JFP **21** (1): 21–58, 2011. © Cambridge University Press 2010
doi:10.1017/S0956796810000249 First published online 19 November 2010

A calculus for hardware description[*]

SUNGWOO PARK and HYEONSEUNG IM[†]
Department of Computer Science and Engineering,
Pohang University of Science and Technology, Republic of Korea
(*e-mail:* {gla,genilhs}@postech.ac.kr)

Abstract

In efforts to overcome the complexity of the syntax and the lack of formal semantics of conventional hardware description languages, a number of functional hardware description languages have been developed. Like conventional hardware description languages, however, functional hardware description languages eventually convert all source programs into netlists, which describe wire connections in hardware circuits at the lowest level and conceal all high-level descriptions written into source programs. We develop a calculus, called $l\lambda$ (linear lambda), which may serve as an intermediate functional language just above netlists in the hierarchy of hardware description languages. In order to support higher-order functions, $l\lambda$ uses a linear type system, which enforces the linear use of variables of function type. The translation of $l\lambda$ into structural descriptions of hardware circuits is sound and complete in the sense that it maps expressions only to realizable hardware circuits, and that every realizable hardware circuit has a corresponding expression in $l\lambda$. To illustrate the use of $l\lambda$ as a practical intermediate language for hardware description, we design a simple hardware description language that extends $l\lambda$ with polymorphism, and use it to implement a fast Fourier transform circuit and a bitonic sorting network.

1 Introduction

In efforts to overcome the complexity of the syntax and the lack of formal semantics of conventional hardware description languages (most notably Verilog and VHDL), a number of approaches based on functional languages have been proposed (Sharp & Rasmussen 1995; O'Donnell 1995; Bjesse *et al.* 1998; Matthews *et al.* 1998; Li & Leeser 2000; Mycroft & Sharp 2000; Axelsson *et al.* 2005; Grundy *et al.* 2006; Ghica 2007). In fact, the idea of using functional languages for hardware design dates back as early as in the 1980s (Cardelli & Plotkin 1981; Boute 1984; Johnson 1984; Sheeran 1984; Meshkinpour & Ercegovac 1985), which saw the birth of currently popular hardware description languages. The merits of functional

[*] This is an extended version of the paper that appeared in the 13th ACM SIGPLAN International Conference on Functional Programming (Park *et al.* 2008).

[†] This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / the National Research Foundation of Korea (NRF) (Grant 2010-0001726), the National IT Industry Promotion Agency (NIPA) under the program of Software Engineering Technologies Development and Experts Education, and the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2009-0077543).

languages as hardware description languages can be attributed to the fact that basic building blocks for hardware circuits are equivalent to mathematical functions, while functional languages lend themselves to creating and composing mathematical functions.

Like conventional hardware description languages, however, functional hardware description languages eventually convert all source programs into *netlists*, the de facto assembly language for hardware description. Netlists describe wire connections in hardware circuits at the lowest level and conceal all high-level descriptions written into source programs. Such a translation of functional hardware description languages into netlists could be compared to a direct translation of functional languages into an assembly language rather than the lambda calculus, the core calculus for functional languages.

Our goal is to develop a syntax-directed translation of a calculus similar to the lambda calculus into structural descriptions of hardware circuits so that we can use it as a "high-level assembly language" for functional hardware description languages.[1] We intend to use the calculus as an "assembly" language in the sense that its definition consists only of a minimal set of primitive constructs each of which corresponds to a specific method of combining hardware components, e.g., linking two separate components or building feedback circuits. In comparison with netlists, the calculus is still a "high-level" language because it makes no explicit use of low-level constructs, such as ports and wires, characterizing netlists. Thus, we wish to use the calculus as an intermediate functional language just above netlists in the hierarchy of hardware description languages.

The basic idea for the translation is already in use by existing functional hardware description languages: functions represent hardware circuits taking input streams to emit output streams while applications link two separate components. The problem is still interesting, however, because we allow higher-order functions as in conventional functional languages. (The translation becomes trivial if higher-order functions are not allowed.) The use of higher-order functions improves the expressive power of the calculus as an intermediate language for hardware description. For example, we can express various higher-order combinators within the calculus itself without recourse to additional metaprogramming constructs or another host language.

To correctly translate higher-order functions, we need to take into consideration the fact that hardware circuits are physical resources that cannot be shared in general. Consider a function (written in the syntax of the lambda calculus)

$$k = \lambda x : \mathbf{1}. \, \mathtt{and} \; x \; x$$

where $\mathbf{1}$ is a base type for bitstreams and $\mathtt{and}$ denotes a binary AND gate. Since a bitstream can be shared by multiple wires, k may use x twice in its body. Now, consider a higher-order function

$$g = \lambda f : \mathbf{1} \to \mathbf{1}. \, f \; (f \; 0)$$

[1] Cardelli & Plotkin (1981) call their algebra for hardware description "a high-level chip assembly language."

where $\mathbf{1}\rightarrow\mathbf{1}$ is a type for functions taking a bitstream to emit another bitstream. Since f represents a hardware circuit that cannot be shared by multiple hardware components, g may not use f twice in its body. A workaround is to rewrite g as

$$g' = \lambda f_1 : \mathbf{1}\rightarrow\mathbf{1}.\, \lambda f_2 : \mathbf{1}\rightarrow\mathbf{1}.\, f_1\ (f_2\ 0)$$

and expand every application $g\ e$ into $g'\ e\ e$ by duplicating e.

Unfortunately, it is not always possible to determine how many times we need to duplicate each expression. As an example, consider another higher-order function

$$h = \lambda f : (\mathbf{1}\rightarrow\mathbf{1})\rightarrow\mathbf{1}.\, f\ (\lambda x : \mathbf{1}.\, x).$$

Since it is unknown how many times f uses its argument $\lambda x : \mathbf{1}.\, x$, we cannot expand $f\ (\lambda x : \mathbf{1}.\, x)$ in the same way that we expand $g\ e$ in the previous example. A quick fix is to annotate $(\mathbf{1}\rightarrow\mathbf{1})\rightarrow\mathbf{1}$ with an integer n indicating how many times f uses its argument:

$$h' = \lambda f : (\mathbf{1}\rightarrow\mathbf{1})^n\rightarrow\mathbf{1}.\, f\ (\lambda x : \mathbf{1}.\, x).$$

A further development of this idea leads to a type system in which variables of function type are used exactly once, i.e., linearly.

Building upon these observations, we design a calculus $l\lambda$ (linear lambda), which borrows its syntax from the standard lambda calculus and uses a *linear type system* to enforce the linear use of variables of function type. (We do not develop equational theories for $l\lambda$, for example, based on β-reductions and η-expansions.) The type system of $l\lambda$ draws a distinction between *sharable types* and *linear types*. A function with a sharable input type (e.g., $\mathbf{1}$) may use its argument more than once; a function with a linear input type (e.g., $\mathbf{1}\rightarrow\mathbf{1}$) must use its argument exactly once. Hence, there arise two kinds of function types: one with a sharable input type and the other with a linear input type. These function types in turn constitute linear types of $l\lambda$. The linear type system of $l\lambda$ is similar in spirit to the affine type system of (Ghica 2007) in that both type systems prevent erroneous sharing of hardware circuits.

We develop a syntax-directed translation of $l\lambda$ into structural descriptions of hardware circuits. The translation is sound and complete in the following sense:

- The translation is sound in the sense that it maps expressions only to *realizable* hardware circuits. A hardware circuit is realizable if it contains no input terminal (accepting a single bitstream) connected with multiple wires.
- The translation is complete in the sense that every realizable hardware circuit has a corresponding expression in $l\lambda$.

In addition, the type system of $l\lambda$ is sound and complete with respect to the translation in the sense that expressions are mapped to hardware circuits if and only if they are well-typed. These properties of the translation allow $l\lambda$ to serve as a practical intermediate language for hardware description. For example, we may convert source programs in a typical hardware description language into well-typed expressions in $l\lambda$, which are guaranteed to be realizable and are also more amenable to layout analysis and behavior simulation than equivalent netlist specifications.

To illustrate the use of $l\lambda$ as a high-level assembly language for representing hardware circuits, we design a simple hardware description language that extends

$l\lambda$ with polymorphism. An expression of a polymorphic type describes a family of hardware circuits with essentially the same layout of hardware components, and polymorphism offers a simple form of metaprogramming, which is particularly useful for writing higher-order combinators. We use the extension of $l\lambda$ to implement a fast Fourier transform (FFT) circuit and a bitonic sorting network. The actual code for the FFT circuit is 60 lines long and expands to 5158 lines of Verilog code by our translator of $l\lambda$; the actual code for the bitonic sorting network is 43 lines long and expands to 5175 lines of Verilog code.

As Sheeran (2005) notes, *"functional programming and hardware design are a perfect match."* Hence, it is actually no surprise that there is already an extensive literature on functional hardware description languages. What comes as a surprise, however, is that there has been little effort to formally interpret the lambda calculus, the core calculus for all functional languages, directly in terms of structural descriptions of hardware circuits. The development of $l\lambda$ has been motivated by a desire for such a formal interpretation of the lambda calculus. Since we interpret expressions in $l\lambda$ only as structural descriptions of hardware circuits and not as their behavioral specifications, we do not investigate equational theories for $l\lambda$ in this work.

This paper is organized as follows. Section 2 presents the abstract syntax and the type system of $l\lambda$ and explains basic ideas behind the translation of $l\lambda$. Section 3 presents a few examples of mapping expressions to hardware circuits and formulates the translation of $l\lambda$. Section 4 proves the soundness and completeness of the translation and the type system. Section 5 discusses an alternative translation of $l\lambda$ and how to eliminate redundant wires. Section 6 presents an FFT circuit and a bitonic sorting network implemented in $l\lambda$ extended with polymorphism. Section 7 discusses related work and Section 8 concludes. Selected proofs are given in Appendix.

2 Basics of $l\lambda$

This section presents the abstract syntax and the type system of $l\lambda$. It also formalizes structural specifications of hardware circuits to be employed in the translation of $l\lambda$.

2.1 Abstract syntax and type system

Figure 1 shows the abstract syntax of $l\lambda$, which builds on the simply typed lambda calculus with product types. A type τ is either a sharable type θ or a linear type κ. (Sharable types and linear types are disjoint.) Sharable types correspond to base types in general programming languages (e.g., 32-bit integers) or their combinations. For the sake of simplicity, we use only single-bit type $\mathbf{1}$ and product types $\theta_1 \times \theta_2$, which suffice for supporting general forms of sharable types. Linear types are another name for function types in $l\lambda$. A function of type $\theta \rightarrow \tau$ may use its argument (of sharable type θ) more than once in its body, but a function of type $\kappa \multimap \tau$ must use its argument (of linear type κ) exactly once. Note that $l\lambda$ uses product types of sharable types only (i.e., no product types of linear types).

type	τ	::=	θ	sharable type
			κ	linear type
sharable type	θ	::=	$\mathbf{1}$	single-bit
			$\theta \times \theta$	sharable product
linear type	κ	::=	$\theta \rightarrow \tau$	sharable input
			$\kappa \multimap \tau$	linear input
expression	e	::=	x	sharable variable
			f	linear variable
			$\lambda x{:}\theta\,.e$	sharable input function
			$e\ e$	sharable input application
			$\hat{\lambda} f{:}\kappa.e$	linear input function
			$e\,\hat{}\,e$	linear input application
			(e,e)	pair
			$\mathbf{proj}\ e\ \mathbf{of}\ (x,x)\ \mathbf{in}\ e$	projection
			$\mathbf{fix}\ x{:}\theta\,.e$	fixed point expression
			c	constant
sharable typing context	Γ	::=	$\cdot \mid \Gamma, x:\theta$	
linear typing context	Δ	::=	$\cdot \mid \Delta, f:\kappa$	

Fig. 1. Abstract syntax of $l\lambda$.

$$\frac{x:\theta \in \Gamma}{\Gamma; \cdot \vdash x:\theta}\ \mathrm{Var} \qquad \frac{}{\Gamma; f:\kappa \vdash f:\kappa}\ \mathrm{LVar}$$

$$\frac{\Gamma, x:\theta; \Delta \vdash e:\tau}{\Gamma; \Delta \vdash \lambda x{:}\theta\,.e : \theta \rightarrow \tau}\ \rightarrow\mathrm{I} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : \theta \rightarrow \tau \quad \Gamma; \Delta_2 \vdash e_2 : \theta}{\Gamma; \Delta_1, \Delta_2 \vdash e_1\ e_2 : \tau}\ \rightarrow\mathrm{E}$$

$$\frac{\Gamma; \Delta, f:\kappa \vdash e:\tau}{\Gamma; \Delta \vdash \hat{\lambda} f{:}\kappa.e : \kappa \multimap \tau}\ \multimap\mathrm{I} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : \kappa \multimap \tau \quad \Gamma; \Delta_2 \vdash e_2 : \kappa}{\Gamma; \Delta_1, \Delta_2 \vdash e_1\,\hat{}\,e_2 : \tau}\ \multimap\mathrm{E}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \theta_1 \quad \Gamma; \Delta_2 \vdash e_2 : \theta_2}{\Gamma; \Delta_1, \Delta_2 \vdash (e_1, e_2) : \theta_1 \times \theta_2}\ \times\mathrm{I} \qquad \frac{\Gamma; \Delta \vdash e : \theta_1 \times \theta_2 \quad \Gamma, x_1:\theta_1, x_2:\theta_2; \Delta' \vdash e' : \tau}{\Gamma; \Delta, \Delta' \vdash \mathbf{proj}\ e\ \mathbf{of}\ (x_1, x_2)\ \mathrm{in}\ e' : \tau}\ \times\mathrm{E}$$

$$\frac{\Gamma, x:\theta; \Delta \vdash e:\theta}{\Gamma; \Delta \vdash \mathbf{fix}\ x{:}\theta\,.e : \theta}\ \mathrm{Fix}$$

Fig. 2. Type system of $l\lambda$.

In order to simplify the presentation of the definition of $l\lambda$, we choose to syntactically distinguish between *sharable variables* x (of sharable type) and *linear variables* f (of linear type). Accordingly, we use two kinds of functions and applications: $\lambda x{:}\theta.e$ and $e\ e$ for sharable input types and $\hat{\lambda} f{:}\kappa.e$ and $e\,\hat{}\,e$ for linear input types. Pairs (e,e) and projections **proj** e **of** (x,x) **in** e are expressions for product types. Fixed-point expressions **fix** $x{:}\theta.e$ permit only sharable variables in their binders, and build feedback circuits and do not synthesize hardware circuits simulating recursive functions. Constants c denote atomic hardware components. For example, we may use a constant `reg` for a single-bit register and another constant `and` for an AND gate.

Figure 2 shows the type system of $l\lambda$. It uses a typing judgment $\Gamma; \Delta \vdash e : \tau$ which means that under *sharable typing context* Γ and *linear typing context* Δ, expression e has type τ. Given a binding $x : \theta$ in Γ, we may use x zero or more times in e, but given a binding $f : \kappa$ in Δ, we must use f exactly once in e. Thus, for example, the rule Var uses an empty linear typing context, and in the rules $\rightarrow$E and $\multimap$E, the linear typing context in the conclusion is split into two in the premises. Each constant assumes a unique type reflecting its behavioral characteristics. For example, `reg` has a linear type $\mathbf{1} \multimap \mathbf{1}$, because it emits a bitstream fed as input (after a delay). For `and`, we assign either $\mathbf{1} \rightarrow (\mathbf{1} \multimap \mathbf{1})$ or $(\mathbf{1} \times \mathbf{1}) \multimap \mathbf{1}$.

2.2 Structural specifications of hardware circuits

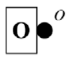
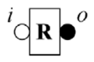
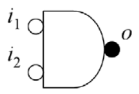
If we are to interpret expressions in $l\lambda$ as descriptions of hardware circuits, we need a formal system for specifying hardware circuits at a lower structural level. We depart from the standard netlist specification (which declares all input terminals, output terminals, and wires individually) in favor of a more concise system described below.

At the physical level, a hardware circuit consists of hardware components and connecting wires. A hardware component has one or more terminals to which external wires can be connected. We assume that every wire is unidirectional and never alternates the direction of the bitstream it transmits. Hence, a wire always connects an output terminal o, emitting a bitstream, to an input terminal i, receiving a bitstream. Schematically, we write an input terminal as $\circ$ and an output terminal as $\bullet$. Then we can draw a wire connecting an output terminal o to an input terminal i as follows:

Note that a wire only connects an output terminal to an input terminal and does not have its own terminals. We assume that input and output terminals are syntactically distinguished, i.e., $i = o$ never holds.

The translation of $l\lambda$ refines the physical view of hardware circuits by supplanting wires by *connection constraints*. A connection constraint $o \mapsto i$ specifies that the bitstream emitted from output terminal o be fed into input terminal i. To realize $o \mapsto i$ in a hardware circuit, we can either connect o to i via a wire or just superimpose o on i (which is equivalent to connecting o to i via a wire of zero length).

Now, we can specify the structure of a hardware circuit with a set H of atomic hardware components and a set C of connection constraints. Examples of atomic hardware components are a constant (zero) generator written as $\mathsf{0}[o]$, a single-bit register written as $\mathsf{reg}[i, o]$, and an AND gate written as $\mathsf{and}[i_1, i_2, o]$:

atomic hardware component		schematic diagram
$\mathsf{0}[o]$	$\Leftrightarrow$	
$\mathsf{reg}[i, o]$	$\Leftrightarrow$	
$\mathsf{and}[i_1, i_2, o]$	$\Leftrightarrow$	

We write $|H|$ and $|C|$ for the set of input and output terminals in H and C, respectively. For example, we have $|H, \mathsf{reg}[i, o]| = |H| \cup \{i, o\}$ and $|C, o \mapsto i| = |C| \cup \{o, i\}$.

We say that a set of connection constraints is realizable if no input terminal receives bitstreams from multiple output terminals:

Definition 2.1
C is realizable if there is no input terminal i such that $o \mapsto i \in C$, $o' \mapsto i \in C$, and $o \neq o'$.

