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 reFLect [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 \rightarrow 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&Sones. It is essentially a picture-drawing language and perhaps the earliest applicative language created for hardware layout. Given a textual input written in the language, the implemented tool can draw a picture on a graphical terminal. Expressions in Sticks&Sones 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: \( \mu \)FP by Sheeran [13] and Daisy by Johnson [14].

\( \mu \)FP. This is a combinator language based on Backus’s FP. Unlike the “typed” first-order function definitions of Lee and Boute, \( \mu \)FP 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, \( \mu \)FP 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 \( \mu \) operator. The use of combinators makes \( \mu \)FP 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 \( \mu \)FP is the longest continued research in the area of functional hardware languages.

Daisy. In parallel with the development of \( \mu \)FP, Johnson [14, 16] started his applicative digital design project, which included the creation of a functional hardware language Daisy. Like \( \mu \)FP, Daisy is untyped and it also uses streams as the main data structure. While \( \mu \)FP 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.

\( \nu \)FP/FHDL. In 1985, Meshkinpour and Ercegovac proposed another FP style language FHDL [17–19], based on a related M.S. thesis in 1981. Like \( \mu \)FP, 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 HDClaml, 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 HDClaml 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 as in other ML family languages; another is the “lifted” circuit.

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 Pel-lauer [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-EV AL/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. Reflection 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 \( \lambda \)-calculus \( \Lambda_{pex} \) 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

\[
\text{add3bits: (Bit i) \rightarrow (Bit j) \rightarrow (Bit k) \rightarrow Binary Bit #2}
\]

\[
\{ \text{plus } \{ \text{plus } j \, k \} \, i \}.
\]

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., reFLect), 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., \( \mu \)FP), 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., HDCCaml), 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., reFLect). An FHL can be a general purpose programming language (e.g., reFLect), 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., \( \mu \)FP), 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., \( \mu \)FP), 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 ($\text{reFL}^{e\text{ct}}$).

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., $\text{reFL}^{e\text{ct}}$).

Despite these significant progresses, functional hardware languages are still facing many challenging problems. (1) There are only few published circuit designs developed 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 visualized using timing analysis, which depends not only on the critical path length, but also on factors like wire length, wire load, 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 structure 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 $\text{reFL}^{e\text{ct}}$.

Acknowledgment

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

References


