An Inlining Approach to Formal Hardware Semantics

by

Joonwon Choi

B.S., Seoul National University (2013)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by .....
Arvind
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

An Inlining Approach to Formal Hardware Semantics

by

Joonwon Choi

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Hardware components are extremely complex due to concurrency. Modularity has been considered as an effective way to design and understand such complex hardware components. Among various hardware description languages (HDLs), Bluespec allows designers to develop hardware not only based on modularity, but also based on the notion of guarded atomic actions (GAAs). Following the concepts of modularity and GAA, we have been defining a framework called Kami to specify, verify, and synthesize Bluespec-style hardware components. However, modular semantics has an inherent weakness in that it is hard to infer internal changes. In this thesis, I present a new semantic approach based on inlining. Inlining semantics is defined for open hardware systems and resolves the weakness by construction. An implication from modular semantics to inlining semantics is also formally proven; thus the inlining semantics can be used to efficiently prove hardware properties.

Thesis Supervisor: Arvind
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank a number of people who have contributed to this work or have given me thoughtful advice. Without help from these people, I would not have been able to develop the work and to write the thesis.

First, I would like to thank my advisors, Professor Arvind and Adam Chlipala for giving me a significant amount of intuitions and motivations for the work. Before joining the master degree program, I had almost no experience in hardware verification. Thanks to advice from Arvind and Adam, I could learn a lot of knowledge and skills fast and efficiently.

I would also like to thank the CSG and PLV group people for invaluable advice. I especially give my gratitude to Muralidaran Vijayaraghavan for discussing and developing the Kami project with me.

Last but not least, I would like to thank Kwanjeong Educational Foundation, which supported me financially for my master's degree.

Contents

1 Introduction **11**

2 Background **15**

2.1 Hardware Design Paradigms 15

2.2 Semantics of the Bluespec Language 16

2.3 Previous Semantic Designs 19

3 An Inlining Approach to Hardware Semantics **23**

3.1 Syntax 24

3.2 Modular Semantics 27

3.3 Manipulating Module Structures 33

3.4 Well-formedness of Modules 35

3.5 Inlining Semantics 38

3.5.1 Inlining Operation 38

3.5.2 Inlining Semantics 41

4 Proving the Implication from Modular to Inlining Semantics **43**

4.1 Meaning of the Semantic Implication 43

4.2 The Implication Proof: From Modular to Inlining Semantics 45

5 Discussion and Conclusions **57**

List of Figures

2-1 A pipelined system 15
2-2 The pipelined system implemented with Bluespec 17
2-3 A weakness of modular semantics 20

3-1 Two methods forming disjoint state updates 36
3-2 A method where double writes happen in some cases 37
3-3 A call cycle during inlining 40
3-4 Inlining efficiency with respect to the Step semantics 42

4-1 Inlining semantics allows an execution of f and g, while the modular one does not. 44
4-2 Inlining a method hides label elements of the method 51

Chapter 1

Introduction

Hardware components are extremely complex. The main reasons for the complexity are concurrency and the sheer size of components. For example, realistic processors make use of instruction pipelining to execute multiple instructions concurrently. Complexity comes with such executions, since one must ensure that the concurrent executions behave as if they are executed in order. The size of components also grows by complexity. More realistic processors require additional hardware components such as a reorder buffer. The logic is further complicated with these components.

Modularity has been considered as an effective way to design and understand such complex hardware components. A complex hardware component can be constructed by composing several simple modules. The notion of modules is ubiquitous, and all commercial hardware description languages (HDLs) like Verilog, VHDL, Bluespec, and Chisel [9, 3, 4, 5] support it.

Even though most HDLs allow users to design hardware in a modular manner, they do not necessarily support independent development of modules. The difficulty arises because of clock-timing issues. For instance, when two modules, a producer of a value and a consumer of that value, are connected with wires, and the producer is optimized to use fewer clock cycles to produce the value, then the consumer should also be modified to consume the values faster; failure to do so will result in an incorrect overall design. Bluespec [4, 2], in addition to modularity, allows designers to develop hardware based on the notion of guarded atomic actions (GAAs) [10]. GAAs simplify

hardware design and its corresponding verification.

Following the concepts of modularity and GAA, we have been defining a framework called Kami [1, 12], which is for specifying, verifying, and synthesizing Bluespec-style hardware components. Kami presents a domain specific language similar to Bluespec. In addition, we define the formal semantics of the Kami language to enable proving properties of hardware. The framework has been built on the Coq proof assistant; hence, verification can be performed by mechanized proofs with a high degree of automation.

The notion of modularity used in designing hardware can be extended to verification. For modular verification, we want to be able to verify the individual modules of a large system and compose their proofs to verify the full system. In many cases, we reuse such simple components in different larger designs. From the perspective of proof, reusing a hardware component implies that we can also reuse its related proofs. Thanks to the modular semantics in Kami, it is indeed possible to use the proof of a component whenever it is used in a larger design.

Modules communicate with other modules by calling each other's methods. In our modular semantics, we model these communications using *labels*. The semantics of these communicating modules is similar to those of labeled transition systems (LTS) [8]. Each module changes its internal state and potentially calls methods of other modules during its state transition. These method calls become labels in our semantics. Modular semantics, therefore, also defines behaviors of *open systems*. Open systems have external interactions, whose specifics cannot be determined unless they are connected with other modules which can respond to them.

However, modular semantics has an inherent weakness in that it is hard to infer internal changes. Even if modular semantics allows us to consider individual modules, we should eventually be able to understand a part of a design that is still composed of several modules when the part cannot be further decomposed. Modular semantics defines behaviors of such modules by giving the behaviors of each small module. In this case, we should consider *all possible combinations* of the small module behaviors. However, we certainly do not want to consider all such cases, since we know only a

few cases are feasible by the design itself.

In this thesis, I present a new semantic approach based on inlining. Inlining semantics is defined for open hardware systems and resolves the weakness by construction. The semantics uses a static inlining operation to substitute internal calls to their method bodies. This operator erases all internal calls in a module, so that we do not need to consider internal communications.

An implication from modular semantics to inlining semantics is also formally proven. Thus, it can be used in order to efficiently prove properties of hardware systems. Since the two semantics do not have equal capabilities, we give the implication proof saying that modular semantics implies inlining semantics.

To sum up, the main contributions of this thesis are

- To define a new semantic approach based on inlining, for open hardware systems.
- To prove the implication from the modular semantics to the inlining semantics.

Overview The thesis is organized as follows: Chapter 2 introduces a number of prerequisites to understanding the hardware design concept of Bluespec. Related works are also provided to compare previous approaches to hardware semantics. Chapter 3 presents the two semantics used in the Kami framework: modular and inlining semantics. Chapter 4 compares the capability of each semantics, and presents the implication proof between them. Lastly, we draw conclusions and consider future work in Chapter 5.

Chapter 2

Background

2.1 Hardware Design Paradigms

Traditional register-transfer level (RTL) designs require hardware designers to consider scheduling for all resources. During hardware design, any resource elements must be correctly employed, *e.g.,* one should not write a register more than once in the same cycle, which would cause an uncertain behavior. If a particular input port of a hardware component is used more than once, we do not have any guarantees for the output value, which also causes an uncertain use of values. A well-known RTL design framework, Verilog [9], lets users take responsibility for this, and they should carefully investigate if there is an assignment that makes an uncertain state transition.

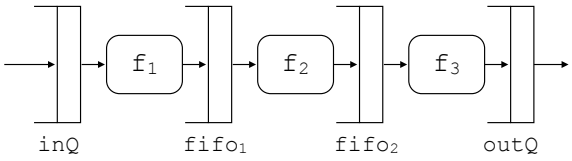


Figure 2-1: A pipelined system

A disadvantage of this approach is that having no separation between functional parts and scheduling logic makes maintenance harder. Let us revisit the pipelined system where three systems, f_1, f_2, and f_3, are connected by fifos, as shown in Figure 2-1. For simplicity, suppose that only one element can reside in a fifo. Since

there is only one element in a fifo, push and pop cannot be requested in the same cycle, since it causes a double-write. Now the problem occurs because we have no information whether push and pop will be requested simultaneously or not. In this case, we cannot avoid an implementation tightly coupled with scheduling logic; for each case of push and pop, there is no way for user except to define control flags for push and pop.

For this reason, in some frameworks like Verilog, designers tend to specify the data path first and then define the controlling logic that is entangled with the data path design. The problem with this conventional design approach is that it is not flexible from the perspective of verification. If a designer specifies the controlling logic clearly for a hardware system, it would be easily verified. However, since the specification is only for the parcicular hardware system, the verification approach used for the system may not be reusable for other systems, even if they have similar designs.

Guarded atomic action (GAA) is a design paradigm used by Bluespec for correct and effective scheduling that makes up for shortcomings of the RTL designs. It is different from the traditional RTL designs mentioned above. The main idea is that any hardware system has a (structural) state component that can be captured by a set of variables that represent registers or storage. State transition is done by a set of rules, where a rule is a series of actions with *guards* on this state. A guard of a rule is a predicate that should be satisfied to execute the rule. Each action (including one for rules) should be atomic – an execution of an action should be guaranteed to make a state transition purely caused by the action.

2.2 Semantics of the Bluespec Language

Module structure Modularity forms the base semantics of the Bluespec language. It is simply to follow a natural hardware design concept. Designers usually develop complex hardware components in a modular manner; small module components are designed first, and they are composed to build a large and complex one.

Bluespec supports such module definitions. Each module has a set of registers,

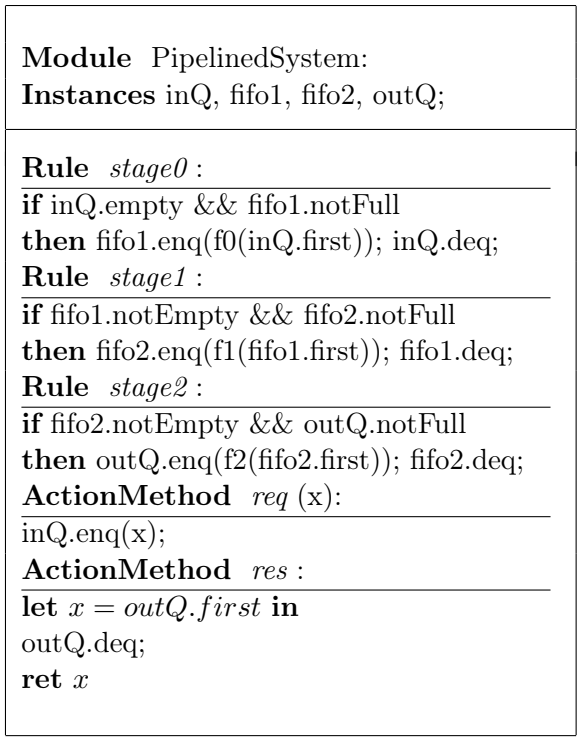


Figure 2-2: The pipelined system implemented with Bluespec

called the *internal state* of the module. The internal state can be modified only by the module itself. In order to induce a state change from the outside of the module, it should be requested by method calls. The module also has a set of *rules*. Each rule is composed of GAAs. Rules are executed by a global rule scheduler. The scheduler selects the maximal number of rules where the guards of the rules do not conflict with each others. Once the scheduler confirms the validity of the guards, they are executed concurrently. Lastly, the module has a set of *methods*. Bluespec separates the notion of *Method* and *ActionMethod*. Method is like a macro, which evaluates an expression. From the perspective of hardware, Method can be synthesized to a combinational circuit. Whereas, ActionMethod is composed of GAAs, which can be called only by external modules. Thus, it acts like an interface of the module. ActionMethod is the only way to manipulate internal states from outside.

Figure 2-2 describes the psuedo-code Bluespec module definition of the pipelined system presented in Figure 2-1. The big module PipelinedSystem has four module instances, named inQ, fifo1, fifo2, and outQ. It has three rules (stage0, stage1, and

stage2); each rule pulls an element from a fifo, applies an operation, and pushes the resulting value to the next fifo. The module has two interface methods: "req" is called when there is a request with argument "x", and "res" is called to obtain the result value.