If an expression is translated to a pair of H and C, we have to show that C is realizable. Otherwise, unpredictable behavior may occur because of input terminals receiving multiple bitstreams from independent sources.

Proposition 2.2
If both C_1 and C_2 are realizable, and there is no input terminal $i \in |C_1| \cap |C_2|$, then $C_1 \cup C_2$ is realizable.

Proof
Suppose that $C_1 \cup C_2$ is not realizable: there is an input terminal i such that $o \mapsto i \in C_1 \cup C_2$, $o' \mapsto i \in C_1 \cup C_2$, and $o \neq o'$. Since $i \notin |C_1| \cap |C_2|$, we have either $o \mapsto i \in C_1$, $o' \mapsto i \in C_1$ (meaning that C_1 is not realizable) or $o \mapsto i \in C_2$, $o' \mapsto i \in C_2$ (meaning that C_2 is not realizable). Both cases result in a contradiction because of the assumption that both C_1 and C_2 are realizable. $\square$

2.3 *Connection points*

In $l\lambda$, we can describe not only actual hardware circuits but also patterns of connecting several wires. For example, $\lambda x : \mathbf{1}. x$ describes a pattern of relaying a bitstream without actually linking two wires. Another example is $\lambda x : \mathbf{1}. (x, x)$ which describes a pattern of replicating a bitstream into two without actually connecting an input wire to two output wires. In order to translate such expressions, $l\lambda$ uses a special kind of hardware components called *connection points*.

A connection point consists of an input terminal i and an output terminal o adjacent to each other and is written as $\mathsf{pt}[i, o]$:

$\mathsf{pt}[i, o] \quad \Leftrightarrow$

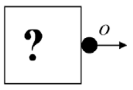
We may think of $\mathsf{pt}[i, o]$ as transmitting a bitstream from i to o (not from o to i) via a wire of zero length. Although it has its own terminals (unlike wires), a connection point just serves as a special mark for linking separate wires and does not occupy a physical area when realized as a hardware circuit. Section 3.1 shows examples of using connection points in the translation of $l\lambda$. Section 5.1 discusses an alternative way of translating $l\lambda$ without using connection points.

2.4 *Output and input interfaces*

In order to map expressions to hardware circuits, the translation of $l\lambda$ needs to know not only how to describe the structure of hardware circuits, but also how to interface with them. For example, a hardware circuit generated from an application $e_1\ e_2$ includes two separate hardware circuits generated from e_1 and e_2, and composing the two hardware circuits requires us to identify which input and output terminals need to be connected together. Thus the compositional nature of the translation leads us to define *output interfaces*, which consist of input and output terminals through which external hardware circuits communicate. That is, only those terminals in the output interface are exposed to external hardware circuits, and we essentially abstract hardware circuits with their output interfaces.

Example 1. An expression e_1 of single-bit type $\mathbf{1}$ is mapped to a hardware circuit emitting a bitstream through an output terminal o:

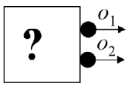
$$e_1 : \mathbf{1} \quad \Rightarrow$$



Hence, the output interface for e_1 consists only of output terminal o, while all other terminals are hidden.

Example 2. An expression e_2 of product type $\mathbf{1} \times \mathbf{1}$ is mapped to a hardware circuit emitting two bitstreams through two output terminals o_1 and o_2:

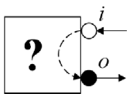
$$e_2 : \mathbf{1} \times \mathbf{1} \quad \Rightarrow$$



Hence, the output interface for e_2 consists only of output terminals o_1 and o_2, while all other terminals are hidden. We write $o_1 \times o_2$ for the output interface for e_2.

Example 3. An expression e_3 of function type $\mathbf{1} \rightarrow \mathbf{1}$ is mapped to a hardware circuit accepting a bitstream from an input terminal i and emitting a bitstream through an output terminal o:

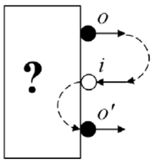
$$e_3 : \mathbf{1} \rightarrow \mathbf{1} \quad \Rightarrow$$



Hence, the output interface for e_3 consists only of input terminal i and output terminal o, while all other terminals are hidden. For the output interface for e_3, we write $i \rightarrow o$ to indicate that a bitstream flows from i to o.

Example 4. An expression e_4 of type $(\mathbf{1} \rightarrow \mathbf{1}) \multimap \mathbf{1}$ is mapped to a hardware circuit that first communicates with an external hardware circuit through an output terminal o and an input terminal i and then emits a bitstream through another output terminal o':

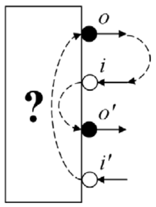
$$e_4 : (\mathbf{1} \rightarrow \mathbf{1}) \multimap \mathbf{1} \quad \Rightarrow$$



The external hardware circuit should be generated from an expression of type $\mathbf{1} \rightarrow \mathbf{1}$. For the output interface for e_4, we write $(o \rightarrow i) \multimap o'$ to indicate that a bitstream flows from o to i (not from i to o) and eventually exits at o'.

Example 5. An expression e_5 of type $(\mathbf{1}\rightarrow\mathbf{1})\multimap(\mathbf{1}\rightarrow\mathbf{1})$ is mapped to a hardware circuit that accepts a bitstream from an input terminal i', communicates with an external hardware circuit through an output terminal o and an input terminal i, and emits a bitstream through an output terminal o':

$$e_5 : (\mathbf{1}\rightarrow\mathbf{1})\multimap(\mathbf{1}\rightarrow\mathbf{1}) \quad \Rightarrow$$

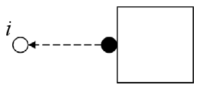


The external hardware circuit should be generated from an expression of type $\mathbf{1}\rightarrow\mathbf{1}$. For the output interface for e_5, we write $(o\rightarrow i)\multimap(i'\rightarrow o')$ to indicate that a bitstream flows from i' to o' and from o to i.

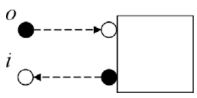
Examples 3–5 illustrate that output interfaces for expressions of function type consist not only of output terminals but also of input terminals. For example, the output interface $i\rightarrow o$ for expression e_3 includes input terminal i to receive a bitstream from an external hardware circuit (generated from an expression of type $\mathbf{1}$); the output interface $(o\rightarrow i)\multimap o'$ for expression e_4 uses input terminal i, as well as output terminal o, to communicate with an external hardware circuit (generated from an expression of type $\mathbf{1}\rightarrow\mathbf{1}$). We refer to these input and output terminals that are to be connected with external hardware circuits as *input interfaces*.

To exploit an existing hardware circuit, we first have to prepare an input interface compatible with its output interface. Here are a couple of examples:

Example 6. To exploit a hardware circuit producing a bitstream, we need an input interface consisting of a single input terminal i:



Example 7. To exploit a hardware circuit accepting a bitstream and emitting another bitstream, we need an input interface consisting of an output terminal o and an input terminal i:



We write such an input interface as $o\rightarrow i$ (not as $i\rightarrow o$) to indicate that a bitstream flows from o to i.

From Examples 3 and 6, we see that an output interface for type $\theta\rightarrow\tau$ includes an input interface for type θ. From Examples 4 and 7, we see that an output interface for type $\kappa\multimap\tau$ includes an input interface for type κ.

$$\frac{}{o \rhd \mathbf{1}}\ \mathbf{1}\rhd \qquad \frac{}{i \lhd \mathbf{1}}\ \mathbf{1}\lhd$$

$$\frac{O_1 \rhd \theta_1 \quad O_2 \rhd \theta_2}{O_1 \times O_2 \rhd \theta_1 \times \theta_2}\ \times\rhd \qquad \frac{I_1 \lhd \theta_1 \quad I_2 \lhd \theta_2 \quad \sharp\{I_1, I_2\}}{I_1 \times I_2 \lhd \theta_1 \times \theta_2}\ \times\lhd$$

$$\frac{I \lhd \theta \quad O \rhd \tau \quad \sharp\{I, O\}}{I \to O \rhd \theta \to \tau}\ \to\rhd \qquad \frac{O \rhd \theta \quad I \lhd \tau \quad \sharp\{O, I\}}{O \to I \lhd \theta \to \tau}\ \to\lhd$$

$$\frac{I \lhd \kappa \quad O \rhd \tau \quad \sharp\{I, O\}}{I \multimap O \rhd \kappa \multimap \tau}\ \multimap\rhd \qquad \frac{O \rhd \kappa \quad I \lhd \tau \quad \sharp\{O, I\}}{O \multimap I \lhd \kappa \multimap \tau}\ \multimap\lhd$$

Fig. 3. Rules for assigning types to output and input interfaces.

Generalizing these observations, we inductively define output and input interfaces as follows:

$$\begin{array}{rccl} \text{output interface} & O & ::= & o \mid O \times O \mid I \to O \mid I \multimap O \\ \text{input interface} & I & ::= & i \mid I \times I \mid O \to I \mid O \multimap I. \end{array}$$

In order to clarify the meaning of each form of output and input interfaces, we introduce two judgments $O \rhd \tau$ and $I \lhd \tau$, which assign types to output and input interfaces. Informally, $O \rhd \tau$ means that O is an output interface of a hardware circuit generated from an expression of type τ, or simply that O is an output interface of type τ. Similarly, $I \lhd \tau$ means that we can connect input interface I with any output interface of type τ, or simply that I is an input interface of type τ.

Figure 3 shows the rules for the judgments $O \rhd \tau$ and $I \lhd \tau$. We write $\sharp\{S, S'\}$ to mean that S and S' share no terminals, where S and S' range over output and input interfaces. That is, $\sharp\{S, S'\}$ holds if and only if $|S| \cap |S'| = \emptyset$ holds, where $|S|$ denotes the set of input and output terminals in S:

$$\begin{array}{rcl@{\qquad}rcl} |i| & = & \{i\} & |o| & = & \{o\} \\ |I_1 \times I_2| & = & |I_1| \cup |I_2| & |O_1 \times O_2| & = & |O_1| \cup |O_2| \\ |O \to I| & = & |O| \cup |I| & |I \to O| & = & |I| \cup |O| \\ |O \multimap I| & = & |O| \cup |I| & |I \multimap O| & = & |I| \cup |O|. \end{array}$$

Note that unlike the rule $\times\lhd$, the rule $\times\rhd$ does not require $\sharp\{O_1, O_2\}$ because a single output terminal can be connected to multiple input terminals.

Proposition 2.3

If $O \rhd \theta$, then there is no input terminal $i \in |O|$.
If $I \lhd \theta$, then there is no output terminal $o \in |I|$.

We write $O \bowtie I$ for the set of connection constraints for connecting output interface O and input interface I:

$$\begin{array}{rcl} o \bowtie i & = & \{o \mapsto i\} \\ O_1 \times O_2 \bowtie I_1 \times I_2 & = & O_1 \bowtie I_1 \cup O_2 \bowtie I_2 \\ I \to O \bowtie O' \to I' & = & O \bowtie I' \cup O' \bowtie I \\ I \multimap O \bowtie O' \multimap I' & = & O \bowtie I' \cup O' \bowtie I. \end{array}$$

$O \bowtie I$ implicitly assumes that O and I are syntactically compatible (e.g., $O_1 \times O_2 \bowtie O' \rightarrow I'$ is invalid). Proposition 2.6 shows that an output interface and an input interface of the same type can be safely connected if they share no input terminal.

Proposition 2.4
If $O \rhd \tau$ and $I \lhd \tau$, then $O \bowtie I$ is valid.

Proof
By induction on the structure of τ. $\square$

Proposition 2.5
If $O \bowtie I$ is valid, where $O \rhd \tau$ and $I \lhd \tau'$, then $\tau = \tau'$.

Proof
By induction on the size of $O \bowtie I$. $\square$

Proposition 2.6
If $O \rhd \tau$, $I \lhd \tau$, and there is no input terminal $i \in |O| \cap |I|$, then $O \bowtie I$ is realizable.

The translation of $l\lambda$ given in the next section maps a given expression to a triple (H, C, O) consisting of a set H of hardware components, a set C of connection constraints, and an output interface O. Thus, it uses not only H and C to specify how to connect hardware components, but also O to specify how to interface with the generated hardware circuit.

3 Translation of $l\lambda$

This section presents the translation of $l\lambda$. To develop an intuition for it, we begin with a few examples of mapping expressions to hardware circuits. Then, we formulate it with rules for translating types and expressions.

3.1 Examples

The translation uses a judgment $e \Rightarrow (H, C, O)$ to mean that expression e describes a hardware circuit specified by triple (H, C, O). An invariant here is that if expression e has type τ, output interface O has the same type, i.e., $O \rhd \tau$. We assume three constants `zero` of type $\mathbf{1}$, `reg` of type $\mathbf{1} \rightarrow \mathbf{1}$, and `and` of type $\mathbf{1} \rightarrow (\mathbf{1} \rightarrow \mathbf{1})$, which are mapped to constant (zero) generators, single-bit registers, and AND gates, respectively; for visual clarity, we use traditional set notation to write H and C:

$$\begin{array}{rcl} \texttt{zero} & \Rightarrow & (\{\mathsf{0}[o]\}, \emptyset, o) \\ \texttt{reg} & \Rightarrow & (\{\mathsf{reg}[i, o]\}, \emptyset, i \rightarrow o) \\ \texttt{and} & \Rightarrow & (\{\mathsf{and}[i_1, i_2, o]\}, \emptyset, i_1 \rightarrow (i_2 \rightarrow o)). \end{array}$$

Note that the translation uses a declarative style in that no constants specify specific identifiers for terminals. Hence, for example, different instances of `zero` generate different hardware components $\mathsf{0}[o]$ and $\mathsf{0}[o']$. The translation, however, ensures that different instances of the same constant never share identifiers for terminals.

In the examples below, we realize a connection constraint $o \mapsto i$ as a wire connecting o to i.

3.1.1 Sharable input function

Consider an identify function $\lambda x:\mathbf{1}.\,x$ of type $\mathbf{1}\to\mathbf{1}$. Since it passes an input bitstream without change, $\lambda x:\mathbf{1}.\,x$ requires no hardware component other than a single connection point, say, $\mathsf{pt}[i,o]$. We generate such a hardware circuit consisting of $\mathsf{pt}[i,o]$ in the following way.

When interpreting the binder $x:\mathbf{1}$ in $\lambda x:\mathbf{1}.\,x$, we associate $\mathsf{pt}[i,o]$ with x so that an input bitstream is fed into i and an output bitstream is emitted from o. In essence, the translation needs to specify an input interface and an output interface for the variable in each binder, which are i and o, respectively, in the case of x. When interpreting the body of $\lambda x:\mathbf{1}.\,x$, however, we use only o as the output interface for x. Then the output interface for $\lambda x:\mathbf{1}.\,x$ becomes $i\to o$ because as a function of type $\mathbf{1}\to\mathbf{1}$, it receives a bitstream via i to emit another bitstream via o:

$$\lambda x:\mathbf{1}.\,x \Rightarrow (\{\mathsf{pt}[i,o]\},\varnothing,i\to o).$$

Here, output interface $i\to o$ also has type $\mathbf{1}\to\mathbf{1}$, i.e., $i\to o \rhd \mathbf{1}\to\mathbf{1}$.

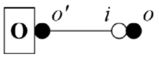
Note that it is not the instance of x in the body but the binder $x:\mathbf{1}$ that generates $\mathsf{pt}[i,o]$. For example, even if the body changes from x to (x,x), we do not generate an additional connection point. Instead, we only update the output interface from $i\to o$ to $i\to(o\times o)$, which is feasible because output terminal o can be shared by both instances of x:

$$\lambda x:\mathbf{1}.\,(x,x) \Rightarrow (\{\mathsf{pt}[i,o]\},\varnothing,i\to(o\times o)).$$

Thus, $\lambda x:\mathbf{1}.\,(x,x)$ in effect replicates an input bitstream into two output bitstreams.

Now, let us build an expression exploiting such two output bitstreams. An application $(\lambda x:\mathbf{1}.\,(x,x))$ `zero` associates a new hardware component $\mathsf{0}[o']$ with `zero` and connects output terminal o' to existing input terminal i:

$$(\lambda x:\mathbf{1}.\,(x,x))\ \texttt{zero} \Rightarrow (\{\mathsf{pt}[i,o],\mathsf{0}[o']\},\{o'\mapsto i\},o\times o)$$

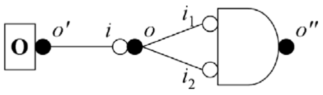


To bind the two instances of o in output interface $o\times o$ to different sharable variables, we use a projection. For example, the following expression binds the two instances of o to sharable variables y and z:

$$\textbf{proj}\ (\lambda x:\mathbf{1}.\,(x,x))\ \texttt{zero}\ \textbf{of}\ (y,z)\ \textbf{in}\ \mathsf{and}\ y\ z.$$

By associating a new hardware component $\mathsf{and}[i_1,i_2,o'']$ with and, we obtain the following mapping:

$$\begin{aligned}&\textbf{proj}\ (\lambda x:\mathbf{1}.\,(x,x))\ \texttt{zero}\ \textbf{of}\ (y,z)\ \textbf{in}\ \mathsf{and}\ y\ z\\ &\quad\Rightarrow (\{\mathsf{pt}[i,o],\mathsf{0}[o'],\mathsf{and}[i_1,i_2,o'']\},\{o'\mapsto i,o\mapsto i_1,o\mapsto i_2\},o'').\end{aligned}$$

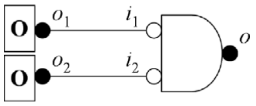


Another expression $(\lambda x : \mathbf{1}. \texttt{and}\ x\ x)\ \texttt{zero}$ (with no projection in it) produces a hardware circuit with the same structure:

$$(\lambda x : \mathbf{1}. \texttt{and}\ x\ x)\ \texttt{zero} \Rightarrow (\{\mathsf{pt}[i,o], \mathsf{0}[o'], \mathsf{and}[i_1,i_2,o'']\}, \{o' \mapsto i, o \mapsto i_1, o \mapsto i_2\}, o'')$$

If we simplify it to $\texttt{and zero zero}$, however, we obtain a hardware circuit with a different structure:[2]

$$\texttt{and zero zero} \Rightarrow (\{\mathsf{0}[o_1], \mathsf{0}[o_2], \mathsf{and}[i_1,i_2,o]\}, \{o_1 \mapsto i_1, o_2 \mapsto i_2\}, o).$$

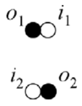


3.1.2 *Linear input function*

Consider another identify function $\hat{\lambda} f : \mathbf{1} \rightarrow \mathbf{1}. f$ of type $(\mathbf{1} \rightarrow \mathbf{1}) \multimap (\mathbf{1} \rightarrow \mathbf{1})$. First, we have to specify an input interface and an output interface for linear variable f. Recall from the previous example that an output interface of type $\mathbf{1} \rightarrow \mathbf{1}$ consists of a pair of input and output terminals. Thus an input interface of type $\mathbf{1} \rightarrow \mathbf{1}$ consists of a pair of output and input terminals.

