on Hardware Design and Functional Languages*, A. Martin, C.-J. Seger, and M. Sheeran, Eds., March 2007.

[112] S. Singh, "Declarative programming techniques for many-core architectures," in *Proceedings of the International Workshop on Hardware Design and Functional Languages*, A. Martin, C.-J. Seger, and M. Sheeran, Eds., March 2007.

[113] M. Sheeran, "Searching for prefix networks to fit in a context using a lazy functional programming language," in *Proceedings of the International Workshop on Hardware Design and Functional Languages, Participants*, A. Martin, C.-J. Seger, and M. Sheeran, Eds., March 2007.

[114] R. S. Boyer and W. A. Hunt Jr., "The e language," in *Proceedings of the International Workshop on Hardware Design and Functional Languages*, A. Martin, C.-J. Seger, and M. Sheeran, Eds., March 2007.

[115] C.-J. H. Seger, "High-level micro-architectural transformations and cycle-accurate high-level models," in *Proceedings of the International Workshop on Hardware Design and Functional Languages*, A. Martin, C.-J. Seger, and M. Sheeran, Eds., March 2007.

[116] K. Claessen and G. Pace, "Embedded hardware description languages: exploring the design space," in *Proceedings of the International Workshop on Hardware Design and Functional Languages*, A. Martin, C.-J. Seger, and M. Sheeran, Eds., March 2007.

[117] M. Naylor, E. Axelsson, and C. Runciman, "Lightweight relational programming for wired," in *Proceedings of the International Workshop on Hardware Design and Functional Languages*, A. Martin, C.-J. Seger, and M. Sheeran, Eds., March 2007.

[118] A. K. Martin and A. Gheith, "A framework for designing hardware in ocaml," in *Proceedings of the International Workshop on Hardware Design and Functional Languages*, A. Martin, C.-J. Seger, and M. Sheeran, Eds., March 2007.

[119] N. Chong and S. Ishtiaq, "Functional programming for hardware definition, verification and modelling," in *Proceedings of the International Workshop on Hardware Design and Functional Languages*, A. Martin, C.-J. Seger, and M. Sheeran, Eds., March 2007.

[120] M. Schmidt-Schauß and D. Sabel, "Program transformation for functional circuit descriptions," in *Proceedings of the International Workshop on Hardware Design and Functional Languages*, A. Martin, C.-J. Seger, and M. Sheeran, Eds., March 2007.

[121] J. B. Note and J. Vuillemin, "Towards automatically compiling efficient fpga hardware," in *Proceedings of the International Workshop on Hardware Design and Functional Languages*, A. Martin, C.-J. Seger, and M. Sheeran, Eds., March 2007.

[122] T. Sheard, "Design principles for hardware description," in *Proceedings of the International Workshop on Hardware Design and Functional Languages*, A. Martin, C.-J. Seger, and M. Sheeran, Eds., March 2007.

[123] W. Taha, Y. Alkabani, C. Andraos et al., "Hardware descriptions as two level computations," in *Proceedings of the International Workshop on Hardware Design and Functional Languages*, A. Martin, C.-J. Seger, and M. Sheeran, Eds., March 2007.

[124] J. Gillenwater, G. Malecha, C. Salama et al., "Synthesizable high level hardware descriptions: using statically typed two-level languages to guarantee Verilog synthesizability," in *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '08)*, pp. 41–50, ACM, New York, NY, USA, 2008.