Well-formedness of a module Syntactic correctness of a module does not imply correct execution. In other words, the module should satisfy an additional number of conditions in order for valid execution. First, a rule or a method should not perform a write operation to the same register twice (double-write). It also should not call the same method twice (double-call). This is simply because actions in a rule (a method) are executed simultaneously. A second condition is that method calls cannot form a cycle among modules, since the cycle has no valid behaviors in terms of hardware design. The last condition is that register reads and writes should be defined within the module in which the register is defined.

These conditions are sometimes called the well-formedness of a module. It is usually defined independently with semantics, since it can be captured statically by traversing the module definition. We formally define the well-formedness on Section 3.4, and use it on the correctness proof of an inlining operation. See the section for more details.

Concurrent rule executions A guard of a rule includes facts that which registers are written and which methods are called. Since the guard contains conditions to execute the rule, it is natural for the guard to have conditions for register writes and method calls. If there are no explicit guard conditions on the definition of a rule, then the guard consists only of register and method conditions. Guards contain such conditions in order to ensure that there are no double-writes or double-calls during multiple rule executions. Even if they are caught by forcing the well-formedness condition, it is restricted within a rule or a method. We cannot statically ensure if multiple rules can be executed concurrently without violating double-writes and double-calls.

Based on the guards of rules, a rule scheduler tries to find a maximal set of guards for the rules to be executed concurrently. The guards of concurrently executed rules do not conflict with each other. For instance, all the rules of PipelinedSystem can be executed concurrently. That is, the rule scheduler can verify that the guards of the three rules in PipelinedSystem do not conflict. This is because the rules do not write the same register and do not call the same method.

One-rule-at-a-time semantics Multiple rule executions require an additional condition other than non-conflicting guards, which is called the one-rule-at-a-time semantics. The condition says that multiple rules can be executed if there *exists a permutation of the rules* that yields the same state change if they are executed sequentially. More formally, when we define $(r_1; r_2; \cdots; r_n)$ to be the concurrent execution of the rules, then there exists $\{r_{p_1}, r_{p_2}, \cdots, r_{p_n}\}$, a permutation of the rules, such that

$$\forall s.(r_1; r_2; \cdots; r_n)(s) = r_{p_n}(\cdots(r_{p_2}(r_{p_1}(s)))\cdots),$$

where s is an internal state of the module.

One-rule-at-a-time semantics makes verification easier. As the name indicates, it is fine to consider semantics where at most one rule is executed at a time. Once we verify some properties under the one-rule-at-a-time semantics, the whole system, in which a rule scheduler is attached, can also be verified simply by proving that the scheduler always selects rules based on the one-rule-at-a-time semantics.

2.3 Previous Semantic Designs

Operational semantics An operational big-step semantics for a Bluespec-like language has been defined [7], but not for verification. The semantics was defined to explain how multiple rules are executed by a rule scheduler, using the notion of rule composition. The target language is BTRS, a Bluespec program after type checking and module instantiations. BTRS is defined in order to make the language and the rule scheduling algorithm deterministic. However, no formal proofs are presented for the correctness of the rule scheduling algorithm by using the semantics. Furthermore,

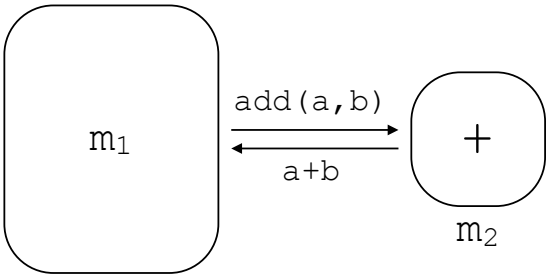


Figure 2-3: A weakness of modular semantics

the semantics is defined only for *closed systems*, not for *open systems*. A closed system implies that all methods that are called are explicitly defined somewhere. In contrast, an open system allows method calls where the methods are not yet defined. When a semantics is defined only for closed systems, it is not suitable for scalable verification since it always requires a fully defined system.

The formal semantics for open hardware systems has subsequently been defined based on modularity [12]. The core concept of the semantics is modularity; it defines the behaviors of small basic modules, and composes them in order to define the behaviors of a big module composed of the small ones. For such modular designs, the semantics should be able to describe behaviors of open systems, since small modules may call methods outside of them. Modular semantics has been formally defined and used in the Kami framework, described by the Coq proof assistant.

However, modular semantics has an inherent weakness in that it is hard to infer the internal state changes by looking only at external communications. The semantics of a composition of two modules can be understood by simply substituting the bodies of the methods in the two modules at their respective places of calls, like β-substitution. The modular semantics, however, does not perform such a syntactic substitution operation. Instead, it composes the effects of the body of the method at its call site.

This definition poses a major problem in our verification effort. For example, as shown in Figure 2-3, suppose we have a module m_1 that calls the `add` method of module m_2 with two arguments and performs a state transition based on the result. The semantics of the composed module $m_1 + m_2$ is defined using the semantics of the body of the `add` method and the semantics of the state transition in m_1. Any analysis

of the composed module requires analyzing these two effects separately and finally combining the effects. However, it is much simpler to analyze the *inlined* version of $m_1 + m_2$, where the body of method `add` is substituted into the body of the state transition of m_1.

The two semantic concepts have trade-offs. If we use semantics defined as a big-step manner (like the one in [7]), then the semantics does not have such a weakness, since it provides deterministic inference rules for drawing state changes. However, it is now difficult to introduce modular reasoning with the big-step semantics, since it is not defined for open systems. On the other hand, modular semantics can describe open system behaviors, but it has the weakness mentioned above. The main contribution of this thesis is to develop a new semantic definition that maintains the modularity concept but resolves the weakness.

Inlining The basic idea of the new semantic definition is to take advantage of a static inlining operation. We maintain modular semantics, and use inlining to concretize all internal communications by substituting method calls to their action bodies. The detailed inlining operation is defined in Section 3.5.

Static inlining is not a new idea in Bluespec. It has been defined for merging rules and methods [11]. It was used to convert between two Bluespec-like languages, where one has a module hierarchy, while the other does not have the hierarchy. The purpose of the use of inlining is similar in that it *flattens* modules to analyze them efficiently. However, the inlining operation defined in this thesis is more complicated in that it is defined for open systems. In other words, the inlining does not affect communications with external modules. Detailed proofs (called the correctness of the inlining operation) are provided in Chapter 4.

Chapter 3

An Inlining Approach to Hardware Semantics

In this chapter, we present a formal definition of inlining. First, we define the inlining operation on the Kami language and the modular semantics [12]. We then present a new semantic approach, which is based on the inlining operation.

Notations Before defining syntax and semantics, we introduce a number of notations to represent general data structures.

- Lists: we use a symbol $\vec{t}$ to represent a type of list whose elements are represented as a value t. [] represents the nil list, and $(hd :: tl)$ represents the list with a head hd and a tail tl.

- Sets: $\varnothing$ represents an empty set, $(s \cup t)$ represents the union of sets s and t. $(a \in S)$ denotes an element a in set S.

- Finite maps: we use a symbol $(K \xrightarrow{\text{fin}} V)$ to represent a finite map with a set of keys K and a set of values V. [] represents an empty map, and $m[k \leftarrow v]$ represents a map update with a new key k and a value v. Singletons are represented by $[k \leftarrow v]$. $(m \setminus ks)$ represents a map where all elements whose keys are in the list ks are removed from m. We use the notation $\in$ also for finite

maps to represent that a map has a certain key; $(k \in m)$ means that a map m has a value for a key k.

3.1 Syntax

Expressions We begin by defining syntax for expressions. Expressions consist of constant, register read, variable, and an inductive operation among expressions.

Definition 1. An expression e is defined inductively as follows:

$$
\begin{array}{lrlll}
\text{Expression} & e & ::= & c & \text{(constant)} \\
 & & \mid & r & \text{(register read)} \\
 & & \mid & x & \text{(variable)} \\
 & & \mid & \mathsf{op}(\overrightarrow{e}) & \text{(operation)}
\end{array}
$$

Constants denote all concrete values used in hardware design. A number of constants with different bit-widths may exist in real hardware design, but they are abstracted into a single type for convenience. Let $\mathcal{C}$ be the set of such constants. $c \in \mathcal{C}$ will be used as a typical character representing a constant throughout the chapter.

$r \in \mathcal{R}$ in expressions represents the value of a register r, where $\mathcal{R}$ is the set of register names. x denotes a variable bound by a continuation. We define syntax as a continuation-passing style (CPS), hence variables are used to get the value of a continuation argument. Lastly, $\mathsf{op}(\overrightarrow{e})$ abstracts all operations among expressions.

Actions As explained in Section 2.1, an action is a unit for describing hardware behaviors, which should ensure atomicity. Actions consist of register write, method call, let bind, conditional branch, assert, and return.

Definition 2. An action a is defined inductively as follows:

$$
\begin{array}{llll}
\text{Action} & a & ::= & r := e\,;\ a & \text{(register write)} \\
& & \mid & f(e)\,;\ \lambda x.a & \text{(method call)} \\
& & \mid & \mathsf{let}\ e\ \mathsf{in}\ \lambda x.a & \text{(let bind)} \\
& & \mid & \mathsf{if}\ e\ \mathsf{then}\ a\ \mathsf{else}\ a\,;\ \lambda x.a & \text{(conditional branch)} \\
& & \mid & \mathsf{assert}\ e\,;\ a & \text{(assert)} \\
& & \mid & \mathsf{return}\ e & \text{(return)}
\end{array}
$$

A register write action takes a register name r and an expression e to assign the evaluated value of e to r. A method call action takes a method name $f \in \mathcal{F}$ to call, and an expression e as an argument. A return value for the call is bound to a variable x in a lambda continuation. $\mathcal{F}$ denotes the set of method names. A let bind action gives a name to an expression e. The name is bound to a variable x in a lambda continuation. A conditional branch action similarly has a lambda binder x to capture the return value from true and false branches. An assert action takes an expression e to be checked to progress the continuation. Lastly, a return action takes e as a return value.

Even if actions are defined in a sequential manner, they are in fact executed concurrently. Semantically, all inference rules for actions will use the current state to obtain values from registers, which implies that no actions use any updated values during the execution (see Definition 6 for details).

Due to the concurrent execution concept, when we *concatenate* two arbitrary action sequences, we can easily imagine the behavior of the concatenated actions simply by merging two corresponding behaviors. The action concatenation is used for various purposes such as inlining a method body. Definition 3 defines a syntactic action concatenation operator. A semantic property will be proven in Lemma 32.

Definition 3. $(::) : \text{Action} \to \text{Action} \to \text{Action}$ is an infix operator, defined inductively with respect to the left-hand side action:

$$
\begin{aligned}
(r := e\,;\ a) :: \lambda x.a_c &\triangleq r := e\,;\ (a :: \lambda x.a_c) \\
(f(e)\,;\ \lambda x.a) :: \lambda x.a_c &\triangleq f(e)\,;\ \lambda x.(a :: \lambda x.a_c) \\
(\mathsf{let}\ e\ \mathsf{in}\ \lambda x.a) :: \lambda x.a_c &\triangleq \mathsf{let}\ e\ \mathsf{in}\ \lambda x.(a :: \lambda x.a_c) \\
(\mathsf{if}\ e\ \mathsf{then}\ a_t\ \mathsf{else}\ a_f\,;\ \lambda x.a) :: \lambda x.a_c &\triangleq \mathsf{if}\ e\ \mathsf{then}\ a_t\ \mathsf{else}\ a_f\,;\ \lambda x.(a :: \lambda x.a_c) \\
(\mathsf{assert}\ e\,;\ a) :: \lambda x.a_c &\triangleq \mathsf{assert}\ e\,;\ (a :: \lambda x.a_c) \\
(\mathsf{return}\ e) :: \lambda x.a_c &\triangleq \mathsf{let}\ e\ \mathsf{in}\ \lambda x.a_c
\end{aligned}
$$

The only nontrivial definition comes from the return statement (return e). Concatenating two actions involves to pass the return value of the left action to the right action as an argument. However, a dynamic evaluation of an expression cannot be involved with the static concatenation operation. The problem can be solved by using let syntax structure, saying that we declare a name in which e is bound, and the value is used in the body ($\lambda x.a_c$). Considering the semantic rule for let, it is natural to define concatenation in such a way.