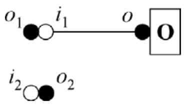
It is important that these output and input terminals for the input interface, say, o and i, must belong to separate connection points so that we can exploit an external hardware circuit providing an output interface of type $\mathbf{1} \rightarrow \mathbf{1}$ in the intended way, i.e., by transmitting a bitstream via o (as input to the external hardware circuit) and receiving the resultant bitstream via i (as output from the external hardware circuit). If o and i happen to belong to the same connection point, any hardware circuit connected with the input interface degenerates into a closed-loop circuit. Thus we associate two separate connection points $\mathsf{pt}[i_1, o_1]$ and $\mathsf{pt}[i_2, o_2]$ with f, and use $o_1 \rightarrow i_2$ for its input interface and $i_1 \rightarrow o_2$ for its output interface. Then the output interface for $\hat{\lambda} f : \mathbf{1} \rightarrow \mathbf{1}. f$ becomes $(o_1 \rightarrow i_2) \multimap (i_1 \rightarrow o_2)$:

$$\hat{\lambda} f : \mathbf{1} \rightarrow \mathbf{1}. f \Rightarrow (\{\mathsf{pt}[i_1,o_1], \mathsf{pt}[i_2,o_2]\}, \emptyset, (o_1 \rightarrow i_2) \multimap (i_1 \rightarrow o_2)).$$



If the body changes from f to f $\texttt{zero}$, we associate a new hardware component $\mathsf{0}[o]$ with $\texttt{zero}$ and connect output terminal o to the input terminal in the output interface for f, namely i_1. The output interface changes to $(o_1 \rightarrow i_2) \multimap o_2$ because i_1 in the output interface for f is now hidden:

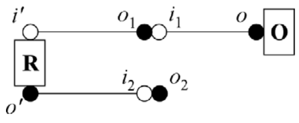
$$\hat{\lambda} f : \mathbf{1} \rightarrow \mathbf{1}. f\ \texttt{zero} \Rightarrow (\{\mathsf{pt}[i_1,o_1], \mathsf{pt}[i_2,o_2], \mathsf{0}[o]\}, \{o \mapsto i_1\}, (o_1 \rightarrow i_2) \multimap o_2)$$



[2] Thus applying a β-reduction to an expression does not necessarily preserve the structure of the hardware circuit that it describes.

Let us apply the resultant function to reg. We associate a new hardware component $\mathsf{reg}[i', o']$ with reg, and introduce two connection constraints so that the output interface $i' \to o'$ for reg matches with the input interface $o_1 \to i_2$ for f. The output interface changes to o_2, which is now the only terminal exposed to external hardware components:

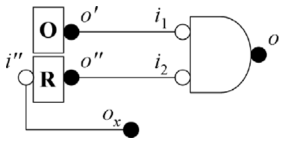
$$\begin{aligned}&(\hat{\lambda} f : \mathbf{1} \to \mathbf{1}.\, f\ \mathtt{zero})\,\hat{}\,\mathtt{reg}\\ &\quad \Rightarrow (\{\mathsf{pt}[i_1, o_1], \mathsf{pt}[i_2, o_2], \mathsf{0}[o], \mathsf{reg}[i', o']\}, \{o \mapsto i_1, o_1 \mapsto i', o' \mapsto i_2\}, o_2).\end{aligned}$$



3.1.3 Fixed-point expression

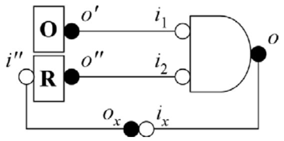
A fixed-point expression **fix** $x : \theta.\, e$ builds a feedback circuit whose output is accessible to itself via sharable variable x. As an example, let us build a feedback circuit from **fix** $x : \mathbf{1}.$ and zero (reg x). We associate hardware components $\mathsf{and}[i_1, i_2, o]$, $\mathsf{0}[o']$, and $\mathsf{reg}[i'', o'']$ with and, zero, and reg, respectively. Under the assumption that the output interface for x is a hypothetical output terminal o_x, the body and zero (reg x) generates a hardware circuit connecting o_x to i'' and providing an output interface o:

$$\mathtt{and\ zero\ (reg}\ x) \Rightarrow (\{\mathsf{and}[i_1, i_2, o], \mathsf{0}[o'], \mathsf{reg}[i'', o'']\}, \{o' \mapsto i_1, o'' \mapsto i_2, o_x \mapsto i''\}, o).$$



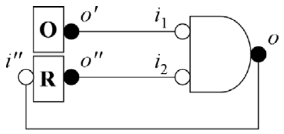
Now, there are two ways to complete the feedback circuit, depending on whether we generate a connection point for x or not. First, we associate an actual connection point $\mathsf{pt}[i_x, o_x]$ with x and connect o to i_x:

$$\begin{aligned}&\textbf{fix}\ x : \mathbf{1}.\ \mathtt{and\ zero\ (reg}\ x)\\ &\quad \Rightarrow (\{\mathsf{and}[i_1, i_2, o], \mathsf{0}[o'], \mathsf{reg}[i'', o''], \mathsf{pt}[i_x, o_x]\}, \{o' \mapsto i_1, o'' \mapsto i_2, o_x \mapsto i'', o \mapsto i_x\}, o).\end{aligned}$$



Second, we do not generate such a connection point by identifying output terminal o_x with output interface o for the whole hardware circuit, i.e., by enforcing $o_x = o$:

$$\begin{aligned}&\textbf{fix}\ x : \mathbf{1}.\ \mathtt{and\ zero\ (reg}\ x)\\ &\quad \Rightarrow (\{\mathsf{and}[i_1, i_2, o], \mathsf{0}[o'], \mathsf{reg}[i'', o'']\}, \{o' \mapsto i_1, o'' \mapsto i_2, o \mapsto i''\}, o).\end{aligned}$$



The two hardware circuits are operationally equivalent because a connection point does not occupy a physical area.

A problem with the second approach is that a well-typed fixed-point expression may not have a corresponding hardware circuit and the soundness of the type system (Theorem 4.7) fails to hold. To be specific, $\mathbf{fix}\ x:\theta.\, e$ has no corresponding hardware circuit if variable x and body e have common output terminals in their output interfaces. For example, $\mathbf{fix}\ x:\mathbf{1}.\, x$, a well-typed fixed-point expression of type $\mathbf{1}$, produces a hardware circuit consisting only of a single output terminal, which cannot be represented as a triple (H, C, O). Thus, we are led to use the first approach in the translation of $l\lambda$, which does not need to deal with equations between output terminals such as $o_x = o$. (In order to use the second approach, we need a more sophisticated type system that rejects such abnormal fixed-point expressions.)

A fixed-point expression in $l\lambda$ does not permit a linear variable in its binder. A fixed-point expression of the form $\mathbf{fix}\ f:\kappa.\, e$ is certainly conceivable, but interpreting it as a description of a hardware circuit necessitates behavioral hardware synthesis, which, unlike structural hardware description, rewrites an expression by analyzing its behavior so that it can be mapped directly to a hardware circuit. (For example, when only a single-adder circuit is available, the translation of an expression adding three integers needs to insert an additional control circuit and thus involves behavioral hardware synthesis.) As $l\lambda$ is concerned only with structural hardware description, we do not consider fixed-point expressions permitting linear variables in their binders.

3.2 *Translation of types*

We have seen that a binder $x : \tau$ or $f : \tau$ generates connection points in accordance with type τ. We split terminals in these connection points into an input interface and an output interface for variable x or f. Hence, we need rules for translating types before developing rules for translating expressions.

We use a judgment $\tau \lhd\!\rhd (H, I, O)$ to mean that a variable of type τ may use I and O as its input and output interfaces, and that all terminals in I and O belong to connection points in H. Operationally, we may think of $\tau \lhd\!\rhd (H, I, O)$ as translating input τ into output (H, I, O) (where identifiers for terminals in H are not uniquely determined by τ). Thus, given a binder $x : \tau$ or $f : \tau$, we first generate H, I, and O such that $\tau \lhd\!\rhd (H, I, O)$, and then use I and O as input and output interfaces for x or f.

Figure 4 shows the rules for the judgment $\tau \lhd\!\rhd (H, I, O)$. We continue to write $\sharp\{S, S'\}$ to mean that S and S' share no terminals, i.e., $|S| \cap |S'| = \emptyset$, where S and S' now range over sets of hardware components as well as output and input interfaces. Note that $\mathbf{1}\lhd\!\rhd$ is the only rule that actually generates a connection point, which implies that H in $\tau \lhd\!\rhd (H, I, O)$ has the same number of connection points as the number of $\mathbf{1}$'s in τ.

Lemma 3.1
If $\tau \lhd\!\rhd (H, I, O)$, then

(1) $|H| = |I| \cup |O|$,

(2) $\sharp\{I, O\}$,

(3) $I \lhd \tau$ and $O \rhd \tau$.

$$\frac{}{\mathbf{1} \lhd\!\rhd (\mathsf{pt}[i,o], i, o)}\ \mathbf{1}\!\lhd\!\rhd$$

$$\frac{\theta_1 \lhd\!\rhd (H_1, I_1, O_1) \quad \theta_2 \lhd\!\rhd (H_2, I_2, O_2) \quad \sharp\{H_1, H_2\}}{\theta_1 \times \theta_2 \lhd\!\rhd (H_1 \cup H_2, I_1 \times I_2, O_1 \times O_2)}\ \times\!\lhd\!\rhd$$

$$\frac{\theta \lhd\!\rhd (H, I, O) \quad \tau \lhd\!\rhd (H', I', O') \quad \sharp\{H, H'\}}{\theta \to \tau \lhd\!\rhd (H \cup H', O \to \mathrm{I}', \mathrm{I} \to O')}\ \to\!\lhd\!\rhd$$

$$\frac{\kappa \lhd\!\rhd (H, I, O) \quad \tau \lhd\!\rhd (H', I', O') \quad \sharp\{H, H'\}}{\kappa \multimap \tau \lhd\!\rhd (H \cup H', O \multimap I', I \multimap O')}\ \multimap\!\lhd\!\rhd$$

Fig. 4. Rules for translating types.

3.3 Translation of expressions

For translating expressions, we generalize the judgment $e \Rightarrow (H, C, O)$ to a new judgment $\mathscr{G}; \mathscr{D} \vdash e \Rightarrow (H, C, O)$, which uses *sharable output context* $\mathscr{G}$ (corresponding to a sharable typing context Γ) and *linear output context* $\mathscr{D}$ (corresponding to a linear typing context Δ) to record output interfaces for variables in e:

$$\begin{array}{rrcl} \text{sharable output context} & \mathscr{G} & ::= & \cdot \mid \mathscr{G}, x :: O \\ \text{linear output context} & \mathscr{D} & ::= & \cdot \mid \mathscr{D}, f :: O. \end{array}$$

A binding $x :: O$ in $\mathscr{G}$ means that O is the output interface for variable x; as in the type system of $l\lambda$, we may use x zero or more times. Similarly, a binding $f :: O$ in $\mathscr{D}$ means that O is the output interface for variable f, and we must use f exactly once.

The judgment $\mathscr{G}; \mathscr{D} \vdash e \Rightarrow (H, C, O)$ requires that $\mathscr{G}$ and $\mathscr{D}$ be well-formed with respect to certain typing contexts Γ and Δ as follows:

- We write $\mathscr{G} \sim \Gamma$ to mean that $x :: O \in \mathscr{G}$ holds if and only if $x : \theta \in \Gamma$ and $O \rhd \theta$ hold, i.e., O has the same type as x.
- We write $\mathscr{D} \sim \Delta$ to mean that $f :: O \in \mathscr{D}$ holds if and only if $f : \kappa \in \Delta$ holds with $O \rhd \kappa$, i.e., O has the same type as f. In addition, $f_1 :: O_1 \in \mathscr{D}$ and $f_2 :: O_2 \in \mathscr{D}$ with $f_1 \neq f_2$ mean $\sharp\{O_1, O_2\}$.

Note that while output interfaces in $\mathscr{D}$ do not share terminals, output interfaces in $\mathscr{G}$ may share terminals. Then variables declared in projections (e.g., x and y in **proj** e **of** (x, y) **in** e') can reuse existing output terminals without having to generate new connection points, as will be explained later.

Figure 5 shows the rules for the judgment $\mathscr{G}; \mathscr{D} \vdash e \Rightarrow (H, C, O)$. We assume that α-conversion has been applied to every expression so that all variables in it are distinct. As before, we write $\sharp\{S, S'\}$ to mean $|S| \cap |S'| = \emptyset$. We calculate $|\mathscr{G}|$ and $|\mathscr{D}|$ as follows:

$$\begin{array}{rcl} |\mathscr{G}| & = & \bigcup\{|O| \mid x :: O \in \mathscr{G}\} \\ |\mathscr{D}| & = & \bigcup\{|O| \mid f :: O \in \mathscr{D}\}. \end{array}$$

For S and S' in $\sharp\{S, S'\}$, we allow unions of different kinds of sets written as $\mathscr{G} + \mathscr{D}$, $\mathscr{D} + H$, $\mathscr{G} + \mathscr{D} + H$, and *etc.* For such a union S of sets, we calculate $|S|$ as the union of sets of terminals calculated from individual sets in S. For example, we have $|\mathscr{G} + \mathscr{D} + H| = |\mathscr{G}| \cup |\mathscr{D}| \cup |H|$.

$$\frac{x :: O_x \in \mathscr{G}}{\mathscr{G}; \cdot \vdash x \Rightarrow (\varnothing, \varnothing, O_x)}\;\mathit{Var} \qquad \frac{}{\mathscr{G}; f :: O_f \vdash f \Rightarrow (\varnothing, \varnothing, O_f)}\;\mathit{LVar}$$

$$\frac{\theta \lhd\!\rhd (H_x, I_x, O_x) \quad \mathscr{G}, x :: O_x; \mathscr{D} \vdash e \Rightarrow (H, C, O) \quad \sharp\{O_x, \mathscr{G} + \mathscr{D}\} \quad \sharp\{I_x, \mathscr{D} + H\}}{\mathscr{G}; \mathscr{D} \vdash \lambda x{:}\theta \triangleright e \Rightarrow (H_x \cup H, C, I_x \rightarrow O)}\;{\rightarrow}I$$

$$\frac{\kappa \lhd\!\rhd (H_f, I_f, O_f) \quad \mathscr{G}; \mathscr{D}, f :: O_f \vdash e \Rightarrow (H, C, O) \quad \sharp\{O_f, \mathscr{G} + \mathscr{D}\} \quad \sharp\{I_{\mathrm{f}}, \mathscr{G} + \mathscr{D} + H\}}{\mathscr{G}; \mathscr{D} \vdash \hat{\lambda} f{:}\kappa \triangleright e \Rightarrow (H_f \cup H, C, I_f \multimap O)}\;{\multimap}I$$

$$\frac{\mathscr{G}; \mathscr{D}_1 \vdash e_1 \Rightarrow (H_1, C_1, I_1 \rightarrow O_1) \quad \mathscr{G}; \mathscr{D}_2 \vdash e_2 \Rightarrow (H_2, C_2, O_2) \quad \sharp\{\mathscr{D}_1 + H_1, \mathscr{D}_2 + H_2\}}{\mathscr{G}; \mathscr{D}_1, \mathscr{D}_2 \vdash e_1\ e_2 \Rightarrow (H_1 \cup H_2, C_1 \cup C_2 \cup O_2 \bowtie I_1, O_1)}\;{\rightarrow}E$$

$$\frac{\mathscr{G}; \mathscr{D}_1 \vdash e_1 \Rightarrow (H_1, C_1, I_1 \multimap O_1) \quad \mathscr{G}; \mathscr{D}_2 \vdash e_2 \Rightarrow (H_2, C_2, O_2) \quad \sharp\{\mathscr{D}_1 + H_1, \mathscr{D}_2 + H_2\}}{\mathscr{G}; \mathscr{D}_1, \mathscr{D}_2 \vdash e_1, e_2 \Rightarrow (H_1 \cup H_2, C_1 \cup C_2 \cup O_2 \bowtie I_1, O_1)}\;{\multimap}E$$

$$\frac{\mathscr{G}; \mathscr{D}_1 \vdash e_1 \Rightarrow (H_1, C_1, O_1) \quad \mathscr{G}; \mathscr{D}_2 \vdash e_2 \Rightarrow (H_2, C_2, O_2) \quad O_1 \rhd \theta_1 \quad O_2 \rhd \theta_2 \quad \sharp\{\mathscr{D}_1 + H_1, \mathscr{D}_2 + H_2\}}{\mathscr{G}; \mathscr{D}_1, \mathscr{D}_2 \vdash (e_1, e_2) \Rightarrow (H_1 \cup H_2, C_1 \cup C_2, O_1 \times O_2)}\;{\times}I$$

$$\frac{\mathscr{G}; \mathscr{D} \vdash e \Rightarrow (H, C, O_1 \times O_2) \quad \mathscr{G}, x_1 :: O_1, x_2 :: O_2; \mathscr{D}' \vdash e' \Rightarrow (H', C', O') \quad \sharp\{\mathscr{D} + H, D' + H'\}}{\mathscr{G}; \mathscr{D}, \mathscr{D}' \vdash \textbf{proj}\ e\ \textbf{of}\ (x_1, x_2)\ \textbf{in}\ e' \Rightarrow (H \cup H', C \cup C', O')}\;{\times}E$$

$$\frac{\theta \lhd\!\rhd (H_x, I_x, O_x) \quad \mathscr{G}, x :: O_{\mathrm{x}}; \mathscr{D} \vdash e \Rightarrow (H, C, O) \quad \sharp\{O_x, \mathscr{G} + \mathscr{D}\} \quad \sharp\{I_{\mathrm{x}}, \mathscr{D} + H\}}{\mathscr{G}; \mathscr{D} \vdash \textbf{fix}\ x{:}\theta \triangleright e \Rightarrow (H_x \cup H, C \cup O \bowtie I_x, O)}\;\mathit{Fix}$$

Fig. 5. Rules for translating expressions.