Modules A module is a unit of hardware systems which has its own state (a set of registers), rules to induce internal state changes, and methods to communicate with other modules.

Definition 4. A module m is inductively defined as follows:

$$
\begin{array}{rrcll}
\text{Module} & m & ::= & (\overrightarrow{(r,c)}, \overrightarrow{(s,a)}, \overrightarrow{(f,\lambda x.a)}) & \text{(basic module)} \\
 & & | & m \oplus m & \text{(composed module)}
\end{array}
$$

A module is either a *basic module* or a *composed module*. In a basic module, $\overrightarrow{(r,c)}$ represents registers where the initial value of register r is c. $\overrightarrow{(s,a)}$ represents rules where s is the name of a rule and a is the body of it. $\overrightarrow{(f,\lambda x.a)}$ represents methods where f is the name of a method and $\lambda x.a$ is the body of it. x in the lambda form serves to take an argument of the method.

All names for registers, rules, and methods are assumed to be globally unique in the module. A simple static checker can be implemented to confirm that there are no name conflicts for all modules.

3.2 Modular Semantics

In this section, we present a modular semantics for hardware systems. A version of modular semantics has been defined in [12]. However, this section defines a slightly different version, which is also used in the current Kami framework. The motivation is unchanged: semantics is defined with respect to module definitions, which includes the one for combined modules. The semantics defined in this section will be used throughout the thesis, including the implication proof in Chapter 4. Specifically, semantic definitions for expressions and actions are completely borrowed from [12]. Other higher semantics differ.

Expressions We start by defining semantics for expressions. Definition 5 describes deterministic and denotational semantics $[\![\cdot]\!]_{\mathsf{e}}$ for expressions.

Definition 5. $[\![\cdot]\!]_{\mathsf{e}} : (\mathcal{R} \xrightarrow{\text{fin}} \mathcal{C}) \to \mathcal{C}$ is defined as follows:

$$\begin{aligned}
[\![c]\!]_{\mathsf{e}}\ o &= c \\
[\![r]\!]_{\mathsf{e}}\ o &= o(r) \\
[\![x]\!]_{\mathsf{e}}\ o &= \mathsf{undefined} \\
[\![\mathsf{op}(\overrightarrow{e})]\!]_{\mathsf{e}}\ o &= [\![\mathsf{op}]\!](\overrightarrow{[\![e]\!]_{\mathsf{e}}\ o})
\end{aligned}$$

Semantics for constant and register read are straightforward. The constant itself is returned for a constant. Register read is performed by reading its value from the *old state* $o : \mathcal{R} \xrightarrow{\text{fin}} \mathcal{C}$.

Semantics for variable is not defined, since the variable is always substituted to the value by β-reduction at higher semantics, before we need the semantic definition for it. Technically, in the Kami framework, syntax is defined with PHOAS [6] terms, thus the variable term is abstracted into a syntax constructor Var. In this case, we can define $[\![\mathsf{Var}\ v]\!]_{\mathsf{e}}\ o = v$, where v is a value already substituted but still contained in the Var constructor.

Lastly, semantics for operation is defined by using the semantic definition for op, denoted as $[\![\mathsf{op}]\!]$. Arguments are inductively evaluated by the same semantic function.

Actions Actions are the basic unit where the communication among modules is triggered by method calls. As defined in Definition 2, actions contain a method call. A method call can be either internal (calling a method in the module), or external (calling a method not in the module).

In the modular semantics, all method calls are first treated as they are external calls. In other words, the semantics for method calls, on the level of action, is defined as if we know the return value of the method for every argument. The validity of such an assumption is checked at higher-level semantics.

Definition 6. The judgment $o \vdash (a) \Downarrow \langle u, cs, v \rangle$ is defined as follows:

$$\text{ActionWriteReg:}\frac{[\![e]\!]_{\mathsf{e}}\ o = v_r \qquad o \vdash (a) \Downarrow \langle u, cs, v\rangle}{o \vdash (r := e\,;\ a) \Downarrow \langle u[r \leftarrow v_r], cs, v\rangle}$$

$$\text{ActionCall:}\frac{[\![e]\!]_{\mathsf{e}}\ o = v_a \qquad o \vdash ((\lambda x.a)\ v_r) \Downarrow \langle u, cs, v\rangle}{o \vdash (f(e)\,;\ \lambda x.a) \Downarrow \langle u, cs[f \leftarrow (v_a, v_r)], v\rangle}$$

$$\text{ActionLet:}\frac{[\![e]\!]_{\mathsf{e}}\ o = v_l \qquad o \vdash ((\lambda x.a)\ v_l) \Downarrow \langle u, cs, v\rangle}{o \vdash (\mathsf{let}\ e\ \mathsf{in}\ \lambda x.a) \Downarrow \langle u, cs, v\rangle}$$

$$\text{ActionIfElseT:}\frac{[\![e]\!]_{\mathsf{e}}\ o = \mathsf{true} \qquad o \vdash (a_t) \Downarrow \langle u_t, cs_t, v_t\rangle \qquad o \vdash ((\lambda x.a)\ v_t) \Downarrow \langle u, cs, v\rangle}{o \vdash (\mathsf{if}\ e\ \mathsf{then}\ a_t\ \mathsf{else}\ a_f\,;\ \lambda x.a) \Downarrow \langle u_t \cup u, cs_t \cup cs, v\rangle}$$

$$\text{ActionIfElseF:}\frac{[\![e]\!]_{\mathsf{e}}\ o = \mathsf{false} \qquad o \vdash (a_f) \Downarrow \langle u_f, cs_f, v_f\rangle \qquad o \vdash ((\lambda x.a)\ v_f) \Downarrow \langle u, cs, v\rangle}{o \vdash (\mathsf{if}\ e\ \mathsf{then}\ a_t\ \mathsf{else}\ a_f\,;\ \lambda x.a) \Downarrow \langle u_f \cup u, cs_f \cup cs, v\rangle}$$

$$\text{ActionAssert:}\frac{[\![e]\!]_{\mathsf{e}}\ o = \mathsf{true} \qquad o \vdash (a) \Downarrow \langle u, cs, v\rangle}{o \vdash (\mathsf{assert}\ e\,;\ a) \Downarrow \langle u, cs, v\rangle}$$

$$\text{ActionReturn:}\frac{[\![e]\!]_{\mathsf{e}}\ o = v}{o \vdash (\mathsf{return}\ e) \Downarrow \langle [], [], v\rangle}$$

Semantics for actions has a form of a judgment relation $o \vdash (a) \Downarrow \langle u, cs, v\rangle$, where $o : \mathcal{R} \xrightarrow{\text{fin}} \mathcal{C}$ is the old state, a is the target action, $u : \mathcal{R} \xrightarrow{\text{fin}} \mathcal{C}$ is the *updated state* after executing a, $cs : \mathcal{F} \xrightarrow{\text{fin}} \mathcal{C} \times \mathcal{C}$ is the map from method names to pairs of

argument and return values. Lastly, v is the return value of a.

Semantics for actions does not check double-writes of registers or double-calls of method calls. For instance, in the rule ActionWriteReg, an evaluated value is simply updated to the updated state u by the register name r, without confirming u does not have the value for r. Similarly, ActionCall does not check cs already has the value for f. As explained in Section 2.2, we separately define semantics and well-formedness conditions. Detailed static well-formedness checks and properties will be discussed in Section 3.4.

An assert and a return action have straightforward semantics. Assert requires its argument e to be true, in order to continue execution. There is no semantics when e is evaluated to false. Return simply evaluates the expression e to return the value.

Substep Once the semantics for actions is defined, it should be used to define the semantics for rules or methods. As explained in Section 2.2, Bluespec and Kami follow the one-rule-at-a-time semantics; only one rule is executed in a cycle, while multiple method calls are allowed to be executed. The Substep semantics defines a single execution by a rule or a method. Definitions are simply lifted from the semantics for actions, since a rule or a method is defined from actions.

A notion of label is used in this semantics, in order to represent communications with other external modules. A label has the form $\langle \alpha, ds, cs \rangle$, where α is a tag to indicate whether the label is formed by a rule or not. If the label is formed by a rule $s \in \mathcal{L}$, it takes $(\mathsf{Rule}\ s)$. Otherwise, it takes Meth. ds denotes *defined methods*, which are executed in an atomic action. Similarly, cs denotes *called methods*, which are called in an atomic action.

The Substep semantics has a form of a judgment relation $\langle m, o \rangle \Downarrow \langle u, \langle \alpha, ds, cs \rangle \rangle$, where m is the target module, o is the old state, u is the updated state, and $\langle \alpha, ds, cs \rangle$ is the label formed by the execution.

Definition 7. The judgment $\langle m, o \rangle \Downarrow \langle u, \langle \alpha, ds, cs \rangle \rangle$ is defined as follows:

$$\text{EmptyRule:}\frac{}{\langle m, o\rangle \Downarrow \langle [], \langle \mathsf{Rule}\ \epsilon, [], []\rangle\rangle}$$

$$\text{EmptyMeth:}\frac{}{\langle m, o\rangle \Downarrow \langle [], \langle \mathsf{Meth}, [], []\rangle\rangle}$$

$$\text{SingleRule:}\frac{(s, a) \in \mathsf{rulesOf}\ m \qquad o \vdash (a) \Downarrow \langle u, cs, v\rangle}{\langle m, o\rangle \Downarrow \langle u, \langle \mathsf{Rule}\ s, [], cs\rangle\rangle}$$

$$\text{SingleMeth:}\frac{(f, \lambda x.a) \in \mathsf{methodsOf}\ m \qquad o \vdash ((\lambda x.a)\ v_a) \Downarrow \langle u, cs, v_r\rangle}{\langle m, o\rangle \Downarrow \langle u, \langle \mathsf{Meth}, [f \leftarrow (v_a, v_r)], cs\rangle\rangle}$$

EmptyRule and EmptyMeth describe the cases where no progress is made in a cycle. An empty step by a rule and one by a method should be separately defined in the Substep semantics, in order to maintain the information needed when defining the refinement relation [12].

SingleRule and SingleMeth describe the case where a rule or a method is executed in a cycle, respectively. In SingleRule, a rule (s, a) should be defined in the module m. $\mathsf{rulesOf}\ m$ is a function which collects all rules in the module, inductively defined as follows:

Definition 8.

$$\begin{aligned} \mathsf{rulesOf}\ (\_, \overrightarrow{(s, a)}, \_) &= \overrightarrow{(s, a)} \\ \mathsf{rulesOf}\ (m1 \oplus m2) &= (\mathsf{rulesOf}\ m1) \cup (\mathsf{rulesOf}\ m2) \end{aligned}$$

Similarly, a method $(f, \lambda x.a)$ should be defined in m in the SingleMeth case. $\mathsf{methodsOf}\ m$ is also similarly defined as follows:

Definition 9.

$$\begin{aligned} \mathsf{methodsOf}\ (\_, \_, \overrightarrow{(f, \lambda x.a)}) &= \overrightarrow{(f, \lambda x.a)} \\ \mathsf{methodsOf}\ (m1 \oplus m2) &= (\mathsf{methodsOf}\ m1) \cup (\mathsf{methodsOf}\ m2) \end{aligned}$$

Once a rule or a method is found, proper labels are formed by using information from the action semantics.

Substeps The "Substeps" semantics collects multiple substeps which are concurrently executed in a cycle. There are a few conditions whether substeps can be merged or not. Conditions are about the updated states and labels from Substep. Firstly, two substeps with (Rule $\cdot$) labels cannot be merged, since it breaks the one-rule-at-a-time semantics. Two updated states should be disjoint unless it breaks the double-write policy. Similarly, two defined methods and two called methods should be disjoint not to break double-call.

The Substeps semantics has a form of a judgment relation $\langle m, o\rangle \Downarrow^* \langle u, l\rangle$, where arguments mostly have the same meaning as in Substep. l denotes the label formed in a cycle.

Definition 10. The judgment $\langle m, o\rangle \Downarrow^* \langle u, l\rangle$ is defined as follows:

$$\text{SubstepsNil:}\frac{}{\langle m, o\rangle \Downarrow^* \langle [], \langle \mathsf{Meth}, [], []\rangle\rangle}$$

$$\text{SubstepsCons:}\frac{\langle m, o\rangle \Downarrow^* \langle u_1, l_1\rangle \quad \langle m, o\rangle \Downarrow \langle u_2, l_2\rangle \quad u_1 * u_2 \quad l_1 * l_2}{\langle m, o\rangle \Downarrow^* \langle u_1 \cup u_2, l_1 \uplus l_2\rangle}$$

In SubstepsCons, disjointnesses of states and labels are both denoted by the operation $*$, while definitions differ. For states u_1 and u_2, $u_1 * u_2$ defines the two finite state maps are disjoint, which can be defined naturally.

Definition 11.

$$u_1 * u_2 \triangleq \forall r. r \notin u_1 \vee r \notin u_2$$

For labels l_1 and l_2, $l_1 * l_2$ defines the two labels are combinable.

Definition 12.

$$\begin{aligned} l_1 * l_2 &\triangleq (\mathsf{annot}\ l_1) * (\mathsf{annot}\ l_2) \wedge (\mathsf{defs}\ l_1) * (\mathsf{defs}\ l_2) \wedge (\mathsf{calls}\ l_1) * (\mathsf{calls}\ l_2), \text{ where} \\ \alpha_1 * \alpha_2 &\triangleq (\alpha_1 = \mathsf{Meth}) \vee (\alpha_2 = \mathsf{Meth}) \end{aligned}$$

Using the disjointness conditions, we define a label-merging operation $(l_1 \uplus l_2)$, defined as follows:

Definition 13.

$$l_1 \uplus l_2 \triangleq \langle \mathsf{annot}\ l_1 \uplus \mathsf{annot}\ l_2, \mathsf{defs}\ l_1 \cup \mathsf{defs}\ l_2, \mathsf{calls}\ l_1 \cup \mathsf{calls}\ l_2 \rangle, \text{ where}$$
$$\alpha_1 \uplus \alpha_2 \triangleq \mathsf{if}\ \alpha_1 = \mathsf{Rule}\ s\ \mathsf{then}\ \alpha_1\ \mathsf{else}\ \alpha_2$$

The only nontrivial definition is $(\alpha_1 \uplus \alpha_2)$, which takes rule annotations from two annotations. It returns Meth if both of them are not a rule annotation. Note that the two annotations cannot both be the rule annotation by the condition $(\alpha_1 * \alpha_2)$, which is following the concept of the one-rule-at-a-time semantics.

Step The Step semantics describes behaviors of a module where all internal communications are hidden. It employes the Substep semantics, and hides all internal calls of the label from the substeps. Hiding internal calls includes to check whether the calls are correctly defined, *i.e.,* if a label of substeps contains a called method f with an argument and a return value (v_a, v_r), then the label also should contain a defined method with the same name and the value.

The Step semantics has a form of a judgment relation $(o \xrightarrow[m]{l} u)$, arguments have the same meaning as in Substeps.

Definition 14. The judgment $o \xrightarrow[m]{l} u$ is defined as follows:

$$\text{StepIntro:}\frac{\langle m, o\rangle \Downarrow^* \langle u, l\rangle \qquad \mathsf{wellHidden}\ m\ (\mathsf{hide}\ l)}{o \xrightarrow[m]{l} u}$$

(hide l) computes the label where all internal (corresponding) calls are removed.

Definition 15.

$$\mathsf{hide}\ \langle \alpha, ds, cs\rangle = \langle \alpha, ds - cs, cs - ds\rangle.$$

A finite map subtraction operation $(-)$ is naturally defined; it removes all pairs (k, v) on the left-hand side map if the right-hand side map contains (k, v).

(wellHidden m l) ensures l does not contain internal calls with respect to m.

Definition 16.

$$\mathsf{wellHidden}\ m\ \langle \alpha, ds, cs \rangle \triangleq (\mathsf{domain}\ ds * \mathsf{callsOf}\ m) \wedge (\mathsf{domain}\ cs * \mathsf{methodsOf}\ m).$$

($\mathsf{callsOf}\ m$) statically collects all method names which are called in m.

Definition 17.

$$\begin{array}{lcl}
\mathsf{callsOf}\ (_, \overrightarrow{(s, a_r)}, \overrightarrow{(f, \lambda x.a_m)}) & = & (\bigcup_{(s_i, a_{r_i}) \in \overrightarrow{(s, a_r)}} \mathsf{callsOfA}\ a_{r_i}) \cup \\
& & (\bigcup_{(f_i, \lambda x.a_{m_i}) \in \overrightarrow{(f, \lambda x.a_m)}} \mathsf{callsOfA}\ a_{m_i}) \\
\mathsf{callsOf}\ (m_1 \oplus m_2) & = & \mathsf{callsOf}\ m_1 \cup \mathsf{callsOf}\ m_2 \\
\mathsf{callsOfA}\ (r := e\,;\ a) & = & \mathsf{callsOfA}\ a \\
\mathsf{callsOfA}\ (f(e)\,;\ \lambda x.a) & = & \{f\} \cup \mathsf{callsOfA}\ a \\
\mathsf{callsOfA}\ (\mathsf{let}\ e\ \mathsf{in}\ \lambda x.a) & = & \mathsf{callsOfA}\ a \\
\mathsf{callsOfA}\ (\mathsf{if}\ e\ \mathsf{then}\ a_t\ \mathsf{else}\ a_f\,;\ \lambda x.a) & = & \mathsf{callsOfA}\ a_t \cup \mathsf{callsOfA}\ a_f \cup \mathsf{calls}\ a \\
\mathsf{callsOfA}\ (\mathsf{assert}\ e\,;\ a) & = & \mathsf{callsOfA}\ a \\
\mathsf{callsOfA}\ (\mathsf{return}\ e) & = & \varnothing
\end{array}$$

3.3 Manipulating Module Structures

Before introducing an inlining operation and its corresponding semantics, we present one of main properties of modular semantics, which is also related to inlining. The property says that how modules are composed does not affect behaviors of the modules. In other words, two modules are semantically equivalent if they have the same set of registers, rules, and defined methods. It is formally described as follows:

Lemma 18.

For two modules m_1 and m_2,

$$\begin{array}{l}
\mathsf{regsOf}\ m_1 = \mathsf{regsOf}\ m_2 \rightarrow \\
\mathsf{rulesOf}\ m_1 = \mathsf{rulesOf}\ m_2 \rightarrow \\
\mathsf{methodsOf}\ m_1 = \mathsf{methodsOf}\ m_2 \rightarrow \\
(o \xrightarrow[m_1]{l} u) \rightarrow (o \xrightarrow[m_2]{l} u)
\end{array}$$

Having the same set of registers, rules, and defined methods does not imply two modules are syntactically equal. For example, two modules $(m_1 \oplus m_2) \oplus m_3$ and

$m_1 \oplus (m_2 \oplus m_3)$ have different module structures (thus syntactically different), but they have the same set of registers, rules, and defined methods (from three modules m_1, m_2, and m_3).

Proving Lemma 18 is straightforward, simply by following inductive definitions of semantics. Since all inference rules from Substep to Step do not look at the structure of modules (how they are composed), the proof does not need any tricks. Using Lemma 18, we obtain two more intuitive corollaries, which directly say that module structures are nothing to do with semantics.

Corollary 19.

For two modules m_1 and m_2,

$$(o \xrightarrow[m_1 \oplus m_2]{l} u) \to (o \xrightarrow[m_2 \oplus m_1]{l} u)$$

Proof. Straightforward by applying Lemma 18. ■

Corollary 20.

For modules m_1, m_2 and m_3,

$$(o \xrightarrow[(m_1 \oplus m_2) \oplus m_3]{l} u) \to (o \xrightarrow[m_1 \oplus (m_2 \oplus m_3)]{l} u)$$

Proof. Straightforward by applying Lemma 18. ■

The independence between module structures and modular semantics allows us to *simplify* the module structure without affecting its semantics. For instance, we can define and use a *flattening operation* for modules. The flattening operation $\overline{\cdot}$ converts a target module to one big basic module, by breaking the module structure.

Definition 21.

$$\overline{\overline{m}} \triangleq (\mathsf{regsOf}\ m, \mathsf{rulesOf}\ m, \mathsf{methodsOf}\ m)$$

It is obvious that m and $\overline{\overline{m}}$ satisfy the conditions for applying Lemma 18, thus we finally obtain the following property:

Corollary 22.

For a module m,

$$(o \xrightarrow[m]{l} u) \to (o \xrightarrow[\overline{\overline{m}}]{l} u)$$

Proof. Straightforward by applying Lemma 18. ■

Once a module is collapsed by flattening, the only concern for efficiently dealing with modules comes from internal calls. In order to handle such internal calls, we use inlining; given a flattened module, the inlining operation eliminates all internal calls by substituting their bodies to call sites. Hence, after flattening and inlining, we obtain a *raw module*, in which all internal structures are collapsed. See Definition 26 for details how the flattening operation is used during inlining.

3.4 Well-formedness of Modules

In this section, we formally define one of well-formedness conditions of modules. Well-formedness for a programming language usually means a set of conditions which can be *statically checked* with a given program. Since the conditions are static, they are usually defined independent to semantic definitions, which has a dynamic aspect. For instance, as briefly discussed in Section 2.2, it is not allowed for a rule or a method to write the same register twice, or to call the same method twice. These conditions can be statically and approximately determined by examining actions.

Well-formedness conditions should be checked not only for correct synthesis, but also for verification. Two representative well-formedness conditions are: 1) there are no double writes or double calls in each action, and 2) there are no call cycles in a module. If a hardware design does not satisfy the former, then the synthesized circuit makes structural hazards. If not satisfying the latter, then the call cycles form a combinational cycle, whose behavior is nonpredictable. Meanwhile, in terms of verification, a main reason for checking well-formedness conditions is that some semantic properties cannot be proven without them. For instance, a correctness proof of an inlining operation requires a well-formedness condition as a hypothesis (see Chapter 4 for details).

Whereas, certain conditions cannot be statically checked, thus should be defined

in part of semantics. For example, when combining two substeps, two corresponding updated states and labels are required to be disjoint, defined in the Substeps semantics. However, in some cases, it is too challenging to statically ensure the updated states and the labels are disjoint.

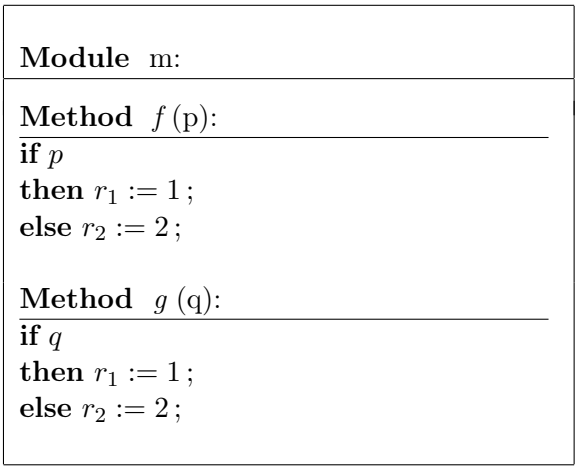


Figure 3-1: Two methods forming disjoint state updates

Figure 3-1 shows the case where two methods f and g yield disjoint state updates when $q = \neg p$, thus they can be concurrently executed. When a condition expression p is true, then f writes r_1 and g writes r_2. Otherwise, if p is false, then q is true so f writes r_2 and g writes r_1. However, it is generally challenging to provide whether two predicates (p and q in the example) are disjoint or not. How can we collect possible pairs of (p, q) if the module m is too complicated to figure out callers of f and g?

In this thesis, we define one of well-formedness conditions, which ensures that there are no double writes or double calls for given actions. For a given module m, (WfDouble m) ensures such a property.