Each rule in Figure 5 has its counterpart in the type system of $l\lambda$ (e.g., *Var* for Var, *LVar* for LVar, and so on). Here are a few further remarks:

- By the rules *Var* and *LVar*, variables generate no new hardware components and connection constraints.
- Connection points are generated only by the rules $\rightarrow I$, $\multimap I$, and *Fix*.
- Connection constraints are generated only by the rules $\rightarrow E$, $\multimap E$, and *Fix*.
- In the rules $\rightarrow I$ and *Fix*, $\sharp\{I_x, \mathscr{D} + H\}$ implies $\sharp\{I_x, \mathscr{G} + \mathscr{D} + H\}$ because $I_x \lhd \theta$ holds by Lemma 3.1, $|I_x|$ contains no output terminals by Proposition 2.3, and $|\mathscr{G}|$ contains only output terminals by Proposition 2.3.
- In the rule $\multimap I$, $\mathscr{D}, f :: O_f \sim \Delta, f : \kappa$ holds from $\mathscr{D} \sim \Delta$, $O_f \rhd \kappa$, and $\sharp\{O_f, \mathscr{D}\}$.
- The premise of the rule $\times I$ requires that both O_1 and O_2 be output interfaces for sharable types.
- The rule $\times E$ binds sharable variables x_1 and x_2 to output interfaces O_1 and O_2, which may share output terminals with $\mathscr{G}$. It explains why output interfaces in a sharable output context may share terminals.
- Because of sharable variables declared in projections, $\mathscr{G}; \mathscr{D} \vdash e \Rightarrow (H, C, O)$ may not satisfy $\sharp\{\mathscr{G}, \mathscr{D}\}$. That is, $\mathscr{G}$ and $\mathscr{D}$ may not be completely disjoint. For example, assuming $f :: I \rightarrow O_1 \times O_2$, a projection $\textbf{proj}\ f\ e'\ \textbf{of}\ (x_1, x_2)\ \textbf{in}\ e$ eventually binds x_1 and x_2 to O_1 and O_2, respectively.

In addition to the rules in Figure 5, we need a rule for each constant. A constant generates a corresponding hardware component and an output interface consistent with its type. We assign a fresh identifier to each terminal in the hardware component so that different instances of the same constant result in separate hardware components. For example, assuming that and has type $\mathbf{1} \rightarrow (\mathbf{1} \rightarrow \mathbf{1})$, we may use the following rule:

$$\frac{\{i_1, i_2, o\} \not\subset |\mathscr{G}| \quad i_1 \neq i_2}{\mathscr{G}; \cdot \vdash \mathsf{and} \Rightarrow (\{\mathsf{and}[i_1, i_2, o]\}, \varnothing, i_1 \rightarrow (i_2 \rightarrow o))} \ \textit{And}$$

Although we may read $\mathscr{G}; \mathscr{D} \vdash e \Rightarrow (H, C, O)$ operationally by regarding $\mathscr{G}$, $\mathscr{D}$, and e as input and (H, C, O) as output, all the rules in Figure 5 are written in a declarative style. For example, no rule specifies how to generate identifiers for terminals; rather each rule only specifies that identifiers for terminals be all different.

4 Properties of $l\lambda$

This section investigates properties of $l\lambda$. We prove the soundness and completeness of the translation of $l\lambda$ with respect to realizability:

- Soundness: expressions are mapped only to realizable hardware circuits (Theorem 4.1).
- Completeness: every realizable hardware circuit has a corresponding expression (Theorem 4.12).

We also prove the soundness and completeness of the type system of $l\lambda$ with respect to the translation:

- Soundness: all well-typed expressions are mapped to hardware circuits (Theorem 4.7).
- Completeness: only well-typed expressions are mapped to hardware circuits (Theorem 4.1).

In combination, these properties imply that all realizable hardware circuits have corresponding well-typed expressions and that all well-typed expressions describe realizable hardware circuits.

4.1 Soundness of the translation and completeness of the type system

Theorem 4.1
If $\cdot; \cdot \vdash e \Rightarrow (H, C, O)$, then $\cdot; \cdot \vdash e : \tau$ and $O \rhd \tau$ for some type τ, and C is realizable.

Theorem 4.1 proves both the soundness of the translation with respect to realizability and the completeness of the type system with respect to the translation at once. It implies that only well-typed expressions are mapped to hardware circuits, which are always realizable. Its proof follows from Propositions 4.4 and 4.6.

Lemma 4.2
If $\mathscr{G}; \mathscr{D} \vdash e \Rightarrow (H, C, O)$, then
(1) $|C| \subset |\mathscr{G} + \mathscr{D} + H|$,
(2) $|O| \subset |\mathscr{G} + \mathscr{D} + H|$.

Proof
By induction on the structure of the proof of $\mathscr{G}; \mathscr{D} \vdash e \Rightarrow (H, C, O)$. The proof does not require $\mathscr{G} \sim \Gamma$ and $\mathscr{D} \sim \Delta$. □

Corollary 4.3
If $\mathscr{G} \sim \Gamma$, $\mathscr{D} \sim \Delta$, and $\mathscr{G};\mathscr{D} \vdash e \Rightarrow (H, C, O)$, then
(1) $i \in |C|$ implies $i \in |\mathscr{D} + H|$,
(2) $i \in |O|$ implies $i \in |\mathscr{D} + H|$.

Proposition 4.4
If $\mathscr{G};\mathscr{D} \vdash e \Rightarrow (H, C, O)$ with $\mathscr{G} \sim \Gamma$ and $\mathscr{D} \sim \Delta$, then $\Gamma;\Delta \vdash e : \tau$ and $O \rhd \tau$.

Proof
See Appendix A. □

Lemma 4.5
If $\mathscr{G};\mathscr{D} \vdash e \Rightarrow (H, C, O)$ with $\mathscr{G} \sim \Gamma$ and $\mathscr{D} \sim \Delta$, then if $i \in |O|$, then $o \mapsto i \notin C$.

Proof
See Appendix A. □

Proposition 4.6
If $\mathscr{G};\mathscr{D} \vdash e \Rightarrow (H, C, O)$ with $\mathscr{G} \sim \Gamma$ and $\mathscr{D} \sim \Delta$, then C is realizable.

Proof
By induction on the structure of the proof of $\mathscr{G};\mathscr{D} \vdash e \Rightarrow (H, C, O)$. The proof reuses the result from the proof of Proposition 4.4 that all output contexts are well formed. □

4.2 Soundness of the type system

Theorem 4.7
If $\cdot;\cdot \vdash e : \tau$, then there exists (H, C, O) such that $\cdot;\cdot \vdash e \Rightarrow (H, C, O)$.

Theorem 4.7 proves the soundness of the type system with respect to the translation: all well-typed expressions are mapped to hardware circuits. Its proof follows from Proposition 4.9.

Lemma 4.8
If $\mathscr{G} \sim \Gamma$, $\mathscr{D} \sim \Delta$, and $\mathscr{G};\mathscr{D} \vdash e \Rightarrow (H, C, O)$, then $\sharp\{\mathscr{G} + \mathscr{D}, H\}$.

Proof
By induction on the structure of the proof of $\mathscr{G};\mathscr{D} \vdash e \Rightarrow (H, C, O)$. □

Proposition 4.9
If $\Gamma;\Delta \vdash e : \tau$, then for $\mathscr{G} \sim \Gamma$ and $\mathscr{D} \sim \Delta$, there exists (H, C, O) such that $\mathscr{G};\mathscr{D} \vdash e \Rightarrow (H, C, O)$ and $O \rhd \tau$.

Since the translation uses a declarative style, a strict proof of Proposition 4.9 requires us to rewrite all the rules in Figure 5 in an algorithmic style. Instead of rewriting the rules, we operationally interpret the judgments $\tau \lhd\rhd (H, I, O)$ and $\mathscr{G};\mathscr{D} \vdash e \Rightarrow (H, C, O)$ to simplify the proof.

For $\tau \lhd\rhd (H, I, O)$, we take τ as input and (H, I, O) as output. Since all terminals are eventually introduced by the rule $\mathbf{1}\lhd\rhd$ (except for those belonging to atomic hardware components), we assume that each application of the rule $\mathbf{1}\lhd\rhd$ creates fresh identifiers i and o. Then $\tau \lhd\rhd (H, I, O)$ implies $\sharp\{I, S\}$ and $\sharp\{O, S'\}$ for any S and S'. (If not, we just generate different identifiers not found in S and S'.) Thus the proof of Proposition 4.9 assumes that the last two premises in each of the rules $\rightarrow I$, $\multimap I$, and Fix automatically hold.

For $\mathscr{G}; \mathscr{D} \vdash e \Rightarrow (H, C, O)$, we take $\mathscr{G}$, $\mathscr{D}$, and e as input and (H, C, O) as output. Since H shares no terminals with $\mathscr{G}$ and $\mathscr{D}$ by Lemma 4.8, we further assume that all terminals in H are assigned fresh identifiers. That is, given $\mathscr{G}; \mathscr{D} \vdash e \Rightarrow (H, C, O)$, we assume that $\sharp\{H, S\}$ holds for any S. (If not, we just generate different identifiers not found in S.) Thus the proof of Proposition 4.9 assumes that the last premise in each of the rules $\rightarrow E$, $\multimap E$, $\times I$, $\times E$ automatically holds.

Proof of Proposition 4.9

By induction on the structure of the proof of $\Gamma; \Delta \vdash e : \tau$. Details are in Appendix A. $\square$

4.3 Completeness of the translation

In order for $l\lambda$ to be an intermediate language for hardware description, its translation should be not only sound with respect to realizability, but also complete in the sense that every realizable hardware circuit has a corresponding expression in $l\lambda$. Below, we first explain informally that $l\lambda$ is indeed expressive enough to describe every realizable hardware circuit, and then give a formal proof (Theorem 4.12). We assume tuple types $\theta_1 \times \cdots \times \theta_n$ generalizing product types, tuples $(e_1, \ldots, e_n)$ generalizing pairs, tuple patterns $(x_1, \ldots, x_n)$ generalizing pair patterns (where $n \geqslant 1$), and allow tuple patterns in fixed-point expressions and projections (e.g., $\mathbf{fix}\ (x_1, \ldots, x_n) : \theta. e$ and $\mathbf{proj}\ e\ \mathbf{of}\ (x_1, \ldots, x_n)\ \mathbf{in}\ e'$).

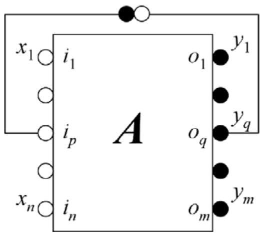
Consider a hardware circuit $\mathscr{A}$ with n input terminals $i_1, \ldots, i_n$ and m output terminals $o_1, \ldots, o_m$. Here, we enumerate all output terminals belonging to $\mathscr{A}$, but exclude those "hidden" input terminals to which wires are already connected. That is, we consider only those input terminals exposed to external hardware circuits. We assume that $\mathscr{A}$ is described by an expression

$$\lambda x_1 : \mathbf{1}. \cdots \lambda x_n : \mathbf{1}. e$$

where x_p corresponds to input terminal i_p $(1 \leqslant p \leqslant n)$ and e has type $\mathbf{1} \times \cdots \times \mathbf{1}$ whose qth element corresponds to output terminal o_q $(1 \leqslant q \leqslant m)$.

We observe that there are two ways to augment $\mathscr{A}$. First, we add a wire connecting an output terminal o_q to an input terminal i_p. We describe the resultant hardware circuit by exploiting a fixed-point expression with dummy variables $y_1, \ldots, y_{q-1}, y_{q+1}, \ldots, y_m$:

$$\begin{array}{l} \lambda x_1 : \mathbf{1}. \cdots \lambda x_{p-1} : \mathbf{1}. \lambda x_{p+1} : \mathbf{1}. \cdots \lambda x_n : \mathbf{1}. \\ \quad \mathbf{fix}\ (y_1, \ldots, y_{q-1}, x_p, y_{q+1}, \ldots, y_m) : \mathbf{1} \times \cdots \times \mathbf{1}. e \end{array}$$



Second, we combine $\mathscr{A}$ with another hardware circuit $\mathscr{A}'$ (without linking them with wires). Let us assume that $\mathscr{A}'$ is described by

$$\lambda x_1' : \mathbf{1}. \cdots \lambda x_l' : \mathbf{1}. e'$$

where x_r' corresponds to its rth input terminal ($1 \leqslant r \leqslant l$) and e' produces k output terminals. Then $\mathscr{A}$ combined with $\mathscr{A}'$ is described by the following expression:

$$\begin{array}{l} \lambda x_1 : \mathbf{1}. \cdots \lambda x_n : \mathbf{1}. \lambda x_1' : \mathbf{1}. \cdots \lambda x_l' : \mathbf{1}. \\ \quad \mathbf{proj}\ e\ \mathbf{of}\ (y_1, \ldots, y_m)\ \mathbf{in} \\ \quad\quad \mathbf{proj}\ e'\ \mathbf{of}\ (y_1', \ldots, y_k')\ \mathbf{in}\ (y_1, \ldots, y_m, y_1', \ldots, y_k'). \end{array}$$

Note that in both cases, the resultant hardware circuit is described by a function declaring the same number of sharable variables as the number of input terminals exposed to external hardware circuits, as is the case for the original hardware circuit $\mathscr{A}$.

Since every hardware circuit is eventually decomposed into atomic hardware components and connecting wires, it now suffices to show that each atomic hardware component with n input terminals can be described by a function of the form $\lambda x_1 : \mathbf{1}. \cdots \lambda x_n : \mathbf{1}. e$. In our case, the problem reduces to converting each constant to such a function, which is trivial (e.g., and to $\lambda x_1 : \mathbf{1}. \lambda x_2 : \mathbf{1}. \mathsf{and}\ x_1\ x_2$).

Theorem 4.12 formally states the completeness of the translation. It uses an extended notion of realizability that considers a pair of hardware components H and connection constraints C (Definition 4.10) and the notion of reduction (Definition 4.11).

Definition 4.10
(H, C) is realizable if H is finite and nonempty, C is realizable, and $|C| \subset |H|$.

Definition 4.11
Suppose that H consists of atomic hardware components with no connection points and that H' is H augmented with a set of connection points. We say that (H', C') reduces to (H, C) if the following two conditions hold:

- If $o \mapsto i \in C$ and $\{o, i\} \subset |H|$,
  then $\{o \mapsto i_1, o_1 \mapsto i_2, \ldots, o_{n-1} \mapsto i_n, o_n \mapsto i\} \subset C'$ (where $n \geqslant 0$)
  and $\{\mathsf{pt}[i_1, o_1], \mathsf{pt}[i_2, o_2], \ldots, \mathsf{pt}[i_{n-1}, o_{n-1}], \mathsf{pt}[i_n, o_n]\} \subset H'$.
  That is, if a bitstream flows from o to i in (H, C), so does it in (H', C'), but via a sequence of intermediate connection points.
- If $\{o \mapsto i_1, o_1 \mapsto i_2, \ldots, o_{n-1} \mapsto i_n, o_n \mapsto i\} \subset C'$ (where $n \geqslant 0$), $\{\mathsf{pt}[i_1, o_1], \mathsf{pt}[i_2, o_2], \ldots, \mathsf{pt}[i_{n-1}, o_{n-1}], \mathsf{pt}[i_n, o_n]\} \subset H'$, and $\{o, i\} \subset |H|$,
  then $o \mapsto i \in C$.

That is, if a bitstream flows from o to i in (H', C'), so does it in (H, C), but without visiting intermediate connection points (where both o and i are assumed to belong to H).

Definition 4.11 implies that if (H', C') reduces to (H, C), two hardware circuits specified by (H', C') and (H, C) are operationally equivalent.

Theorem 4.12
Suppose that (H, C) is realizable and H contains no connection points. Then there exists an expression e such that $\cdot;\cdot \vdash e \Rightarrow (H', C', O')$, H' is H augmented with a set of connection points, and (H', C') reduces to (H, C).

For the sake of proving the completeness of the translation, it is safe to assume in Theorem 4.12 that H contains no connection points, since connection points are not actual hardware components. That is, we exploit the fact that every hardware circuit has a corresponding realizable pair (H, C) such that H contains only atomic hardware components and no connection points.

Instead of incrementally building e from (H, C), the proof of Theorem 4.12 analyzes (H, C) at once and builds e in a single step, which allows us to dispense with projections in e and considerably simplifies the proof. This result implies that for the purpose of describing hardware circuits with no connection points, the first-order subset of $l\lambda$ without projections suffices, which in turn implies that linear input functions and applications can both be regarded as metaprogramming constructs such that $(\hat{\lambda} f : \kappa. e)\,\hat{}\,e'$ is rewritten as $[e'/f]e$, which replaces the only occurrence of f in e by e'. See Appendix B for the proof of Theorem 4.12.

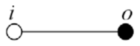
5 Discussion

This section presents an alternative translation of $l\lambda$ and explains how to eliminate redundant wires in hardware circuits generated from expressions.

5.1 *Mapping variables to wires*

The translation of $l\lambda$ maps variables to connection points, which are hardware components with their own input and output terminals. Since all input and output terminals belong to some hardware components, wires are secondary components, which have no input and output terminals of their own and serve only to connect other hardware components.