Well-formedness for avoiding double writes and calls Even if detecting double writes or calls is difficult within a module, we can provide a *sound* static checker for actions. A sound checker implies that if it says "no double writes or calls", then it is correct. However, it might say "there exist double writes or calls" even if such double writes or calls cannot happen.

Figure 3-2 shows the case where double writes happen in some cases, but not in

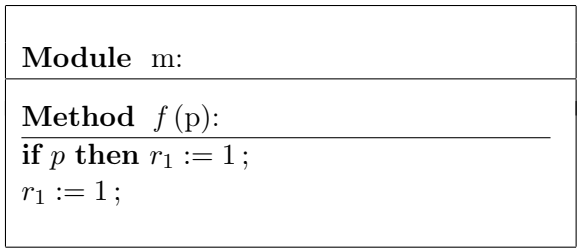


Figure 3-2: A method where double writes happen in some cases

the other cases. If p is true, then double writes happen. Otherwise, double writes do not happen. However, as explained with the case in Figure 3-1, analyzing predicates is difficult in most cases. Thus, we should define a checker in a sound manner, by assuming that p can be sometimes true.

Now we define a static checker for ensuring that there are no double writes or calls in an action. (WfDoubleA' a) is inductively defined as follows:

Definition 23.

$$
\begin{array}{lcl}
\mathsf{WfDoubleA}\ a & = & \mathsf{WfDoubleA'}\ a\ \varnothing\ \varnothing, \text{ where} \\
\mathsf{WfDoubleA'}\ (r := e\,;\ a)\ rs\ cs & = & \mathsf{if}\ r \in rs \\
 & & \mathsf{then\ false} \\
 & & \mathsf{else\ WfDoubleA'}\ a\ (rs \cup \{r\})\ cs \\
\mathsf{WfDoubleA'}\ (f(e)\,;\ \lambda x.a)\ rs\ cs & = & \mathsf{if}\ f \in cs \\
 & & \mathsf{then\ false} \\
 & & \mathsf{else\ WfDoubleA'}\ a\ rs\ (fs \cup \{f\}) \\
\mathsf{WfDoubleA'}\ (\mathsf{let}\ e\ \mathsf{in}\ \lambda x.a)\ rs\ cs & = & \mathsf{WfDoubleA'}\ a\ rs\ cs \\
\mathsf{WfDoubleA'}\ (\mathsf{if}\ e\ \mathsf{then}\ a_t\ \mathsf{else}\ a_f\,;\ \lambda x.a) & = & (\mathsf{WfDoubleA'}\ a_t :: \lambda x.a\ rs\ cs)\ \&\& \\
 & & (\mathsf{WfDoubleA'}\ a_f :: \lambda x.a\ rs\ cs) \\
\mathsf{WfDoubleA'}\ (\mathsf{assert}\ e\,;\ a)\ rs\ cs & = & \mathsf{WfDoubleA'}\ a\ rs\ cs \\
\mathsf{WfDoubleA'}\ (\mathsf{return}\ e)\ rs\ cs & = & \mathsf{true}
\end{array}
$$

(WfDoubleA a) is defined using (WfDoubleA' $a\ rs\ cs$), where rs and cs are registers and method calls collected while iterating an action, respectively. When an action is a register write or a method call, then (WfDoubleA' a) checks if the register or the method is already written or called, respectively. If not, it collects the register and the

method, and continues to check for next actions. Note that the action concatenation operation (::) is used to define the well-formedness of conditional branch actions. This is to collect register writes and called methods for true and false branches in parallel. See Definition 3 for the definition of (::).

The definition of (WfDoubleA a) is naturally extended to (WfDouble m), by applying it for each action in m.

Definition 24.

$$\begin{aligned}
&\mathsf{WfDouble}\ ((\overrightarrow{(r,c)}, \overrightarrow{(s,a_s)}, \overrightarrow{(f,\lambda x.a_f)})) &&= (\mathsf{foreach}\ a_s.\ \mathsf{WfDoubleA}\ a_s = \mathsf{true})\ \&\& \\
&&&\quad (\mathsf{foreach}\ a_f.\ \mathsf{WfDouble}\ a_f = \mathsf{true}) \\
&\mathsf{WfDouble}\ (m_1 \oplus m_2) &&= (\mathsf{WfDouble}\ m_1)\ \&\&\ (\mathsf{WfDouble}\ m_2)
\end{aligned}$$

3.5 Inlining Semantics

In this section, we formally define an inlining operation and its corresponding semantics. The main purpose of inlining is to handle possible behaviors of modules easily. As explained in Section 2.3, internal calls are the main hurdle analyzing module behaviors. However, in consequence of inlining, a module will no longer have internal calls, which implies that the inlined module is easy to analyze.

3.5.1 Inlining Operation

Inlining a method There are several ways to implement an inlining function. One way is to define it like a breadth-first search; during the iteration of a target action, called methods are inlined for each method call. This inlining process is applied repeatedly until there are no internal method calls.

We define an inlining function in a somewhat different way for proof efficiency. First, we pick a function which will be inlined throughout a target module. Then for every action in the target module, we search for all method calls to the target method, and inline all of them. This process is applied for every defined method in the module. The reason we chose this way is that inlining a single method is closely

related to semantic label manipulation. See Chapter 4 for the detailed reason why this inlining is easier to prove properties.

For a target action a and an inlining method dm, we define a method-inlining operator for an action, denoted as $(\hookleftarrow)$.

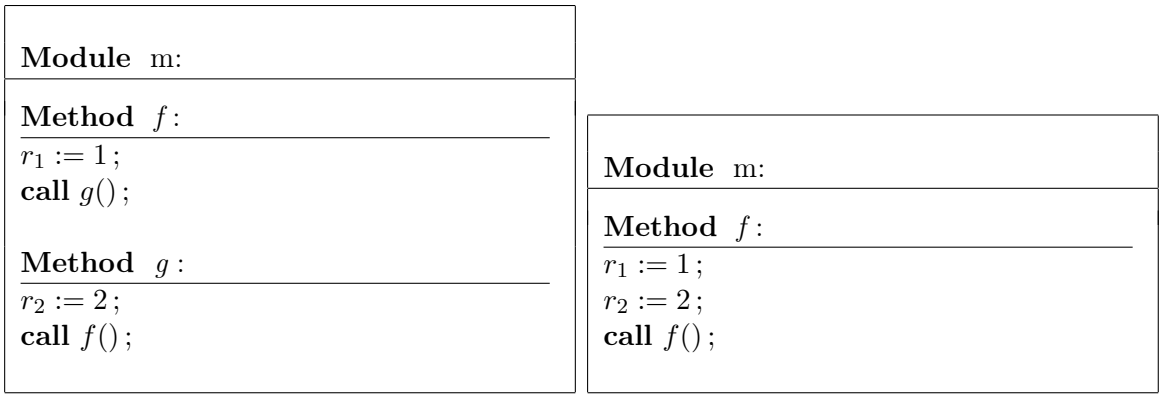
Definition 25. $(\hookleftarrow)$: Action $\to$ Method $\to$ Action is an infix operator, defined inductively with respect to the left-hand side action:

$$
\begin{aligned}
(r := e\,;\ a) \hookleftarrow dm &\triangleq r := e\,;\ (a \hookleftarrow dm) \\
(f(e)\,;\ \lambda x.a) \hookleftarrow (f_i, \lambda x.a_f) &\triangleq \textbf{if } f = f_i \\
&\quad \textbf{then } (\mathsf{let}\ e\ \mathsf{in}\ \lambda y.(\lambda x.a_f)\ y) :: \lambda x.(a \hookleftarrow (f_i, \lambda x.a_f)) \\
&\quad \textbf{else } f(e)\,;\ \lambda x.(a \hookleftarrow (f_i, \lambda x.a_f)) \\
(\mathsf{let}\ e\ \mathsf{in}\ \lambda x.a) \hookleftarrow dm &\triangleq \mathsf{let}\ e\ \mathsf{in}\ \lambda x.(a \hookleftarrow dm) \\
(\mathsf{if}\ e\ \mathsf{then}\ a_t\ \mathsf{else}\ a_f\,;\ \lambda x.a) \hookleftarrow dm &\triangleq \mathsf{if}\ e\ \mathsf{then}\ (a_t \hookleftarrow dm)\ \mathsf{else}\ (a_f \hookleftarrow dm)\,;\ \lambda x.(a \hookleftarrow dm) \\
(\mathsf{assert}\ e\,;\ a) \hookleftarrow dm &\triangleq \mathsf{assert}\ e\,;\ (a \hookleftarrow dm) \\
(\mathsf{return}\ e) \hookleftarrow dm &\triangleq \mathsf{return}\ e
\end{aligned}
$$

A method-inlining operator for a module is naturally defined by extending the operator for an action. For a target module m and an inlining *method name* f, we define a method-inlining operator for a module, denoted as $(\hookleftarrow_m)$. Note that module is flattened (as defined in Definition 21) during the inlining process.

Definition 26. $(\hookleftarrow_m)$: Module $\to \mathcal{F} \to$ Module is an infix operator, defined as follows:

$$
\begin{aligned}
m \hookleftarrow_m f \triangleq\ & \textbf{let } dm = (\mathsf{methodsOf}\ m)[f] \textbf{ in} \\
& \textbf{if } \mathsf{isRecursive}\ dm \textbf{ then } \mathsf{fail} \\
& \textbf{else} \\
& \textbf{let } \overrightarrow{(r, c)} = \mathsf{regsOf}\ m \textbf{ in} \\
& \textbf{let } \overrightarrow{(s, a_s)} = \mathsf{rulesOf}\ m \textbf{ in} \\
& \textbf{let } \overrightarrow{(f, \lambda x.a_f)} = \mathsf{methodsOf}\ m \textbf{ in} \\
& (\overrightarrow{(r, c)}, \overrightarrow{(s, (a \hookleftarrow dm))}, \overrightarrow{(f, \lambda x.(a \hookleftarrow dm))})
\end{aligned}
$$



(a) Before inlining g (b) After inlining g

Figure 3-3: A call cycle during inlining

Inlining all defined methods Using the method-inlining operator $(\hookleftarrow_m)$, now it is straightforward to define an inlining function for a module. We simply apply $(\hookleftarrow_m)$ for all defined methods in the module.

However, inlining gets complicated when there is *a call cycle* in the module. Figure 3-3 shows the case where inlining for such a module is problematic. A module m has two methods f and g. It has a call cycle since f calls g and g calls f. When inlining g first, f becomes a *self-recursive* call (a right figure). In this case, inlining for f cannot progress, since inlining itself will cause register double-writes by r_1 and r_2. In order to avoid such cases, the method-inlining operator checks whether the method which will be inlined has a self-recursive method call or not. As described in Definition 26, (isRecursive dm) ensures that dm is not self-recursive.

Now we naturally extend the method-inlining operator to a methods-inlining operator $(\hookleftarrow^*)$.

Definition 27. $(\hookleftarrow_m)$: Module $\to$ list $\mathcal{F} \to$ Module is an infix operator, defined as follows:

$$m \hookleftarrow^* [] \triangleq m$$
$$m \hookleftarrow^* f :: fs \triangleq (m \hookleftarrow_m f) \hookleftarrow^* fs$$

Using the methods-inlining operator $(\hookleftarrow^*)$, we finally define the inlining operator $|\cdot|$ for a target module m.

Definition 28. $|\cdot|$: Module $\to$ Module is defined as follows:

$$|m| \triangleq m \hookleftarrow^* (\mathsf{namesOf}\ (\mathsf{methodsOf}\ m))$$

Why are method names used for inlining, instead of actual method bodies? The reason is that when we do inlining once for a particular method, then all related method bodies are changed. In other words, when a method-inlining is done, we should take a new method which will be inlined next, by taking it from the *inlined module*, not from the original one.

Hiding internal methods The last step of inlining is to hide internal methods so that they cannot be called from external modules. It is to ensure an original module and the inlined one have the same behaviors. According to the modular semantics, all interal methods cannot be called by external methods. In terms of inlining, when all internal calls are inlined, then the module should not have interfaces for these inlined methods.

How do we know which defined methods should be hidden? The way to hide such methods is simple; we simply filter defined methods by checking whether the target method is internally called by some rules or methods.