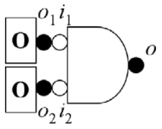
An alternative translation of $l\lambda$ dispenses with connection points and maps variables directly to wires. The idea is to treat wires as independent hardware components with their own input and output terminals. We can obtain such a translation by reusing the previous translation of $l\lambda$ with a different interpretation of $\mathsf{pt}[i, o]$ and $o \mapsto i$. Specifically, we use $\mathsf{pt}[i, o]$ to represent a wire with input terminal i and output terminal o and a connection constraint $o \mapsto i$ to specify that o and i be placed at the same physical location:

$\mathsf{pt}[i, o] \Leftrightarrow$ 

$o \mapsto i \Leftrightarrow$

The new translation is unrealistic, however, because closed expressions with no variables produce no wires at all. For example, `and zero zero` is mapped to a hardware circuit with no wires:

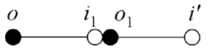
$$\texttt{and zero zero} \Rightarrow (\{\mathsf{0}[o_1], \mathsf{0}[o_2], \mathsf{and}[i_1, i_2, o]\}, \{o_1 \mapsto i_1, o_2 \mapsto i_2\}, o)$$



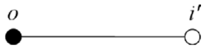
If we again choose to realize connection constraints as wires, it suffices to interpret $\mathsf{pt}[i, o]$ as a wire of zero length, i.e., as a connection point. Then we obtain the original translation of $l\lambda$ given in Section 3.

5.2 Eliminating redundant wires

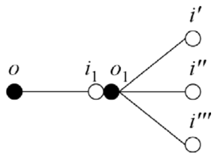
The translation of $l\lambda$ ensures that well-typed expressions are always mapped to realizable hardware circuits, but it sometimes produces redundant wires if all connection constraints are realized as wires. For example, $(\hat{\lambda} f : \mathbf{1} \to \mathbf{1}.\, f\ \texttt{zero})\,\hat{}\,\texttt{reg}$ in Section 3.1 produces two wires linked via a connection point $\mathsf{pt}[i_1, o_1]$:



Since the bitstream emitted from output terminal o eventually arrives at input terminal i', it is safe to merge the two wires into a single wire directly connecting o to i':



The merged wire results from eliminating the left wire (connecting o to i_1) and stretching the right wire (connecting o_1 to i') over to output terminal o. Note that eliminating the right wire and stretching the left wire does not work in general, because multiple wires can be connected to output terminal o_1, as in the following example:



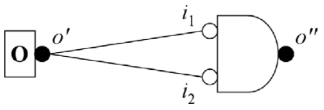
If we wish to eliminate such redundant wires, we can treat input terminals of connection points in the following way. We write $\overline{o}$ for the input terminal of a connection point whose output terminal is o. Now every connection point is written as $\mathsf{pt}[\overline{o}, o]$:

$$\frac{}{\mathbf{1} \lhd\!\rhd (\mathsf{pt}[\overline{o}, o], \overline{o}, o)}\ \mathbf{1}\lhd\!\rhd.$$

We realize $o \mapsto i$ as a wire connecting o to i as before, but interpret $o \mapsto \overline{o'}$ as an equation $o = o'$, in the presence of which every connection constraint $o' \mapsto i$ is automatically replaced by $o \mapsto i$, and the connection point $\mathsf{pt}[\overline{o'}, o']$ is removed. Thus

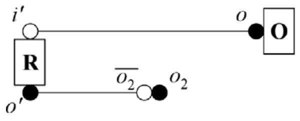
$o \mapsto \overline{o'}$ effectively superimposes o' on o and eliminates otherwise redundant wires. As an example, $(\lambda x : \mathbf{1}.\, \mathsf{and}\ x\ x)$ `zero` in Section 3.1 now produces a hardware circuit with no redundant wire:

$$(\lambda x : \mathbf{1}.\, \mathsf{and}\ x\ x)\ \mathtt{zero} \Rightarrow (\{\mathsf{0}[o'], \mathsf{and}[i_1, i_2, o'']\}, \{o' \mapsto i_1, o' \mapsto i_2\}, o'')$$



It does not, however, suggest that all connection points are unnecessary. For example, $(\hat{\lambda} f : \mathbf{1} \to \mathbf{1}.\, f\ \mathtt{zero})\hat{}\ \mathtt{reg}$ in Section 3.1 still requires a connection point:

$$(\hat{\lambda} f : \mathbf{1} \to \mathbf{1}.\, f\ \mathtt{zero})\hat{}\ \mathtt{reg} \Rightarrow (\{\mathsf{pt}[\overline{o_2}, o_2], \mathsf{0}[o], \mathsf{reg}[i', o']\}, \{o \mapsto i', o' \mapsto \overline{o_2}\}, o_2)$$



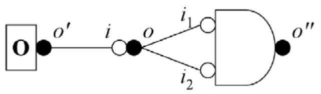
If we wish to realize connection points as physical marks so that hardware circuits with no redundant wires are distinguished from operationally equivalent hardware circuits with redundant wires, we need an additional construct in $l\lambda$ that associates output terminals with variables without creating new connection points. For example, we could permit single variables as patterns in projections:

$$\frac{\Gamma; \Delta \vdash e : \theta \quad \Gamma, x : \theta; \Delta' \vdash e' : \tau}{\Gamma; \Delta, \Delta' \vdash \mathbf{proj}\ e\ \mathbf{of}\ x\ \mathbf{in}\ e' : \tau}\ \times\mathsf{E}'$$

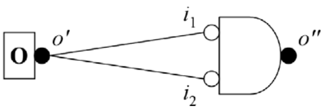
$$\frac{\mathscr{G}; \mathscr{D} \vdash e \Rightarrow (H, C, O) \quad \mathscr{G}, x :: O; \mathscr{D}' \vdash e' \Rightarrow (H', C', O') \quad \sharp\{\mathscr{D} + H, \mathscr{D}' + H'\}}{\mathscr{G}; \mathscr{D}, \mathscr{D}' \vdash \mathbf{proj}\ e\ \mathbf{of}\ x\ \mathbf{in}\ e' \Rightarrow (H \cup H', C \cup C', O')}\ \times E'$$

Then, for example, $(\lambda x : \mathbf{1}.\, \mathsf{and}\ x\ x)$ `zero` and **proj** `zero` **of** x **in** and $x\ x$ produce operationally equivalent but structurally different hardware circuits:

$(\lambda x : \mathbf{1}.\, \mathsf{and}\ x\ x)\ \mathtt{zero} \quad \Rightarrow$



proj `zero` **of** x **in** and $x\ x \quad \Rightarrow$



6 Extension of $l\lambda$

As an intermediate language for hardware description, $l\lambda$ is not intended as a hardware description language in itself. Nevertheless a simple extension of $l\lambda$ gives a hardware description language that is expressive enough to describe nontrivial hardware circuits in a concise way. This section presents an extension of $l\lambda$ with polymorphism and an implementation of an FFT circuit and a bitonic sorting network.

6.1 Polymorphism

As in general programming languages, an expression of polymorphic type represents a family of expressions of monomorphic type. In the case of $l\lambda$, an expression of monomorphic type describes a hardware circuit, which in turn implies that an expression of polymorphic type describes a family of hardware circuits. These hardware circuits differ only in the number of wires transmitting data streams and use essentially the same layout of hardware components.

We introduce a polymorphic type $\forall\alpha.\sigma$, where α is a type variable and σ is a metavariable ranging over polymorphic types. For the sake of simplicity, we restrict α to range over not all monomorphic types τ but only sharable types θ. (Letting α range over linear types κ as well poses no technical difficulty, but does not seem to be particularly useful.) We use $\Lambda\alpha.\,e$ for type abstractions and $e\,\langle\theta\rangle$ for type applications. In the rule $\forall I$ below, $tvar(\Gamma \cup \Delta)$ stands for the set of type variables in Γ and Δ.

$$\begin{array}{rcll}
\text{sharable type} & \theta & ::= & \cdots \mid \alpha \\
\text{polymorphic type} & \sigma & ::= & \tau \mid \forall\alpha.\sigma \\
\text{expression} & e & ::= & \cdots \mid \Lambda\alpha.\,e \mid e\,\langle\theta\rangle
\end{array}$$

$$\frac{\Gamma;\Delta \vdash e : \sigma \quad \alpha \notin tvar(\Gamma \cup \Delta)}{\Gamma;\Delta \vdash \Lambda\alpha.\,e : \forall\alpha.\sigma}\ \forall I \qquad \frac{\Gamma;\Delta \vdash e : \forall\alpha.\sigma}{\Gamma;\Delta \vdash e\,\langle\theta\rangle : [\theta/\alpha]\sigma}\ \forall E$$

Instead of extending the translation of $l\lambda$ for polymorphic types, we treat both type abstractions and type applications as metaprogramming constructs for generating expressions of monomorphic type. To be specific, without directly associating hardware circuits with type abstractions and type applications, we identify a type application of the form $(\Lambda\alpha.\,e)\,\langle\theta\rangle$ with $[\theta/\alpha]e$, which substitutes θ for all occurrences of α in e (where we assume that type variable captures do not arise):

$$(\Lambda\alpha.\,e)\,\langle\theta\rangle \;=\; [\theta/\alpha]e$$

Only when $(\Lambda\alpha.\,e)\,\langle\theta\rangle$ yields an expression of monomorphic type do we use the translation of $l\lambda$ to generate a description of a hardware circuit.

As it provides just a simple form of metaprogramming, polymorphism does not add to the expressive power of $l\lambda$. In conjunction with linear types, however, polymorphic types enable us to write various higher-order combinators within $l\lambda$ itself, thereby greatly facilitating the design of hardware circuits in which the same pattern of combining hardware components is repeated. A few examples of such higher-order combinators are given below.

6.2 Examples

We implement an FFT circuit and a bitonic sorting network in polymorphic $l\lambda$. Our implementation uses every construct available in $l\lambda$ except fixed point expressions. In addition, we use the following types and expressions all of which can be shown to be syntactic sugar.

We define a sharable type θ^{2^n} inductively on n:

$$\begin{array}{rcll} \theta^2 & = & \theta \times \theta & \\ \theta^{2^n} & = & \theta^{2^{n-1}} \times \theta^{2^{n-1}} & \quad (n > 1) \end{array}$$

We allow a pattern p in a sharable input function $\lambda p : \theta.\, e$, where p is either a sharable variable or a pair of patterns:

$$\text{pattern} \quad p \quad ::= \quad x \mid (p, p)$$

A linear input function $\hat{\lambda} f^n : \kappa.\, e$ uses f exactly n times in e. It has a linear type $\kappa \multimap^n \tau$, which is defined inductively on n:

$$\begin{array}{rcl} \kappa \multimap^1 \tau & = & \kappa \multimap \tau \\ \kappa \multimap^n \tau & = & \kappa \multimap (\kappa \multimap^{n-1} \tau) \end{array}$$

A linear input application $e\,\hat{}^{\,n}e'$ uses e' as an argument exactly n times:

$$\begin{array}{rcl} e\,\hat{}^{\,1}e' & = & e\,\hat{}\,e' \\ e\,\hat{}^{\,n}e' & = & (e\,\hat{}^{\,n-1}e')\,\hat{}\,e' \end{array}$$

Thus $e\,\hat{}^{\,n}e'$ is an abbreviation of n consecutive linear input applications, which make n syntactic copies of e'. As an example, consider two linear input functions `double` and `double2`:

$$\begin{array}{rcl} \texttt{double} & = & \hat{\lambda} g^2 : \mathbf{1} \to \mathbf{1}.\, \lambda x : \mathbf{1}.\, \mathsf{and}\ (g\ x)\ (g\ x) \\ \texttt{double2} & = & \hat{\lambda} f^2 : (\mathbf{1} \to \mathbf{1}) \multimap^2 (\mathbf{1} \to \mathbf{1}).\, \hat{\lambda} g^4 : \mathbf{1} \to \mathbf{1}.\, \lambda x : \mathbf{1}.\, \mathsf{and}\ (f\,\hat{}^{\,2}g\ x)\ (f\,\hat{}^{\,2}g\ x) \end{array}$$

The two functions have the following types:

$$\begin{array}{rcl} \texttt{double} & : & (\mathbf{1} \to \mathbf{1}) \multimap^2 (\mathbf{1} \to \mathbf{1}) \\ \texttt{double2} & : & ((\mathbf{1} \to \mathbf{1}) \multimap^2 (\mathbf{1} \to \mathbf{1})) \multimap^2 (\mathbf{1} \to \mathbf{1}) \multimap^4 (\mathbf{1} \to \mathbf{1}) \end{array}$$

Hence, `double2`$\,\hat{}^{\,2}$`double` type checks and has type $(\mathbf{1} \to \mathbf{1}) \multimap^4 (\mathbf{1} \to \mathbf{1})$. Note that when writing a higher-order combinator with a linear input type such as `double` and `double2`, we have to manually count the number of uses of its argument. For example, we observe that `double2` uses its argument g not twice but four times because $f\,\hat{}^{\,2}g$ syntactically expands to $(f\,\hat{}\,g)\,\hat{}\,g$. Since $l\lambda$ is intended as an intermediate language for hardware description, such a syntactic analysis (along with type inference) can be delegated to the translator for a higher-level hardware description language.

$e_1 \circ e_2$ composes e_1 of type $\theta' \to \theta''$ and e_2 of type $\theta \to \theta'$ to yield a sharable input function of type $\theta \to \theta''$:

$$e_1 \circ e_2 \quad = \quad \lambda x : \theta.\, e_1\ (e_2\ x)$$

We use $\circ$ as a right associative operator.

6.2.1 Fast Fourier transform

Figure 6 shows part of the code for an FFT circuit of size 16 which is inspired by the implementation in (Bjesse *et al.* 1998). The code consists of a series of declarations each of which yields a closed expression of $l\lambda$. We assume a sharable type `c` for

$$\begin{array}{lll}
\text{twoC} & : \forall\alpha.(\alpha\rightarrow\alpha)\multimap^{2}(\alpha^{2}\rightarrow\alpha^{2}) & = \Lambda\alpha.\hat{\lambda} f^{2}:\alpha\rightarrow\alpha.\lambda(x,y):\alpha^{2}.(f\,x, f\,y) \\
\text{prodC} & : \forall\alpha.(\alpha^{2}\rightarrow\alpha)\multimap^{2}(\alpha^{4}\rightarrow\alpha^{2}) & = \Lambda\alpha.\hat{\lambda} f^{2}:\alpha^{2}\rightarrow\alpha.\lambda((x,y),(z,w)):\alpha^{4}.(f\,(x,z), f\,(y,w)) \\
\text{riffleC} & : \forall\alpha.(\alpha^{2}\rightarrow\alpha^{2})\multimap^{2}(\alpha^{4}\rightarrow\alpha^{4}) & = \Lambda\alpha.\hat{\lambda} f^{2}:\alpha^{2}\rightarrow\alpha^{2}.\lambda((x,y),(z,w)):\alpha^{4}.(f\,(x,z), f\,(y,w)) \\
\text{unriffleC} & : \forall\alpha.(\alpha^{2}\rightarrow\alpha^{2})\multimap^{2}(\alpha^{4}\rightarrow\alpha^{4}) & = \\
 & \multicolumn{2}{l}{\Lambda\alpha.\hat{\lambda} f^{2}:\alpha^{2}\rightarrow\alpha^{2}.\lambda(p,q):\alpha^{4}.\,\mathbf{proj}\ f\ p\ \mathbf{of}\,(x,z)\ \mathbf{in}\ \mathbf{proj}\ f\ q\ \mathbf{of}\,(y,w)\ \mathbf{in}\ ((x,y),(z,w))}
\end{array}$$

$$\begin{array}{llcl}
\text{riffle}_1 & : & \mathrm{c}^{2}\rightarrow\mathrm{c}^{2} & = \ \lambda p:\mathrm{c}^{2}.p \\
\text{riffle}_2 & : & \mathrm{c}^{4}\rightarrow\mathrm{c}^{4} & = \ (\text{riffle}C\langle\mathrm{c}\rangle)\,\hat{}^{2}\text{riffle}_1 \\
\text{riffle}_3 & : & \mathrm{c}^{8}\rightarrow\mathrm{c}^{8} & = \ (\text{riffleC}\langle\mathrm{c}^{2}\rangle)\,\hat{}^{2}\text{riffle}_2 \\
\text{riffle}_4 & : & \mathrm{c}^{16}\rightarrow\mathrm{c}^{16} & = \ (\text{riffleC}\langle\mathrm{c}^{4}\rangle)\,\hat{}^{2}\text{riffle}_3 \\
 \\
\text{unriffle}_1 & : & \mathrm{c}^{2}\rightarrow\mathrm{c}^{2} & = \ \lambda p:c^{2}.p \\
\text{unriffle}_2 & : & \mathrm{c}^{4}\rightarrow\mathrm{c}^{4} & = \ (\text{unriffleC}\langle\mathrm{c}\rangle)\,\hat{}^{2}\text{unriffle}_1 \\
\text{unriffle}_3 & : & \mathrm{c}^{8}\rightarrow\mathrm{c}^{8} & = \ (\text{unriffleC}\langle\mathrm{c}^{2}\rangle)\,\hat{}^{2}\text{unriffle}_2 \\
\text{unriffle}_4 & : & \mathrm{c}^{16}\rightarrow\mathrm{c}^{16} & = \ (\text{unriffleC}\langle\mathrm{c}^{4}\rangle)\,\hat{}^{2}\text{unriffle}_3 \\
 \\
\text{g}_1 & : & \mathrm{c}^{2}\rightarrow\mathrm{c}^{2} & = \ \lambda p:c^{2}.\,(\text{cplus}\ p,\ \text{cminus}\ p) \\
\text{g}_2 & : & \mathrm{c}^{4}\rightarrow\mathrm{c}^{4} & = \ \text{twoC}\langle\mathrm{c}^{2}\rangle\,\hat{}^{2}\ {}_{1} \\
\text{g}_3 & : & \mathrm{c}^{8}\rightarrow\mathrm{c}^{8} & = \ \text{twoC}\langle\mathrm{c}^{4}\rangle\,\hat{}^{2}\text{g}_2 \\
\text{g}_4 & : & \mathrm{c}^{16}\rightarrow\mathrm{c}^{16} & = \ \text{twoC}\langle\mathrm{c}^{8}\rangle\,\hat{}^{2}\text{g}_3 \\
 \\
\text{bfly}_1 & : & \mathrm{c}^{2}\rightarrow\mathrm{c}^{2} & = \ \text{unriffle}_1\circ\text{g}_1\circ\text{riffle}_1 \\
\text{bfly}_2 & : & \mathrm{c}^{4}\rightarrow\mathrm{c}^{4} & = \ \text{unriffle}_2\circ\text{g}_2\circ\text{riffle}_2 \\
\text{bfly}_3 & : & \mathrm{c}^{8}\rightarrow\mathrm{c}^{8} & = \ \text{unriffle}_3\circ\text{g}_3\circ\text{riffle}_3 \\
\text{bfly}_4 & : & \mathrm{c}^{16}\rightarrow\mathrm{c}^{16} & = \ \text{unriffle}_4\circ\text{g}_4\circ\text{riffle}_4 \\
 \\
\text{prod}_1 & : & \mathrm{c}^{2}\rightarrow\mathrm{c} & = \ \text{cmult} \\
\text{prod}_2 & : & \mathrm{c}^{4}\rightarrow\mathrm{c}^{2} & = \ \text{prodC}\langle\mathrm{c}\rangle\,\hat{}^{2}\text{prod}_1 \\
\text{prod}_3 & : & \mathrm{c}^{8}\rightarrow\mathrm{c}^{4} & = \ \text{prodC}\langle\mathrm{c}^{2}\rangle\,\hat{}^{2}\text{prod}_2 \\
\text{prod}_4 & : & \mathrm{c}^{16}\rightarrow\mathrm{c}^{8} & = \ \text{prodC}\langle\mathrm{c}^{4}\rangle\,\hat{}^{2}\text{prod}_3 \\
 \\
\text{factor}_1 & : & \mathrm{c} & = \ W_2^0 \\
\text{factor}_2 & : & \mathrm{c}^{2} & = \ (W_4^0, W_4^1) \\
\text{factor}_3 & : & \mathrm{c}^{4} & = \ ((W_8^0, W_8^1),(W_8^2, W_8^3)) \\
\text{factor}_4 & : & \mathrm{c}^{8} & = \ (((W_{16}^0, W_{16}^1),(W_{16}^2, W_{16}^3)),((W_{16}^4, W_{16}^5),(W_{16}^6, W_{16}^7))) \\
 \\
\text{f}_1 & : & \mathrm{c}^{2}\rightarrow\mathrm{c}^{2} & = \ \lambda(x,y):\mathrm{c}^{2}.\,\text{bfly}_1\,(x,\ \text{prod}_1\,(y,\ \text{factor}_1)) \\
\text{f}_2 & : & \mathrm{c}^{4}\rightarrow\mathrm{c}^{4} & = \ \lambda(x,y):\mathrm{c}^{4}.\,\text{bfly}_2\,(x,\ \text{prod}_2\,(y,\ \text{factor}_2)) \\
\text{f}_3 & : & \mathrm{c}^{8}\rightarrow\mathrm{c}^{8} & = \ \lambda(x,y):\mathrm{c}^{8}.\,\text{bfly}_3\,(x,\ \text{prod}_3\,(y,\ \text{factor}_3)) \\
\text{f}_4 & : & \mathrm{c}^{16}\rightarrow\mathrm{c}^{16} & = \ \lambda(x,y):\mathrm{c}^{16}.\,\text{bfly}_4\,(x,\ \text{prod}_4\,(y,\ \text{factor}_4)) \\
 \\
\text{fft}_1 & : & \mathrm{c}^{2}\rightarrow\mathrm{c}^{2} & = \ \text{f}_1 \\
\text{fft}_2 & : & \mathrm{c}^{4}\rightarrow\mathrm{c}^{4} & = \ \text{f}_2\circ(\text{twoC}\langle\mathrm{c}^{2}\rangle\ \text{f}_1) \\
\text{fft}_3 & : & \mathrm{c}^{8}\rightarrow\mathrm{c}^{8} & = \ \text{f}_3\circ(\text{twoC}\langle\mathrm{c}^{4}\rangle\ \text{f}_2) \\
\text{fft}_4 & : & \mathrm{c}^{16}\rightarrow\mathrm{c}^{16} & = \ \text{f}_4\circ(\text{twoC}\langle\mathrm{c}^{8}\rangle\ \text{f}_3)
\end{array}$$

Fig. 6. Fast Fourier transform in $l\lambda$.

complex numbers and three constants `cplus`, `cminus`, and `cmult`, all of type $\mathtt{c}^2\rightarrow\mathtt{c}$, as operators on complex numbers. Twiddle factors W_i^j, indexed by i and j, are constants of type `c`.

The code in Figure 6 demonstrates how to use higher-order combinators of polymorphic type (`twoC`, `prodC`, `riffleC`, and `unriffleC`). For example, `twoC` takes a sharable input function f of type $\alpha\rightarrow\alpha$ and applies f to each element of a pair (x, y) of type α^2. Each use of `twoC` instantiates α to a sharable type (e.g., $\mathtt{c}^2$, $\mathtt{c}^4$, or $\mathtt{c}^8$) and requires a sharable input function for f. Without polymorphic types in $l\lambda$, we would have to expand each instance of `twoC` into the linear input function given in its declaration. We can also define $\circ$ as another higher order combinator of polymorphic type

$$\forall\alpha.\forall\alpha'.\forall\alpha''.(\alpha' \to \alpha'') \multimap ((\alpha \to \alpha') \multimap (\alpha \to \alpha''))$$

but here we use it as syntactic sugar assuming that the type system is capable of expanding $e_1 \circ e_2$ correctly after inferring the types of e_1 and e_2.

The actual code for the FFT circuit is 60 lines long, which includes declarations of twiddle factors and additional functions for reordering the output. Our translator of $l\lambda$ expands it to 5158 lines of Verilog code (not including blank lines), which can be thought of as an equivalent netlist specification because it consists mostly of declarations of wires and assignments between wires. We have tested the generated Verilog code for correctness on Aldec's Active-HDL simulator.

6.2.2 Bitonic sorting network

Figure 7 shows part of the code for a bitonic sorting network of size 16. Like the code for the FFT circuit, it consists of a series of declarations each of which yields a closed expression of $l\lambda$. We assume a sharable type `r` for real numbers and two constants `rmin` and `rmax`, of type $\mathtt{r}^2 \to \mathtt{r}$, as operators on real numbers. We reuse the higher-order combinators `twoC`, `riffleC`, and `unriffleC` from Figure 6, whose type variable α is now instantiated to sharable types for real numbers ($\mathtt{r}$, $\mathtt{r}^2$, $\mathtt{r}^4$, and $\mathtt{r}^8$). The actual code for the bitonic sorting network is 43 lines long and expands to 5175 lines of Verilog code by our translator.

6.3 Metaprogramming

Although polymorphism provides a basic form of metaprogramming in $l\lambda$, a practical hardware description language based on $l\lambda$ needs additional metaprogramming constructs. For example, the code in Figure 6 assumes all twiddle factors W_i^j as precalculated constants, but a more realistic approach is to calculate these constants at the metaprogramming level (e.g., with a program written in a general programming language) and then use a metaprogramming construct to import the results. A general solution is to design a metaprogramming language that uses $l\lambda$ as an object language. Such a metaprogramming language enables us to write a program that generates the code for an FFT circuit of any given size by exploiting the regular patterns of composing expressions. For example, we may think of the code in Figure 6 as the result of running the program with an input size of 2^4.

7 Related work

There are several hardware description languages embedded into existing functional languages. Hydra (O'Donnell 1995), Lava (Bjesse *et al.* 1998), Hawk (Matthews *et al.* 1998), and Wired (Axelsson *et al.* 2005) are embedded into Haskell, and HML (Li & Leeser 2000) is embedded into ML. An example of a functional language designed

$$
\begin{array}{llll}
\texttt{riffle}_1 & : & \texttt{r}^2 \rightarrow \texttt{r}^2 & = & \lambda\, p\!:\!\texttt{r}^2.\, p \\
\texttt{riffle}_2 & : & \texttt{r}^4 \rightarrow \texttt{r}^4 & = & (\texttt{riffleC}\,\langle\texttt{r}\rangle)\,\hat{}^2\texttt{riffle}_1 \\
\texttt{riffle}_3 & : & \texttt{r}^8 \rightarrow \texttt{r}^8 & = & (\texttt{riffleC}\,\langle\texttt{r}^2\rangle)\,\hat{}^2\texttt{riffle}_2 \\
\texttt{riffle}_4 & : & \texttt{r}^{16} \rightarrow \texttt{r}^{16} & = & (\texttt{riffleC}\,\langle\texttt{r}^4\rangle)\,\hat{}^2\texttt{riffle}_3 \\
 & & & & \\
\texttt{unriffle}_1 & : & \texttt{r}^2 \rightarrow \texttt{r}^2 & = & \lambda\, p\!:\!\texttt{r}^2.\, p \\
\texttt{unriffle}_2 & : & \texttt{r}^4 \rightarrow \texttt{r}^4 & = & (\texttt{unriffleC}\,\langle\texttt{r}\rangle)\,\hat{}^2\texttt{unriffle}_1 \\
\texttt{unriffle}_3 & : & \texttt{r}^8 \rightarrow \texttt{r}^8 & = & (\texttt{unriffleC}\,\langle\texttt{r}^2\rangle)\,\hat{}^2\texttt{unriffle}_2 \\
\texttt{unriffle}_4 & : & \texttt{r}^{16} \rightarrow \texttt{r}^{16} & = & (\texttt{unriffleC}\,\langle\texttt{r}^4\rangle)\,\hat{}^2\texttt{unriffle}_3 \\
 & & & & \\
\texttt{inc}_1 & : & \texttt{r}^2 \rightarrow \texttt{r}^2 & = & \lambda\, p\!:\!\texttt{r}^2.\,(\texttt{rmin}\ p, \texttt{rmax}\ p) \\
\texttt{inc}_2 & : & \texttt{r}^4 \rightarrow \texttt{r}^4 & = & (\texttt{twoC}\,\langle\texttt{r}^2\rangle)\,\hat{}^2\texttt{inc}_1 \\
\texttt{inc}_3 & : & \texttt{r}^8 \rightarrow \texttt{r}^8 & = & (\texttt{twoC}\,\langle\texttt{r}^4\rangle)\,\hat{}^2\texttt{inc}_2 \\
\texttt{inc}_4 & : & \texttt{r}^{16} \rightarrow \texttt{r}^{16} & = & (\texttt{twoC}\,\langle\texttt{r}^8\rangle)\,\hat{}^2\texttt{inc}_3 \\
 & & & & \\
\texttt{dec}_1 & : & \texttt{r}^2 \rightarrow \texttt{r}^2 & = & \lambda\, p\!:\!\texttt{r}^2.\,(\texttt{rmax}\ p, \texttt{rmin}\ p) \\
\texttt{dec}_2 & : & \texttt{r}^4 \rightarrow \texttt{r}^4 & = & (\texttt{twoC}\,\langle\texttt{r}^2\rangle)\,\hat{}^2\texttt{dec}_1 \\
\texttt{dec}_3 & : & \texttt{r}^8 \rightarrow \texttt{r}^8 & = & (\texttt{twoC}\,\langle\texttt{r}^4\rangle)\,\hat{}^2\texttt{dec}_2 \\
\texttt{dec}_4 & : & \texttt{r}^{16} \rightarrow \texttt{r}^{16} & = & (\texttt{twoC}\,\langle\texttt{r}^8\rangle)\,\hat{}^2\texttt{dec}_3 \\
 & & & & \\
\texttt{Mg\_inc}_1 & : & \texttt{r}^2 \rightarrow \texttt{r}^2 & = & \texttt{unriffle}_1 \circ \texttt{inc}_1 \circ \texttt{riffle}_1 \\
\texttt{Mg\_inc}_2 & : & \texttt{r}^4 \rightarrow \texttt{r}^4 & = & ((\texttt{twoC}\,\langle\texttt{r}^2\rangle)\,\hat{}^2\texttt{Mg\_inc}_1) \circ \texttt{unriffle}_2 \circ \texttt{inc}_2 \circ \texttt{riffle}_2 \\
\texttt{Mg\_inc}_3 & : & \texttt{r}^8 \rightarrow \texttt{r}^8 & = & ((\texttt{twoC}\,\langle\texttt{r}^4\rangle)\,\hat{}^2\texttt{Mg\_inc}_2) \circ \texttt{unriffle}_3 \circ \texttt{inc}_3 \circ \texttt{riffle}_3 \\
\texttt{Mg\_inc}_4 & : & \texttt{r}^{16} \rightarrow \texttt{r}^{16} & = & ((\texttt{twoC}\,\langle\texttt{r}^8\rangle)\,\hat{}^2\texttt{Mg\_inc}_3) \circ \texttt{unriffle}_4 \circ \texttt{inc}_4 \circ \texttt{riffle}_4 \\
 & & & & \\
\texttt{Mg\_dec}_1 & : & \texttt{r}^2 \rightarrow \texttt{r}^2 & = & \texttt{unriffle}_1 \circ \texttt{dec}_1 \circ \texttt{riffle}_1 \\
\texttt{Mg\_dec}_2 & : & \texttt{r}^4 \rightarrow \texttt{r}^4 & = & ((\texttt{twoC}\,\langle\texttt{r}^2\rangle)\,\hat{}^2\texttt{Mg\_dec}_1) \circ \texttt{unriffle}_2 \circ \texttt{dec}_2 \circ \texttt{riffle}_2 \\
\texttt{Mg\_dec}_3 & : & \texttt{r}^8 \rightarrow \texttt{r}^8 & = & ((\texttt{twoC}\,\langle\texttt{r}^4\rangle)\,\hat{}^2\texttt{Mg\_dec}_2) \circ \texttt{unriffle}_3 \circ \texttt{dec}_3 \circ \texttt{riffle}_3 \\
\texttt{Mg\_dec}_4 & : & \texttt{r}^{16} \rightarrow \texttt{r}^{16} & = & ((\texttt{twoC}\,\langle\texttt{r}^8\rangle)\,\hat{}^2\texttt{Mg\_dec}_3) \circ \texttt{unriffle}_4 \circ \texttt{dec}_4 \circ \texttt{riffle}_4
\end{array}
$$

$$
\begin{array}{lllll}
\texttt{bitonic\_inc}_1 & : & \texttt{r}^2 \rightarrow \texttt{r}^2 & = & \texttt{Mg\_inc}_1 \\
\texttt{bitonic\_dec}_1 & : & \texttt{r}^2 \rightarrow \texttt{r}^2 & = & \texttt{Mg\_dec}_1 \\
\texttt{bitonic\_inc}_2 & : & \texttt{r}^4 \rightarrow \texttt{r}^4 & = & \lambda\,(x,y)\!:\!\texttt{r}^4.\,\textbf{proj}\ \texttt{bitonic\_inc}_1\ x\ \textbf{of}\ (x_1,x_2)\ \textbf{in} \\
 & & & & \quad \textbf{proj}\ \texttt{bitonic\_dec}_1\ y\ \textbf{of}\ (y_1,y_2)\ \textbf{in} \\
 & & & & \quad\ (\texttt{Mg\_inc}_2\ ((x_1,x_2),(y_1,y_2))) \\
\texttt{bitonic\_dec}_2 & : & \texttt{r}^4 \rightarrow \texttt{r}^4 & = & \lambda\,(x,y)\!:\!\texttt{r}^4.\,\textbf{proj}\ \texttt{bitonic\_dec}_1\ x\ \textbf{of}\ (x_1,x_2)\ \textbf{in} \\
 & & & & \quad \textbf{proj}\ \texttt{bitonic\_inc}_1\ y\ \textbf{of}\ (y_1,y_2)\ \textbf{in} \\
 & & & & \quad\ (\texttt{Mg\_dec}_2\ ((x_1,x_2),(y_1,y_2))) \\
\texttt{bitonic\_inc}_3 & : & \texttt{r}^8 \rightarrow \texttt{r}^8 & = & \lambda\,(x,y)\!:\!\texttt{r}^8.\,\textbf{proj}\ \texttt{bitonic\_inc}_2\ x\ \textbf{of}\ (x_1,x_2)\ \textbf{in} \\
 & & & & \quad \textbf{proj}\ \texttt{bitonic\_dec}_2\ y\ \textbf{of}\ (y_1,y_2)\ \textbf{in} \\
 & & & & \quad\ (\texttt{Mg\_inc}_3\ ((x_1,x_2),(y_1,y_2))) \\
\texttt{bitonic\_dec}_3 & : & \texttt{r}^8 \rightarrow \texttt{r}^8 & = & \lambda\,(x,y)\!:\!\texttt{r}^8.\,\textbf{proj}\ \texttt{bitonic\_dec}_2\ x\ \textbf{of}\ (x_1,x_2)\ \textbf{in} \\
 & & & & \quad \textbf{proj}\ \texttt{bitonic\_inc}_2\ y\ \textbf{of}\ (y_1,y_2)\ \textbf{in} \\
 & & & & \quad\ (\texttt{Mg\_dec}_3\ ((x_1,x_2),(y_1,y_2))) \\
\texttt{bitonic\_inc}_4 & : & \texttt{r}^{16} \rightarrow \texttt{r}^{16} & = & \lambda\,(x,y)\!:\!\texttt{r}^{16}.\,\textbf{proj}\ \texttt{bitonic\_inc}_3\ x\ \textbf{of}\ (x_1,x_2)\ \textbf{in} \\
 & & & & \quad \textbf{proj}\ \texttt{bitonic\_dec}_3\ y\ \textbf{of}\ (y_1,y_2)\ \textbf{in} \\
 & & & & \quad\ (\texttt{Mg\_inc}_4\ ((x_1,x_2),(y_1,y_2))) \\
\texttt{bitonic\_dec}_4 & : & \texttt{r}^{16} \rightarrow \texttt{r}^{16} & = & \lambda\,(x,y)\!:\!\texttt{r}^{16}.\,\textbf{proj}\ \texttt{bitonic\_dec}_3\ x\ \mathrm{of}\ (x_1,x_2)\ \textbf{in} \\
 & & & & \quad \textbf{proj}\ \texttt{bitonic\_inc}_3\ y\ \textbf{of}\ (y_1,y_2)\ \textbf{in} \\
 & & & & \quad\ (\texttt{Mg\_dec}_4\ ((x_1,x_2),(y_1,y_2)))
\end{array}
$$

Fig. 7. Bitonic sorting network in $l\lambda$.

specifically for hardware design is *reFL^ect^* (Grundy *et al.* 2006). As it is capable of constructing and decomposing its own expressions, we may think of *reFL^ect^* as a hardware description language embedded into itself.