Definition 29. $\|\cdot\| : \text{Module} \to \text{Module}$ is defined as follows:

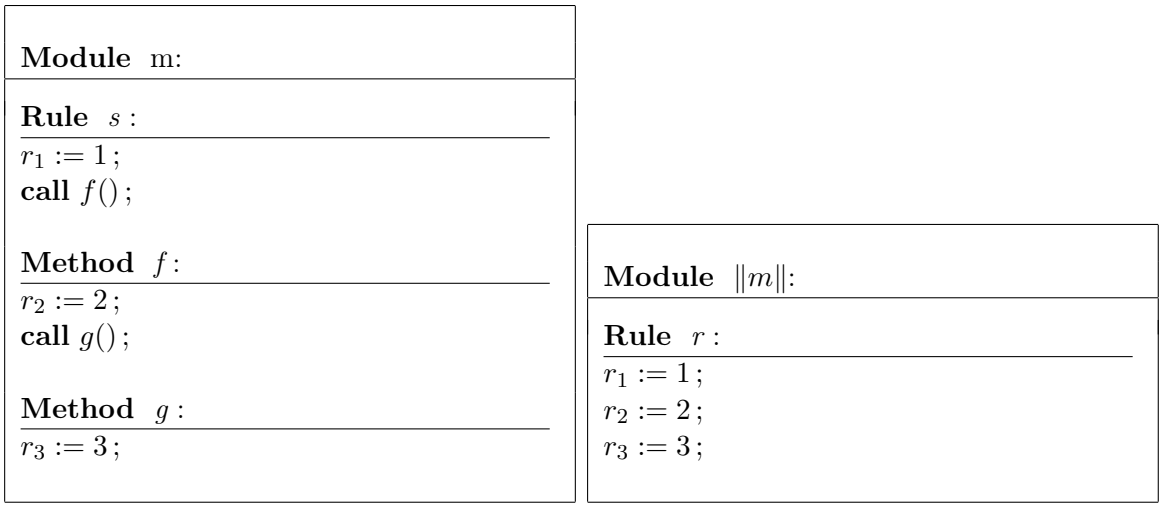
$$\|m\| \triangleq (\mathsf{regsOf}\ |m|, \mathsf{rulesOf}\ |m|, (\mathsf{methodsOf}\ |m|) \setminus (\mathsf{callsOf}\ m)),$$

where the map filtering operator $(m \setminus l)$ filters out elements of m where the key is not in l.

3.5.2 Inlining Semantics

Using the complete inlining operator, we finally define the inlining semantics. It borrows the Step semantics from the modular semantics. The StepInl semantics have a form of a judgment relation $(o \overset{l}{\underset{m}{\hookrightarrow}} u)$, arguments have the same meaning as in Step.

Definition 30. The judgment $o \overset{l}{\underset{m}{\hookrightarrow}} u$ is defined as follows:



(a) Before applying the inlining operator $\|\cdot\|$ (b) After applying the inlining

Figure 3-4: Inlining efficiency with respect to the Step semantics

$$\text{StepInlIntro:}\frac{o \xrightarrow[\|m\|]{l} u}{o \overset{l}{\underset{m}{\hookrightarrow}} u}$$

Step semantics can be efficiently handled by using inlining. In other words, dealing with Step with inlined modules is easier than ordinary modules. Figure 3-4 describes two modules, where the left module is an original module (which is not inlined), while the right one is inlined. We can clearly see that the inlined module is simpler than the original one. More formally, the inlined module is easily handled, since 1) it has fewer defined methods, and 2) the module has no internal calls.

Chapter 4

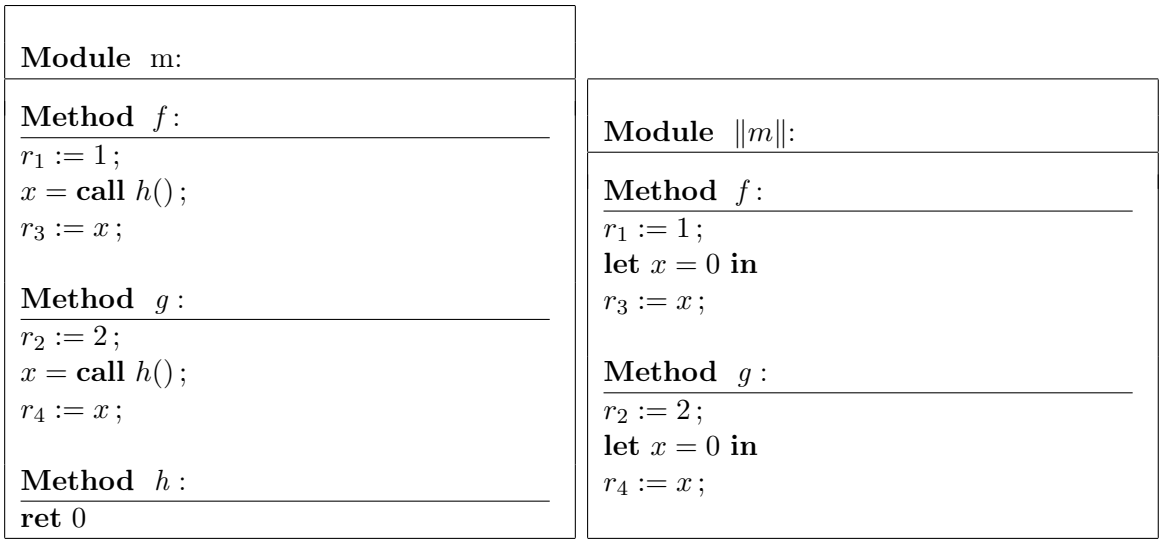
Proving the Implication from Modular to Inlining Semantics

4.1 Meaning of the Semantic Implication

The semantics defined so far does not have equal capabilities. The inlining semantics can simulate more design cases than the modular semantics. In other words, it is possible to prove an implication from the modular semantics to the inlining one, but not for the inverse.

Figure 4-1 shows the case that the inlining semantics allows a specific execution, while the modular one does not. In a module m, methods f and g cannot be executed concurrently, since both methods call a method h. More specifically, the Substeps judgement containing substeps for f and g cannot be constructed due to the disjointness condition of called methods. However, in the inlined module $\|m\|$, f and g can be executed concurrently, since h simply returns a constant zero.

Does this case hurt the capacity of the modular semantics? According to the example, when h does not write registers or call methods, any two methods both calling h should be able to be executed concurrently. This case is quite practical in real-world hardware design; for example, if m has a register r_{cnt} which acts as a counter, and h calculates the next value of the counter by returning $(r_{\text{cnt}} + 1)$, any rules and methods should be able to call h, even though they are executed



(a) A module before inlining (b) A module after inlining

Figure 4-1: Inlining semantics allows an execution of f and g, while the modular one does not.

concurrently.

We can resolve this issue by converting designs to the ones without such methods by inlining. Bluespec has two notations for methods, called **ActionMethod** and **Method**. **ActionMethod** can write registers, while **Method** cannot contain such actions. Hence, if we inline all **Method**s while converting designs from Bluespec to Kami, the issue will not happen (though the inlining in conversion will remain as a part of trusted base).

Based on the analysis so far, we give the main theorem for semantic implication. Theorem 31 claims that the set of possible behaviors of the modular semantics is a subset of the one in the inlining semantics.

Theorem 31 (Modular semantics implies inlining semantics)**.**

$$\forall m.\forall o.\forall u.\forall l.\ (o \xrightarrow[m]{l} u) \Longrightarrow (o \overset{l}{\underset{m}{\hookrightarrow}} u)$$

4.2 The Implication Proof: From Modular to Inlining Semantics

On this section, we present the proof of Theorem 31. The proof inductively follows the definition of the complete inlining operator $\|\cdot\|$. In other words, we gradually present the correctness lemma from the concatenation operator (::), which is the most basic operator for inlining, to the complete inlining operator.

All correctness lemmas have a similar form, which describes that if we have a judgement for a target object which calls a target method and a judgement for the target method, then two judgements can be *merged* by a proper inlining operator. More formally, correctness lemmas have a following form:

(a judgement calling an inlining method) $\rightarrow$
(conditions) $\rightarrow$
(a judgement of an inlining method) $\rightarrow$
(a merged judgement where the method is inlined by a certain operator).

Correctness of the concatenation operator Since the concatenation operator (::) merely appends two actions, the correctness lemma also states the corresponding semantic merge.

Lemma 32 (Correctness of the action concatenation)**.**
For every $o, a_1, u_1, cs_1, v_1, a_2, u_2, cs_2$, and v_2,

$$o \vdash (a_1) \Downarrow \langle u_1, cs_1, v_1 \rangle \rightarrow$$
$$o \vdash ((\lambda x.a_2)\ v_1) \Downarrow \langle u_2, cs_2, v_2 \rangle \rightarrow$$
$$o \vdash (a_1 :: \lambda x.a_2) \Downarrow \langle u_1 \cup u_2, cs_1 \cup cs_2, v_2 \rangle$$

Proof. Straightforward by the inductive definition of (::). ■

Note that two updates u_1 and u_2 and two called methods cs_1 and cs_2 *do not need to be disjoint* – a normal union $\cup$ is used to merge updates and method calls, instead of a disjoint union $\uplus$. This is because the action semantics allows double writes or calls. These conditions are defined as one of the well-formedness conditions, which is independent to the semantics.

Correctness of the method-inlining operator Using the correctness lemma for the concatenation operator, we give one for the method-inlining operator. The correctness lemma is described first with action structures, and lifted to Substep and Substeps.

There are two variations of the correctness lemma: the *intact case* is designed for objects which are not related to the method which is inlined. For example, if an action does not call the method, then a judgement of the action should not be changed. The *call case* is designed for objects which call the method. In this case, we will obtain a merged judgement as explained.

Lemma 33 (Correctness of $(\hookleftarrow)$ for actions – intact case)**.**
For every o, a_f, a, u, cs, v_r, and f,

$$\begin{aligned}
&o \vdash (a) \Downarrow \langle u, cs, v_r \rangle \to \\
&f \notin cs \to \\
&o \vdash (a \hookleftarrow (f, a_f)) \Downarrow \langle u, cs, v_r \rangle
\end{aligned}$$

Proof. By induction on the action judgement for a. The only nontrivial case is a method call action. If $a = (f'(e)\,;\ \lambda x.a')$, then f' cannot be equal to f since $f' \in cs$ by the inductive definition of the method call, but $f \notin cs$. Therefore, $\langle u, cs, v_r \rangle$ is never changed by the inlining. ■

Below call case is where the well-formedness condition occurs for the first time.

Lemma 34 (Correctness of $(\hookleftarrow)$ for actions – call case)**.**
For every $o, a_f, a, u_f, u, cs_f, cs, v_r, f, v_a, v_r$, and v_f,

$$\begin{aligned}
&o \vdash (a) \Downarrow \langle u, cs, v_r \rangle \to \\
&\mathsf{WfDoubleA}\ a \to \\
&u_f * u \to cs_f * cs \to \\
&cs[f] = (v_a, v_f) \to \\
&o \vdash ((\lambda x.a_f)\ v_a) \Downarrow \langle u_f, cs_f, v_f \rangle \to \\
&o \vdash (a \hookleftarrow (f, a_f)) \Downarrow \langle u_f \uplus u, cs_f \uplus cs, v_r \rangle
\end{aligned}$$

Proof. By induction on the action judgement for a. The only nontrivial case is a method call action. When $a = (f'(e)\,;\ \lambda x.a')$, we have to consider following two cases:

- If $f' = f$, then the method call should have the argument value v_a and the return value v_r, since $cs[f] = (v_a, v_f)$. Since $(\hookleftarrow)$ is defined using $(::)$, we can use the judgement for $(\lambda x.a_f)\ v_a$ and Lemma 32 to prove the goal.
- If $f' \neq f$, then it is same as the intact case.

■

When is the well-formedness condition $(\mathsf{WfDouble}\ a)$ used during the proof? Suppose there is a double call in a. If the argument-return pairs of the two calls differ, then one of them is overwritten by the other one while evaluating cs, according to the action semantics. In this case, when encountering the method call where the pair is overwritten, we cannot apply Lemma 32 since the pair does not match.