A technical problem with embedding a hardware description language into Haskell is that feedback circuits may give rise to infinite data structures for representing netlists because Haskell is a lazy functional language. As a solution to the problem, O'Donnell (1993) proposes to add a tag to the data type representing hardware circuits; Claessen & Sands (1999) propose an extension to Haskell called observable sharing. Such a problem does not arise in $l\lambda$ because it uses a syntax-directed translation and thus never evaluates expressions.

muFP (Sheeran 1984) is a functional hardware description language complete in itself. A characteristic feature of muFP is a small number of *combining forms*, which are higher-order combinators that can be applied to primitive or derived functions to build new functions. Combining forms contain information not only about operational behavior of hardware circuits (i.e., what they actually compute) but also about their layout (i.e., how to realize them physically). They enable us to write concise descriptions of hardware circuits that also produce compact layouts when physically realized, which is the key strength of muFP.

In comparison with muFP, $l\lambda$ has no combining forms and lacks the ability to specify the physical layout of hardware circuits, as its focus is on how to connect hardware components without regard to their relative placement. On the other hand, $l\lambda$ allows us to use $\lambda x:\theta.e$ and $\hat{\lambda} f:\kappa.e$ to directly define new functions, including higher-order combinators. If we are concerned only with operational behavior of hardware circuits, we can incorporate combining forms into $l\lambda$ as constants with appropriate translation rules. In order to express the physical layout of hardware circuits in $l\lambda$, however, we need to extend the judgment for translating expressions, which is left as future work.

T-Ruby (Sharp & Rasmussen 1995) is a functional hardware description language similar to $l\lambda$ in that its syntax is based on the standard lambda calculus. Its type system features parametric polymorphism and dependent product types, which enable programmers to write various higher-order combinators in T-Ruby itself. Like its predecessor Ruby (Jones & Sheeran 1990), however, T-Ruby adopts a relational approach to describing hardware circuits by modeling a hardware circuit as a relation between two data streams. Hence, it does not explicitly specify the direction of data flow in hardware circuits.

Although our work is concerned with structural hardware description, it is worth mentioning that there are functional languages designed for behavioral hardware synthesis such as SAFL (Mycroft & Sharp 2000). Ghica (2007) uses Basic SCI (bSCI) (O'Hearn 2003) as a higher-order functional language for hardware synthesis. The affine type system of bSCI prevents functions from sharing identifiers with their arguments, thereby achieving controlled uses of hardware circuits that cannot be shared. The linear type system of $l\lambda$ also achieves controlled uses of hardware circuits, but in the context of hardware description (rather than hardware synthesis) and with a different motivation.

8 Conclusion

We present a calculus $l\lambda$, which may serve as an intermediate functional language just above netlists in the hierarchy of hardware description languages. A characteristic feature of $l\lambda$ is its use of a linear-type system, which enforces the linear use of variables of function type and enables us to use higher-order functions. We develop a translation of $l\lambda$ into structural descriptions of hardware circuits and illustrate the feasibility of using $l\lambda$ as a practical intermediate language for hardware description by implementing an FFT circuit and a bitonic sorting network.

Although $l\lambda$ is designed primarily as an intermediate language for hardware description, developing it into a full functional hardware description language is certainly feasible. We are considering two directions in which to extend $l\lambda$. The first is to add more metaprogramming constructs, which do not increase the expressive power of $l\lambda$ but simplifies programming tasks. In addition to polymorphism discussed in Section 6.1, higher-order modules appear to be particularly attractive. The second is to define a new judgment for translating expressions so as to increase the expressive power of $l\lambda$. For example, we could extend the syntax of $l\lambda$ and incorporate a dependent-type system to express such physical properties of hardware circuits as layout, area, wiring, and power consumption. Combined together, these two directions will turn $l\lambda$ into a practical functional hardware description language.

References

Axelsson, E., Claessen, K. & Sheeran, M. (2005) Wired: Wire-aware circuit design. In *Proceedings of Conference on Correct Hardware Design and Verification Methods (CHARME)*. Springer-Verlag, pp. 5–19.

Bjesse, P., Claessen, K., Sheeran, M. & Singh, S. (1998) Lava: Hardware design in Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ACM Press, pp. 174–184.

Boute, R. T. (1984) Functional languages and their application to the description of digital systems. *Journal A*, **25** (1), 27–33.

Cardelli, L. & Plotkin, G. (1981) An algebraic approach to VLSI design. In *Proceedings of the VLSI 81 International Conference*, pp. 173–182.

Claessen, K. & Sands, D. (1999) Observable sharing for functional circuit description. In *ASIAN '99: Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science*. Springer-Verlag, pp. 62–73.

Ghica, D. R. (2007) Geometry of synthesis: A structured approach to VLSI design. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, pp. 363–375.

Grundy, J., Melham, T. & O'Leary, J. (2006) A reflective functional language for hardware design and theorem proving, *J. Funct. Program.*, **16** (2), 157–196.

Johnson, S. D. (1984) Applicative programming and digital design. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, pp. 218–227.

Jones, G. & Sheeran, M. (1990) Circuit design in Ruby, *Formal Methods VLSI Des.*, 13–70.

Li, Y. & Leeser, M. (2000) HML, a novel hardware description language and its translation to VHDL, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, **8** (1), 1–8.

Matthews, J., Cook, B. & Launchbury, J. (1998) Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages*. IEEE, p. 90.

Meshkinpour, F. & Ercegovac, M. D. (1985) A functional language for description and design of digital systems: Sequential constructs. In *Proceedings of the 22nd ACM/IEEE Conference on Design Automation*. ACM Press, pp. 238–244.

Mycroft, A. & Sharp, R. (2000) A statically allocated parallel functional language. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP 2000)*. Springer-Verlag, pp. 37–48.

O'Donnell, J. J. (1995) From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Proceedings of the First International Symposium on Functional Programming Languages in Education*. Springer-Verlag, pp. 195–214.

O'Donnell, J. T. (1993) Generating netlists from executable circuit specifications. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Springer-Verlag, pp. 178–194.

O'Hearn, P. (2003) On bunched typing. *J. Funct. Program.*, **13** (4), 747–796.

Park, S., Kim, J. & Im, H. (2008) Functional netlists. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ACM Press, pp. 353–365.

Sharp, R. & Rasmussen, O. (1995) Using a language of functions and relations for VLSI specification. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. ACM Press, pp. 45–54.

Sheeran, M. (1984) muFP, a language for VLSI design. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. ACM Press, pp. 104–112.

Sheeran, M. (2005) Hardware design and functional programming: A perfect match. *J. Universal Comput. Sci.*, **11** (7), 1135–1158.

Appendix A. Selected proofs

Below we show proofs of Proposition 4.4, Lemma 4.5, and Proposition 4.9. To simplify the proofs, we consider only sharable input functions and applications. All other cases are trivial or analogous.

Proof of Proposition 4.4

By induction on the structure of the proof of $\mathscr{G};\mathscr{D} \vdash e \Rightarrow (H, C, O)$.

Case $\mathscr{G};\mathscr{D} \vdash \lambda x{:}\theta.e \Rightarrow (H_x \cup H, C, I_x \to O)$ with $\mathscr{G} \sim \Gamma$ and $\mathscr{D} \sim \Delta$:

$$\begin{array}{ll}
\left.\begin{array}{ll}
(1) & \theta \lhd\!\rhd (H_x, I_x, O_x) \\
(2) & \mathscr{G}, x :: O_x; \mathscr{D} \vdash e \Rightarrow (H, C, O) \\
(3) & \sharp\{I_x, \mathscr{D} + H\}
\end{array}\right\} & \text{by the rule } \to I \\
\left.\begin{array}{ll}
(4) & I_x \lhd \theta \\
(5) & O_x \rhd \theta \\
(6) & \sharp\{I_x, O_x\}
\end{array}\right\} & \text{by Lemma 3.1 with (1)} \\
(7) \quad \sharp\{I_x, \mathscr{G} + \mathscr{D} + H\} & \text{by Proposition 2.3 with (3) and (4)} \\
(8) \quad \mathscr{G}, x :: O_x \sim \Gamma, x : \theta & \text{from } \mathscr{G} \sim \Gamma \text{ and (5)} \\
\left.\begin{array}{ll}
(9) & \Gamma, x : \theta; \Delta \vdash e : \tau \\
(10) & O \rhd \tau
\end{array}\right\} & \text{by IH on (2) with (8)} \\
\underline{\Gamma; \Delta \vdash \lambda x{:}\theta.e : \theta \to \tau} & \text{by the rule} \to\!\mathsf{I} \text{ with (9)} \\
(11) \quad |O| \subset |\mathscr{G} + O_x + \mathscr{D} + H| & \text{by Lemma 4.2 with (2)} \\
(12) \quad \sharp\{I_x, O\} & \text{from (6), (7) and (11)} \\
\underline{I_x \to O \rhd \theta \to \tau} & \text{by the rule} \to\rhd \text{ with (4), (10) and (12)}
\end{array}$$

Case $\mathscr{G};\mathscr{D}_1,\mathscr{D}_2 \vdash e_1\ e_2 \Rightarrow (H_1 \cup H_2, C, O_1)$ with $\mathscr{G} \sim \Gamma$, $\mathscr{D}_1 \sim \Delta_1$, and $\mathscr{D}_2 \sim \Delta_2$:

$$\left.\begin{array}{ll}(1) & \mathscr{G};\mathscr{D}_1 \vdash e_1 \Rightarrow (H_1, C_1, I_1 \to O_1)\\ (2) & \mathscr{G};\mathscr{D}_2 \vdash e_2 \Rightarrow (H_2, C_2, O_2)\\ (3) & C = C_1 \cup C_2 \cup O_2 \bowtie I_1\end{array}\right\} \quad \text{by the rule } \to E$$

$$\left.\begin{array}{ll}(4) & \Gamma;\Delta_1 \vdash e_1 : \tau_1\\ (5) & I_1 \to O_1 \rhd \tau_1\end{array}\right\} \quad \text{by IH on (1) with } \mathscr{G} \sim \Gamma \text{ and } \mathscr{D}_1 \sim \Delta_1$$

$$\left.\begin{array}{ll}(6) & \tau_1 = \theta \to \tau_1'\\ (7) & I_1 \lhd \theta\\ & \underline{O_1 \rhd \tau_1'}\end{array}\right\} \quad \text{by the rule } \to\rhd \text{ with (5)}$$

$$\left.\begin{array}{ll}(8) & \Gamma;\Delta_2 \vdash e_2 : \tau_2\\ (9) & O_2 \rhd \tau_2\end{array}\right\} \quad \text{by IH on (2) with } \mathscr{G} \sim \Gamma \text{ and } \mathscr{D}_2 \sim \Delta_2$$

(10) $\tau_2 = \theta$ by Proposition 2.5 with (3), (7) and (9)

$\underline{\Gamma;\Delta_1,\Delta_2 \vdash e_1\ e_2 : \tau_1'}$ by the rule →E with (4), (6), (8) and (10) □

Proof of Lemma 4.5
By induction on the structure of the proof of $\mathscr{G};\mathscr{D} \vdash e \Rightarrow (H, C, O)$.

Case $\mathscr{G};\mathscr{D} \vdash \lambda x{:}\theta.\, e \Rightarrow (H_x \cup H, C, I_x \to O)$ with $\mathscr{G} \sim \Gamma$ and $\mathscr{D} \sim \Delta$:

$$\left.\begin{array}{ll}(1) & \theta \lhd\rhd (H_x, I_x, O_x)\\ (2) & \mathscr{G}, x :: O_x;\mathscr{D} \vdash e \Rightarrow (H, C, O)\\ (3) & \sharp\{I_x, \mathscr{D} + H\}\end{array}\right\} \quad \text{by the rule } \to I$$

$$\left.\begin{array}{ll}(4) & I_x \lhd \theta\\ (5) & O_x \rhd \theta\\ (6) & \sharp\{I_x, O_x\}\end{array}\right\} \quad \text{by Lemma 3.1 with (1)}$$

(7) $\sharp\{I_x, \mathscr{G} + \mathscr{D} + H\}$ by Proposition 2.3 with (3) and (4)

$i \in |I_x \to O|$ assumption

if $i \in |O|$,

(8) $\mathscr{G}, x :: O_x \sim \Gamma, x : \theta$ from $\mathscr{G} \sim \Gamma$ and (5)

$o \mapsto i \notin C$ by IH on (2) with (8)

if $i \in |I_x|$,

(9) $\sharp\{I_x, \mathscr{G} + O_x + \mathscr{D} + H\}$ from (6) and (7)

(10) $|C| \subset |\mathscr{G} + O_x + \mathscr{D} + H|$ by Lemma 4.2 with (2)

(11) $\sharp\{I_x, C\}$ from (9) and (10)

$o \mapsto i \notin C$ from $i \in |I_x|$ and (11)

Case $\mathscr{G};\mathscr{D}_1,\mathscr{D}_2 \vdash e_1\ e_2 \Rightarrow (H_1 \cup H_2, C, O_1)$ with $\mathscr{G} \sim \Gamma$, $\mathscr{D}_1 \sim \Delta_1$, and $\mathscr{D}_2 \sim \Delta_2$:

$$\left.\begin{array}{ll}(1) & \mathscr{G};\mathscr{D}_1 \vdash e_1 \Rightarrow (H_1, C_1, I_1 \to O_1)\\ (2) & \mathscr{G};\mathscr{D}_2 \vdash e_2 \Rightarrow (H_2, C_2, O_2)\\ (3) & C = C_1 \cup C_2 \cup O_2 \bowtie I_1\\ (4) & \sharp\{\mathscr{D}_1 + H_1, \mathscr{D}_2 + H_2\}\end{array}\right\} \quad \text{by the rule } \to E$$

$$\left.\begin{array}{ll}(5) & i \in |O_1|\\ (6) & \mathscr{D}_1 \sim \Delta_1\\ (7) & \mathscr{D}_2 \sim \Delta_2\end{array}\right\} \quad \text{assumption}$$

(8) $I_1 \to O_1 \rhd \tau$ by Lemma 4.4 with (1)

$$\begin{array}{ll}
\quad (9) \quad \sharp\{I_1, O_1\} & \text{by the rule } \to\!\rhd \text{ with (8)} \\
\quad (10) \quad i \in |I_1 \to O_1| & \text{from (5)} \\
\quad (11) \quad i \notin |I_1| & \text{from (5) and (9)} \\
\quad (12) \quad i \in |\mathscr{D}_1 + H_1| & \text{by Corollary 4.3 with (1), (6) and (10)} \\
o \mapsto i \notin C_1 & \text{by IH on (1) with (10)} \\
\text{if } o \mapsto i \in C_2, & \\
\quad (13) \quad i \in |\mathscr{D}_2 + H_2| & \text{by Corollary 4.3 with (2), (7), and } i \in |C_2| \\
\quad \text{contradiction} & \text{from (4), (12), and (13)} \\
\text{if } o \mapsto i \in O_2 \bowtie I_1, & \\
\quad (14) \quad i \in |O_2| & \text{from (11)} \\
\quad (15) \quad i \in |\mathscr{D}_2 + H_2| & \text{by Corollary 4.3 with (2) and (14)} \\
\quad \text{contradiction} & \text{from (4), (12), and (15)} \\
o \mapsto i \notin C & \text{from } o \mapsto i \notin C_1,\ o \mapsto i \notin C_2, \text{ and } o \mapsto i \notin O_2 \bowtie I_1 \quad \square
\end{array}$$

Proof of Proposition 4.9

By induction on the structure of the proof of $\Gamma;\Delta \vdash e : \tau$.