Correctness of the method-inlining operator for modules We lift Lemma 33 (the intact case) and Lemma 34 (the call case) to the level of Substep and Substeps.

Lemma 35 (Correctness of $(\hookleftarrow_m)$ for Substep – intact case)**.**
For every m, o, u, ds, cs, and f,

$$\begin{aligned}&\langle m, o\rangle \Downarrow \langle u, \langle \alpha, ds, cs\rangle\rangle \to \\ &f \notin ds \to \\ &\langle m \hookleftarrow_m f, o\rangle \Downarrow \langle u, \langle \alpha, ds, cs\rangle\rangle\end{aligned}$$

Proof. Straightforward by the definition of Substep. Destructing Substep gives two cases: one for a rule, and the other for a method. In both cases, we directly apply Lemma 33 to prove the goal. ■

Lemma 36 (Correctness of $(\hookleftarrow_m)$ for Substep – call case)**.**
For every $m, o, u_f, u, ds, cs_f, cs, f, a_f, v_a$, and v_r,

$$\langle m, o\rangle \Downarrow \langle u, \langle \alpha, ds, cs\rangle\rangle \rightarrow$$
$$\mathsf{WfDouble}\ m \rightarrow$$
$$o \vdash ((\lambda x.a_f)\ v_a) \Downarrow \langle u_f, cs_f, v_r\rangle \rightarrow$$
$$u_f * u \rightarrow cs_f * cs \rightarrow$$
$$cs[f] = (v_a, v_r) \rightarrow$$
$$\langle m \hookleftarrow_m f, o\rangle \Downarrow \langle u_s \uplus u, \langle \alpha, ds, cs_s \uplus cs \setminus f\rangle\rangle$$

Proof. Straightforward by the definition of Substep. Destructing Substep gives two cases: one for a rule, and the other for a method. In both cases, we directly apply Lemma 34 to prove the goal. ■

Note that (WfDouble m) is given as a condition instead of (WfDoubleA a). This is because we cannot directly obtain the action body from Substep. Since (WfDouble m) implies that each action a in m satisfies (WfDoubleA a), we eventually obtain what is needed.

Now we lift two theorems Lemma 35 and Lemma 36 to the level of Substeps. Since Substeps is composed of a series of Substep, we have more fine-grained cases in order to describe possible cases of each Substep:

- An *intact case* (Lemma 37) describes the case where the inlined method is not called anywhere in the substeps. As shown in above similar lemmas, the judgement will not be changed.

- A *rule case* (Lemma 38) describes the case where a substep in the substeps is defined by a rule, and the rule calls the inlined method. Also, the inlined method is defined as a substep in the substeps.

- A *method case 1* (Lemma 39) describes the case where a substep in the substeps is defined by a method g, but it is not the inlined method. Also, the substeps call g somewhere.

- A *method case 2* (Lemma 40) describes the case where a substep in the substeps is defined by a method f, which is the inlined method.

Lemma 37 (Correctness of $(\hookleftarrow_m)$ for Substeps – intact case)**.**

For every m, o, u, l, and f,

$$\begin{aligned}&\langle m, o\rangle \Downarrow^* \langle u, l\rangle \to \\ &f \notin (\mathsf{calls}\ l) \to \\ &\langle m \hookleftarrow_m f, o\rangle \Downarrow^* \langle u, l\rangle\end{aligned}$$

Proof. By induction on Substeps. Each substep in the substeps does not call f, since $f \notin (\mathsf{calls}\, l)$ and l is the merged label of all substeps, according to the definition. Thus, by Lemma 35, for each substep $\langle m, o\rangle \Downarrow \langle u_i, l_i\rangle$, we obtain $\langle m \hookleftarrow_m f, o\rangle \Downarrow \langle u_i, l_i\rangle$. Since $l = \uplus_i\, l_i$, we obtain the goal by the definition of Substeps. ■

Lemma 38 (Correctness of $(\hookleftarrow_m)$ for Substeps – rule case)**.**

For every $m, o, u_s, u, ds, cs_s, cs, s$, and f,

$$\begin{aligned}&\langle m, o\rangle \Downarrow \langle u_s, \langle \mathsf{Rule}\ s, [], cs_s\rangle\rangle \to \\ &\mathsf{WfDouble}\ m \to \\ &\langle m, o\rangle \Downarrow^* \langle u, \langle \mathsf{Meth}, ds, cs\rangle\rangle \to \\ &f \in cs_s \to f \in ds \to u_s * u \to cs_s * cs \to \\ &\langle m \hookleftarrow_m f, o\rangle \Downarrow^* \langle u_s \uplus u, \langle \mathsf{Rule}\ s, ds \setminus f, cs_s \uplus cs \setminus f\rangle\rangle\end{aligned}$$

Proof. By induction on substeps containing $f \in ds$. By the definition of Substeps, a substep for f exists in the substeps. Using the substep for f and the substep for the rule s calling f, we obtain the merged substep by Lemma 36. The other substeps except for f are treated as an intact case, since we obtain $f \notin cs$ by $f \in cs_s$ and $cs_s * cs$. ■

Lemma 39 (Correctness of $(\hookleftarrow_m)$ for Substeps – method case 1)**.**

For every $m, o, u_f, u, \alpha, ds, cs_f, cs, g, v_a, v_r$, and f,

$$\begin{aligned}&\langle m, o\rangle \Downarrow \langle u_f, \langle \mathsf{Meth}, [g \leftarrow (v_a, v_r)], cs_f\rangle\rangle \to \\ &\mathsf{WfDouble}\ m \to \\ &\langle m, o\rangle \Downarrow^* \langle u, \langle \alpha, ds, cs\rangle\rangle \to \\ &f \in cs_f \to f \in ds \to g \notin ds \to \\ &u_f * u \to cs_f * cs \to \\ &\langle m \hookleftarrow_m f, o\rangle \Downarrow^* \langle u_f \uplus u, \langle \alpha, [g \leftarrow (v_a, v_r)] \uplus ds \setminus f, cs_f \uplus cs \setminus f\rangle\rangle\end{aligned}$$

Proof. The proof is similar to Lemma 38, done by induction on substeps containing $f \in ds$. A substep for f exists in the substeps, and with the substep for g, we obtain the merged substep by Lemma 36. The other substeps except for f are treated as an intact case. Additionally, we do not have a conflict on defined methods while merging, since $g \notin ds$. ■

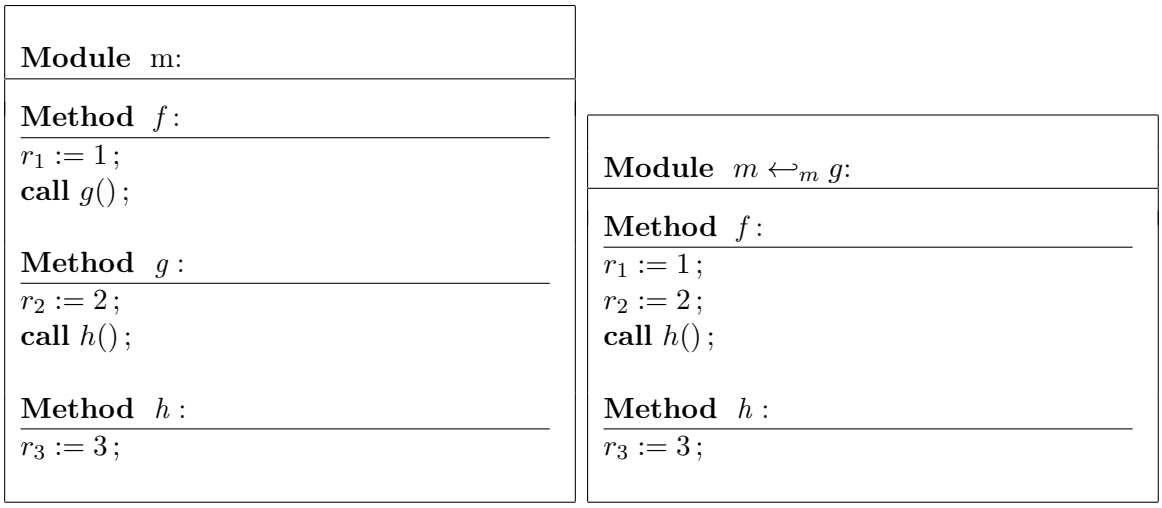
Lemma 40 (Correctness of $(\hookleftarrow_m)$ for Substeps – method case 2)**.**
For every $m, o, u_f, u, \alpha, ds, cs_f, cs, v_a, v_r$, and f,

$$\begin{aligned}
&\langle m, o\rangle \Downarrow \langle u_f, \langle \mathsf{Meth}, [f \leftarrow (v_a, v_r)], cs_f\rangle\rangle \rightarrow \\
&\mathsf{WfDouble}\ m \rightarrow \\
&\langle m, o\rangle \Downarrow^* \langle u, \langle \alpha, ds, cs\rangle\rangle \rightarrow \\
&f \in cs \rightarrow f \notin ds \rightarrow \\
&u_f * u \rightarrow cs_f * cs \rightarrow \\
&\langle m \hookleftarrow_m f, o\rangle \Downarrow^* \langle u_f \uplus u, \langle \alpha, ds, cs_f \uplus cs \setminus f\rangle\rangle
\end{aligned}$$

Proof. The proof is similar to Lemma 39, done by induction on substeps calling $f \in cs_f$. A substep calling f exists in the substeps, and with the substep for f, we obtain the merged substep by Lemma 36. The other substeps cannot call f by the definition of Substeps, thus they are treated as an intact case. ■

Giving the relation between a label and inlining Before proving the correctness of the method-inlining operator $(\hookleftarrow_m)$, which can be proved by applying the four lemmas (Lemma 37, Lemma 38, Lemma 39, and Lemma 40) for each proper case, we explain an intuitive relation between a label and inlining. The intuition starts from a basic question: what is the meaning of inlining in terms of semantics?

When inlining a method, label elements related to the method are hidden. Figure 4-2 presents an example where a method g is inlined. Considering a step judgements of f, before inlining, corresponding substeps should contain f, g, and h since f calls g and g calls h. Thus, the label of the substeps should contain f, g, and h. Whereas, after inlining, corresponding substeps now contain f and h, thus the label contains only f and h. Therefore, inlining a method is interpreted as *hiding the method from a label*, in terms of semantics.



(a) Before inlining g (b) After inlining g

Figure 4-2: Inlining a method hides label elements of the method

Following the intuition, we define hideMeth and hideMeths to form labels in which specific methods are hidden. (hideMeth l f) hides a method f from a given label l, which is defined as follows:

Definition 41.

$$\begin{aligned}\mathsf{hideMeth}\ l\ f\ =\ & \mathsf{if}\ (\mathsf{defs}\ l)[f] = (\mathsf{calls}\ l)[f] \\ & \mathsf{then}\ \langle \mathsf{annot}\ l, \mathsf{defs}\ l \setminus f, \mathsf{calls}\ l \setminus f \rangle \\ & \mathsf{else}\ l\end{aligned}$$

(hideMeths l fs) is a natural extension of hideMeth, hiding list of methods fs from a given label l:

Definition 42.

$$\begin{aligned}\mathsf{hideMeths}\ l\ [] &= l \\ \mathsf{hideMeths}\ l\ (f : fs) &= \mathsf{hideMeths}\ (\mathsf{hideMeth}\ l\ f)\ fs\end{aligned}$$

Finally, we are ready to state the lemma for the correctness of $(\hookleftarrow_m)$. For given substeps, they either call the inlining target method or not. If they call the method, they are required to have a substep for the method in order to use the substep to form the merged substep.