Case $\Gamma;\Delta \vdash \lambda x\!:\!\theta.\,e : \theta \to \tau$ with $\mathscr{G} \sim \Gamma$ and $\mathscr{D} \sim \Delta$:

$$\begin{array}{ll}
(1) \quad \Gamma, x : \theta; \Delta \vdash e : \tau & \text{by the rule } \to\mathsf{I} \\
\left.\begin{array}{l} (2) \quad \theta \lhd\!\rhd (H_x, I_x, O_x) \\ (3) \quad \sharp\{O_x, \mathscr{G} + \mathscr{D}\} \\ (4) \quad \sharp\{I_x, \mathscr{D} + H\} \end{array}\right\} & \text{assumption} \\
\left.\begin{array}{l} (5) \quad I_x \lhd \theta \\ (6) \quad O_x \rhd \theta \end{array}\right\} & \text{by Lemma 3.1 with (2)} \\
(7) \quad \mathscr{G}, x :: O_x \sim \Gamma, x : \theta & \text{from } \mathscr{G} \sim \Gamma \text{ and (6)} \\
\left.\begin{array}{l} (8) \quad \mathscr{G}, x :: O_x; \mathscr{D} \vdash e \Rightarrow (H, C, O) \\ (9) \quad O \rhd \tau \end{array}\right\} & \text{by IH on (1) with (7)} \\
\quad \underline{\mathscr{G};\mathscr{D} \vdash \lambda x\!:\!\theta.\,e \Rightarrow (H_x \cup H, C, I_x \to O)} & \text{by the rule } \to\! I \text{ with (2), (3), (4), and (8)} \\
(10) \quad \sharp\{I_x, O\} & \text{by Lemma 4.2 with (4) and (8)} \\
\quad \underline{I_x \to O \rhd \theta \to \tau} & \text{by the rule } \to\!\rhd \text{ with (5), (9), and (10)}
\end{array}$$

Case $\Gamma;\Delta_1,\Delta_2 \vdash e_1\ e_2 : \tau$ with $\mathscr{G} \sim \Gamma$, $\mathscr{D}_1 \sim \Delta_1$, $\mathscr{D}_2 \sim \Delta_2$, and $\sharp\{\mathscr{D}_1, \mathscr{D}_2\}$:

$$\begin{array}{ll}
\left.\begin{array}{l} (1) \quad \Gamma;\Delta_1 \vdash e_1 : \theta \to \tau \\ (2) \quad \Gamma;\Delta_2 \vdash e_2 : \theta \end{array}\right\} & \text{by the rule } \to\mathsf{E} \\
\left.\begin{array}{l} (3) \quad \mathscr{G};\mathscr{D}_1 \vdash e_1 \Rightarrow (H_1, C_1, O_1') \\ (4) \quad O_1' \rhd \theta \to \tau \end{array}\right\} & \text{by IH on (1) with } \mathscr{G} \sim \Gamma \text{ and } \mathscr{D}_1 \sim \Delta_1 \\
(5) \quad \sharp\{H_1, \mathscr{D}_2\} & \text{assumption} \\
\left.\begin{array}{l} \quad O_1' = I_1 \to O_1 \\ (6) \quad I_1 \lhd \theta \\ \quad \underline{O_1 \rhd \tau} \end{array}\right\} & \text{by the rule } \to\!\rhd \text{ with (4)} \\
\left.\begin{array}{l} (7) \quad \mathscr{G};\mathscr{D}_2 \vdash e_2 \Rightarrow (H_2, C_2, O_2) \\ (8) \quad O_2 \rhd \theta \end{array}\right\} & \text{by IH on (2) with } \mathscr{G} \sim \Gamma \text{ and } \mathscr{D}_2 \sim \Delta_2 \\
(9) \quad \sharp\{H_2, \mathscr{D}_1 + H_1\} & \text{assumption}
\end{array}$$

(10) $O_2 \bowtie I_1$ valid by Proposition 2.4 with (6) and (8)
(11) $\sharp\{\mathscr{D}_1 + H_1, \mathscr{D}_2 + H_2\}$ from $\sharp\{\mathscr{D}_1, \mathscr{D}_2\}$, (5) and (9)

$$\frac{}{\mathscr{G};\mathscr{D}_1,\mathscr{D}_2 \vdash e_1\ e_2 \Rightarrow (H_1 \cup H_2, C_1 \cup C_2 \cup O_2 \bowtie I_1, O_1)}$$

by the rule $\rightarrow E$ with (3), (7), (10), and (11) □

Appendix B. Proof of the completeness of the translation

We extend $l\lambda$ with patterns to be used in fixed-point expressions:

$$\begin{array}{lcll} \text{pattern} & p & ::= & x \mid (p,p) \\ \text{expression} & e & ::= & \cdots \mid \textbf{fix}\ p\!:\!\theta.\, e \end{array}$$

A tuple pattern $(p_1, p_2, \ldots, p_n)$ is syntactic sugar for $(p_1, (p_2, (\ldots, p_n)\cdots))$, and a tuple type $\theta_1 \times \theta_2 \times \cdots \times \theta_n$ is syntactic sugar for $\theta_1 \times (\theta_2 \times (\cdots \times \theta_n)\cdots))$. An output interface $O_1 \times O_2 \times \cdots \times O_n$ and an input interface $I_1 \times I_2 \times \cdots \times I_n$ are similarly defined as syntactic sugar.

We use a new judgment $p;\theta \lhd\!\rhd (H, I, O);\mathscr{G}$ for translating patterns. It means that pattern p of type θ may use I and O as its input and output interfaces and that $\mathscr{G}$ associates output terminals in O with variables in p.

$$\frac{\theta \lhd\!\rhd (H, I, O)}{x;\theta \lhd\!\rhd (H, I, O); x :: O}\ \textit{VarPat} \lhd\!\rhd$$

$$\frac{p_1;\theta_1 \lhd\!\rhd (H_1, I_1, O_1);\mathscr{G}_1 \quad p_2;\theta_2 \lhd\!\rhd (H_2, I_2, O_2);\mathscr{G}_2 \quad \sharp\{H_1, H_2\}}{(p_1, p_2);\theta_1 \times \theta_2 \lhd\!\rhd (H_1 \cup H_2, I_1 \times I_2, O_1 \times O_2);\mathscr{G}_1, \mathscr{G}_2}\ \textit{PairPat} \lhd\!\rhd$$

We revise the rule *Fix* which is the only rule using the judgment $p;\theta \lhd\!\rhd (H, I, O);\mathscr{G}$ (because patterns are used only in fixed-point expressions):

$$\frac{p;\theta \lhd\!\rhd (H_p, I_p, O_p);\mathscr{G}' \quad \mathscr{G}, \mathscr{G}';\mathscr{D} \vdash e \Rightarrow (H, C, O) \quad \sharp\{O_p, \mathscr{G} + \mathscr{D}\} \quad \sharp\{I_p, \mathscr{D} + H\}}{\mathscr{G};\mathscr{D} \vdash \textbf{fix}\ p\!:\!\theta.\, e \Rightarrow (H_p \cup H, C \cup O \bowtie I_p, O)}\ \textit{Fix}$$

Proof of Theorem 4.12

Suppose that H consists of atomic hardware components a_q with input terminals $i_{q1}, \ldots, i_{qj_q}$ and output terminals $o_{q1}, \ldots, o_{qk_q}$ where $1 \leqslant q \leqslant n$:

$$H = \{a_1[i_{11}, \ldots, i_{1j_1}, o_{11}, \ldots, o_{1k_1}], \ldots, a_n[i_{n1}, \ldots, i_{nj_n}, o_{n1}, \ldots, o_{nk_n}]\}$$

We assume that all atomic hardware components have their corresponding constants $c_1, \ldots, c_n$:

$$\begin{gathered} \cdot;\cdot \vdash c_1 \Rightarrow (\{a_1[i_{11}, \ldots, i_{1j_1}, o_{11}, \ldots, o_{1k_1}]\}, \varnothing, i_{11} \rightarrow \cdots i_{1j_1} \rightarrow (o_{11} \times \cdots \times o_{1k_1})) \\ \vdots \\ \cdot;\cdot \vdash c_n \Rightarrow (\{a_n[i_{n1}, \ldots, i_{nj_n}, o_{n1}, \ldots, o_{nk_n}]\}, \varnothing, i_{n1} \rightarrow \cdots i_{nj_n} \rightarrow (o_{n1} \times \cdots \times o_{nk_n})). \end{gathered}$$

Let m be the number of input terminals in H that have no associated connection constraints in C, i.e., $m = |\{i \mid i \in |H|, i \notin |C|\}|$. Then we construct a new expression

$$\begin{array}{rcl} e & = & \lambda x_1 : \mathbf{1} \ldots \lambda x_m : \mathbf{1}.\, \mathbf{fix}\ ((y_{11}, \ldots, y_{1k_1}), \ldots, (y_{n1}, \ldots, y_{nk_n})) : \theta. \\ & & \qquad ((c_1\ z_{11}\ \ldots\ z_{1j_1}), \ldots, (c_n\ z_{n1}\ \ldots\ z_{nj_n})) \\ \theta & = & (\underbrace{\mathbf{1} \times \cdots \times \mathbf{1}}_{k_1 \text{ times}}) \times \cdots \times (\underbrace{\mathbf{1} \times \cdots \times \mathbf{1}}_{k_n \text{ times}}) \end{array}$$

where z_{st} $(1 \leqslant s \leqslant n,\ 1 \leqslant t \leqslant j_s)$ is determined as follows:

- We set z_{st} to y_{qr} if $o_{qr} \mapsto i_{st} \in C$, where $1 \leqslant q \leqslant n$, $1 \leqslant r \leqslant k_q$. That is, y_{qr} corresponds to output terminal o_{qr}.
- We set z_{st} to x_l for some l $(1 \leqslant l \leqslant m)$ if $i_{st} \notin |C|$. That is, x_l corresponds to a certain input terminal exposed to external hardware circuits. Since there are exactly m input terminals i_{st} not in $|C|$, we can assign a unique x_l to z_{st}.

Now we show that e is an expression describing the given hardware circuit. We first prove $\cdot;\cdot \vdash e \Rightarrow (H'', C'', O'')$ and then prove that (H'', C'') reduces to (H, C). In our proof, we write $\mathcal{G}(x)$ to denote O when $x :: O \in \mathcal{G}$.

We introduce sets of hardware components, sharable output contexts, and sets of connection constraints as follows:

$$\begin{array}{rcl} H_1 & = & \{a_1[i_{11}, \ldots, i_{1j_1}, o_{11}, \ldots, o_{1k_1}]\} \\ \vdots & & \\ H_n & = & \{a_n[i_{n1}, \ldots, i_{nj_n}, o_{n1}, \ldots, o_{nk_n}]\} \\ H_p & = & \{\mathsf{pt}[i'_{11}, o'_{11}], \ldots, \mathsf{pt}[i'_{1k_1}, o'_{1k_1}], \ldots, \mathsf{pt}[i'_{n1}, o'_{n1}], \ldots, \mathsf{pt}[i'_{nk_n}, o'_{nk_n}]\} \\ H_x & = & \{\mathsf{pt}[i'_1, o'_1], \ldots, \mathsf{pt}[i'_m, o'_m]\} \\ H' & = & H \cup H_p \\ H'' & = & H \cup H_p \cup H_x \end{array}$$

$$\begin{array}{rcl} \mathcal{G} & = & \mathcal{G}', \mathcal{G}'' \\ \mathcal{G}' & = & y_{11} :: o'_{11}, \ldots, y_{1k_1} :: o'_{1k_1}, \ldots, y_{n1} :: o'_{n1}, \ldots, y_{nk_n} :: o'_{nk_n} \\ \mathcal{G}'' & = & x_1 :: o'_1, \ldots, x_m :: o'_m \end{array}$$

$$\begin{array}{rcl} C_1 & = & \{\mathcal{G}(z_{11}) \mapsto i_{11}, \ldots, \mathcal{G}(z_{1j_1}) \mapsto i_{1j_1}\} \\ \vdots & & \\ C_n & = & \{\mathcal{G}(z_{n1}) \mapsto i_{n1}, \ldots, \mathcal{G}(z_{nj_n}) \mapsto i_{nj_n}\} \\ C' & = & C_1 \cup \cdots \cup C_n \\ C'' & = & C' \cup \{o_{11} \mapsto i'_{11}, \ldots, o_{1k_1} \mapsto i'_{1k_1}, \ldots, o_{n1} \mapsto i'_{n1}, \ldots, o_{nk_n} \mapsto i'_{nk_n}\} \end{array}$$

- We have $H = H_1 \cup \cdots \cup H_n$.
- H_p contains connection points $\mathsf{pt}[i'_{qr}, o'_{qr}]$ $(1 \leqslant q \leqslant n,\ 1 \leqslant r \leqslant k_q)$ to be created from variables y_{qr} in the fixed-point expression. $\mathcal{G}'$ binds variables y_{qr} to output terminals o'_{qr}.
- H_x contains connection points $\mathsf{pt}[i'_l, o'_l]$ $(1 \leqslant l \leqslant m)$ to be created from variables x_l. $\mathcal{G}''$ binds variables x_l to output terminals o'_l.
- We assume that all input and output terminals in H'' are distinct.
- Every connection constraint $o' \mapsto i$ in C' connects output terminal o' of some connection point to input terminal i of some atomic hardware component.

- Every connection constraint $o \mapsto i'$ in $C'' - C'$ connects output terminal o of some atomic hardware component to input terminal i' of some connection point.

The proof of $\cdot;\cdot \vdash e \Rightarrow (H'', C'', O'')$ proceeds as follows.

$$(1)\quad \left.\begin{array}{l} \mathscr{G};\cdot \vdash c_1 \Rightarrow (H_1, \emptyset, i_{11} \rightarrow \cdots i_{1j_1} \rightarrow (o_{11} \times \cdots \times o_{1k_1})) \\ \qquad\qquad\vdots \\ \mathscr{G};\cdot \vdash c_n \Rightarrow (H_n, \emptyset, i_{n1} \rightarrow \cdots i_{nj_n} \rightarrow (o_{n1} \times \cdots \times o_{nk_n})) \end{array}\right\}$$

by weakening the assumption on each atomic hardware component

$$(2)\quad \left.\begin{array}{l} \mathscr{G};\cdot \vdash c_1\ z_{11}\ \cdots\ z_{1j_1} \Rightarrow (H_1, \emptyset, C_1, o_{11} \times \cdots \times o_{1k_1}) \\ \qquad\qquad\vdots \\ \mathscr{G};\cdot \vdash c_n\ z_{n1}\ \cdots\ z_{nj_n} \Rightarrow (H_n, \emptyset, C_n, o_{n1} \times \cdots \times o_{nk_n}) \end{array}\right\} \text{by the rule} \rightarrow\! E \text{ with (1)}$$

(3) $O' = (o_{11} \times \cdots \times o_{1k_1}) \times \cdots \times (o_{n1} \times \cdots \times o_{nk_n})$ assumption

(4) $\mathscr{G};\cdot \vdash ((c_1\ z_{11}\ \ldots\ z_{1j_1}), \ldots, (c_n\ z_{n1}\ \ldots\ z_{nj_n})) \Rightarrow (H, C', O')$

by the rule $\times I$ with (2) and $H = H_1 \cup \cdots \cup H_n$

$$(5)\quad \left.\begin{array}{l} I_p = (i'_{11} \times \cdots \times i'_{1k_1}) \times \cdots \times (i'_{n1} \times \cdots \times i'_{nk_n}) \\ O_p = (o'_{11} \times \cdots \times o'_{1k_1}) \times \cdots \times (o'_{n1} \times \cdots \times o'_{nk_n}) \end{array}\right\} \quad \text{assumption}$$

(6) $((y_{11}, \ldots, y_{1k_1}), \ldots, (y_{n1}, \ldots, y_{nk_n})); \theta \lhd\!\rhd (H_p, I_p, O_p); \mathscr{G}'$

by the rule *PairPat* $\lhd\!\rhd$ with (5)

(7) $O' \bowtie I_p = \{o_{11} \mapsto i'_{11}, \ldots, o_{1k_1} \mapsto i'_{1k_1}, \ldots, o_{n1} \mapsto i'_{n1}, \ldots, o_{nk_n} \mapsto i'_{nk_n}\}$

from (3) and (5)

$$(8)\quad \mathscr{G}'';\cdot \vdash \begin{array}{l} \textbf{fix}\ ((y_{11}, \ldots, y_{1k_1}), \ldots, (y_{n1}, \ldots, y_{nk_n})) : \theta. \\ \quad ((c_1\ z_{11}\ \ldots\ z_{1j_1}), \ldots, (c_n\ z_{n1}\ \ldots\ z_{nj_n})) \end{array} \Rightarrow (H', C'', O')$$

by the rule *Fix* with (4), (6), and (7)

$\underline{\cdot;\cdot \vdash e \Rightarrow (H'', C'', O'')}$ where $O'' = i'_1 \rightarrow \cdots \rightarrow i'_m \rightarrow O'$ by the rule $\rightarrow\! I$ with (8)

The proof that (H'', C'') reduces to (H, C) uses the following property of C'':

(9) If $o \mapsto i \in C''$, then either $o \in |H|$ and $i \notin |H|$ or $o \notin |H|$ and $i \in |H|$.

- If $o \mapsto i \in C$ and $\{i, o\} \subset |H|$,

  (10) $i = i_{st}$ for some s and t $(1 \leqslant s \leqslant n, 1 \leqslant t \leqslant j_s)$ from $i \in |H|$

  (11) $o = o_{qr}$ for some q and r $(1 \leqslant q \leqslant n, 1 \leqslant r \leqslant k_q)$ from $o \in |H|$

  (12) $z_{st} = y_{qr}$ from the definition of e with $o_{qr} \mapsto i_{st} \in C$

  (13) $z_{st} :: o'_{qr} \in \mathscr{G}$ from (12)

  $\underline{o \mapsto i'_{qr} \in C''}$ from $o_{qr} \mapsto i'_{qr} \in C''$ and (11)

  $\underline{o'_{qr} \mapsto i \in C''}$ from $\mathscr{G}(z_{st}) \mapsto i_{st} \in C''$, (10), and (13)

  $\underline{\mathsf{pt}[i'_{qr}, o'_{qr}] \in H''}$ from assumption of H''

- If $\{o \mapsto i_1, o_1 \mapsto i_2, \ldots, o_{u-1} \mapsto i_u, o_u \mapsto i\} \subset C''$ (where $u \geqslant 0$), $\{\mathsf{pt}[i_1, o_1], \mathsf{pt}[i_2, o_2], \ldots, \mathsf{pt}[i_{u-1}, o_{u-1}], \mathsf{pt}[i_u, o_u]\} \subset H''$, and $\{i, o\} \subset |H|$,

  (14) $u = 1$ from (9)

  (15) $\{o \mapsto i_1, o_1 \mapsto i\} \subset C''$ from (14)

  (16) $i = i_{st}$ for some s and t $(1 \leqslant s \leqslant n, 1 \leqslant t \leqslant j_s)$ from $i \in |H|$

  (17) $o = o_{qr}$ for some q and r $(1 \leqslant q \leqslant n, 1 \leqslant r \leqslant k_q)$ from $o \in |H|$

(18)	$i_1 = i'_{qr}$	from $o_{qr} \mapsto i'_{qr} \in C''$, (15), and (17)
(19)	$o_1 = \mathscr{G}(z_{st})$	from $\mathscr{G}(z_{st}) \mapsto i_{st} \in C''$, (15), and (16)
(20)	$o_1 = o'_{qr}$	from $\mathsf{pt}[i'_{qr}, o'_{qr}] \in H''$, (18), and $\mathsf{pt}[i_1, o_1] \in H''$
(21)	$z_{st} :: o'_{qr} \in \mathscr{G}$	from (19) and (20)
(22)	$z_{st} = y_{qr}$	from (21)
	$\underline{o \mapsto i \in C}$	from the definition of e with (16), (17), and (22)

$\square$