Lemma 43 (Correctness of $(\hookleftarrow_m)$)**.**

For every m, o, u, l, and f,

$$\begin{aligned}
&\langle m, o\rangle \Downarrow^* \langle u, l\rangle \rightarrow \\
&f \notin (\mathsf{calls}\ l) \vee (\mathsf{defs}\ l)[f] = (\mathsf{calls}\ l)[f] \rightarrow \\
&\langle m \hookleftarrow_m f, o\rangle \Downarrow^* \langle u, \mathsf{hideMeth}\ l\ f\rangle
\end{aligned}$$

Proof. The proof is largely divided into two cases: $f \notin (\mathsf{calls}\ l)$ or $(\mathsf{defs}\ l)[f] = (\mathsf{calls}\ l)[f]$.

- If $f \notin (\mathsf{calls}\ l)$, it is exactly an intact case so we apply Lemma 37 to obatin the goal. Note that we obtain $(\mathsf{hideMeth}\ l\ f) = l$, because $f \notin (\mathsf{calls}\ l)$.

- If $(\mathsf{defs}\ l)[f] = (\mathsf{calls}\ l)[f]$, the proof is done by induction on the given substeps. Picking a head substep from the substeps forms following four cases:

  - If the head substep is defined for f, then Lemma 40 is applied to obtain the goal. cs_f in the lemma cannot contain f, since the substep is defined for f, and we know $(\neg\ \mathsf{isRecursive}\ f)$ (f is not self-recursive), which are obtained from the definition of $(\hookleftarrow_m)$. Thus, we obtain $f \in cs$ so Lemma 40 can be applied properly.

  - If the head substep is defined for a method g where $g \neq f$, but calls f, then Lemma 39 is applied to obtain the goal. All conditions of the lemma are introduced by the definition of Substeps.

  - If the head substep is defined for a rule s, but calls f, then Lemma 38 is applied to obtain the goal. Similarly, all conditions are introduced by the definition of Substeps.

  - If the head substep is defined not for f and does not call f, then Lemma 35 is applied to obtain the substep after inlining. The induction hypothesis gives the substeps after inlining. Thus we simply merge the substep and the substeps to obtain the goal.

■

Correctness of the methods-inlining operator Lemma 44 is simply an extension of the correctness of the method-inlining operator, described in Lemma 43.

Lemma 44 (Correctness of $(\hookleftarrow^*)$ w.r.t. $\mathsf{hideMeths}$)**.**
For every m, o, u, l, and fs,

$$\begin{aligned}&\langle m, o\rangle \Downarrow^* \langle u, l\rangle \to \\ &\mathsf{wellHidden}\ m\ (\mathsf{hide}\ l) \to \\ &\langle m \hookleftarrow^* fs, o\rangle \Downarrow^* \langle u, \mathsf{hideMeths}\ l\ fs\rangle\end{aligned}$$

Proof. Straightforward by induction on fs. For every element f of fs, we can apply Lemma 43 to obtain the goal. Note that the necessary condition $(f \notin (\mathsf{calls}\ l) \vee (\mathsf{defs}\ l)[f] = (\mathsf{calls}\ l)[f])$ always holds by the condition $(\mathsf{wellHidden}\ m\ (\mathsf{hide}\ l))$. ■

Lemma 45 fills the gap between $\mathsf{hideMeths}$ and hide, which implies that when inlining is done for all defined methods in a module, then its effect to the label is as same as the hide operation.

Lemma 45 (Relation between $\mathsf{hideMeths}$ and hide)**.**
For every l and fs,

$$\mathsf{defs}\ l \subseteq fs \to \mathsf{hideMeths}\ l\ fs = \mathsf{hide}\ l$$

Proof. Straightforward by the definitions of $\mathsf{hideMeths}$ and hide. ■

Finally, by using Lemma 44 and Lemma 45, we can prove the correctness of $(\hookleftarrow^*)$.

Lemma 46 (Correctness of $(\hookleftarrow^*)$ w.r.t. Substeps)**.**
For every m, o, u, l, and fs,

$$\begin{aligned}&\langle m, o\rangle \Downarrow^* \langle u, l\rangle \to \\ &\mathsf{wellHidden}\ m\ (\mathsf{hide}\ l) \to \\ &\langle m \hookleftarrow^* (\mathsf{namesOf}\ (\mathsf{methodsOf}\ m)), o\rangle \Downarrow^* \langle u, \mathsf{hide}\ l\rangle\end{aligned}$$

Proof. Straightforward by applying Lemma 44 and Lemma 45. A statement $\mathsf{defs}\ l \subseteq (\mathsf{namesOf}\ (\mathsf{methodsOf}\ m))$ is trivial, since the label is always formed by merging substeps constructed by a rule or defined methods. ■

Lemma 47 (Correctness of $(\hookleftarrow^*)$)**.**

For every m, o, u, l, and fs,

$$(o \xrightarrow[m]{l} u) \to$$
$$fs = \mathsf{namesOf}\ (\mathsf{methodsOf}\ m) \to$$
$$(o \xrightarrow[(m \hookleftarrow^* fs)]{l} u)$$

Proof. This lemma is almost as same as Lemma 46. The only gap is filled by showing $\mathsf{hide}\ (\mathsf{hide}\ l) = \mathsf{hide}\ l$, which can be proved easily by destructing the definition of hide. ■

Correctness of the inlining operator At last, we show the correctness of the overall inlining operation. Since Lemma 47 directly gives the correctness of $|\cdot|$, it suffices to consider the internal method filtering defined in $\|\cdot\|$.

Lemma 48 (Correctness of $|\cdot|$)**.** For every m, o, u, and l,

$$(o \xrightarrow[m]{l} u) \to (o \xrightarrow[|m|]{l} u)$$

Proof. By the definition of $|\cdot|$, it is simply another representation of Lemma 47. ■

Lemma 49 (Correctness of $\|\cdot\|$)**.** For every m, o, u, and l,

$$(o \xrightarrow[m]{l} u) \to (o \xrightarrow[\|m\|]{l} u)$$

Proof. Applying Lemma 48, if suffices to show: $(o \xrightarrow[|m|]{l} u) \to (o \xrightarrow[\|m\|]{l} u)$. According to the definition $\|m\| \triangleq (\mathsf{regsOf}\ |m|, \mathsf{rulesOf}\ |m|, (\mathsf{methodsOf}\ |m|) \setminus (\mathsf{callsOf}\ m))$, the rules are preserved and the methods are filtered whether they are called internally or not. Now for every substep in the step:

- If it is defined by a rule, the substep is trivially preserved.
- If if is defined by a method, it suffices to show that the label l does not contain anything about internal calls. $(\mathsf{defs}\ l)$ cannot contain internal calls, since $\mathsf{wellHidden}\ m\ l$ by the definition of Step. $(\mathsf{calls}\ l)$ cannot contain internal calls, since $|m|$ does not have internal calls any longer by the definition. Thus, l is independent to the internal calls, which implies that the substep is also preserved.

■

Note that now Theorem 31 is no more than another representation of Lemma 49, according to the definition of the inlining semantics.

Chapter 5

Discussion and Conclusions

Inlining for Parametrized Modules The inlining operator $\| \cdot \|$ manipulates module definitions in a static way. Therefore, it requires a computation with module structures. All definitions about the operator generally perform two tasks; they either change action structures (by substituting call sites to their actual bodies), or compare two method names (in order to check a method call name equals the inlining target while examining actions). Each definition is either an ordinary function or a recursive function – there are no relational definitions. We call such definitions *computational*, which informally means that no proofs are involved.

However, in the Coq proof assistant on which Kami is built, variables make the computation stuck. Kami allows module definitions to be *parametric*, aiming for general designs and corresponding verifications. For example, a fifo implementation can be parametrized with respect to the size of the fifo. For a processor implementation, various sizes such as addresses, values, and register files also can be parametrized. However, once a module has parameters, inlining cannot be computationally performed, since variables cannot be compared unless they have concrete values. As a result, the inlining operation is no longer computational when it comes to parametrzied modules.

In order to solve this problem, a *term rewriting theorem* was defined (proved) and has been used to rewrite parameter comparisons. Even if we cannot define a general way to compare two parameters without any hypotheses, at least we can prove a

reflexive term rewriting theorem, saying $\forall s.\ \mathsf{is\_equal}\ s\ s = \mathsf{true}$. It is certainly not a complete theorem to include all possible parameter-comparison scenarios. However, we claim this kind of theorem is enough to perform inlining for practical designs. For instance, parameters are used to index processors for designing multi-core processors. However, since processors do not communicate directly with each others, method calls are used only within processors. This design fact indicates that parameter comparisons always show two aspects: either they have different concrete prefixes ($p \neq q \rightarrow p_i \neq q_j$), or they are exactly the same even with the index ($p_i = p_i$). Therefore, in this case, the term rewriting for reflexivity is enough to rewrite comparisons.

However, because of the term rewriting, the inlining operator loses its original performance. Term rewriting is considered to be slow in Coq. For a simple processor definition (approximately 200 lines of code), it took around 30 seconds to inline the processor module. One might say this is tolerable, but considering inlining as a trivial and simple operation, it may not be tolerable. Furthermore, a profiler in Coq produced a result that over 80 percentage of the time was due to term rewritings; inlining could have taken around 5 seconds without parameters.

Inlining and Synthesis Inlining is designed for efficient proofs, not for synthesis. Inlined modules can be synthesized into actual hardware components, but they should not be used as synthesis targets. The main reason is performance. In terms of synthesis, substituting action bodies to call sites implies that the corresponding circuit is duplicated throughout the module. It certainly increases the size of the synthesized circuit.

Thanks to the semantic implication proof, we can synthesize an original module, while verification is performed on the inlined one. Suppose that we want to prove a property about a module m, saying $P(m)$, where P is related to semantics. The semantic implication theorem (Theorem 31 in Chapter 4) allows us to reduce the goal to $P'(\|m\|)$, where P' is also related to semantics. P' is generally easier to prove, since we can take advantage of properties of inlined modules; *e.g.,* there are no internal calls in inlined modules.

Conclusions and Future Work We have built the Kami framework for general hardware verification, which requires a series of general concepts. With the belief that modularity is the key concept of hardware designs, we have defined modular semantics. However, modular semantics has an inherent weakness in that it is hard to infer internal state changes. Hence, we adopt a new semantic concept based on inlining in order to eliminate the weakness. We also prove an implication from the modular semantics to the inlined one, thus the inlining can be freely employed during verification.

The grand goal of the Kami framework is to provide efficient tools for general and practical hardware verification. Modular semantics and inlining are the main tools of the framework. In addition to them, we have built other interfaces such as renaming, decomposition, and modular refinement [12].

Using these various tools, we aim to verify real-world hardware components. Modern hardware designs such as a pipelined processor or a multi-level cached memory are too complicated to provide completely general verifications. However, we believe that efficient and high-level verification interfaces can make such verifications much easier. Modular semantics and the refinement theorem will separate verification into small pieces. Inlining will help us easily verify each small component. Equipped with an effective framework, it is no longer a dream to verify real-world hardware.

Bibliography

[1] The kami project website. http://plv.csail.mit.edu/kami/.

[2] *Bluespec^{TM} SystemVerilog Reference Guide*, January 2012.

[3] Peter J. Ashenden. *The Designer's Guide to VHDL, Volume 3, Third Edition (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, third edition, 2008.

[4] Lennart Augustsson, Jacob Schwarz, and Rishiyur S. Nikhil. Bluespec Language definition, 2001. Sandburst Corp.

[5] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Aviienis, J. Wawrzynek, and K. Asanovi. Chisel: Constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, June 2012.

[6] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP'08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, September 2008.

[7] Nirav Dave, Arvind, and Michael Pellauer. Scheduling as rule composition. In *MEMOCODE*, pages 51–60. IEEE Computer Society, 2007.

[8] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[9] Samir Palnitkar. *Verilog®Hdl: A Guide to Digital Design and Synthesis, Second Edition*. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2003.

[10] D. L. Rosenband and Arvind. Hardware synthesis from guarded atomic actions with performance specifications. In *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*, pages 784–791, Nov 2005.

[11] Daniel L. Rosenband. *A Performance Driven Approach for Hardware Synthesis of Guarded Atomic Actions*. PhD thesis, Massachusetts Institute of Technology, August 2005.

[12] Muralidaran Vijayaraghavan. *Modular Verification of Hardware Systems*. PhD thesis, Massachusetts Institute of Technology, June 2016.